2

DTIC FILE COPY

RADC-TR-90-21
Final Technical Report
April 1990

AD-A225 199

PARALLEL LOGIC PROGRAMMING ARCHITECTURES

Auburn University

William B. Day

DTIC
ELECTE
AUG 09 1990
S B D

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

90 [illegible] 036

This report has been reviewed by the RADC Public Affairs Division (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-90-21 has been reviewed and is approved for publication.

APPROVED: John J. Crowter

JOHN J. CROWTER
Project Engineer

APPROVED: [illegible]

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

FOR THE COMMANDER: [illegible]

IGOR G. PLONISCH
Directorate of Plans & Programs

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

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

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data needed, and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE April 1990	3. REPORT TYPE AND DATES COVERED Final Jun 88 to May 89

4. TITLE AND SUBTITLE
PARALLEL LOGIC PROGRAMMING ARCHITECTURES

5. FUNDING NUMBERS
C - F30602-88-D-0027
PE - 63728F
PR - 2532
TA - 01
WU - P5

6. AUTHOR(S)
William B. Day

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Auburn University
Department of Computer Science & Engineering
Auburn AL 36849

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Air Development Center (COES)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RADC-TR-90-21

11. SUPPLEMENTARY NOTES
RADC Project Engineer: John J. Crowter/COES/(315) 330-3564
The prime contractor of this effort is Syracuse University.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The objective of this effort was to investigate methods for distributing a knowledge base over multiple processors using the logic programming paradigm. A three-fold approach was employed and is discussed in this report. The first phase involved the testing of the Parallel Knowledge-based Execution System (PKES). Secondly, memory allocation techniques were investigated and evaluated. Lastly, a study of the implementation of data-typing in a logic programming paradigm is discussed.

14. SUBJECT TERMS
Parallel Architectures, Distributed Systems, Logic Programming, Prolog, Distributed Knowledge Bases, Parallel Algorithms

15. NUMBER OF PAGES
132

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19 SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

TABLE OF CONTENTS

Accession For	
NTIS GRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

QUALITY INSPECTED 4

List of Figures

1. INTRODUCTION

The objective of this effort is to investigate methods for distributing a knowledge base over multiple processors using the logic programming paradigm. A threefold approach has been employed. First, the Parallel Knowledge-based System (PKS) has been tested more thoroughly. Secondly, memory allocation schemes have been investigated and evaluated using the PKS. The third phase has studied the employment of data typing in the logic programming schema.

In Section 2 we present an AND-parallel execution scheme for logic programs which are distributed in a large computer network. The clauses in each processing unit (called a node) are assumed to be statically allocated and will not be passed from one node to another. A three-layer architecture is proposed to separate the problems of program interpretation, parallel execution and communication. The simple concept of variable holding provides the semaphore mechanism to exploit AND-parallel literals concurrently without causing binding conflicts; no data dependency analysis is needed. The forward execution and backtracking algorithms are described as transitions of variable states and literal states. A set of communication protocols describes how the distributed computation is achieved with parallel processing.

Section 3 examines a class of static allocation schemes, each of which decomposes a logic program into knowledge sources or nodes. Each scheme assigns a weight to a procedural bundle (predicate) and then uses a scale-balancing technique to assign bundles to different nodes. Two of the allocation schemes are applied to A* search.

Section 4 presents two applications of the PKS. The first application is an example of network management and system control for communications systems, as described in [CONRY 88]. The problem deals with restoring end-to-end user connectivity in a network following disruption of a link. The second application is a blocks world example with multi-agent planning. This planning example is devised to illustrate these points: hierarchical planning for multiple agents, knowledge organization and allocation, and cooperative parallel execution.

Section 5 is an initial investigation into efficiency considerations within the PKS and provides a general background for data typing. In this section four knowledge base management techniques are presented. Also advanced machine learning methods are addressed vis-a-vis data typing.

The last section is a summary of this report and describes future extensions of the theory and applications of the PKS.

2. AN AND-PARALLEL EXECUTION MODEL FOR DISTRIBUTED LOGIC PROGARMS

In this section we present an AND-parallel exeution scheme for logic programs which are distributed in a large computer network. This model extends the initial description of the parallel knowledge-based system, as given in [CHUNG 87]. Section 2.1 is an introduction to parallel problem solving on a distributed knowledge- based system. Section 2.2 discusses the overall system organization and strategies for parallel and distributed operation. Sections 2.3 and 2.4 describe the mechanisms of forward execution and backtracking, respectively. Several simple examples are also included in these two sections to elucidate the details. Finally, Section 2.5 describes the required communications between processors of the distributed system.

2.1 INTRODUCTION

Logic programming has been widely applied to the construction of deductive databases, expert systems and many other knowledge-based systems. A sophisticated application requires a large amount of knowledge, which is frequently distributed among the nodes of a computer network. In this paper we address the problem of parallel problem solving on a distributed knowledge-base system, assuming that logic programming is adopted for the common framework of knowledge representation and the common semantics of system operation. Three major issues are involved in our problem domain: (1) problem solving under the semantics of logic programming, (2) parallelism detection and deployment, and (3) communication and coordination. The body of this section presents details of the Parallel Knowledge-based System (PKS), which is an integrated approach to these problems.

The potential application of this cooperative distributed problem solving scheme is unlimited. A typical example is the knowledge-based network switching problem. Based on the prior experience of the network operation, each region of the network has some knowledge of the typical patterns of local switching requests at a given time, and the performance of different switching strategies in different situations. All that knowledge, together with the dynamic information regarding the local state in which the region is currently operating, may be coded as a logic program for local decision making. Thus, a global switching plan is partitioned as a region switching plan and a set of detailed switching plans within each involved region, with the latter plans derived independently from the local knowledge bases of the regions. This example can be easily generalized as a large class of problems in which an overall problem may be decomposed as a set of sub-problems. Each sub-problem is then solved distributedly by a local agent with its local knowledge.

The ample parallelism in logic programs provides the opportunity for concurrent problem solving. There are two major types of parallelism in logic programs. AND-parallelism refers to a simultaneous execution of ANDed literals in a clause body, while OR-parallelism refers to a simultaneous exploration of ORed clauses for a given goal. Parallel problem solving in a distributed environment mimics coordinating many experts to solve a complicated problem concurrently. This report is primarily concerned with AND-parallelism, and the readers are assumed to know the basis of logic programming.

2.1.1 AND-PARALLELISM

The fundamental problem of AND-parallel execution is binding conflicts of literals which share variables. Given a clause such as

```
p(X) :- q(X), r(X), s(X).
```

if procedure p is called with an unbound variable, then a parallel execution of literals q(X), r(X) and s(X) could produce different values for the same variable X. Two major approaches have been taken to avoid this binding conflict problem. The first approach requires a programmer to specify literal synchronization constraints by different parallel language constructs (see [CLARK 86], [SHAPIRO 86], and [UEDA 86]). The programmer assumes full responsibility of controlling the parallel execution. The second approach requires the system to generate partial orders of ANDed literals based on a data dependency analysis of shared variables. As long as these partial orders are observed, literals may be executed in parallel.

Unfortunately, in logic programs the data dependency is not static. As the program execution proceeds, variables may be bound and the data dependency changes. In the previous example, depending on which literal binds (grounds) the variable X, the execution sequence may be described by one of the three graphs in Figure 1. When variables are instantiated to dependent terms, the data dependency gets even more complicated. Consider the following clause

```
    t(X,Y) :- u(X), v(Y).
```

If the procedure call contains dependent arguments such as t(A,A) or t(A,f(A)), then parallel execution of u(X) and v(Y) may also cause binding conflicts. This type of data dependency may be nested inside structures several levels deep.

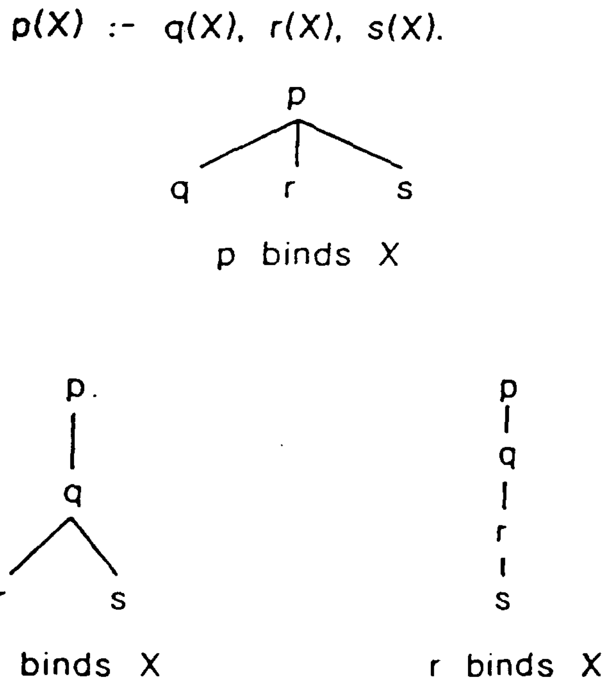


Figure 1. Different execution graphs for a clause.

Therefore, in order to avoid binding conflicts, the system needs to make sure that a group of literals does not share non-ground, dependent variables before it starts the parallel execution for them. Two major questions are (1) detecting the literals which share non-ground, dependent variables and (2) preventing their parallel execution.

2.1.2 PREVIOUS APPROACHES

Several models have been proposed to handle AND-parallel execution of logic programs in a multiprocessing environment. They employ different strategies regarding when and how the detection and the prevention could be done. They share one common theme in addressing the second question: some literals are (at least implicitly) ordered to prevent the parallel execution.

[CONERY 81] proposed a set of elaborate algorithms which order literals dynamically, based on a run-time data dependency analysis. Since the algorithms have to be applied to re-order the literals each time a non-ground binding is generated, the run-time overhead is generally considered very high.

[CHANG 85] devised a static data dependency analysis to reduce the run-time overhead. Their scheme requires a programmer to provide the possible activation modes for each entry procedure. The analyzer considers the worst-case (in terms of the possibilities of variables being unbound and dependent) to determine if literals may be executed in parallel. This worst-case consideration results in a reduction of AND-parallelism that could be exploited.

DeGroot's Restricted AND-Parallelism (RAP)(see [DEGROOT 84]) performs a compile-time data dependency analysis and then encodes execution graph expressions associated with the results of run-time groundness and dependency tests. Two approximation algorithms are designed to make the groundness test and the dependency test very simple. Because of the limitation of execution graph expressions, some types of AND-parallelism are restricted.

[LIN 86] proposed a token-passing model to avoid the binding conflict problem. Each unbound variable in that model has a token. The token contains a list of left-to-right ordered literals which share the variable. The leftmost literal in the list initially has the token. When this literal is solved, it passes the token to the next literal according to the literal order in the token. When a literal receives all the tokens for its unbound variables, it is granted execution. While this scheme easily avoids the problem of binding conflicts, the predetermined literal order limits the opportunity of AND-parallel execution.

2.1.3 A DISTRIBUTED APPROACH

The approaches in the last section constitute several process-oriented execution models, in which the execution of a literal is typically considered an independent process. The execution algorithms in those models give the scheduling strategies for these processes. While process-oriented models may be efficiently implemented on shared-memory multiprocessors, they are not compatible with the distributed processing requirements in our problem domain (unless each node is assumed to have a redundant copy of the global knowledge base). In a distributed environment the execution of a literal may or may not be an independent process, depending on whether the local knowledge base has the clauses to solve the literal or not. Consequently, a scheduling algorithm has to take this consideration into account.

Another difference in our approach to the scheduling problem is that we abandon the concept of data dependency analysis and replace it with semaphores for mutually exclusive literals. Instead of determining a set of literals which may run in parallel, the system simply holds shared variables to prevent any possible conflicts. This change makes distributed decision-making very simple and efficient. We showed that this variable holding approach may produce a high degree of parallelism with minimal overhead [CHUNG 89]. Since data dependency of ANDed literals changes dynamically, a total ordering strategy will waste the system's computation resource, or restrict the opportunity of parallel execution, or both. In Conery and Kibler's approach, for example, the system makes an overall plan, consults a small portion of the plan (regarding which literals may be executed now), and then re-plans the whole execution sequence based on the binding situation of these executed literals. A large portion of the sequence is repeatedly constructed but rarely used.

Based on these observations of scheduling strategies, we designed a different execution algorithm for distributed environment. The rest of this section presents this PKS model for parallel and distributed execution of logic programs. Section 2.2 outlines the overall system architecture and basic execution strategies. Section 2.3 presents the forward execution algorithms, followed by examples to illustrate forward execution. Section 2.4 contains a companion backtracking algorithm and examples to continue the problem solving process in the previous examples. Section 2.5 discusses communication problems and how the system is implemented in each node.

2.2 ABSTRACT INTERPRETATION

This section discusses the overall system organization and the basic strategies for parallel and distributed operation. The architecture of the system will be introduced first, followed by a description on the communication primitives for cooperative problem solving. The argument modes, variable states and literal states are then presented to show how the semaphore mechanism is devised to regulate parallel execution.

2.2.1 SYSTEM ARCHITECTURE

Similar to many sophisticated systems [TANENBAUM 88], this distributed system is constructed in several layers, with each layer supporting a specific function of the system. Figure 2 depicts the three-layer structure of the current system architecture.

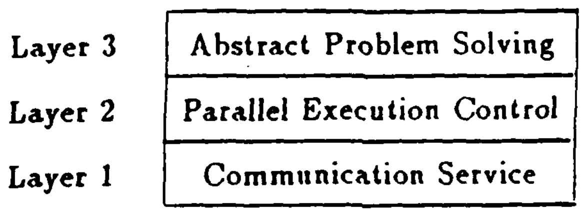


Figure 2. System architecture.

The highest layer, layer 3, is responsible for abstract problem solving. Assuming that the lower level system furnishes a goal and a local clause, this level unifies the goal with the head of the clause and records the binding states of the problem solving. Depending on the success or failure of the unification, this layer decides either forward execution or backtracking should take place. As all the problems involving parallel and distributed processing are abstracted away from this layer, the problems it faces are identical to those in sequential logic programming.

The forward execution and the backtracking strateqies are implemented in the layer 2 system. When a goal successfully unifies with the head of a clause, the forward execution strategy is charged with selecting literals for AND-parallel execution. If the literal is locally solvable, the forward execution algorithm will present the goal to the layer 3 system for local execution; otherwise, the forward execution algorithm will use the communication protocols (supported in layer 1) to coordinate distributed problem solving. Backtracking activities are achieved in a similar way.

The layer 1 system is concerned with communication service for distributed problem solving. When the layer 2 system decides to send out a message, the layer 1 system takes care of the real message sending. This may include packeting, buffering, routing, error handling, and bookkeeping services, depending on the network architecture. Similarly, when the layer 1 system receives a message, it has to process the message before it invokes a forward execution or backtracking in the layer 1 system.

2.2.2 COMMUNICATION PRIMITIVES FOR DISTRIBUTED EXECUTION

Five execution commands are used as the communication primitives to coordinate the distributed processing. They are request, redo, cancel, success and failure. A request message designates the sending node and the receiving node as the requestor and the servant with respect to the service. A non-redundant message ID is assigned by the requestor to identify the service it requests. This service is considered an order of the requestor and a job of the servant. Subsequently, the requestor of an order may send out redo or cancel messages to the servant, and the servant may report the problem solving state with success or failure messages.

As far as a user is concerned, he may work on any node of the network, use this node as an interface to the network, and request this interface node to solve a goal using the distributed knowledge base in the network. This request becomes an order of the node and starts up a forward execution on this node. As the forward execution proceeds, this node may send out messages to other nodes to request that some subproblems be solved using their local knowledge bases. These request messages in turn start up new jobs in the servant nodes. Therefore, a global problem solving is accomplished by a tree of orders and jobs in the nodes of the network. Each node may be a requestor and a servant at the same time with respect to different orders and jobs. The message ID and the communication primitive of a message allow a node to identify the corresponding job and its problem solving state.

While users on the network may issue queries and spawn many trees of jobs and orders, the tree structure is not visible to any single node. As far as a node is concerned, an incoming message starts a task for the corresponding job of the node. Each task represents a local problem solving step of the job. In other words, a node handles each job by dispatching one or more tasks; each task accomplishes some progress in local forward execution or backtracking. The forward execution and the backtracking algorithms are defined as literal state transitions, which are in turn defined by argument modes and variable states.

2.2.3 MODE DECLARATION

Similar to DEC-10 Prolog, our execution scheme allows its programmers to declare the activation mode of an argument. There are three modes: in, out and in_out. Semantically, they designate the corresponding argument as an input argument, an output argument and any argument, respectively. Operationally, a literal may not be executed unless all its input arguments are available and ground, all its output arguments are available and non-ground, and none of its in_out argument depends on a held variable. An output mode violation is a program error; an input mode violation simply means that the literal is not ready for execution yet. An undeclared predicate is understood to have the default in_out mode for all its arguments. The default mode definition makes Prolog a subset language; a Prolog program may be executed without modification.

There are two motivations to incorporate this mode declaration: semantically it documents the intended use of a declared argument, and operationally it specifies the binding conditions for execution. Many evaluable system predicates in standard Prolog are predefined to have input or output modes. When AND-parallelism takes the left-to-right sequence away, this mode declaration helps a programmer to make sure that a goal will not be attempted prematurely.

2.2.4 VARIABLE STATES

The interpreter associates each term with one of the three states: ground, non-ground or pending. A variable initially has the non-ground state, a constant term has the ground state, and a structure term with at least one inner variable has the pending state. The "typing algorithm" in [DEGROOT 84] shows that the state of a variable may be updated easily at the run-time: when a non-ground variable is instantiated to a ground or a pending term, its state is changed accordingly. Coupled variables share the same state. In addition, a pending variable may be examined to see if it is already ground. On the other hand, when a binding is retracted during backtracking, the variable state will be reset (from ground or pending) to pending or non-ground.

In addition to these states, the availability of a variable is defined. A non-ground or pending variable may be held by a certain literal for possible binding. A held variable is committed to the holding literal, which may instantiate the variable and consequently changes the variable's state to pending or ground. When a literal finishes its computation (or during backtracking), held variables may be released and becomes available again. A variable is either available or held; all variables are initially available. The possible transitions of variable state and availability are summarized in Figure 3.

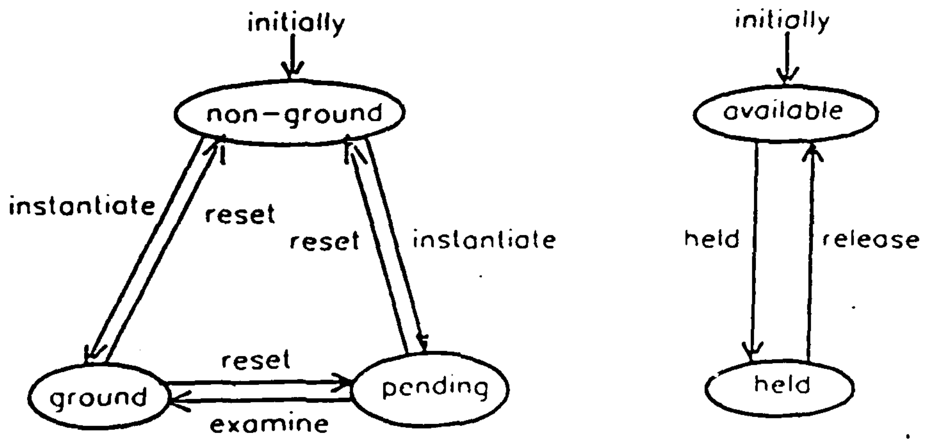


Figure 3. Variable states and availability transitions.

2.2.5 LITERAL STATES

A literal has five possible states: waiting, ready, ordered, solving and solved. When a goal successfully unifies with the head of a clause, each literal in the body of the clause has the waiting state. A waiting literal becomes ready when all its arguments satisfy the mode constraints. As soon as a literal is ready, it is granted execution. If the literal is locally solvable, it has the solving state, and the node is engaged in solving this literal. Otherwise, a request message is sent to the node which has the clauses to solve the literal, and the literal has the ordered state. When an ordered or a solving literal successfully solves the goal, it has the solved state. Notice that the ready state is transient and is defined simply as a conceptual aid. During backtracking, an ordered, solving or solved literal may be reset to the waiting state. These possible state transitions are summarized in Figure 4.

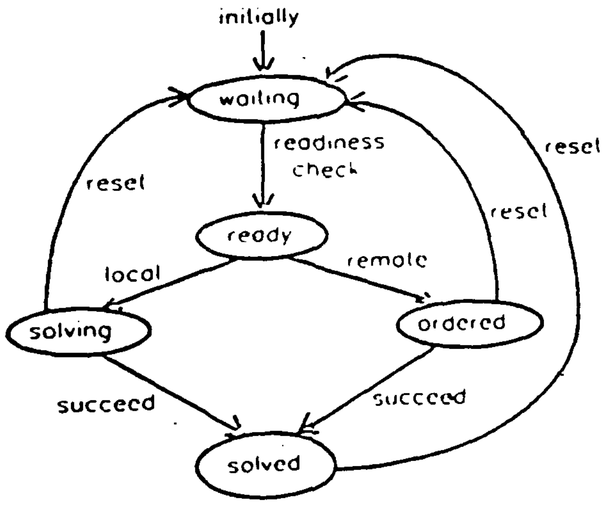


Figure 4. Literal state transitions.

2.3 FORWARD EXECUTION

In our abstract interpretation model (see Figure 2), when the current goal successfully unifies with the head of the current clause, the layer 3 system makes a service call to the layer 2 system and starts up an AND-parallel forward execution. There are four major algorithms for the AND-parallel forward execution. The readiness check algorithm determines which body literals may be scheduled for execution, the literal execution algorithm determines when and which node should execute a ready literal, the inter-clause forward execution algorithm controls the execution when there is no ready literal in the current clause, and the task termination algorithm determines when the task should terminate. This section discusses these forward execution algorithms and explains how they work together. Two examples are given to illustrate the execution strategy.

2.3.1 READINESS CHECK

When a goal successfully unifies with the head of a clause, all the literals in the clause body are in the waiting state and are candidates for AND-parallel execution. The system starts to check the readiness of these literals. Of course, the system has to do this sequentially, one literal after another. A programmer-suggested sequence, a heuristic sequence or any other systematic sequence may be adopted to perform this readiness checking. To follow the Prolog convention and without losing any generality, we assume that the left-to-right sequence is adopted. We will see how this sequence affects the parallel execution.

A literal is determined to be ready when all its input arguments are available and ground, all its output arguments are available and non-ground, and none of its in_out arguments depends on a held variable. As soon as a literal is determined ready, it is executed immediately (see the description in the next subsection), and all the non-ground variables in the literal are held. On the other hand, if a literal is not ready yet, the node continues checking the readiness of the other literals until the rightmost literal is reached. While the node scans the body literals to check their readiness, it also updates whether all the body literals it has scanned so far are in the solved state. This state information will be used to guide the subsequent forward execution (see the subsection entitled "Inter-Clause Forward Execution").

2.3.2 LITERAL EXECUTION

A ready literal may be executed locally or at a remote node, depending on whether the local knowledge base has the clauses to solve the literal or not. If the local knowledge base does not have the clauses to solve the literal, the node formulates a request and sends the request to the node which has the clauses to solve the literal. The request formulation is a binding reference process: each variable is replaced by the concrete term to which it is bound. This formulation process makes it possible to have a distributed execution without passing variable bindings from one node to another. It also makes the readiness check simply a local operation.

When a ready literal is locally solvable, the node proceeds with a local execution of the ready literal, which results in a depth-first execution. The semaphore scheduling policy is a greedy algorithm in the sense that when a literal is determined ready, it is granted execution. This scheduling algorithm ignores the possibility that some literals to the right of this ready literal may also be ready, and checking their readiness could mean more parallelism. Nevertheless, this greedy algorithm offers simplicity and great flexibility for the distributed execution.

For example, the body literals in a clause could be pre-ordered such that locally solvable literals always preceed remotely solvable literals. In that case, the locally solvable literals have a higher priority to bind the variables, and inter-node backtracking will be greatly reduced. This is analogous to the cooperative scheme in which an expert always deliberates about a plan and makes sure that the plan agrees with all his knowledge before he proposes the plan to other experts. Conversely, we can also pre-order the body literals such that remotely solvable literals always proceed the locally solvable literals. This arrangement will produce higher degree of parallelism with the cost of higher frequency of inter-node backtracking. This is analogous to the cooperative scheme in which a manager asks his subordinates to work on different plans, and he approves or disapproves them. A trade-off between more parallelism or less inter-node backtracking has to consider communication cost and delay, granularity of parallelism, possibility of backtracking and several other factors. This greedy algorithm provides the flexibility and opportunity for fine-tuning the performance without having to change the execution strategy.

2.3.3 INTER-CLAUSE FORWARD EXECUTION

As the body literals are checked from left to right for their readiness, eventually the rightmost body literal is reached and the forward execution has to proceed from one clause to another. The execution strategy at this point depends on the problem solving state of the current clause. There are three possible situations:

1. The current clause is a unit clause.
2. All the body literals of the current clause have been solved; that is, all of them are in the solved state.
3. Some body literals are not solved yet.

In the first or the second situation, the calling literal (which successfully unified with the head of the current clause) has been successfully solved. Its state is changed from solving to solved, and all the variables which are held by this literal should be released at this point. The forward execution continues with checking the readiness of the brothers of the calling literal, starting from the leftmost brother. This is because this calling literal may have grounded some varables and made its left brother ready. In order to facilitate backtracking, when a literal is solved, the node also determines the order in which the literal is solved. Thus, a literal may be the third one, among its brothers being executed, but the first one being solved. The execution order is unimportant and is not even determined; the solved order is remembered and will be used in backtracking.

In the third situation, the calling literal still has the solving state. The forward execution continues with checking the readiness of the brother literal next to (to the right of) this calling literal. Note that readiness checking is performed on waiting literals only; a literal in any other state will be skipped in the checking process.

2.3.4 TASK TERMINATION

If the inter-clause forward execution keeps causing the node to resume the problem solving activity in the parent clause, eventually the execution will reach the original goal of the job. In this case, the task terminates. If the goal literal is solved, this node formulates an answer and sends it back to the requestor with a success message. If the goal is not solved yet, the task terminates without any message communication. The node will check its message buffer and start another task. We can expect that an incoming message will get the node back to work on this job again. This will be described in Section 2.5.

The answer formulation is, in fact, the same as the request formulation. The variables in the original goal will be replaced by the concrete terms to which they are bound in this node. This answer is not only sent back to the requestor, but also cached (see [ROWE 87]) into the knowledge base of this servant node as a unit clause. It turns out that this type of learning (also known as lemma generation; see [HOGGER 84]) much simplifies the backtracking process of the system. This will be illustrated in Section 2.5.

We summarize that the forward execution may cause a success message, or one or more request messages to be sent out. When a task terminates during forward execution, the node may have solved the original goal, or may be waiting for a message to continue the execution. It is easy to see that once the node requests another node to solve a literal for this job, the task will terminate the forward execution and wait for the responding message.

2.3.5 EXAMPLES

In all the following examples, we will assume that the clauses for predicates p1, p2, p3, and p4 are allocated to nodes N1, N2, N3, and N4, respectively. The clauses for predicate p5 are allocated to node N3; the clauses for predicate p6 are allocated to node N1. The following mode information is declared for the predicates.

```
mode(p1, out, in_out).
mode(p2, in, in).
mode(p3, out, out).
mode(p4, in, in_out).
mode(p5, out, in).
mode(p6, in_out, in_out).
```

Example 1.

Suppose node N3 has the following clauses for predicates p3 and p5:

```
(3.1)  p5(A,B) :- p1(A,B), p2(B,C), p3(C,D), p4(A,D).
```

```
(3.2)  p3(c,d).
(3.3)  p3(c1,d1).
```

A request message with the goal

```
?- p5(A,b).
```

starts a request task in node N3. The node finds clause (3.1) and unifies the goal with the clause head. Thus, the body of the clause becomes

```
p1(A,b), p2(b,C), p3(C,D), p4(A,D).
```

Now the node starts to check whether these literals are ready for execution. Literal p1(A,b) is ready, because the output argument is a non-ground variable A and the in_out argument is ground to "b". Therefore, p1(A,b) holds variable A for binding, its state is changed to ordered, and a request message is sent to node N1. Next the node finds that literal p2(b,C) is not ready, because variable C is a non-ground input argument. So the node continues to check the state of literal p3(C,D). With both arguments of the output mode and non-ground, p3(C,D) is ready and locally solvable. The node changes its state to solving and uses clause (3.2) to solve it. Of course, the unification succeeds and the forward execution returns to clause (3.1). So the node marks literal p3 as the first solved literal and restarts the readiness checking from the leftmost literal. Because literal p1(A,b) is ordered, it is skipped. Literal p2(b,C) is found ready this time because variable C is ground to "c" after p3(C,D) is solved. The node sends a request to node N2 and marks p2(b,C) as ordered. The readiness checking skips the solved literal p3(C,D) and finds p4(A,D) not ready yet, because variable A is held by literal p1(A,b). At this point the forward execution returns to the original goal, p5(A,b), which currently has the solving status. Therefore, the request task terminates and leaves the following literal states:

p1(A,b) : ordered (node N1);

p2(b,c) : ordered (node N2);

p3(c,d) : first solved;

p4(A,d) : waiting.

Example 2

We continue the discussion in Example 1 and consider the forward execution in node N1. Suppose node N1 has the following relevant clauses:

```
(1.1) p1(X,Y) :- p6(Y,X).
(1.2) p6(b,a).
```

Now the request message from node N3 is received. The goal

```
?- p1(A,b).
```

starts a request task in this node. Clause (1.1) is chosen to solve the goal, and the unification succeeds. Literal p6 is ready immediately because both the ground and the non-ground arguments satisfy in_out modes. Clause (1.2) is thus used to solve the literal and successfully binds the literal to p6(b,a). The forward execution thus returns to clause(1.1) with literal p6 in the solved state, and then returns again to the original goal p1(A,b), which is successfully solved. Therefore, the answer p1(a,b) is formulated and sent back to node N3 with a success message. In the meantime, node N1 caches this answer as a unit clause

(1.3) p1(a,b).

into its knowledge base and terminates the task.

2.4 BACKTRACKING

According to the abstract interpretation model in Figure 2, when the unification between the current goal and the head of the current clause fails, backtracking takes place. This backtracking is considered a service request from the layer 3 system to the layer 2 system, just like the case of forward execution.

There are two types of backtracking: shallow backtracking tries to find another clause to solve the current goal, while deep backtracking discards the current goal and tries to establish a new current goal. When a unification fails, shallow backtracking is called, and the system tries to use the next clause to solve the current goal. Therefore, a shallow backtracking algorithm is simply a definition of what the next clause is. This is defined in the PKS system as the standard top-to-bottom sequence, with the exception that all facts will be used before rules. This is justified by the observation that facts generally represent boundary conditions or exceptions which should be examined before rules. This strategy also makes sure that the cached results (facts) may be used to enhance the performance.

When there are no more clauses to solve the current goal, the system has to establish a new current goal and to backtrack deeply. The deep backtracking algorithm has to identify a backtrack literal; i.e., which literal shall be the new current goal and be re-solved. Depending on where the backtrack literal is, deep backtracking is further classified into intra-clause backtracking, inter-clause backtracking and inter-node back-tracking. This section first describes the procedures for these three types of deep backtracking. A couple of examples are then given to illustrate them.

2.4.1 INTRA-CLAUSE BACKTRACKING

When the shallow backtracking algorithm finds no more clauses to solve the current goal, the system first tries to have an intra-clause backtracking; i.e., to backtrack to a brother of the failed literal. It is easy to see that only solved brother literals are likely to have caused the failure and therefore are candidates for re-solvng. Because the semaphore mechanism requires that a solving or `rdered literal holds all its non-ground variables, the failed literal could not have referenced these variables and failed as a result of that. The latest solved brother literal is identified by the intra-clause backtracking algorithm as the backtrack literal to be re-solved.

Since re-solving the backtrack literal will produce a different variable binding environment and make the solving or ordered literals obsolete, these executing literals have to be retracted. Retracting a literal covers two possibilities:

1. If the literal is in the ordered state, the corresponding request is canceled and the literal is reset to waiting;
2. If the literal is in the solving state, all the bindings which were made while solving this literal are undone, and all the requests are canceled.

After all the executing literals are retracted, the node re-solves the backtrack literal. Re-solving a literal covers three possibilities:

1. If the literal was not locally solved, a redo message is sent to the node which solved it;
2. If the literal was locally solved and was solved in one unification (a unit clause was used to solve the literal), the node tries to use another clause to solve the literal.
3. If the literal was locally solved with more than one unification, then the node re-solves the last solved literal in the whole subtree corresponding to the solving of this literal.

Notice that re-solve is recursively defined.

Re-solving a literal technically causes a shallow backtracking of either the backtrack literal or a dependent of the backtrack literal. When the shallow backtracking algorithm finds the next clause, it presents the new current goal and the new current clause to the layer 3 system for unification. Depending on whether the unification is successful or not, the layer 3 system will continue the problem solving activity with forward execution or backtracking. However, if the node is not currently running a request task and the backtracking causes a redo message to be sent out, then the current task terminates after sending out this redo message, because nothing is left to the layer 3 system of this node.

Although this backtracking scheme is essentially a naive backtracking upon the solved literals, it instantly weeds out all waiting, ordered and solving literals as candidates for the backtrack literal. A more intelligent backtracking algorithm may be devised with the observation that a literal must have held an input or in_out variable of the failed literal to be responsible for the failure. A recording of the variable holding history will help the system to weed out irrelevant solved literals and make the backtracking scheme smarter. Currently we think that the overhead of remembering and analyzing the variable holding history can not be justified, and the simple backtracking scheme is adopted.

2.4.2 INTER-CLAUSE BACKTRACKING

When a literal fails and there is no solved brother literal to be re-solved, inter-clause backtracking takes place. The calling literal, which unified with the head of the current clause, becomes the backtrack literal. The ordered and solving literals in the body of the clause, if any, are first retracted. Then the node tries to use another clause to re-solve the calling literal. Note that the calling literal had a local unification with the head of the current clause. Therefore, an inter-clause backtracking will cause a local shallow backtracking of the calling literal.

2.4.3 INTER-NODE BACKTRACKING

When the backtrack literal happens to be the original goal of this job, and the shallow backtracking algorithm fails to find any more clauses to solve the goal, the whole job fails. A fail message is sent to the requesting node and the task terminates.

In summary, backtracking may cause a fail message, or several redo and cancel messages to be sent out. When a task terminates during backtracking, the node may have failed to solve the original goal, or may be waiting for a response to a redo message.

2.4.4 EXAMPLES

Example 3.

In this example we consider the execution of node N2 after it receives the request message, which was sent out by node N3 during the forward execution described in Example 1. Suppose node N2 has the following relevant clauses:

(2.1) p2(b1,c1).

(2.2) p2(b2,c2).

The request from node N3 has the goal

```
?- p2(b,c).
```

Node N2 first chooses clause (2.1) to solve the goal. The unification fails. Shallow backtracking takes place and node N2 chooses clause (2.2) to solve the goal again. This also fails and there is no other clause to solve the goal. The calling literal therefore fails. Since this calling literal is the original goal, node N2 gets into inter-node backtracking. A fail message is sent to node N3 and the (request) task terminates.

Example 4.

In this example we consider that node N3 receives the fail message from node N2 and starts its backtracking. We assume that node N3 receives this fail message before the success message from node N1 (see Example 2) arrives. This assumption should not surprise a reader, although node N3 did order node N1 to solve literal p1 before it ordered node N2 to solve literal p2. One can think that node N1 was busy working on other tasks and therefore could not process the request before node N2 finished solving literal p2.

When the fail message is received, the literals in clause (3.1) have the states which are listed in the end of Example 1. Literal p2 is the failed literal; its state is reset to waiting. Literal p3 is the only solved literal and is chosen as the backtrack literal. Literal p1 has to be retracted because it has the ordered state. So node N3 sends a cancel message to node N1 and resets p1 to waiting. Literal p3 also has to be retracted before being re-solved. The bindings of variables C and D are undone, which makes these two variables non-ground again. Now node N3 uses clause (3.3) to re-solve literal p3. At this point, the literals have the following states:

p1(A,b) : waiting;

p2(b,C) : waiting;

p3(C,D) : solving;

p4(A,D) : waiting.

The backtracking process terminates by reporting to the layer 3 system the new current goal p3, and the new current clause (3.3). The successful unification in the layer 3 system causes variables C and D to be bound to constants c1 and d1, respectively. The following readiness checking finds literals p1 and p2 ready. Two request messages are sent to node N1 and node N2 and these two literals are again set to ordered. The forward execution stops when it finds literal p4 not ready. Therefore, the task terminates with the following literal statuses:

P1(A, b) : ordered (node N1);

p2(b,c1) : ordered (node N2);

p3(c1,d1) : first solved;

p4(A,d1) : waiting.

A reader may criticize that node N3 is sort of stupid in its backtracking, because it sends a request message to node N1, cancels it later, and then sends out an identical request again. This criticism is valid; but it is also valid that node N3 did not spend any time trying to analyze the reason for failure and to identify a better backtrack literal. We will see the impact of this "stupid" message on node N1 and node N3 in the examples coming up in the next section.

2.5 DISTRIBUTED IMPLEMENTATION

The last two sections described the layer 2 system execution from the viewpoint of the layer 3 system: they explain how the system controls the forward execution and backtracking to regulate the parallel problem solving. Sometimes the execution extends outside the nodal boundary, and messages are sent to other nodes to continue the execution. In this section, we describe the layer 1 system and show how it is integrated with the other layers for distributed processing. The internal structure of a node will be presented first, followed by the communication protocols which explain how the incoming messages drive a node to start or continue the forward execution and backtracking.

2.5.1 INTERNAL STRUCTURE OF A NODE

There are four major data structures of a node: a knowledge base, a proof tree, a job table and an order table. The knowledge base contains the internal representation of all the program clauses of a node. The proof tree of a node has many frames to record the state information of the program execution. The job table and the order table are used to store pointers to the proper frames which started the job or requested the order, and to do the conversion between internal names and external names.

The format of the messages which are passed from one node to another is shown in Figure 5. The sender field specifies the identification of the sender node. This information, together with the message ID, uniquely identifies an order of the requestor node and a job of the servant node. The command field holds one of the five communication primitives previously described, and the data field contains the goal to be solved or the solution to be reported. The communication protocols associated with the five commands are described in the following subsections.

Sender	Message ID	Command	Data

Figure 5. Message format.

2.5.2 REQUEST

A request message signifies the beginning of a new job. The node logs the requestor and the message ID, associated with the starting frame for this job, on the job table. Then it starts a task to process the job. If forward execution continues and eventually the task successfully terminates, the node sends a success message to the requestor; but this job entry on the job table and all the relevant frames in the proof tree are retained. This information will be needed if later the requestor wants the goal to be re-solved and sends a redo message to this node. On the contrary, if the task encounters inter-node backtracking, the node sends a failure message to the requestor, purges the proof tree and deletes this job entry.

During forward execution, when the node needs to request another node to solve a literal, it first generates an order ID, which is different from all the order ID's in the current order table. This order ID, the servant node and the frame (which spawns this request) are logged on the order table. The node then sends the request message to the servant node with this order ID as the message ID.

During backtracking, the node looks up the order table and finds the order ID and the servant node to which it should send a redo or cancel message. When the node sends out a redo message, it only looks up the order table; when the node sends out a cancel message, it also deletes the order entry on the order table.

2.5.3 REDO

When a node receives a redo message, it first looks up the job table for the starting frame of the job. Recall that a redo message is an inter-node version of re-solving a literal. The node handles a redo message exactly the same way it re-solves a locally solved literal: if the goal was solved in one unification, it tries to use another clause to solve the goal; otherwise, it traces down the proof tree (from the starting frame) and re-solves the last solved literal. Note that if the last solved literal is not locally solved, another redo message will be sent out consequently.

2.5.4 SUCCESS

When a success message is received, the node first checks the order table to see if this order is still active. Because of the nature of asynchronized problem solving and communication delay, it is possible that the node may have canceled the order and deleted the corresponding entry on the order table before a responding success (or fail) message is received. In this case the node can not find the order ID, and this task simply terminates because this order has been canceled. Otherwise, the order is still active, and the node identifies the order requesting frame from the order table.

A success message responds to a previous request or redo message, which asks the servant node to solve or re-solve a literal. Since solving or re-solving a literal begins with searching for a clause to unify with the literal, a success message continues the forward execution (from the spawning frame) as if a unit clause were found in the local knowledge base. The answer, which is reported with the success message, is disguised as a unit clause in the requesting node. Then the node unifies the literal with the disguised unit clause and solves the literal locally.

At this point the spawning frame still remembers that the literal was actually solved by another node, and the order table also remembers the servant node and the order ID. If the literal has to be re-solved, a redo message can be sent to the servant node.

2.5.5 FAILURE

The handling of a failure message is similar to the handling of a success message. The node checks if the order is still active. For an active order the node starts backtracking as if it could not find a clause in its knowledge base to solve the literal. As soon as a failure message is received, the corresponding entry on the order table is deleted.

2.5.6 CANCEL

When a cancel message is received, the node also has to check if the job is still active. This is because this servant node may have failed to solve the request and deleted the job entry before receiving the cancel message. Since a cancel message is sent out while the requesting node retracts a literal, one may expect that a cancel task will do the same thing as retracting a literal. This is largely true, except that this servant node does not have to undo variable bindings because these bindings are inaccessible outside this node.

The node finds the starting frame of this job from the job table. It then traverses down the proof tree from the starting frame to identify all the orders spawned by this job. All these orders are canceled. Finally, the node purges the frames from the proof tree and deletes this job from the job table. Notice that a cancel task is handled by the layer 1 system alone, while all other tasks will need the layer 2 system to handle the forward execution or backtracking. As contrast, a cancel task may only send out cancel messages, while other types of tasks may send out all five types of messages.

2.5.7 EXAMPLES

Example 5

To continue our previous examples, we consider the success message (generated in Example 2) from node N1 now arrives at node N3. Node N3 can not find this order because it is already canceled during the backtracking described in Example 4. Therefore, this task terminates immediately.

Example 6

Node N1 now receives the cancel message from node N3, which sent out this message during the backtracking described in Example 4. Node N1 purges the proof tree, deletes the job entry on the job table, and terminates this cancel task.

Example 7

Inevitably, node N1 receives the second request message from node N3. This request message starts a new job (not just a new task) on node N1. However, clause (1.3), instead of clause (1.1), is used to solve the goal this time. This is because the system only uses a rule after all the unit clauses are exhausted. Therefore, the goal is quickly solved again, and a success message is sent to node N3. We see that those "stupid" messages actually do not cause much overhead in node N1.

2.5.8 IMPLEMENTATION STATUS

This PKS model was first implemented as ADA task structures running under VAX/VMS to simulate the parallel and distributed computation. In order to simplify the implementation, a reliable communication mechanism was assumed, and each node task had a dedicated channel task and an associated buffer task to provide the service of non-blocking message sending (via mail boxes) and unlimited message buffering. The proof tree of each node was implemented as fragmented environment stacks using a heap allocation strategy. Each frame in the environment stack kept variable binding information, literal state information, and pointers to its job table entry or the spawning order entries for the processing of the three layers, respectively. The implementation details are documented in [CHUNG 87].

This simulation model has been used to verify and to refine the system computation model in our subsequent investigations. Currently we are re-implementing the system in C language and trying to port it to a 4-node Transputer and a network of UNIX-based work stations. We plan to take some performance measurements of the system running under different processing environments and report our findings in a follow-up paper.

One major direction of our continuing research effort is to incorporate the capability of OR-parallel execution and knowledge base maintenance into the PKS model. The layer 2 system of the present PKS model has to be divided into three sublayers to handle AND-parallel execution control, OR-parallel execution control, and knowledge base maintenance, respectively. The basic idea of this extension is that each node should fully utilize its knowledge base to contribute to the problem solving activity of the system. When the layer 1 system finds that there is no incoming message to start a task of AND-parallel execution, it reports this finding to the layer 2 system. The OR-parallel execution control sublayer schedules a new task, which utilizes the next clause to solve a previously attempted goal and achieves a sort of pipeline OR-parallelism. Furthermore, when there is no OR-parallel solution to investigate, the knowledge base maintenance sublayer starts a task to refine the knowledge base, to enforce consistency and to resolve the conflicts within the knowledge base.

3. STATIC ALLOCATION

Decomposition and allocation are common problems to multi-processor and distributed systems. This section presents a unified decomposition/allocation scheme for the PKS. The Contract Net and its successors (see [SMITH 80], [SMITH 81], and [DAVIS 83]) and the Distributed Vehicle Monitoring Testbed (see [DURFEE 87]) proposed making allocation decisions after the problem decompositions have been made by programmers and system designers. The schemes in this section support this view and also allow for cases where no decomposition has been made.

[BOND 88]cites eleven bases for decomposition. The SAS class directly supports five of these decomposition bases: (1.) the abstraction level, (2.) the availability of co-ordination, (3.) the control dependencies, (4.) the data/logical dependencies, and (5.) the resource minimization. Two other bases, interaction levels and uniqueness avoidance, may be considered after enhancing the PKS. Section 3.4 returns to these bases for a concluding discussion vis-a-vis the allocation schemes of Section 3.1.

3.1. A STATIC ALLOCATION SCHEME (SAS)

Methods that have been used for decomposing distributed problems in artificial intelligence include (1.) picking tasks that are inherently decomposable, (2.) decomposing by a programmer, (3.) planning hierarchically, (3.) load balancing, (4.) using minimally connected subgraphs, and (5.) aggregating by subtask (see

[BOND 88]). While most of these methods depend on either the programmer or on a problem's conceptual representation, the SAS of the PKS can be applied without such prior knowledge.

One must emphasize that there is no substitue for initially representing a problem clearly and that the obvious decompositions from representation and programmers should be used first. It is at the point where further decomposition is needed that the SAS should be invoked. The SAS presented in this section is a form of load balancing, where the weights balance knowledge bases rather than computations.

One can envision that over the life-cycle of a program it will be advantageous to re-configure the PKS-graph many times. Such re-allocation schemes will be elaborations of the initial allocation, augmented with heuristics and statistics developed since the previous allocation.

The partitioning of the knowledge base and the corresponding allocation process observe these two principles: (1) clauses of a procedural bundle, which have the same predicate-head, 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.

When a predicate in the body of a clause is ready for execution, it is either in the SOLVING state if the local processor can solve it or it is ORDERed if an external node can solve it. In determining whether to allocate a procedural bundle to a partially filled node or to an unused node, a metric for measuring these communication and solution mechanism costs is needed. The metric will be hardware-dependent.

Consider now an algorithm 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 +u to weight Wi
2. For each non-unit clause in Bi,

   add +h   to Wi for the clause head

   add +b   to Wi for each literal in the body if it is in Bi

   add +i   to Wi for each built-in literal in the body

   add +b-c to Wi for each literal in the body if it is not in Bi.

The constants u, h, b and i are related to solution mechanism costs. The constant c accounts for the inter-nodal communication costs. The fundamental premise of this algorithm is to add the heaviest unassigned bundle to the Lighter Knowledge Base (LKB). The weight corresponds to processing time from the LKB's point of view. Consequently, a bundle which must communicate with the other KB is using less of the LKB's time. Communication has a negative effect on the LKB's time. Therefore, the weight added to Wi for each external literal in the body of a clause is b-c.

Consider this example:

```
a(1).   b(6).   c(7).   d(6).
a(2).   b(7).   c(8).   d(7).
a(3).           c(9).
a(4).
a(5).

a(X) :- b(X), X<7.
b(X) :- Y is X+1, Y<10, b(Y).
c(X) :- a(X).
d(X) :- b(X).
```

Then with $u = h = b = 1$, $i = 0$, and $c = 2$, one has

$$Wa = 5 + (1 + 1-2 + 0) = 5$$

$$Wb = 2 + (1 + 0 + 0 + 1) = 4$$

$$Wc = 3 + (1 + 1-2) = 3$$

$$Wd = 2 + (1 + 1-2) = 2.$$

Now bundle Ba is assigned to KB1, bundle Bb is assigned to KB2. Since KB2 is still the LKB, bundle Bc is also assigned to KB2. Finally, bundle Bd is assigned to KB1.

This scheme can be modified to re-weigh and re-sort the unassigned bundles after each addition. This affects the weight Wi only when a literal in the body of a clause in Bi has already been assigned to the lighter knowledge base.

In this example, Ba is assigned to KB1. Bundle Bb is then put into KB2. The current LKB is KB2. Now $Wc = 3$ but $Wd = 2 + (1+1) = 4$, after reweighing; hence, bundle Bd is added to KB2. The balance is tipped, and one inserts bundle Bc into KB1. This new distribution (bundles Ba and Bc in KB1 and bundles Bb and Bd in KB2) looks natural since predicate c invokes predicate a and predicate d invokes predicate b.

In general, one has the following algorithm for deciding the appropriate bundle Bi to add to the LKB:

1. Weigh and sort the bundles. Add the heaviest bundle to KB1. Add the second heaviest bundle to KB2. Set LKB to KB2.
2. Until all bundles are assigned to KB1 or KB2, repeat
   a. Re-weigh and re-sort the unassigned bundles.
   b. Assign the heaviest unassigned bundle, Bh, to the LKB.
   c. Re-weigh the bundles of the LKB, other than Bh.
   d. Set LBK equal to the lighter of KB1 and KB2.

This splitting of a knowledge base into two component knowledge bases is the first step in this allocation scheme. The two resulting knowledge bases are then split into four (maybe fewer) KBs. This iterative splitting continues until (1.) there is no unused processor available, (2.) the currently examined knowledge base contains only one bundle, or (3.) the overhead of splitting the currently examined knowledge base exceeds an acceptable threshold (user defined).

3.2 STATIC ALLOCATIOIN OF A* SEARCH

The algorithm for A* Search, as presented in [ROWE 88], is an interesting example to which to apply the SAS. The Prolog implementation of A* Search uses 23 predicates and the built-in predicates ABOLISH, ASSERTA, CALL, FAIL, NOT, RETRACT, WRITE, !, =, +, and <. If we assume that each node is provided with all necessary built-in predicates and if the SAS weight of each built-in predicate is zero (i=0), it is unnecessary to acknowledge them during weighing.

Exceptions to this rule for built-in predicates are those which use another predicate as an argument; e.g.,

asserta(successor(STATE)).

For the purpose of weighing, we shall treat such literals as if the built-in predicate were absent; thus, asserta(successor(STATE)) is weighed the same as successor(STATE). For the A* search, these exceptions involve only NOT, ASSERTA, and RETRACT.

We use the following key to express the required predicates of the A* Search:

P1 = add_state	P12 = countup
P2 = add_successors	P13 = eval
P3 = agenda	P14 = fix_agenda
P4 = agenda_check	P15 = goalreached
P5 = append	P16 = measurework
P6 = astarsearch	P17 = pick_best_state
P7 = beststate	P18 = repeatifagenda
P8 = check_abolish	P19 = replace_front
P9 = cleandatabase	P20 = special_less_than
P10 = cost	P21 = successor
P11 = counter	P22 = usedstate
	P23 = usedstate_check

It is then possible to express the bundles with the following notation:

B1 : 3(P1) + 2(P10) + (P4) + (P23) + 2(P13) + (P3)

B2 : 3(P2) + (P15) + (P21) + (P1) + (P3) + (P22)

B3 : Facts only; a variable number, which is >= 0.

B4 : 3(P4) + 3(P3)

B5 : 3(P5)

B6 : (P6) + (P9) + (P1) + (P18) + (P17) + (P2) + 2(P3) + (P16)

B7 : (P7), one fact

B8 : 2(P8)

B9 : (P9) + 4(P8)

B10: Group of predicates, none of which depends on these bundles

B11: (P11), one fact

B12: 2(P12) + 6(P11)

B13: Group of predicates, none of which depends on these bundles

B14: (P14) + 3(P3) + (P19) + (P10)

B15: (P15), one fact

B16: (P16) + 2(P12)

B17: 2(P17) + 6(P7) + (P3) + (P20)

B18: 3(P18) + (P3)

B19: (P19) + 2(P5)

B20: 2(P20)

B21: 24(P21), twenty-four facts

B22: Facts only; a variable number, which is >= 0.

B23: 3(P23) + 4(P22) + (P14)

As an explanation, consider B1. Our notation implies that bundle B1 contains three occurences of predicate P1, two occurences each of predicates P10 and B13, and single occurences of predicates P3, P4, and P23.

Bundles B3, B10, B13, B21, and B24 are dependent on the particular problem to which one wishes to apply the A* Search. We have in mind the Eight Tiles Puzzle; hence, bundle B21 will use 24 facts to account for the successors of the nine possible tile positions.

Bundles B10 and and B13 deal with the cost and evaluation functions which are special to a particular application. These two bundles will not invoke any of the other twenty-one bundles, but may need some special predicates of their own. We shall assigned a weight of ten to each of the two.

Bundles B3 and B22 represent facts that are asserted and retracted as the program runs. Initially, there are no such facts. There is obviously a problem with how much weight to assign statically to a bundle of facts which changes dynamically. We choose to assign zero as the initial value. Another approach is to weight facts and rules differently. This appears in Section 3.3.

3.2.1 ALLOCATION BY THE PROGRAMMER

Before the SAS is applied the programmer has the option of doing a partial allocation. In this subsection we illustrate one such partitioning.

Bundles B3 and B22, each consisting of a variable number of facts, are invoked by these other bundles:

B3 by B1, B2, B4, B6, B14, B17, B18 and

B22 by B23.

Consequently, a natural structure is to group B3 and its invokers into a super-bundle, SB1, and to group B22 and its invoker into another super-bundle, SB2. Other natural groupings, are B8 and B9 as super-bundle SB3; B11, B12, and B16 as super-bundle SB4; B5 and B19 as super-bundle SB5. In summary, we have these bundles to allocate by the SAS:

B7	B10	B13	B15	B20	B21
SB1	SB2	SB3	SB4	SB5.	

The revised bundle constructions are these:

```
B7  : (P7), one fact
B10 : Group of predicates, none of which depend on these bundles
B13 : Group of predicates, none of which depend on these bundles
B15 : (P15), one fact
B20 : 2(P20)
B21 : 24(P21), twenty-four facts
SB1 :  6(P7)  + 3(P10) + 2(P13) + (P15) + (P20) + (P21) +
      36(SB1) + 2(SB2) + (SB3)  + (SB4) + (SB5)
SB2 :   (SB1) + 7(SB2)
SB3 :  7(SB3)
SB4 : 12(SB4)
SB5 :  6(SB5)
```

The initial weights (using u=h=b=1, i=0, c=2) for the bundles are these:

wB7 = 1	wB10 = 10	wB13 = 10	wB15 = 1	wB20 = 2	wB21 = 24
wSB1 = 17	wSB2 = 6	wSB3 = 7	wSB4 = 12	wSB5 = 6	

3.2.2 ALLOCATION BY THE SAS

The iterations and actions of the SAS while cleaving the original knowledge base into KB-1 and KB-2 are shown in Table 1. As a result of the first separation, KB-1 consists of predicates 7-10, 15, 20-23 and KB-2 consists of predicates 1-6, 11-14, and 16-19.

Further subdivision of KB-1 into KB-11 and KB-12 yields the following: KB-11 contains predicates 7, 15, and 21 while KB-12 contains predicates 8-10, 20, and 22-23. There is no need to divide KB-11 further since most of its weight is contained in the single bundle B21. One could divided KB-12 into KB-121, which contains predicates 10 and 20, and KB-122, which contains predicates 8-9 and 22-23. In all three subdivisions of KB-1, the weight of each KB is greater than 10.

In a similar manner, one finds that KB-2 splits into KB-21, which contains only SB1 predicates (1-4, 6, 14, 17-18), and KB-22, which contains predicates 5, 11-13, 16, and 19. Any further division of KB-21 or KB-22 will produce a KB which is too light (defined as weighing less than 10 for this example).

The PKS graph for this SAS application is shown in Figure 6. The required communications in the PKS-graph of Figure 6 are these:

KB-11, KB-121, KB-22 invoke no other node

KB-122 invokes KB-21 only

KB-21 invokes the other four nodes

ITERATION	KB 1 BUNDLE	KB 1 WEIGHT	KB 2 BUNDLE	KB 2 WEIGHT	REWEIGHINGS
1	*B21	24	EMPTY	0	NONE
2	B21	24	*SB1	17	wSB2 = 8
3	B21	24	SB1	19#	NONE
			*SB4	12	
4	B21	24	SB1	19	NONE
	*B10	10	SB4	12	
5	B21	24	SB1	23#	wSB2 = 6
	B10	10	SB4	12	
			*B13	10	
6	B21	24	SB1	23	NONE
	B10	10	SB4	12	
	*SB3	7	B13	10	
7	B21	24	SB1	23	NONE
	B10	10	SB4	12	
	SB3	7	B13	10	
	*SB2	6			
8	B21	24	SB1	25#	NONE
	B10	10	SB4	12	
	SB3	7	B13	10	
	SB2	6	*SB5	6	
9	B21	24	SB1	25	NONE
	B10	10	SB4	12	
	SB3	7	B13	10	
	SB2	6	SB5	6	
	*B20	2			
10	B21	24	SB1	25	NONE
	B10	10	SB4	12	
	SB3	7	B13	10	
	SB2	6	SB5	6	
	B20	2			
	B7	1			
11	B21	24	SB1	25	
	B10	10	SB4	12	
	SB3	7	B13	10	
	SB2	6	SB5	6	
	B20	2			
	B7	1			
	*B15	1			

* denotes the bundle added on this iteration
# denotes an adjustment to this weight because of the particular bundle added on this iteration

TABLE 1

Hence, we have a hierarchy, where KB-21 can be combined with the Interface since the only anticipated query from the user is predicate 6 (astarsearch).

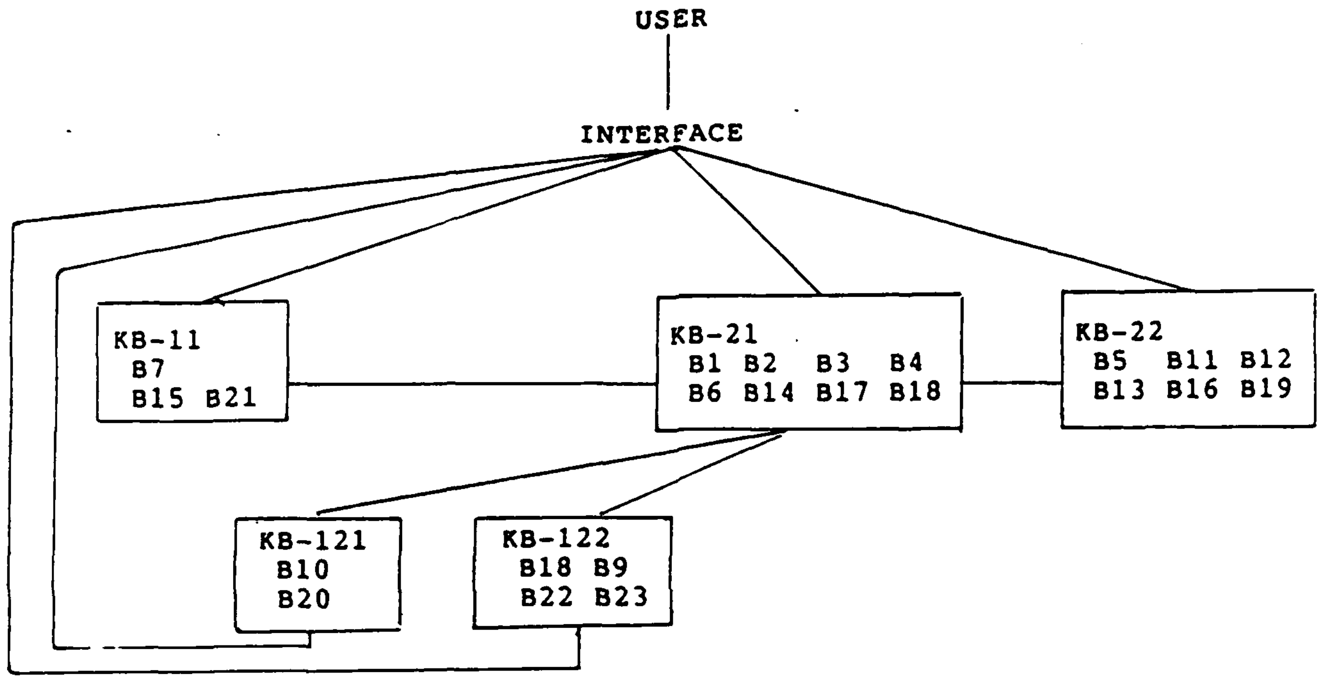


Figure 6. The Initial PKS-Graph for A* Search.

3.3 VARIATIONS OF THE SAS

Refinement of the SAS begins with reassessing the assignment of equal weights to facts and rule heads. In the proving mechanism of each node of the PKS, facts are much easier to deal with than are rules. Consequently, a fairer weight distribution is to use each fact with u=1 and each rule head with h=10. Rule heads should also be more expensive than the literals in the body.

Therefore, we weight each local literal in the body the same as a fact (b=1) and each external literal in the body as b-c = 1-6 = -5. Thus, we have the following new notation for weighing bundle B1:

B1 : 1[0] + 10[3] -5[2(P10) + (P4) + (P23) + 2(P13) + (P3)] + 1[0]

to indicate that there are zero facts for P1, there are 3 clause heads for P1, there are seven external literals in the bodies (two each for P10 and P13, and one each for P4, P23 and P3) and there are no P1 literals in the bodies.

After completing the programmer's allocation into the five super-bundles, as in Section 3.2, the weights become these:

wB7= 1	wB10=10	wB13=10	wB15= 1	wB20=20	wB21=24
wSB1=50	wSB2=29	wSB3=34	wSB4=49	wSB5=33.	

Repeated application of the SAS (until a KB weighs < 11 or contain only one bundle) yields Figure 7. Again the PKS-graph is a hierarchy, with KB-11 controlling the other nodes.

3.4 DISCUSSION

The SAS unites decomposition and allocation for distributed problem-solving into a automated procedure, which can be advised by a system designer or programmer.

The SAS can be viewed as decomposition/allocation based on abstraction levels. The SAS support may be nominal for a problem which is well-structured as in distributed path search of [COMRY 88], Or the SAS support may produce a natural hierarchy, as seen in the examples of this section.

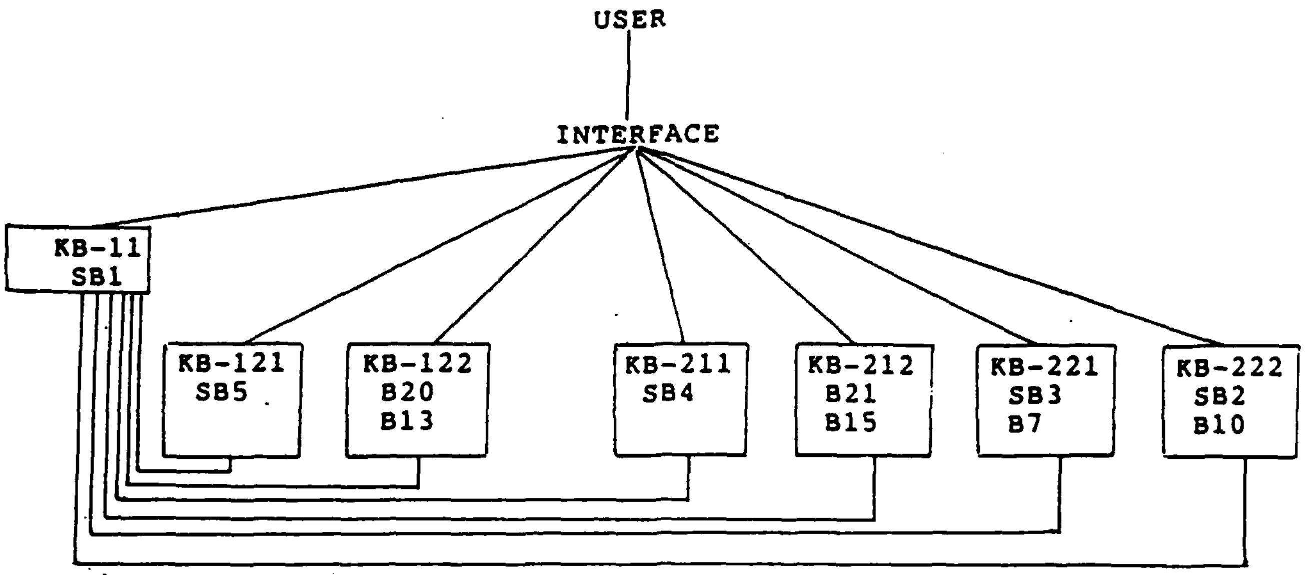


Figure 7. A Second PKS-Graph for A* Search.

Since the PKS provides the global and local control structures needed in distributed problem-solving, the decomposition/allocation basis of availability of coordination is not as restrictive for the SAS (or other allocation schemes of the PKS) as it may be for other distribution systems.

The SAS can be viewed as decomposition/allocation based on control dependencies. Since it is only through clausal bodies that communications with other nodes occur, the SAS weighs the importance of local versus foreign JOB requests. The SAS provides a programmer/designer a choice for the weights of these requests.

The SAS can be viewed as decomposition/allocation based on data/logical dependencies. Indeed, all allocation schemes of the PKS which conform to the two general rules (construction of bundles and nodal connections based on bundles in clausal bodies) exhibit this basis.

The SAS can be viewed as decomposition/allocation based on resource minimization since the decision of whether to place a bundle in the local node or in a foreign node is made in order to minimize communication costs.

The SAS is not presently viewed as decomposition/allocation based on interaction levels because of the inherent control of the PKS. As more work is done on the second queue (the OR queue) of each node in the PKS, this criterion can be incorporated into an allocation scheme. In particular, a controlled priority system for selecting elements from the first and second queues may be preferable to tending to the second queue only when the first queue is empty.

Finally, the SAS is not presently viewed as decomposition/allocation based on uniqueness avoidance. In fact, each node is generally assumed to hold unique knowledge. But this is only a high-level abstraction. As we move to a lower level, an individual node may itself be a PKS-graph where knowledge can be duplicated or redundant. Hence, the broadcasting of a message from a linked, high-level node to the redundant sub-nodes of another node (and the reciprocal problem of avoiding discrepancies in the replies of the duplicates to the requesting high-level node) will be handled by the channel/buffer configurations of the pertinent nodes. These problems are only now being addressed, but the prospects look bright.

4. APPLICATIONS

This section examines two applications of the PKS. The first example is a network management system and the second example models a blocks world with multi-agent planning.

4.1 A NETWORK EXAMPLE FROM COMMUNICATIONS

The following is an example of network management and system control for communications systems, as described in [CONRY 88]. The problem deals with restoring end-to-end user connectivity in a network following disruption of a link.

Consider the network shown in Figure 8. There are five regions, A, B, C, D, and E, each describing a subnetwork. Typically the network of Figure 8 has specified circuits, such as A-1 to A-3 to C-1 to C-4 to D-1 to D-2, and each link is limited by a capacity or maximum number of circuits that use it. When a link fails, a new route for all the old circuits, which traverse that link, must be found. Ultimately the circuit restoration problem becomes that of finding a path from the initial node to the terminal node of each original circuit after the network is adjusted for a missing link.

Herein is discussed a straight-forward example of the PKS; therefore, such realistic features as heuristics for cost and evaluation functions, priority categories of a circuit, and equipment constraints are not included. Furthermore, this problem is restricted to one of determining a yes/no answer to a query of whether there exists a path from subnode X to subnode Y.

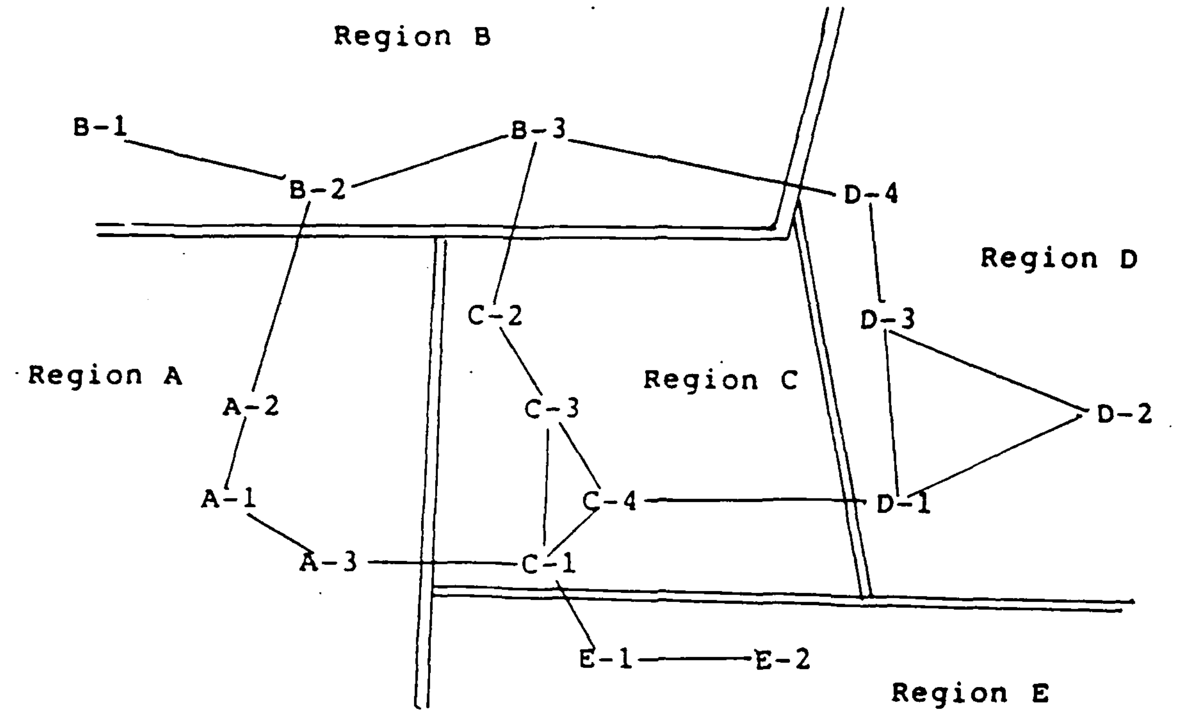


Figure 8. A Network Example

The PKS-graph for this example consists of the five Regional Nodes A . . E, and a manager Node M, which is linked to the other five nodes. The Interface and Node M could be combined as a single node. Node M represents a top-level manager and contains knowledge pertaining only to the connectivity of the regions. Node M does not have knowledge of any subnetwork of Nodes A..E.

Node M contains the following clauses:

```
super_path( a, 2, b, 2).
super_path( b, 2, a, 2).
super_path( a, 3, c, 1).
super_path( c, 1, a, 3).
super_path( b, 3, c, 2).
super_path( c, 2, b, 3).
```

```
    super_path( b, 3, d, 4).
    super_path( d, 4, b, 3).
    super_path( c, 4, d, 1).
    super_path( d, 1, c, 4).
    super_path( c, 1, e, 1).
    super_path( e, 1, c, 1).

    connect( R1, N1, R2, N2) :- super_path( R1, N3, R2, N4),
                                path( R1, N1, N3),  path( R2, N4, N2).

    connect (R1, N1, R2, N2) :- super_path( R1, N3, R3, N5),
                                path( R1, N1, N3),
                                connect( R3, N5, R2, N2).
```

The modes for the predicates of Node M are these:

```
        mode( super_path, IN, IN-OUT, IN-OUT, IN-OUT).

        mode( connect, IN, IN, IN, IN).

        mode( path, IN, IN, IN).
```

Since the particular node that Node M would invoke through "path" depends on the first argument, which must be instantiated when it is invoked, we use the following modified map in Node M:

```
        super_path  ---> M

        connect     ---> M

        path(a,_,_) ---> A

        path(b,_,_) ---> B

        path(c,_,_) ---> C

        path(d,_,_) ---> D

        path(e,_,_) ---> E
```

where the underscore in the second and third arguments of "path" means "don't care".

Each of Nodes A..E holds its links plus these four predicates:

```
link( local, X, X).
link( local, X, Y) :- link( local, Y, X).

path( local, X, Y) :- link( local, X, Y).
path( local, X, Y) :- link( local, X, Z),
                      path( local, Z, Y).
```

where "local" is a, b, c, d, or e for Nodes A, B, C, D, or E, respectively. Again this is a slight modification from the original desciption of the PKS. It is reasonable that certain predicates may be duplicated in several nodes. For very large knowledge bases, one would certainly want all the "built-in" predicates of Prolog to be duplicated. From another viewpoint, the predicates "link" and "path" are not duplicates, but are specialized by the first argument.

The data special to Nodes A..E are these:

```
Node A:     link( a, 1, 2).
            link( a, 1, 3).

Node B:     link( b, 1, 2).
            link( b, 2, 3).

Node C:     link( c, 1, 3).
            link( c, 1, 4).
            link( c, 2, 3).
            link( c, 3, 4).

Node D:     link( d, 1, 2).
            link( d, 1, 3).
            link( d, 2, 3).
            link( d, 3, 4).

Node E:     link( 3, 1, 2).
```

The mode for the new predicate "link" is

```
mode( link, IN, IN-OUT, IN-OUT).
```

The map for each of the Nodes A..E contains only these two predicates:

```
link( local, _, _) ---> local.
path( local, _, _) ---> local.
```

Consider the sample query, connect(a, 1, d, 4), from the user. The following steps occur after the query is passed to Node M.

1. The first clause for "connect" fails since there is no "super _path" between regions A and D.
2. The second clause for "connect" binds N3 = 2 and N5 = 2 in the first literal of the body, "super_path".
3. In parallel, Node A is invoked to verify "path(a, 1, 2)" and Node M is invoked again with "connect(b, 2, d, 4)".
4. This time the first clause of "connect" succeeds because of the fact "super_path(b, 3, d, 4)".
5. In parallel, Nodes B and D are invoked to verify "path(b, 2, 3)" and "path(d, 4, 4)", respectively.
6. Node M returns "yes" to the user through the Interface.

If the user now replies with a semi-colon, indicating a wish for additional answers, the system again replies "yes", to signify that there is another solution. This continues until there are no more solutions to the initial query. Then the user receives a "no". With some additional predicates, one could modify this program to print the path traversals.

Deletions and insertions of "links" in the knowledge bases of the nodes easily allow one to query other configurations. For more complex networks, it is possible that many nodes may be working simultaneously.

4.2 MULTI-AGENT PLANNING

This blocks world consists of three robots and seven blocks, distributed throughout a four-room house where rooms 1 and 3 are each connected to rooms 2 and 4. The house is represented as follows:

```
---------------
|      |      |
|  1   |  2   |
|      |      |
|-------------|
|      |      |
|  4   |  3   |
|      |      |
---------------
```

All three robots are capable of moving from one room to a connected room. Robot-1 can also stack blocks and can transport a block between rooms, but it will not enter a dark room. Robot-2 can unstack blocks and can transport a block between rooms, but it will not enter a dark room. Robot-3 will enter a dark room and switch on the light, but it can neither stack, unstack, nor move blocks. When the last robot exits a room, the light automatically goes off. Robot-1 and robot-2 must always have robot-3 precede them into a room.

In order to simplify this discussion, it is assumed that the robots will not physically interfer with each other during their movements.

After initial and final configurations of the blocks and locations of the robots are given, the problem is to devise and execute a plan to move from the initial to the final state. This approach can be generalized to a house with more than four rooms, to more than three robots each of which may perform numerous actions, and to more complicated interactions among subsets of robots.

This problem is devised to illustrate the following points: hierarchical planning for multiple agents, knowledge organization and allocation, and cooperative parallel execution. These difficulties are common to distributed artificial intelligence problems involving multiple agents.

The multi-agent planning for this blocks world is done as a dual-level hierarchy. A master-planner constructs the system plan, and each robot constructs its own execution plan. The capabilites of robot-1 and robot-2 are abstracted as goals which are assigned to them by the master-planner. Each robot executes a goal by constructing its own plan, which may required communication and coordination with other robots. The hierarchical structure of the agents is depicted in the following graph:

$$\begin{tikzcd} & \text{MASTER-PLANNER} \arrow[dl, no head] \arrow[dr, no head] & \\ \text{ROBOT-1} \arrow[dr, no head] & & \text{ROBOT-2} \arrow[dl, no head] \\ & \text{ROBOT-3} & \end{tikzcd}$$

No link between the master-planner and robot-3 is required since the capability of robot-3 (light switching) is not in the abstraction level of the master-planner.

The link between the master-planner and robot-1 represents not only subgoals (either stacking or transporting), which are presented to robot-1, but also robot-1's request to the master-planner to update a block's characteristics. The link between the master-planner and robot-2 is similar. The link between robot-1 and robot-3 represents robot-1's solicitation of robot-3 to switch on the lights in a particular room. An analogous link exists between robot-2 and robot-3.

For example, if the master-planner has determined a stack-plan for robot-1, then robot-1 receives the message

```
go_stack( r1, r2, r3, new, old, code),
```

which orders robot-1 to move from its current location in room "r1" to room "r3" (passing through room "r2" if "code" = diagonal) where it will find the "new" block ready to be stacked onto the "old" block. The planner has already determined that the "old" block is currently on top of a partially built, goal stack in room "r3" and that the "new" block also sits uncovered on the floor of the same room. However, no instructions are given to robot-1 for how to enlist robot-3 since the planner does not have this knowledge. Rather, the knowledge base of robot-1 holds the instructions for soliciting robot-3 and coordinating its movement with robot-1's objective. Similar comments apply to the transport-plan for robot-1 and to the unstack-plan and the transport-plan for robot-2. Robot-3, which occupies the lowest level of the hierarchy, meekly does as it is told.

4.2.1 THE KNOWLEDGE BASES

The master-planner constructs short-range plans for robot-1 and robot-2. These robots must then coordinate their actions with robot-3. When both robot-1 and robot-2 have succeeded in achieving their intermediate goals, the master-planner begins again to plan. There is no link between robot-1 and robot-2 since any contention over the services of robot-3 is decided by robot-3 itself. This problem's solution allows robot-1 and robot-2 to be engaged simultaneously.

The master-planner's knowledge base can be partitioned into two pieces. The primary piece, the planner, consists of unchangeable facts about the house's architecture and the goal for each block. The planner also contains all the rules for planning, as described below.

The second piece, the blackboard, maintains the database of facts for each block. Each fact contains a block's room number, what lies above and below each block, and a status (unstacked, used, or stacked). The blackboard also contains rules for manipulating the database. All of these rules ultimately reduce to retracting an old block-fact and asserting a new block-fact. No planning control is associated with the blackboard.

The planner's strategy is simple: while the goal state has not been reached, find work for robot-1 and robot-2. This is done with the following rules:

```
plan( available, available) :-
    goal_state,
    return_automata_home.
```

```
        plan( available, available) :-
            assign_duty_to_robot_1( S1),
            assign_duty_to_robot_2( S2),
            plan( S1, S2).
```

The variables S1 and S2 in the second clause are used to delay the recursive call to "plan" until robot-1 and robot-2 have executed their assignments.

However, it is the PKS which allows constructing a plan for robot-2 before robot-1 signals completion of its task. Thus, both the master-planner and robot-1 can operate in parallel. In a problem with more than two robots which must receive planning instructions, this pipelining of plans is even more advantageous.

The choices of planning work for robot-1 are stacking and transporting, with stacking given first priority. If the planner cannot locate a room containing a partially built stack and the next required block, then it attempts to find a block that needs transporting to its goal room. Should both of these choices fail, robot-1 is directed to its goal room.

In an analogous manner, the master-planner tries to locate a non-goal stack which robot-2 is directed to disassemble. Otherwise, the master-planner seeks a block that needs transporting or if this fails, orders robot-2 to its goal room.

Two facts, the room location and the goal location of robot-1, reside in the knowledge base of robot-1. A companion rule, which retracts an old room number and asserts a new room number, is also provided to update robot-1's location. A group of rules in this node consists of "soliciting" robot-3's services for moving about the house and of informing the blackboard about a changed block-fact. The sequence of "soliciting" and "moving" is planned by this node and depends on whether robot-1 is transporting or stacking a block. Finally, another small piece of planning code is attached to this node to allow robot-1 to find its goal room after all blocks are properly stacked. This requires asking the planner about the house's architecture or storing duplicate code.

The knowledge base of robot-2 is similar to that of robot-1 except that its planned interactions with robot-3 depend on whether it is transporting or unstacking a block.

The knowledge base of robot-3 contains its room location and its goal location as well as a "retract/assert" rule for moving about the house. This node also holds the "solicit" synchronization rule. A route planning algorithm was deemed redundant for this fearless automaton.

4.2.2 CONFLICT RESOLUTION

One potential conflict, which must be avoided, is the following situation. Robot-1 and robot-2 are both assigned the duty of transporting a block. Because the master-planner may be planning a goal for robot-2 while robot-1 is executing its goal, the state of the block, which robot-1 is transporting, is not yet updated and could be redundantly assigned to robot-2 for transporting. This is prevented by using the "used" status for this block in the blackboard.

Thus, between the time that the planner has decided to order robot-1 to "transport" block X and the time that robot-1 confirms the order, block X is unavailable for transport. While robot-1 is accomplishing its task, the planner will simply pick the next available block for robot-2 to transport.

Another potential conflict can involve robot-1 and robot-2 contending for robot-3. A request by robot-1 or robot-2 for the services of robot-3 is made in the form of JOB requests. Depending on which request arrives first, robot-3 will respond to that request, complete it, and then answer the next request in its JOB queue.

Since the interleaving of soliciting robot-3 and informing the blackboard of an altered block-fact are both ORDERs to two different nodes (and can be done simultaneously), we enforce sequencing by a synchronizing variable, as described for the variables in the planner's strategy. Thus, only after robot-3 has actually moved into a requested room may the requesting robot move there.

As stated in Section 4.2.1, the planner begins to determine a plan for robot-2 as soon as the plan for robot-1 has been sent to the latter. In a problem which requires the planner to construct M plans for M robots, it is possible that there will be no solution to the ith plan or that the ith plan will conflict with earlier plans. In either case the planner must backtrack and redo some earlier plans. The backtracking mechanism is inherent to each node in the PKS and offers no difficulties since the PROOF TREE for each JOB of each node maintains a precise history of all its deductions.

In this problem no backtracking is needed unless one is interested in finding additional solutions. In this case, one must backtrack over the built-in predicates of "assert" and "retract". Standard Prolog will effectively skip such predicates during backtracking. The blocks world problem uses these predicates so transparently that it is possible to modify the backtracking of the PKS and thereby allow true backtracking over assert and retract. Since assert and retract are used only to update the characteristics of a block or a robot and since they always occur in pairs each of which (1.) retracts a property of an object (block or robot) and then (2.) asserts another property of the same object, one would retract an asserted fact and then assert the related, retracted fact during backtracking.

4.2.3 SAMPLE EXECUTION

Figure 9 displays the first two planning cycles of the planner and the resulting configurations of the blocks and the locations of the robots. The initial state is shown as Cycle 0 of Figure 9. The goal state is to have all three robots in room-1, and to have block-a on block-b on block-c in room-4, block-d alone in room-3, and block-e on block-f on block-g in room-2.

Although the actions of robot-1 and robot-2 occur sequentially within a single cycle (read vertically in Figure 9), the given sequence of moves of robot-3 within a single cycle represents one

T: x n/m	means the robot transport box x from room n to room m
S: x	means robot_1 stacks box x
U: x	means robot_2 unstacks box x
M: n/m	means the robot moves from room n to room m
-	means no action is taken by the robot

```
CYCLE               ROBOT ACTIONS                HOUSE CONFIGURATION
           R1             R2            R3

  0        -              -             -        r1 r2 r3  |
                                                           |        b
                                                        a  1|2      c
                                                 ----------------------
                                                           4|3
                                                        d   |       f
                                                        e   |       g

  1        T: a 1/4       M: 1/2        M: 1/2              |          r2
                          U: b          M: 2/3              |
                                        M: 3/4             1|2   b  c
                                                 ----------------------
                                                 r1 r3     4|3
                                                       d    |       f
                                                    a  e    |       g

  2        M: 4/1         M: 2/3        M: 4/3              |
           M: 1/2         U: f          M: 3/4              |
           T: b 2/3                     M: 4/1             1|2      c
                                        M: 1/2   ----------------------
                                        M: 2/3             4|3   r1 r2 r3
                                                       d    |
                                                    a  e    |    b  f  g
```

Figure 9. A Sample Execution.

of several possible sequences. The exact sequence depends on the order n which requests from robot-1 and robot-2 are received. For example, during cycle 1 robot-3 first moves from room 1 to room 2 at the request of robot-2. Then it receives robot-1's request to move to room 4, which it does by moving to room 3 and then to room 4. For the given initial arrangement, the only

other possible sequence for robot-3 in cycle 1 would be this:
M: 1/4, M: 4/1, M: 1/2.

The goal state for the example is reached on the twelfth planning cycle; see Appendix. Since no work was found for robot-2 after the sixth cycle, the top-level of planning is not as efficient as one would wish. More intelligent planning can be achieved by pursuing longer ranged planning. For example, instead of giving priority to robot-2's unstacking, it may be wiser to determine if robot-2 can help robot-1 move some blocks so that stacking can begin earlier.

An equally significant improvement can be realized if the planner node has access to a multi-processor or several other available (possibly remote) single processors. (Note that the PKS views these different resouces as equivalent.) In such cases, we could restructure the planner node as a subnet of seven nodes: the blackboard, the top-level planner, a robot-1-planner, a robot-2-planner, a stack-planner, a transport-planner and an unstack-planner. The top-level planner is now able to request plan generation by robot-1-planner and robot-2-planner simultaneously, rather than in pipeline fashion. The robot-1-planner may then seek help from the stack-planner or the transport-planner, and the robot-2-planner may utilize the unstack-planner and the transport- planner. Acceptable plans would then be conveyed to robot-1 and robot-2. While the robots are complying with their orders, the planning network could begin working on the next cycle of action.

5. DATA TYPING AND KNOWLEDGE BASE MANAGEMENT AND LEARNING

This section is an initial investigation into efficiency considerations within the PKS and provides a general background for data typing, which is the third task of this report.

In section 5.1 four knowledge base management techniques are presented. These include dependency backtracking, query optimization, caching, and compiled optimization. The first two techniques deal with improvements to the execution mechanisms (forward execution and backtracking) of individual nodes and are associated with the first and second queues of the PKS. The last two techniques deal with reorganizing a logic program to take advantage of special features of the PKS. These two techniques are associated with the third queue of the PKS and can be interpreted as either management or learning in the knowledge base of each node.

In section 5.2 advanced machine learning methods are addressed vis-a-vis data typing. The most promising method of data typing is that used in the language LOGIN, as described in [AIT-KACI 88]. Data typing is then used as the basis for learning by discovery (see [LENAT 83]) and learning by conjunctive conceptual clustering (see [Michalski 80]).

5.1 KNOWLEDGE BASE MAINTENACE

5.1.1 DEPENDENCY-BASED BACKTRACKING

Standard Prolog performs backtracking chronologically; i.e., Prolog recovers from a failure by returning to the most recently solved literal. Within the body of a clause, this is always the literal, which lies to the left of the failed literal. Non-chronological or dependency-based backtracking (DB) recovers from failure by moving left from the failed literal to the literal that most recently bound a variable of the failed literal. DB is more intelligent than chronological backtracking, but it pays for it superiority by maintaining two lists for each literal in the body of each clause. The first list for a particular predicate p in the body of a clause contains all the predicates which are to the left of p in the body of the clause and share a variable with p. The second list, a sublist of the first, is used to indicate those literals which actually bind a variable of p.

For example, if the body of a clause is

a(X), b(X,Y), c(Z), d(X,Z), e(X).

then the list for d(X,Z) is [a(X), b(X,Y), c(Z)] and the sublist is [a(X), c(Z)]. If d(X,Z) fails during execution, DB will return to c(Z) (as will chronological backtracking), but with a failure in c(Z), the dependency-based backtracking will return to a(X) by skipping over b(X,Y) since b(X,Y) is not in the sublist of d(X,Z) and therefore did not bind any variables of d. When forward execution begins (by successfully re-binding X in a), b(X,Y) and c(Z) must also be re-solved; hence, d must maintain both its list and its sublist.

In the PKS the backtracking mechanism of each node is a variation of chronological backtracking which is based on the chronology of the solved literals. A more intelligent form of backtracking, such as dependency-based backtracking, was deemed unnecessary for the following reasons. First, intellegent backtracking requires much more bookkeeping and is therefore more costly during consult-time and during run-time. Second, the current backtracking scheme of the PKS does display a modest level of sophistication in that it ignores all the fired and waiting literals as possible backtracking points. It is only those literals which have actually been solved that qualify as possible backtracking points. Third, if a form of caching is implemented, then it may not be necessary to re-solve a request other than matching a cached fact. In such cases the use of more intelligent backtracking becomes counter-productive. After weighing both pluses and minuses, dependency-based backtracking does not appear to be capable of significantly improving the present version of the PKS.

5.1.2 QUERY OPTIMIZATIONS

5.1.2.1 INTER-NODAL AND-OPTIMIZATION

The PKS Version 1.0 handles inter-nodal REQUESTs by using a single goal; thus, if node i ascertains during problem-solving a need to solve predicate p, which lies in node j, then a REQUEST by node i presents the single predicate p to node j as a goal. From this view of requesting a single goal, inter-nodal query optimization is inane.

However, consider that node i holds the following clause with three OUT-variables (non-ground):

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

Assume that node j holds predicates b and d, and that node k holds predicates c and e. The PKS Version 1.0 ORDERs b from node j, sets predicates c to WAIT (since X is held by b), ORDERs d from node j, and sets predicate e to WAIT (since X is held by b and Z is held by d). Assuming that the orders are successfully solved, node i ORDERs predicate c from node k and then ORDERs predicate e from node k.

A reduction in the number of inter-nodal communications can be obtained by letting node i ORDER (b(X), d(Z)) as a single goal from node j. When a success is return by node j, node i can then ORDER the goal (c(X,Y), e(X,Z)) from node k. In this simple example, the minimum number of communications has been cut in half. Once the goal (b(X), d(Z)) is received, the intra-nodal query optimizer (described below) of node j would decide how the query/goal may be optimized.

Adjustments to the PKS Version 1.0 would involve (1.) replacing the depth-first readiness checking of the predicates in the body of a clause by a breadth-first examination, and (2.) generalizing the communicated REQUEST goal from a single predicate to a conjunction of predicates.

5.1.2.2 INTRA-NODAL AND-OPTIMIZATION

The generalization of REQUESTs to conjunctive goals can also be used for intra-nodal optimizing. For the above example, one would obtain the following reordering:

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

Investigating how to reorder the literals in the body of a clause defines another function for the Third Queue. This refinement is applicable to standard Prolog and is addressed in more detail in [ROWE 88].

5.1.3 CACHING

Herein is used the definition of [ROWE 88] for caching: caching means asserting unnecessary or redundant facts to improve efficiency. A spectrum of control-structure concepts for rule-based system ranges from increased flexibility to increased efficiency. Caching is an example of the extreme of increased efficiency. By contrast, meta-rules are the extreme of increased flexibility. The efficiency of backward chaining is often improved by caching some or all of the conclusions derived. As a side-effect, there will be more facts to search after caching. Consequently, one must be selective in which conclusions to cache.

Within the PKS caching must be very selective. Surely a node does not need to remember solutions returned from other nodes if those solutions are not part of a final solution to a user's query. Therefore, a command for caching must be directed from the Interface Node. At the same time that the Interface reports the solution to the User, the Interface must command its original servant node to cache the result. This caching command is then relayed throughout those nodes of the network that contributed to the final solution. Caching causes run-time problems for asserting solutions within each node and for adjusting the allocation maps. Furthermore, the static allocation schemes' principle of non-distribution of bundles will be violated unless the following steps are taken.

Assume the Node N has been directed to cache a solution of predicate p and that this solution was obtained from Node M. Node N will now assert this solution (in some form) as a fact in its private knowledge base.

To avoid adjusting references by other nodes to predicate p and to maintain the principle of non-distribution of predicate bundles, Node N must determine from its proof-tree which clause invoked Node M and accepted the solution for predicate p. Within that clause the predicate p must be replaced by a new predicate, p_1, which has exactly the same arguments as p.

Node N adds this new rule: p_1(arguments) :- p(arguments), where the arguments of p_1 and p are identical. Node N must also add the solution from Node M in the form of p_1(argument-bindings) rather than p(argument-bindings), and this fact must appear before the new rule in Node N's knowledge base. Allocation adjustments must be made to Node N and the Interface to account for the new predicate p_1. Node N must be able to copy the mode-declaration of predicate p from Node M (or from the Interface, if it holds a duplicate of the mode-declaration) in order to reproduce it.

It also appears that there would be no cause for caching unless the inter-nodal solution provides enough information so that the newly asserted fact for p_1 contains only ground arguments.

5.1.4 COMPILED OPTIMIZATIONS

The final technique for improving efficiency in the PKS is the use of compiled optimizations for OR-parallelism. This can be applied to any procedural bundle which is initially unordered. Procedures (or clauses) of such a bundle can be ordered by a metric m( ). This metric can be a function of several different characteristics and can be either static or dynamic. Described below is a static metric. This static metric determines the minimum number of resolutions that are required to instantiate the head of the clause. This value is only a lower bound for the actual number of resolutions that may be required during run-time.

For example, consider that the following two clauses are the only "a" clauses in the program and that the goal is a:

```
C1--->     a :- b, c, d.
C2--->     a :- e, f, g, h, i.
```

If one has already determined m(b)=4, m(c)=1, m(d)=5, m(e)=1, m(f)=2, m(g)=1, m(h)=2, and m(i)=1, then one can assign m(C1) to be 10, m(C2) to be 7, and m(a) to be the minimum of the clausal metrics or 7. Clause C2 is therefore preferred to clause C1. The calculation of m(a) may be needed in another clause in which predicate a is a body literal. The notation m( ) is slightly abused here since the argument may be either a clause or a predicate. In general, one first determines the metric of each individual member of a predicate bundle (by summing the metrics of the predicates in the body) and then calculates the metric of the predicate (bundle) p by choosing the minimum of the metrics of the members of the bundle p.

In algorithmic form, one initially assigns m(predicate) to be one for each predicate that contains at least one fact. Then iteratively assign clausal metrics, followed by predicate metrics. At any step a metric can be assigned to a clause if all predicates in the body of the clause have already been assigned. A predicate metric can be assigned if all members of its bundle have been assigned.

Notice that recursive clauses are finessed since there must always be an accompanying fact which terminates the recursion, and therefore the associated predicate has metric equal to one. If an infinite loop is encountered, it is a sign of erroneous code. Consequently, the algorithm must terminate.

This metric assignment scheme is sequential and takes no account of possible AND-parallelism within an individual clause. For a parallel version of this scheme, a metric will be obtained from the optimal execution graph of the clause. This can also be determined statically.

If the optimal execution graph is deterministic the set of metrics for the clause is a singleton. For example, if all the predicates in the body of a clause are either ground or "indepedent" (i.e., all arguments of all predicates are independent), then all predicates can be evaluated in one step in parallel and the metric for the clause is the maximum of the individual metrics.

In the example of this section, if predicates b through i are ground then

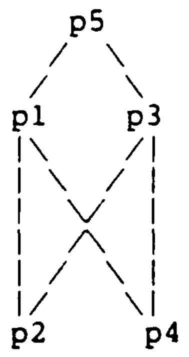
$$m(C1) = \max(4,1,5) = 5,$$
$$m(C2) = \max(1,2,1,2,1) = 2,$$
$$m(a) = \min(5, 2) = 2.$$

For dependent predicates, the metric of the clause will be the sum of the metrics that one finds on each level of the optimal, static execution graph of the clause.

For example, the clause

```
C --->   p5(A,B) :- p1(A,B), p2(B,C), p3(C,D), p4(A,D).
```

has the execution-graph (in the PKS) shown below:



According to the literal ordering of the PKS, predicates p1 and p3 can be executed in parallel since their variables are independent. The same is true for predicates p2 and p4, which are executed on the second level of the execution graph. Thus,

$$m(C\text{-level-}1) = \max\{\ m(p1),\ m(p3)\ \},$$
$$m(C\text{-level-}2) = \max\{\ m(p2),\ m(p4)\ \},$$

and

$$m(C) = \max\{ m(p1), m(p3) \} + \max\{ m(p2), m(p4) \}.$$

For purposes of static assignment of metrics, it is assumed that no variable in the head of a clause is ground.

This same scheme can be adapted to a dynamic form, but the anticipated high run-time overhead discourages implementing it.

Other compiled optimizations for standard Prolog are also possible within individual nodes. For example, rule collapsing can reduce constructing two frames to constructing one frame during forward execution if the two rules a :- b,c,d. and c :- e,f. are collapsed to the single rule a :- b,e,f,d.

5.2 KNOWLEDGE BASE LEARNING AND DATA TYPING

In this section, classical machine learning techniques are explored in a logic programming language which has a native mechanism for data typing, such as LOGIN (see [AIT-KACI 88]). Section 5.2.1 summarizes the pertinent characteristics of LOGIN. Section 5.2.2 examines a mechanism for automatically generating new data types, which are related to a semantic network of old data types. Section 5.2.3 discusses how conjunctive conceptual clustering of [MICHALSKI 80] may be used to convert a standard Prolog program into a LOGIN program.

The primary reason for adding data typing to a language is to allow programs in that language to model more closely real-world objects. Those objects are represented as abstract objects in the program. A specification of a data type includes attributes, values, and operations for that type. [PRATT 84] and [MARCOTTY 86] have lengthy discussions on data typing for procedural high-level languages.

From the persepective of database programming languages, there is a need to support software and data evolution in large, long-lived systems. This in turn demands support for new data types. A complete examination of data typing and persistence is found in [ATKINSON 88].

5.2.1 DATA TYPING USING INHERITANCE IN LOGIC PROGRAMMING

LOGIN [see AIT_KACI 88] is a logic programming language which incorporates inheritance-based reasoning as a part of the unification process. The motivation for LOGIN was to separate inheritance from logical inference. The unification process of LOGIN includes inheritance-based information and is a generalization of the unification algorithm of [HUET 76], which is based on maintenance of equivalence classes of subterms. The algorithm of [HUET 76] is a variation of the classic algorithm for testing the equivalence of two finite automata.

Standard PROLOG and its variants are all able to deal with inheritance with additional clauses (facts and rules). For example,

```
dog(fido).
mammal(X) :- dog(X).
animal(X) :- mammal(X).
```

clearly expresses that fido is a dog, a mammal and an animal. These and other inheritance clauses can be eliminated if the interpreter has some way to know it can instantaiate X to fido anywhere it encounters the predicate dog, mammal, or animal. Since logic programming entails only resolutions and unifications, the obvious alternative is to expand the power of unification in order to include inheritance.

The innovation of LOGIN is to use (data) type-checking during unification. If each argument in a literal is typed and if types collectively define semantic networks for the world of discourse, then inheritance reduces to testing either subset containment or set membership.

From the canine example, fido is a member of the set (data type) of dogs, which is a subset of the set of mammals, which is a subset of the set of animals.

In terms of run-time, LOGIN replaces a series of inheritance resolutions (and the associated frame constructions) by a single inheritance unification, which is more economical. To accomodate an inheritance, unification is generalized to be the computation of a greatest lower bound of two symbols relative to the < ordering. Thus, one must explicitly declare the semantic network; for example, {fido, fifi} < dog, dog < mammal, mammal < animal, {esmerelda} < snake, snake < reptile, reptile < animal.

One limitation of Prolog that must be eliminated to generalize unification is the syntax of a predicate's argument. For example, with fixed arity, a predicate a(X,Y) must always appear with exactly two arguments and in that prescribed order (positional interpretation). This can be corrected by viewing each argument as a record-structure and by explicitly labeling (or data typing) the record fields. Both of these restrictions are eliminated through explicit labeling of record fields (see next example).

Unification in LOGIN permits unification of predicate names (the signature of the predicate) not only for lexically identical names but also for any predicate with a name that is a subset of the orginal name. Thus, unification is a greatest lower bound operation. When the greatest lower bound is empty, unification fails.

LOGIN retains the Prolog property of coreferencing variables; i.e., in p(X,Y,X) the two X-arguments are bound to the same constant.

A nice example from [AIT-KACI 88] is this:

```
     person(id     => name(first => string;
                           last  => X : string);
            born   => date(day => integer;
                           month => monthname;
                           year => integer);
            father => person(id => name(last => X : string))).
```

The ROOT SYMBOL of this well-formed psi-term (wft) is "person". The ATTRIBUTE LABELs of "person" are "id", "born", and "father". The ATTRIBUTE LABELs map the ROOT SYMBOL into the corresponding types (listed after the =>), which in turn are sub-psi-terms. The

TAG (a variable in Prolog) X is used here only for coreferencing. If the second "X:" were not followed by "string", the psi-term would not be well-formed.

As in Pascal record and sub-record structures, one can refer to a subfield through concatenation of LABELs; e.g. father.id.last is the TAG X.

The type signature Sigma is a partially-ordered set of symbols. It is the semantic net used in inheritance. The greatest element of the net is U (the universe of discourse), and the least element of the net is E (the empty set). The partial-ordering is set inclusion of the type symbols. A missing type symbol in a wft is understood to be U.

The final defintion from [AIT-KACI 88] is subtype. A wft t1 is a subtype of wft t2 if (1.) the ROOT SYMBOL of t1 is a subtype in Sigma of the ROOT SYMBOL of t2, (2.) all ATTRIBUTE LABELs of t2 are ATTRIBUTE LABELs of t1, and (3.) all coreference constraints binding in t2 must also be binding in t1.

In an implementation of LOGIN, backtracking will affect both resolution (through frames) and unification. Inheritance type-checking can be done statically before run-time. LOGIN also provides an efficient way for handling database queries which require a "set at a time" mode. Finally, through inheritance, LOGIN provides an object-oriented flavor to logic programming. Although an extension of the PKS to include data typing and inheritance is desirable, it is not clear what restrictions on the signatures will be required. For example, is it necessary that all nodes contain duplicates of the signature(s)?

5.2.2 LEARNING SIGNATURE ELABORATIONS

The type signature Sigma provides a perfect setting for a learning mechanism for the Third Queue of the PKS. Such an expansion of Sigma can be accomplished automatically or in concert with a user.

Consider the following section of a signature Sigma:

```
                   -----------
                   | type Z   |
                   |  att: C  |
                   -----------
                     /      \
                    /        \
                   /          \
          -------------      ------------
          | type X     |     | type Y    |
          |   att: C   |     |  att: C   |
          |        A1  |     |       B1  |
          |        A2  |     |       B2  |
          -------------      ------------
```

Here, type Z has the single attribute C, type X has the attributes C, A1, and A2, and type Y has the attributes Y, B1 and B2. If the attributes A1 and B1 can be shown to be the same, then the new type W can be inserted into Sigma. Type W will have the attributes C and C2 (where C2 = A1 = B1). The given section of Sigma now assumes the following structure:

```
                    -----------
                    | type Z  |
                    |  att: C |
                    -----------
                         |
                         |
                    ------------
                    | type W   |
                    |  att: C  |
                    |       C2 |
                    ------------
                      /       \
                     /         \
                    /           \
           ------------       ------------
           | type X   |       | type Y   |
           |  att: C  |       |  att: C  |
           |      C2  |       |      C2  |
           |      A2  |       |      B2  |
           ------------       ------------
```

The above example is a form of insertion into Sigma through generalization. In general, if two vertices in Sigma are siblings, then Sigma can be expanded between the parent and the siblings if there exists an attribute which is common to the siblings and is not held by the parent. When the siblings hold more than one common, non-parental attribute, then there are many alternative expansions of Sigma that are possible. Let Att_P denote the set of attributes which type P possesses, and let | Att_P | denote the size of set Att_P. Then if type Z is the parent of sibling types X and Y, set S will denote the intersection of (Att_X and Att_Y) minus Att_Z, where minus is set complement, and | S | = N, where N is a non-negative integer, which is bounded above by UB

UB = minimum { | Att_X | , | Att_Y | } - | Att_Z |.

Consider an example in which the parent Z holds only attribute C, one child X holds attributes C, A1, A2, and A3, and the second child Y holds attributes C, B1, B2, B3. Then, | S | = N and 0 <= N <= 3. If N is 0, then no new vertex can be added to Sigma. If N = 1, then Sigma's expansion will be like that of the previous example. If N=2, then Sigma's expansion will be the sub-graph shown below. In the N=2 expansion, it is assumed that some attribute Ai of type X is the same as the Bj attribute of type Y and that attribute has been relabeled as C2. Similarly, some other attribute Am of type X is the same as some other Bn attribute of type Y and that common attribute has been relabeled as C3.

```
                 ------------
                 | type Z    |
                 |  att: C   |
                 ------------
                      |
             -------------------
             |                 |
        ------------      ------------
        | type W1   |     | type W2   |
        |  att: C   |     |  att: C   |
        |      C2   |     |      C3   |
        ------------      ------------
             |                 |
             -------------------
                      |
             -----------------
             | type W12       |
             |  att: C,C2,C3  |
             -----------------
                      |
             -------------------
             |                 |
        ------------      ------------
        | type X    |     | type Y    |
        |  att: C   |     |  att: C   |
        |      A1   |     |      B1   |
        |      A2   |     |      B2   |
        |      A3   |     |      B3   |
        ------------      ------------
```

The extreme case that N is UB (UB=3 in the last example) is special. Either one has discovered that the siblings are twins or the type with more attributes is a subtype of its sibling. In the first case, the prudent choice is to collapse the subgraph to two nodes, the parent and a single child. The latter case should place the supertype between the parent and the subtype.

In general, if 1 <= N <= UB-1, then the complete lattice of N elements can be inserted between the parent and the two children. The Minimal Element (the empty set) of the lattice will coincide with the parent. The Maximal Element (all N elements) is the common parent of the two original children. All elements other than the Minimal Element are new vertices for Sigma. In practice there must be some small number used as a limit on the number of attributes that distinguish a parent from its children because of the exponential number of vertices that are added. If a has two children and the children have k attributes that are not attributes of the parent, then one may potentially add 2**k - 2 vertices between this parent and its children.

For example, suppose N = 3, six type-vertices can be inserted between the parent and the two children. The following graph illustrates this case with the assumption that C2, C3, and C4 are the attributes common to both type X and type Y.

Further generalizations of inserting lattices into a type signature include (1.) parents with more than two children and (2.) a child with multiple parents. The theory is easily extended to larger families (more than two children). The complexity of the problem is, however, increased severly since one must consider all partitions of all combinations of all attributes which are common to the children.

```
                 ------------
                 | type Z   |
                 |  att: C1 |
                 ------------
                      |
     ----------------------------------
      |               |               |
------------     ------------    ------------
| type W2  |     | type W3  |    | type W4  |
|  att: C  |     |  att: C  |    |  att: C  |
|       C2 |     |       C3 |    |       C4 |
------------     ------------    ------------
   |      |        |      |        |      |
   |      -------------   |        |      |
   |               |  |   -------------   |
     --------------------------------------
   |  |            |  |            |  |
------------     ------------    ------------
| type W24 |     | type W23 |    | type W34 |
|  att: C  |     |  att: C  |    |  att: C  |
|       C2 |     |       C2 |    |       C3 |
|       C4 |     |       C3 |    |       C4 |
------------     ------------    ------------
      |               |                |
     -----------------------------------
                      |
          ------------------------
          | type W234            |
          |  att: C, C2, C3, C4  |
          ------------------------
                      |
          ------------------------
           |                     |
      ------------        ------------
      | type X   |        | type Y   |
      |  att: C  |        |  att: C  |
      |       C2 |        |       C2 |
      |       C3 |        |       C3 |
      |       C4 |        |       C4 |
      |     more |        |     more |
      ------------        ------------
```

Extension to multiple parents is also completely analogous to the existing theory. Consider a child-type with four attributes and two parents, each of which has only one of the child's attributes. Then intermediate type-vertices can be defined as shown in the following graph:

```
        ------------    ------------
        | type Z1  |    | type Z2  |
        |  att: A1 |    |  att: A2 |
        ------------    ------------
             |               |
     ---------------   ---------------
     |             |   |             |
------------    ------------   ------------
| type W13 |    | type W12 |   | type W23 |
|  att: A1 |    |  att: A1 |   |  att: A2 |
|       A3 |    |       A2 |   |       A3 |
------------    ------------   ------------
     |               |               |
     ---------------------------------
                     |
                ------------
                | type X   |
                |  att: A1 |
                |       A2 |
                |       A3 |
                ------------
```

These insertions algorithms are the first level of learning by discovery, as originially introduced by [LENAT 83]. In order to implement a practical insertion algorithm, it will be necessary to supply heuristics for pruning the combinatorial explosion of new data types. As a type signature matures, meta-heuristics become necessary. [LENAT 83] suggests that meta-heuristics can also be discovered by applying the learning procedures to the heuristics themselves.

5.2.3 LEARNING BY CONJUNCTIVE CONCEPTUAL CLUSTERING

It is possible that standard Prolog programs can be automatically converted to LOGIN programs through the learning schemes of conjunctive conceptual clustering, as introduced by [MICHALSKI 80]. The necessary prerequisites are a set of objects (the data types), a set of attributes, which characterize the objects, and a body of background knowledge. The result is a hierarchy of object classes. It would be necessary to extend this work from hierarchies to lattices, but the automatic construction of a type signature would provide a meaningful restructuring of a program in order to facilitate human comprehension. This divison of labor may also be another basis for static allocation of logic programs.

One final comment on the advantages of data typing in logic programming is this: data typing can provide a route for logic programs to become "open systems". As defined by [AGHA 85], "open systems" must possess (1.) continuous availability (there is no closed-world hypothesis), (2.) modularity, and (3.) extensibility. The second and third characteristics are enhanced with data typing, although the closed-world assumption remains a problem.

6. CONCLUSIONS

This report has extended the theoretical basis of the Parallel Knowledge-based System and has illustrated typical AI applications which are appropriate for it.

Theoretical issues, which have been examined, include restructuring the functions of individual nodes, methods for dealing with duplicated data, metrics for static allocation schemes, and management of the knowledge bases. There remain problems for the PKS, which have not been examined. No truth maintenance system has been attached to the system. The PKS presently handles uncertainty only in its elementary form; i.e., an extra argument in each predicate can be used for certainty factors, as in standard Prolog. The PKS does not support graceful degradation of the system. This is dependent on having duplicated data and is an area that will be examined in furture research. These theoretical questions can be unified as an effort is made to transform the PKS into an "open system", as defined in Section 5.2.2.

Future work will also move the ADA/VAX simulation to a C-coded distributed system. The anticipated network will be an ethernet with nodes of work stations, PC's, a VAX 11/780, and a micro- VAX. With this network, it will be possible to gather data on the static allocation schemes. These needed data are required for parameter adjustments and benchmarks for communication costs.

Ultimately, the multi-agent planning system will be expanded to run on the distributed system. Issues to be addressed include hierarchical planning, conflict resolution, granularity of parallelism, and fault-tolerance. It is anticipated that solutions to new problems will aid in expanding the capabilities of the Second and Third Queues of each node.

Finally, the distributed system can also be enlarged to encompass the efficient implementation issues described in Section 5.

7. REFERENCES

1. Agha, G. and Hewitt, C. (1985). Concurrent programming using actorsL exploiting large-scale parallelism. AI Memo 865, Massachusetts Institute of Technology.
2. Ait-Kaci, H. and Nasr, R. (1988). Integrating data type inheritance into logic programming. In M. P. Atkinson, P. Buneman, and R. Morrison (Eds.), Data Types and Persistence, (pp. 121-134). New York: Springer-Verlag.
3. Atkinson, M. P., Buneman, P., and Morrison, R. (Eds.) (1988). New York: Springer-Verlag.
4. Bond, A., and Gasser, L. (1988). An analysis of problems and research in DAI. In A. H. Bond and L. Gasser (Eds.), Readings in Distributed Artificial Intelligence (pp. 3-36). San Mateo, CA: Morgan Kaufmann.
5. Chang, J.-H., Despain, Am M., and DeGroot, D. (1985). AND-parallelism of logic programs based on a static data dependency analysis, IEEE COMPCON Spring 85, 218-225.
6. Chung, W.-K., and Day, W. (1987, April). A static allocation approach for parallel execution of logic programs. Paper presented at the IEEE Southeastcon '87, Tampa, Florida.
7. Chung, W.-K. (1987). A Knowledge Based System for Parallel Processing of Logic Programs. Ph.D. dissertation, Auburn University, Auburn, Alabama.

8. Chung, W., and Day, W. (1989, March). Detecting AND-Parallelism in Logic Programs without Literal Ordering. Paper presented at the 1989 IEEE International Phoenix Conference on Computers and Communications, Phoenix, Arizona.

9. Clark, K., and Gregory, S. (1986). PARLOG: parallel programming in logic, ACM Trans. Prog. Sys., Vol. 8, 1-49.

10. Conery, J. S., and Kibler, D. F. (1981). Parallel interpretation of logic programs, Proceedings of the Conference on Functional Programming and Computer Architecture, 163-170.

11. Conery, J. S. (1987). Parallel Execution of Logic Programs. Boston: Kluwer Academic Publishers.

12. Conry, S., Meyer, R., and Lesser, V. (1988). Multistage negotiation in distributed planning. In A. H. Bond and L. Gasser (Eds.), Readings in Distributed Artificial Intelligence (pp. 367-384). San Mateo, CA: Morgan Kaufmann.

13. Davis, R., and Smith, R. (1983). Negotiation as a metaphor for distributed problem solving. Artificial Intelligence, 20(1), 63-109.

14. Day, W. (1987). Cooperative Knowledge Bases. Final Report (Contract SCEEE F30602-81-C-0193, Task 0081). Griffiss AFB, New York: Rome Air Development Center.

15. DeGroot, D. R. (1984). DeGroot, D. Restricted AND-parallelism, Proceedings of International Conference on Fifth Generation Computer Systems, 361-368.

16. DeGroot, D. (1988). A technique for compiling execution graph expressions for restricted AND-parallelism in logic programs, Journal of Parallel and Distributted Computing, Vol. 5, 494-516.

17. Durfee, E., Lesser, V., and Corkill, D. (1987). Coherent cooperation among communicating problem solvers. IEEE Transactions on Computers, C-36, 1275-1291.

18. Genesereth, M. R. and Nilsson, N. J. (1987). Logical Foundations of Artificial Intelligence (pp. 161-176). San Mateo, CA: Morgan Kaufmann.

19. Hogger, C. J. (1984). Introduction to Logic Programming, New York: Academic Press.

20. Knight, K. (1989). Unification: a multidisciplinary survey. ACM Computing Survery, Vol. 21, No. 1, March 1989, 93-124.

21. Lenat, D. (1983). The role of heuristics in learning by discovery: three case studies. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), Machine Learning: An Artificial Intelligence Approach, (pp. 243-306). Palo Alto, CA: Tioga Publishing.

22. Li, Y. (1988, April). DKM - a distributed knowledge representation framework. Paper presented at the Second International Conference on Expert Database Systems, Tysons Corner, Virginia.

23. Lin, Y.-J., and Kumar, V. (1986). A parallel Execution scheme for exploiting AND-parallelism of logic programs, Proceedings 19866 International Conference on Parallel Processing, 972-975.

24. Lin, Y.-J., and Kumar, V. (1988). An execution model for exploiting AND-parallelism in logic programs, New Generation Computing, Vol. 5, 393-425.

25. Marcotty, M. and Ledgard, H. F. (1986). Programming Language Landscape (Second Edition). Chicago, IL: Science Research Associates, Inc.

26. Michalski, R. S. and Stepp, R. E. (1983). Learning from observation: conceptual clustering. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), Machine Learning: An Artificial Intelligence Approach, (pp. 331-364). Palo Alto, CA: Tioga Publishing.

27. Mitchell, T., Utgoff, P. E., and Banerji, R. (1983). Learning by experimentation: acquiring and refining problem-solving heuristics. In R. S. Michalski, J. G. Carbonell, and Tom, M Mitchell (Eds.) Machine Learning: An Artificial Intelligence Approach, (pp. 163-189). Palo Alto, CA: Tioga Publishing.

28. Pratt, T. W. (1984). Programming Languages (Second Edition). Englewood Cliffs, NJ: Prentice-Hall.

29. Rowe, N. C. (1988). Artificial Intelligence Through Prolog. Englewood Cliffs, NJ: Prentice-Hall.

30. Shapiro, E. (1986). Concurrent prolog: a progress report, IEEE Computer, 44-58.

31. Smith, R. (1980). The contract net protocols: high-level communication and control in a distributed problem solver. IEEE Transactions on Computers, C-29(12), 1104-1113.

32. Smith, R., and Davis, R. (1981). Frameworks for cooperation in distributed problem solving. IEEE Transactions on Systems, Man, and Cybernetics, SMC-1(1), 61-70.

33. Tanenbaum, A. S. (1988). Structured Computer Organization, Englewood Cliffs, NJ: Prentice-Hall.

34. Tanenbaum, A. S. (1988). Computer Networks. Englewood Cliffs, NJ: Prentice-Hall.

35. Ueda, K. (1986). Guarded horm clauses. In Doss, G. and Hartmanis, J. (Eds.) Logic Programming '85, (pp. 168-179). New York: Springer- Verlag.

APPENDIX

CODE AND SAMPLE EXECUTION

of

A BLOCKS WORLD WITH MULTI-AGENT PLANNING

HYPOTHESES -

Three automata, seven boxes, and a four-room house.

The map of the house is as follows:

```
---------------------------------
|                |               |
|    Room 1      |    Room 2     |
|                                |
|                                |
|                |               |
|                |               |
-------    -----------    --------
|                |               |
|    Room 4      |    Room 3     |
|                                |
|                                |
|                |               |
|                |               |
---------------------------------
```

It is assumed that each door between rooms is actually a triple door, each of which is maked 1, 2, or 3 and is reserved for Automaton 1, 2, or 3, respectively. Hence, it is possible for two or all three automata to move between the same two rooms simultaneously.

RULES -

Automaton 1 can either stack boxes or move a single box from one room to another. It will not enter a dark room.

Automaton 2 can either unstack boxes or move a single box from one room to another. It will not enter a dark room.

Automaton 3 will enter a dark room and switch on the light. It can neither stack, unstack, nor move boxes.

When an automaton exits an empty (no other automaton remaining) room, the light automatically goes off. Exception: if a second automaton is entering the same room which the first automaton is exiting, the light will stay on (this exception is never coded, nor does the code account for the fact that Automaton 1 does not have need of Automaton 3 if Automaton is entering a room where Automaton 2 is located).

INITIAL STATE -

All three automata are located in Room 1.

Box A is alone in Room 1.

Box B is on top of Box C in Room 2.

Box D is on top of Box E in Room 4.

Box F is on top of Box G in Room 3.

GOAL STATE -

All three automata are located in Room 1.

Box A is on top of Box B, which is on top of Box C in Room 4.

Box E is on top of Box F, which is on top of Box G in Room 2.

Box D is alone in Room 3.

A (HUMAN) SOLUTION: (OPTIMAL?)

	ACTIONS			AUTOMATON LOCATION			BOX LOCATION							LIGHT			
	A1	A2	A3	A1	A2	A3	A	B	C	D	E	F	G	R1	R2	R3	R4
0.	*	*	*	R1	R1	R1	R1	R2	R2	R4	R4	R3	R3	+	-	-	-
1.	*	*	M:1/2	R1	R1	R2	R1	R2	R2	R4	R4	R3	R3	+	+	-	-
2.	*	M:1/2	M:2/3	R1	R2	R3	R1	R2	R2	R4	R4	R3	R3	+	+	+	-
3.	*	U:B	M:3/4	R1	R2	R4	R1	R2	R2	R4	R4	R3	R3	+	+	-	+
4.	T:A1/4	*	M:4/1	R4	R2	R1	R4	R2	R2	R4	R4	R3	R3	+	+	-	+
5.	M:4/1	*	M:1/2	R1	R2	R2	R4	R2	R2	R4	R4	R3	R3	+	+	-	-
6.	M:1/2	*	M:2/3	R2	R2	R3	R4	R2	R2	R4	R4	R3	R3	-	+	+	-
7.	T:B2/3	M:2/3	M:3/4	R3	R3	R4	R4	R3	R2	R4	R4	R3	R3	-	-	+	+
8.	T:B3/4	U:F	M:4/1	R4	R3	R1	R4	R4	R2	R4	R4	R3	R3	+	-	+	+
9.	M:4/1	M:3/4	M:1/2	R1	R4	R2	R1	R4	R2	R4	R4	R3	R3	+	+	-	+
0.	M:1/2	U:D	M:2/3	R2	R4	R3	R4	R4	R2	R4	R4	R3	R3	-	+	+	+
11.	T:C2/3	T:D4/3	M:3/4	R3	R3	R4	R4	R4	R3	R3	R4	R3	R3	-	-	+	+
12.	T:C3/4	*	M:4/3	R4	R3	R3	R4	R4	R4	R3	R4	R3	R3	-	-	+	+
13.	S:B	*	M:3/2	R4	R3	R2	R4	R4	R4	R3	R4	R3	R3	-	+	+	+
14.	S:A	T:G3/2	M:2/3	R4	R2	R3	R4	R4	R4	R3	R4	R3	R2	-	+	+	+
15.	T:E4/3	M:2/3	M:3/2	R3	R3	R2	R4	R4	R4	R3	R3	R3	R2	-	+	+	-
16.	T:E3/2	T:F3/2	M:2/1	R2	R2	R1	R4	R4	R4	R3	R2	R2	R2	+	+	-	-
17.	S:F	M:2/1	*	R2	R1	R1	R4	R4	R4	R3	R2	R2	R2	+	+	-	-
18.	S:E	*	*	R2	R1	R1	R4	R4	R4	R3	R2	R2	R2	+	+	-	-
19.	M:2/1	*	*	R1	R1	R1	R4	R4	R4	R3	R2	R2	R2	+	-	-	-

KEY:

M:i/j	means that this automaton moves from room i to room j
S:X	means that this automaton stacks box X on the correct stack
T:Xi/j	means that this automaton transport box X from room i to room j
U:X	means that this automaton unstacks box X from the correct stack
*	means this automaton is idle
+	means the light is ON in this room
-	means the light is OFF in this room

PKS GRAPH

```
                    USER
                      |
                      |
                 INTERFACE
                      |
                      |
                   MASTER -------------------------- BLACKBOARD
                    |     |                             |    |
                    |     |                             |    |
      ---------------     ---------------               |    |
       |                                |               |    |
       |                                |               |    |
AUTOMATON_1-----AUTOMATON 3-----AUTOMATON 2             |    |
|                                        |              |    |
|                                        |              |    |
|                                       -----------------    |
|                                                            |
|                                                            |
--------------------------------------------------------------
```

ALLOCATION MAP

```
                              MASTER
                 invokes these foreign predicates
                                 |
                                 |
                                 |
          -------------------------------
          |               |               |
          |               |               |
          |               |               |
      AUTOMATON 1     AUTOMATON 2     BLACKBOARD
      place_1         place_2          location
      return_1        return_2         reserve
      carry_1         carry_2
      amble_1         amble_2
      transport_1     transport_2
      go_stack        go_unstack



          AUTOMATON 1                              AUTOMATON 2
  invokes these foreign predicates       invokes these foreign predicates
                |                                      |
                |                                      |
   -------------------------------        -------------------------------
   |         |         |         |        |         |         |         |
   |         |         |         |        |         |         |         |
MASTER    AUT. 2    AUT. 3   BLACKBOARD MASTER    AUT. 1    AUT. 3   BLACKBOARD
adjacent            solicit  move_box   adjacent            solicit  move_box
                             add_box                                 relocate



          AUTOMATON 3                            BLACKBOARD
  invokes no foreign predicates                 invokes only
                                                     |
                                                     |
                                                   MASTER
                                                   box_goal
```

HIERARCHICAL STRUCTURE OF THE KNOWLEDGE BASES

```
                          MASTER KNOWLEDGE BASE
                                  |
       ---------------------------------------------------------
       |                  |              |                    |
    (facts)          (utilites)     (built_in)              PLAN
   SUCCESSOR          PLACE          UNEQUAL                  |
   PREDECESSOR        ADJACENT                                |
   BOX_GOAL                                                   |
   NEXT                                                       |
           --------------------------------------------------------
           |                   |               |                  |
 CURRENT_INTERMEDIATE_STATE    |     RETURN_AUTOMATA_HOME         |
  INTERMEDIATE_STATE           |                                  |
  COMPARE                      |                                  |
  DIFFERENT                    |                                  |
              ASSIGN_DUTY_TO_AUTOMATON_1     ASSIGN_DUTY_TO_AUTOMATON_2
                               |                                  |
            ----------------------------        ------------------------
            |                          |         |                     |
   PARTIAL_STATE_REMAINS      TRANSPORT_REQUIRED          STACK_REMAINS
     FIND_PARTIAL_STACK         FIND_READY_BOX              FIND_STACK
       PARTIAL_STACK              FREE_BOX                    STACK_EXISTS
         PARTIALLY-BUILD        CARRY
/*************************************************************************************/

 BLACKBOARD KNOWLEDGE BASE                                AUTOMATON 3
            |                                                  |
 ----------------------                     ----------------------------
 |                     |                     |              |              |
(facts)     (actions)        (r/a)       (facts)       (actions)        (r/a)
 LOCATION     RELOCATE        RETRACT      PLACE_3       SOLICIT          RETRACT_
              ADD_BOX         LOCATION     PLACE_GOAL_3  MOVE_3             PLACE_3
              RESERVE         ASSERTA_                                    ASSERTA.
              MOVE_BOX          LOCATION                                    PLACE_3
                ADJUST_STATUS
                ADJUST_LOCATION
/*************************************************************************************/

               AUTOMATON 1                                    AUTOMATON 2
                   |                                               |
     ------------------------------               ----------------------------
     |                  |               |             |             |                |
(facts)            (r/a)          (actions)       (facts)       (r/a)          (actions)
PLACE_1            RETRACT          MOVE_1          PLACE_2      RETRACT          MOVE_2
PLACE_GOAL_1         PLACE_1        CARRY_1         PLACE_         PLACE_2        CARRY_2
                   ASSERTA_         AMBLE_1           GOAL_2     ASSERTA_         AMBLE_2
                     PLACE_2        TRANSPORT_1                    PLACE_2        TRANSPORT_2
                                    RETURN_1                                      RETURN_2
                                    ROUTE_1                                       ROUTE_2
                                    GO_STACK                                      GO_UNSTACK
```

INTERFACE NODE

Master Node Predicates -

```
  mode( unequal, in, in).

  mode( successor, in, in_out).
  mode( predecessor, in, in_out).
  mode( box_goal, in_out, in_out, in_out, in_out, in_out).
  mode( next, in, in_out).

  mode( plan, in, in).
  mode( current_intermediate_state, out).
  mode( assign_duty_to_automaton_1, in, out).
  mode( assign_duty_to_automaton_2, in, out).
  mode( return_automata_home).

  mode( intermediate_state, in).
  mode( compare, in).
  mode( different, in, in, in, in, in, in, in, in).

  mode( transport_required, in).
  mode( find_ready_box, in, out, out, out, out, out, out).
  mode( free_box, in, out, out).
  mode( which_way, in, in, out).
  mode( carry, in, in, in, in, in, in, in).

  mode( partial_stack_remains).
  mode( find_partial_stack, in, out, out, out, out)..
  mode( have_next_box, in, out, in).
  mode( partial_stack, in, out).
  mode( partially_built, in, in, in, in, out).

  mode( stack_remains).
  mode( find_stack, in, out, out, out, out, out, out).
  mode( stack_exists, in, out, out, out)

  mode( place, in_out, in).
  mode( adjacent, in, in_out).
```

Blackboard Node Predicates -

```
  mode( location, in_out, in_out, in_out, in_out, in_out).
  mode( relocate, in, in, in, in).
  mode( move_box, in, in, in).
  mode( adjust_status, in, in, in).
  mode( adjust_location, in, in, in).
  mode( add_box, in, in, in).
  mode( reserve, in, in).
  mode( retract_location, in, in, in, in, in).
  mode( asserta_location, in, in, in, in, in).
```

INTERFACE NODE -

Automaton_1 Predicates -

```
mode( place_1, in-out).
mode( place_goal_1, out).
mode( move_1, in, in).
mode( carry_1, in, in, in, in, in, in, in).
mode( amble_1, in, in, in, in).
mode( transport_1, in, in, in).
mode( go_stack( in, in, in, in, in, in).
mode( return_1, out).
mode( route_1, in, in).
mode( retract_place_1, in).
mode( asserta_place_1, in).
```

Automaton_2 Predicates -

```
mode( place_2, in-out).
mode( place_goal_2, out).
mode( move_2, in, in).
mode( carry_2, in, in, in, in, in, in, in).
mode( amble_2, in, in, in, in).
mode( transport_2, in, in, in).
mode( go_unstack, in, in, in, in, in, in, in).
mode( return_2, out).
mode( route_2, in, in).
mode( retract_place_2, in).
mode( asserta_place_2, in).
```

Automaton_3 Predicates

```
mode( place_3, in-out).
mode( place_goal_3, out).
mode( move_3, in, in).
mode( solicit, in, out).
mode( retract_place_3, in).
mode( asserta_place_3, in).
```

INITIAL STATE

```
 ------------------------------------------------------
|                          |                           |
|   A1    A2    A3         |                           |
|                                      -----            |
|                                     | b |             |
|            -----          |          -----            |
|           | a |           |         | c |             |
|            -----        1 | 2        -----            |
 ----------      ------------------------     ----------
|                        4 | 3                         |
|                          |                           |
|            -----         |          -----            |
|           | d |                     | f |             |
|            -----                     -----            |
|           | e |          |          | g |             |
|            -----         |          -----            |
 ------------------------------------------------------
```

....

GOAL STATE

```
 ------------------------------------------------------
|                          |          -----            |
|                          |         | e |             |
|   A1    A2    A3                    -----            |
|                                     | f |             |
|                          |          -----            |
|                          |         | g |             |
|                        1 | 2        -----            |
 ----------      ------------------------     ----------
|            -----       4 | 3                         |
|           | a |          |                           |
|            -----         |                           |
|           | b |                                       |
|            -----                     -----            |
|           | c |          |         | d |             |
|            -----         |          -----            |
 ------------------------------------------------------
```

```
    MASTER NODE KNOWLEDGE BASE -

****************************  FACTS  ******************************/

   successor( r1, r2).
   successor( r2, r3).
   successor( r3, r4).
   successor( r4, r1).

   predecessor( r1, r4).
   predecessor( r4, r3).
   predecessor( r3, r2).
   predecessor( r2, r1).

   box_goal( r2, clear, e, f    , stacked).
   box_goal( r2, e    , f, g    , stacked).
   box_goal( r2, f    , g, floor, stacked).
   box_goal( r3, clear, d, floor, stacked).
   box_goal( r4, clear, a, b    , stacked).
   box_goal( r4, a    , b, c    , stacked).
   box_goal( r4, b    , c, floor, stacked).

   next(a,b).
   next(b,c).
   next(c,d).
   next(d,e).
   next(e,f).
   next(f,g).
   next(g,none).
```

MASTER NODE KNOWLEDGE BASE -

```
 *************************  MASTER PLAN  ****************************/

     plan( available, available) :-
          current_intermediate_state( Synch3),
          assign_duty_to_automaton_1( Synch3, Synch1),
          assign_duty_to_automaton_2( Synch3, Synch2),
          plan( Synch1, Synch2).
     plan :-
          return_automata_home.

/********************  CURRENT INTERMEDIATE STATE  *****************/

     current_intermediate_state( available) :-
          intermediate_state(a).

          intermediate_state( none).
          intermediate_state( X) :-
               compare( X), next( X, Y),
               intermediate_state( Y).

               compare( X) :-
                    location(R1, Above, X, Below, Status1),
                    box_goal(R2, Over, X, Under, Status2),
                    different(R1,R2,Above,Over,Below,Under,Status1,Status2).

                    different(R1,R2,Above,Over,Below,Under,Status1,Status2) :-
                        unequal( R1, R2).
                    different(R1,R2,Above,Over,Below,Under,Status1,Status2) :-
                        unequal( Above, Over).
                    different(R1,R2,Above,Over,Below,Under,Status1,Status2) :-
                         unequal( Below, Under).
                    different(R1,R2,Above,Over,Below,Under,Status1,Status2) :-
                         unequal( Status1, Status2).

 ***************              TRANSPORTATION              *******************/
/*************** THIS CODE IS USED BY AUTOMATA 1 AND 2    *******************/

     transport_required( A) :-
          place( R, A),
          find_ready_box( R, Rmid, Ri, Rf, Box, Code),
          reserve( R, Box),
          carry( A, R, Rmid, Ri, Rf, Box, Code).

          place( R, a1) :-
               place_1( R).
          place( R, a2) :-
               place_2( R).
          place( R, a3) :-
               place_3( R).
```

Master Node Knowledge Base -

```
   find_ready_box( R, R, R, Rf, Box, here) :-
        free_box( R, Z, Box),
        which_way( R, Z, Rf).
   find_ready_box( R, R, Ri, Rf, Box, adjacent) :-
        adjacent( R, Ri),
        free_box( Ri, Z, Box),
        which_way( Ri, Z, Rf).
   find_ready_box( R, Rm, Ri, Rf, Box, diagonal) :-
        successor( R, Rm), successor( Rm, Ri),
        free_box( Ri, Z, Box),
        which_way( Ri, Z, Rf).


        free_box( R, Z, B) :-
             location( R, clear, B, floor, unstacked),
             box_goal( Z, Above, B, Under, stacked),
             unequal( R, Z).


        adjacent( R1, R2) :-
             successor( R1, R2).
        adjacent( R1, R2) :-
             predecessor( R1, R2).


        which_way( R, Z, Z) :-
             successor( R, Z).
        which_way( R, Z, Z) :-
             predecessor( R, Z).
        which_way( R, Z, F) :-
             successor( R, F)
             successor( F, Z).


   carry( a1, R, Rmid, Ri, Rf, Box, Code) :-
        carry_1( R, Rmid, Ri, Rf, Box, Code).
   carry( a2, R, Rmid, Ri, Rf, Box, Code) :-
        carry_2( R, Rmid, Ri, Rf, Box, Code).
```

MASTER NODE KNOWLEDGE BASES -

```
/*********************** ASSIGN AUTOMATON-1 DUTY *********************/

    assign_duty_to_automaton_1( S, available):-
         partial_stack_remains.
    assign_duty_to_automaton_1( S, available) :-
         transport_required(a1).
    assign_duty_to_automaton_1( S, available) :-
         return_1.


    partial_stack_remains :-
         place( R, a1),
         find_partial_stack( R, Rm, Rs, Top, Code),
         have_next_box( Rs, New_top, Top),
         go_stack( R, Rm, Rs, New_top, Top, Code).


         find_partial_stack( R, R, R, Top, here) :-
              partial_stack( R, Top).
         find_partial_stack( R, R, Rs, Top, adjacent) :-
              adjacent( R, Rs),
              partial_stack( Rs, Top).
         find_partial_stack( R, Rm, Rs, Top, diagonal) :-
              successor( R, Rm), successor( Rm, Rs),
              partial_stack( Rs, Top).


              partial_stack( R, Top) :-
                   box_goal( R, Above, Box, floor, stacked),
                   unequal( Above, clear),
                   location ( R, Over, Box, floor, stacked),
                   partially_built( R, Over, Box, floor, stacked, Top).


                   partially_built( R, clear, Box, Under, stacked, Box).
                   partially_built( R, Over, Box, Under, stacked, Top) :-
                        box_goal( R, Above, Over, Box, stacked).
                        unequal( Above, clear),
                        location( R, Up, Over, Box, stacked),
                        partially_built( R, Up, Over, Box, stacked, Top).


         have_next_box( R, New_top, Top) :-
              box_goal( R, New_top, Top, Bottom, stacked),
              location( R, clear, New_top, floor, unstacked).
```

MASTER NODE KNOWLEDGE BASE -

```
/*****************  ASSIGN AUTOMATON-2 DUTY  ************************/

     assign_duty_to_automaton_2( S, available) :-
          stack_remains.
     assign_duty_to_automaton_2( S, available) :-
          transport_required(a2).
     assign_duty_to_automaton_2( S, available) :-
          return_2.


          stack_remains:-
               place( R, a2),
               find_stack( R, Rm, Rs, Top, Support, Under, Code),
               go_unstack( R, Rm, Rs, Top, Support, Under, Code).


               find_stack( R, R, R, Top, Support, Under) :-
                    stack_exists( R, Top, Support, Under).
               find_stack( R, R, Rs, Top, Support, Under) :-
                    adjacent( R, Rs),
                    stack_exists( Rs, Top, Support, Under).
               find_stack( R, Rm, Rs, Top, Support) :-
                    successor( R, Rm), successor( Rm, Rs),
                    stack_exists( Rs, Top, Support, Under).

                    stack_exists( Room, X, Support, Under) :-
                         location( Room, clear, X, Support, unstacked),
                         unequal( Support, floor),
                         location( Room, X, Support, Under, Status).

/**********************  RETURN AUTOMATA HOME  ***************************/

     return_automata_home :-
          return_1,
          return_2.
```

BLACKBOARD NODE KNOWLEDGE BASE (CHANGEABLE FACTS) -

```
location( r1, clear, a, floor, unstacked).
location( r2, clear, b, c    , unstacked).
location( r2, b    , c, floor, unstacked).
location( r3, clear, f, g    , unstacked).
location( r3, f    , g, floor, unstacked).
location( r4, clear, d, e    , unstacked).
location( r4, d    , e, floor, unstacked).


relocate( R, Box, Support, X) :-
     retract_location( R, clear, Box, Support, unstacked),
     asserta_location( R, clear, Box, floor  , unstacked),
     retract_location( R, Box  , Support, X  , unstacked),
     asserta_location( R, clear, Support, X  , unstacked).


move_box( R1, R2, Box) :-
     box_goal( R2, clear, Box, floor, stacked),
     adjust_status( R1, R2, Box).
move_box( R1, R2, Box) :-
     adjust_location( R1, R2, Box).


     adjust_status( R1, R2, Box) :-
          retract_location( R1, clear, Box, floor, used),
          asserta_location( R2, clear, Box, floor, stacked).


     adjust_location( R1, R2, Box) :-
          retract_location( R1, clear, Box, floor, used),
          asserta_location( R2, clear, Box, floor, unstacked).


add_box( R, New, Old) :-
     retract_location( R, clear, Old, Below, stacked),
     asserta_location( R, New,   Old, Below, stacked),
     retract_location( R, clear, New, floor, unstacked),
     asserta_location( R, clear, New, Old,   stacked).


reserve( R, Box) :-
     retract_location( R, clear, Box, floor, unstacked),
     asserta_location( R, clear, Box, floor, used).
```

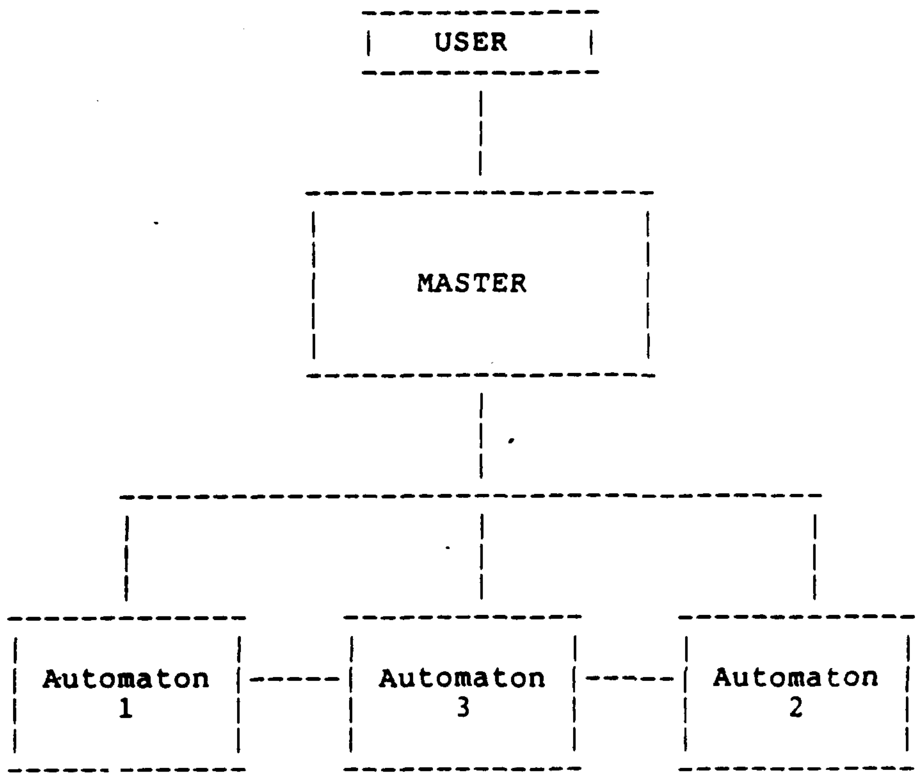
AUTOMATON 1 KNOWLEDGE BASE -

```
place_1( r1).

place_goal_1(r1).

move_1( R1, R2) :-
     retract_place_1( R1),
     asserta_place_1( R2).

carry_1( R1, R2, R3, Rf, Box, Code) :-
     amble_1( R1, R2, R3, Code),
     transport_1( R3, Rf, Box).

amble_1( R1, R2, R3, here).
amble_1( R1, R2, R3, adjacent) :-
     solicit( R3, Rr),
     move_1( R1, Rr).
amble_1( R1, R2, R3, diagonal) :-
     solicit( R2, Rr1),
     move_1( R1, Rr1),
     solicit( R3, Rr2),
     move_1( Rr1, Rr2).

transport_1( Ri, Rf, Box) :-
     solicit( Rf, Rr),
     move_1( Ri, Rr),
     move_box( Ri, Rr, Box).

go_stack( R1, R2, R3, New, Old, here) :-
     add_box( R3, New, Old).
go_stack( R1, R2, R3, New, Old, adjacent) :-
     solicit( R3, Rr),
     move_1( R1, Rr),
     add_box( Rr, New, Old).
go_stack( R1, R2, R3, New, Old, diagonal) :-
     solicit( R2, Rr1),
     move_1( R1, Rr1),
     solicit( R3, Rr2),
     move_1( Rr1, Rr2),
     add_box( Rr2, New, Old).
```

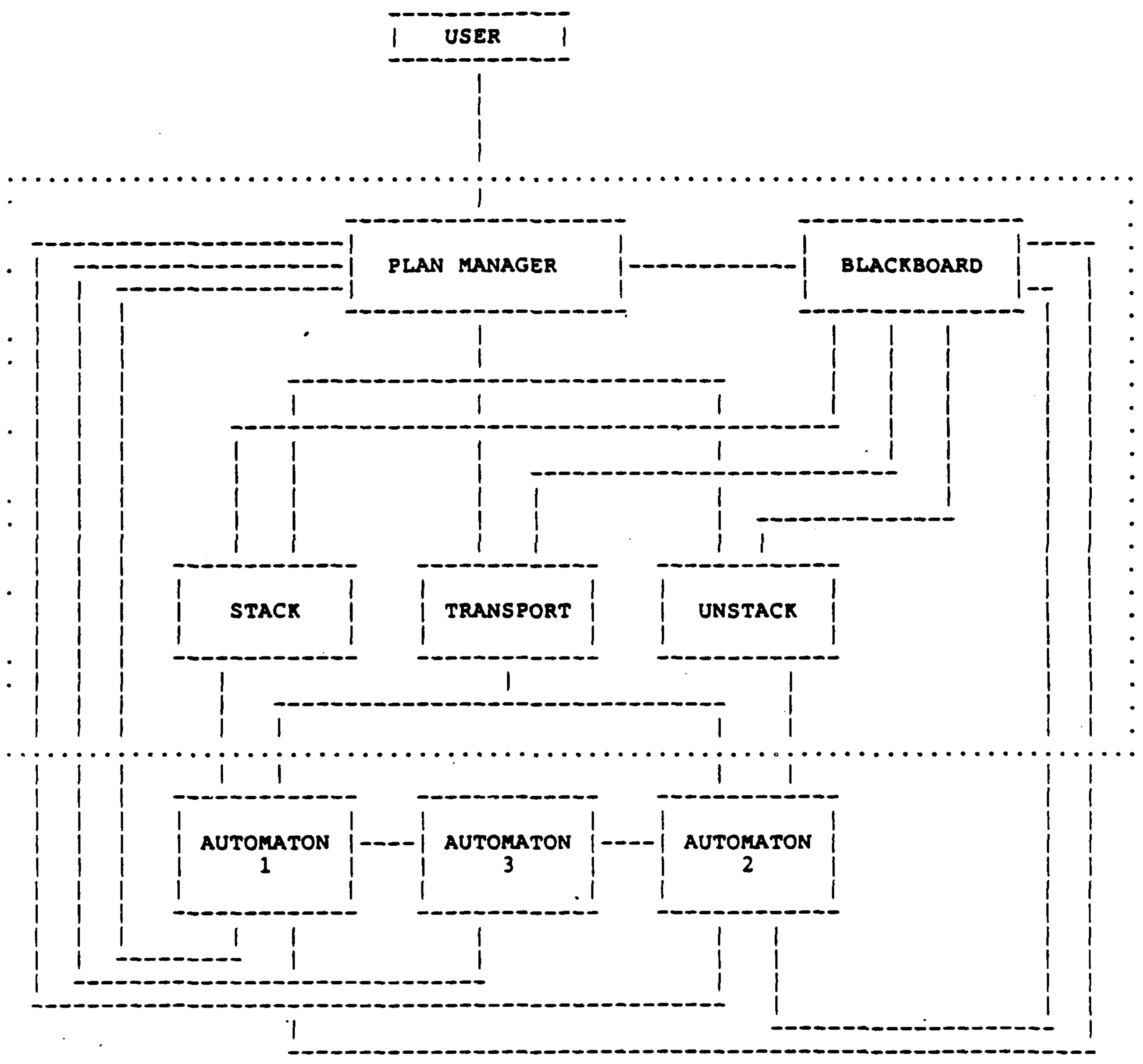
```
return_1 :-
     place_1( R),
     place_goal_1( Rg),
     route_1( R, Rg).

     route_1( R, R).
     route_1( R, Rg) :-
          adjacent( R, Rg),
          solicit( Rg, Rf),
          move_1( R, Rf).
     route_1( R, Rg) :-
          adjacent( R, Ra),
          solicit( Ra, Rr1),
          move_1( R, Rr1),
          solicit( Rg, Rr2),
          move_1( Rr1, Rr2).

/******************************************************************************/


   AUTOMATON 2 KNOWLEDGE BASE -

place_2( r1).

place_goal_2(r1).

move_2( R1, R2) :-
     retract_place_2( R1),
     asserta_place_2( R2).

carry_2( R1, R2, R3, Rf, Box, Code) :-
     amble_2( R1, R2, R3, Code),
     transport( R3, Rf, Box).

amble_2( R1, R2, R3, here).
amble_2( R1, R2, R3, adjacent) :-
     solicit( R3, Rr),
     move_2( R1, Rr).
amble_2( R1, R2, R3, diagonal) :-
     solicit( R2, Rr1),
     move_2( R1, Rr1),
     solicit( R3, Rr2),
     move_2( Rr1, Rr2).


transport_2( Ri, Rf, Box) :-
     solicit( Rf, Rr),
     move_2( Ri, Rr),
     move_box( Ri, Rr, Box).
```

```
go_unstack( R, Rm, Rs, Top, Support, Under, here) :-
     relocate( Rs, Top, Support, Under).
go_unstack( R, Rm, Rs, Top, Support, Under, adjacent) :-
     solicit( Rs, Rr),
     move_2( R, Rr),
     relocate( Rr, Top, Support, Under).
go_unstack( R, Rm, Rs, Top, Support, Under, diagonal) :-
     solicit( Rm, Rr1),
     move_2( R, Rr1),
     solicit( Rs, Rr2),
     move_2( Rr1, Rr2),
     relocate( Rr1, Top, Support, Under).


return_2 :-
     place_2( R),
     place_goal_2( Rg).
     route_2( R, Rg).

     route_2( R, R).
     route_2( R, Rg) :-
          adjacent( R, Rg),
          solicit( Rg, Rr),
          move_2( R, Rr).
     route_2( R, Rg) :-
          adjacent( R, Ra),
          solicit( Ra, Rr1),
          move_2( R, Rr1),
          solicit( Rg, Rr2),
          move_2( Rr1, Rr2).
/*************************************************************************************/


  AUTOMATON 3 KNOWLEDGE BASE -

place_3( r1).

place_goal_3(r1).

move_3( R1, R2) :-
     retract_place_3( R1),
     asserta_place_3( R2).

solicit( Rc, Rc) :-
     place_3( R),
     move_3( R, Rc).
```

PHYSICAL PKS GRAPH



CONCEPTUAL PKS GRAPH



SAMPLE EXECUTION

```
T: x n/m means the robot transport box x from room n to room m
S: x means robot_1 stacks box x
U: x means robot_2 unstacks box x
M: n/m means the robot moves from room n to room m
-  means no action is taken by the robot
```

```
ITERATION          ROBOT ACTIONS              HOUSE CONFIGURATION
              R1         R2         R3

   0          -          -          -       r1 r2 r3    |
                                                        |      b
                                               a       1|2     c
                                            ------------------------
                                                       4|3
                                               d        |      f
                                               e        |      g


   1          T: a 1/4   M: 1/2     M: 1/2              |           r2
                         U: b       M: 2/3              |
                                    M: 3/4             1|2   b  c
                                            ------------------------
                                            r1 r3      4|3
                                                d       |       f
                                             a  e       |       g


   2          M: 4/1     M: 2/3     M: 4/3              |
              M: 1/2     U: f       M: 3/4              |
              T: b 2/3              M: 4/1             1|2      c
                                    M: 1/2  ------------------------
                                    M: 2/3             4|3  r1 r2 r3
                                                d       |
                                             a  e       |   b  f  g


   3          T: b 3/4   M: 3/4     M: 3/4              |
                         U: d                           |
                                                       1|2      c
                                            ------------------------
                                            r1 r2 r3   4|3
                                                        |
                                            a  b  e  d  |       f  g
```

```
4        T: d 4/3   T: e 4/1   M: 4/3    r2 r3       |
                               M: 3/4                |
                               M: 4/1          e    1|2      c
                                         -----------------------
                                                    4|3         r1
                                                     |
                                           a   b     |  d  f  g

5        T: f 3/2   T: e 1/2   M: 1/2                |   r1 r2 r3
                                                     |
                                                    1|2  c  e  f
                                         -----------------------
                                                    4|3
                                                     |
                                           a   b     |     d  g

6        T: c 2/3   M: 2/3     M: 2/3                |      r2 r3
                    T: g 3/2   M: 3/2                |
                                                    1|2  e  f  g
                                         -----------------------
                                                    4|3        r1
                                                     |
                                           a   b     |     c  d

7        T: c 3/4   M: 2/1     M: 2/3    r2          |
                               M: 3/4                |
                                                    1|2  e  f  g
                                         -----------------------
                                         r1  r3     4|3
                                                     |
                                          a  b  c    |    d

8        S: b       -          -         r2          |
                                                     |
                                                    1|2  e  f  g
                                         -----------------------
                                         r1  r3     4|3
                                              b      |
                                          a   c      |    d
```

```
9       S: a       -       -         r2          |
                                                 |
                                                1|2  e  f  g
                                     ---------------------------
                                     r1 r3    a 4|3
                                              b  |
                                              c  |       d

10      M: 4/1     -       M: 4/1    r2          |        r1 r3
        M: 1/2             M: 1/2                |    f
        S: f                                    1|2   g  e
                                     ---------------------------
                                           a    4|3
                                           b     |
                                           c     |       d

11      S: e       -       -         r2          |    e  r1 r3
                                                 |    f
                                                1|2   g
                                     ---------------------------
                                           a    4|3
                                           b     |
                                           c     |       d

12      M: 2/1     -       M: 2/1    r1 r2 r3    |    e
                                                 |    f
                                                1|2   g
                                     ---------------------------
                                           a    4|3
                                           b     |
                                           c     |       d
```

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 (C^3I) 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 C^3I 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 reliability/maintainability and compatibility.