123

PRISM
A Parallel Inference System for Problem Solving

Simon Kasif Madhur Kohli Jack Minker

Department of Computer Science
University of Maryland
College Park, MD 20742
(301) 454-4251
MINKER @ UMCP-CS

Abstract

A Parallel Inference System for Problem Solving (PRISM) has been developed at the University of Maryland. The system is designed to provide a general experimental tool for the construction of large artificial intelligence problem solvers.

We present some of the basic facilities for controlling parallelism and inference provided by the system. PRISM is based on the concept of logic programming with a separate control componenent. The control may either be explicitly specified by the user in his input or alternatively determined dynamically by the system, which takes advantage of the implicit parallelism in the logic of the algorithm. The design makes the underlying virtual architecture transparent to the user. The system supports both AND and OR parallelism.

1. Introduction and Overview

1.1. Introduction to Parallel Problem Solving

In general, problem solving systems have been designed to be executed on sequential machines (i.e. a single processor architecture). However, the complexity of many interesting problems, makes the sequential implementation of these problems infeasible in terms of speed and resource requirements. This implies that it is necessary to examine solutions to these problems in a distributed environment, in order to determine if these solutions will prove more feasible in terms of speed and resources, than those in a sequential environment. Furthermore, the investigation of distributed methods of problem solving is suggested by the structure of the problems themselves. Many interesting AI problems are NP-complete and require exponential time on a deterministic machine, whereas they can be solved in polynomial and sometimes linear time on a nondeterministic machine. A distributed system is necessary, to implement, nondeterministic solutions.

A large amount of work has been done on parallel architectures [Computer 1982a], [Computer 1982b] and algorithms for parallel architectures [Kung 1980]. Work has also been done on parallel languages and environments for parallel architectures for non-AI problems [Hewitt 1977], [Kahn 1977].

In the AI environment several researchers have suggested methods to parallelize certain types of problems, however, few of these schemes have actually been implemented on a distributed system. Kornfeld [1979], and Lieberman [1981] describe systems and languages which have been designed for parallel applications.

PRISM (a Parallel Inference System), which is an experimental tool for the development of distributed AI problem solvers, has been developed at the University of Maryland and has been implemented on ZMOB (Rieger [1980]). PRISM is based on logic programming (Kowalski [1979]).

1.2. Control in Logic Programs

In conventional programming systems the logic and control of an algorithm are combined making it difficult to separate or to modify control without affecting the logic. Logic as the specification language, is neutral with respect to control and specifies only the problem semantics. The method of how the problem is to be solved is external to the logic specification. It has been shown (Kowalski[1979], Pereira[1978], van Emden[1976]) that the complete separation of logic (the specification to be executed) and control (the order in which tasks are executed) allows a great amount of flexibility during execution, thus providing a natural parallel implementation of a program.

This is true since the inherent nondeterminism of logic programs can be exploited in many different directions during excution.

1. Top-down and bottom up execution of a program can be done in parallel.
2. At any time during execution more than one possible goal node (procedure) can be invoked.

3. Since the order of execution of atoms in a goal is usually not specified we can sometimes separate the goal into several independent subgoals to be solved.

4. Logic programs are distinguished from other applicative languages such as LISP due to the fact that more than one procedure can match a procedure call. This seeming disadvantage on a sequential machine becomes an advantage in a highly parallel environment since all or some matching procedures can be executed in parallel.

Thus, a primary issue in achieving a parallel system is developing an effective control specification that exploits parallelism. PRISM permits us to specify the problem independently of the control and allows us to experiment with alternative control possibilities for the same problem.

1.3. ZMOB and Parallel Problem Solving

PRISM has been implemented on ZMOB, which consists of a set of 256 Z80A microprocessors connected on a conveyor belt together with a host VAX-11/780 minicomputer. A description of ZMOB is given in the following section. A description of how parallel problem solving is achieved using ZMOB is described in Section 1.3.2.

1.3.1. ZMOB Description

The particular system to be used is ZMOB, a parallel multi-microprocessor system developed at the University of Maryland (Rieger[1980]). ZMOB is to consist of 256 Z80A microprocessors connected to a host computer (VAX 11/780) which is to communicate between machines via a high speed 48 bit wide, 257 stage shift register called the "Conveyor Belt" (Figure 1). The system is described in detail in Rieger[1980, 1981a, 1981b]. We shall briefly describe here only the communication features necessary to support PRISM.

The Z80A is a microprocessor capable of executing 400,000 instructions/second and has a 64K byte memory. Thus, the whole system is theoretically capable of executing 100 million instructions/second and has a memory capacity of 16 million bytes. Each processor is connected to the conveyor belt via a collection of high-speed 8-bit I/O registers and associated control circuitry, called the "Mail Stop". The registers are in charge of interrupt control, buffering and address control functions.

In general, the Conveyor Belt moves 257 bit patterns (bins) each 48 bits wide. Each processor can theoretically consume any bin that is currently at its mail stop, but it can send out information only in its own bin. The 48-bit message in the bin consists of four fields:

CONTROL	DATA	SOURCE	DESTINATION
8 bits	16 bits	12 bits	12 bits

Figure 1

The control bits allow the implementation of several communication strategies : Let (C X S D) be the content of a bin on the Conveyor Belt, then different control bits specify the following communication formats.

1. Direct addressing - The message X is sent to a processor whose physical address is D.
2. Pattern matching - Message X is sent to the first processor whose pattern (determined by Capability Code and Mask Registers in the Mail stop) matches D.
3. Send to all Processors - Message X is sent to all processors.
4. Send to a set of Processors - Message X is sent to all processors whose patterns match D.

Additionally, different settings of Control Registers in the Mail Stop allow the following :

5. Exclusive Source - This mode provides exclusive conversation between two processors and disables interrupts from other processors.
6. Readback - This mode allows an individual processor to intercept any of its own messages that went around the conveyor belt and was not consumed by any of the destination processors.

The following examples illustrate the utility of the above formats.

(3,5) Permits large blocks of data to be sent in a burst mode to all processors from the host computer. (e.g. to load kernel programs or data to all processors).

(2) Provides the ability to assign to each processor a relation. Logically the relation's name would be the pattern identifying this processor.

(4) Allows a very useful provision of clustering the system into independent sets of logically equivalent processors.

(6,4) Can be used to send a message to a set of processors and in case it was not consumed to activate a recovery routine.

1.3.2. ZMOB Parallel Problem Solving System

We find it useful to separate the static set of clauses representing the logic of a problem from the control which generates a search tree by applying these clauses to a goal clause. This distinction will be seen to be useful in experimenting with the control of a parallel logic programming system.

In particular we shall distinguish three separate portions of the system to which we dedicate microprocessors. These are :

(1) the problem solver (PS),

(2) the extensional database (EDB), the set of assertions, and

(3) the intensional database (IDB), the set of procedure clauses.

The PS administers the search space which consists of a tree of goal clauses. The root of the tree is the original goal, whereas successors of any clause C in the tree are resolvents obtained by resolving program clauses (EDB or IDB clauses) with an atom selected in C. Each leaf node in the tree is either the empty clause; or some indication that the respective branch of the search resulted in a failure; or an open goal clause not yet selected for expansion; or an active clause sent for expansion and not fully expanded.

To generate the successors of an open clause C the PS has to select an atom and send it to that part of the system that handles unification of the atom with procedure heads in the EDB or the IDB. If the tree is distributed among several microprocessors, several atoms of different clauses can be selected simultaneously for expansion. Atoms sent to the EDB/IDB for solving cause the return of information necessary for generating all successor clauses of C.

While waiting for the information the PS in each machine can treat other open clauses in the same way, so that the subproblems of several open nodes in the same machine can be solved in parallel and independent of each other.

A second part of the system is in charge of the assertions and procedure clauses. This is subdivided further into the extensional database (EDB) consisting of all function-free ground assertions, and the intensional database (IDB) that constitute the procedure clauses and non-EDB assertions (i.e. those that contain variables and/or functions).

This distinction was drawn primarily for two reasons. First, the EDB and IDB can use different unification algorithms. In particular, when matching an atom against an EDB entry, it is not necessary to invoke the occur check which is used to determine if a term substituted for a variable contains the variable. Second, there are many applications where the sizes of the EDB and IDB differ considerably. If the set of clauses is used as a database, the number of IDB clauses is likely to be relatively small, whereas there are many EDB clauses corresponding to a relational database in the usual sense. If, on the other hand, the set of clauses represents a program, there are usually few EDB-clauses, but the IDB clauses are generally numerous. In some instances we may wish not to make a distinction between EDB/IDB clauses. We want the system to be sufficiently flexible to be able to react in different ways.

In addition to predicates contained in the EDB and IDB, systems usually contain predefined predicates, e.g. arithmetic predicates or equality predicates. Such atoms are evaluated directly in the PS where encountered and are not sent to the EDB or IDB for evaluation.

Problems can arise if predicates are permitted to have side effects. One such side effect would be the ability to modify the database as, for example, contained in the PROLOG primitives ASSERT and RETRACT. As the system pursues different branches of the search tree in parallel, there is no way of determining the exact point at which the side effects were executed. Since side effects in one branch can influence other branches of the search tree, this fact would render the overall behaviour of the system intolerably unpredictable.

For that reason in this first design, we do not allow any predicates with side effects in a goal clause (and hence they are not permitted in a procedure clause) thus restricting the system to pure logic. This means that such

extralogical tricks as modifications to the database to simulate global variables are not permitted. Of course, the system must provide features other than the ability to solve goal clauses, including such capabilities as adding, deleting, and modifying clauses. Such capabilities are provided at the top level only, so that there is no modification of a knowledge base during problem solving.

The separation of the problem solving system into the problem solver, EDB search, IDB search, IDB monitor and VAX has isolated the functions in the system and has placed them on separate processors. The main link between the processors is the conveyor belt and message passing. There is an uniform message passing facility between machines.

2. Control Issues

2.1. Problem Solving Process

The problem solving process may be outlined as follows:

(1) the problem to be solved is expressed as a conjunction of goals, each of which is a subproblem to be solved;

(2) one or more subgoals may be selected to be solved;

(3) a subgoal is solved if it is matched by some assertion, or it is matched by a procedure which consists of a set of subgoals which can be solved.

The repeated execution of steps (1), (2) and (3) results in a top-down execution of a problem. One can specify a problem solving process which permits bottom-up, middle-out, top-down, or any combination of these reasoning methods. The initial PRISM system is restricted to top-down reasoning (backward chaining from the goal).

2.2. Goal tree and Control Issues

A goal tree is generated in the problem solving process. The goal tree is formed initially by placing the conjunction of goals to be solved in the root node of the tree. In general the tree consists of a set of nodes, where each node consists of a set of goals. Now, given a node, there are several ways in which the node may be executed. One or more goals may be selected to be executed asynchronously. This possibility provides for user control of parallel execution. Subgoals in a node may be characterized to be dependent or independent of one another. A subgoal is dependent if its execution must await the successful execution of another subgoal in the same node. It is independent otherwise. An acyclic partial order expresses such a relationship among subgoals. At any stage of the execution of a node, all those subgoals which are independent may be executed asynchronously. However, goals which are candidates for simultaneous execution must be treated specially if they share unbound variables.

A goal selected for execution must be matched against assertion or procedure heads. There may be several assertion/procedure heads which match the given goal. Any procedure head which matches a goal can potentially lead to the solving of that goal, independent of any other procedure head that may

also match the goal. All matching procedure heads are therefore candidates for asynchronous execution. Furthermore, the user may wish to specify a partial order of execution of procedure bodies, in a similar manner to the partial ordering on subgoals within a node. Thus there is the possibility of specifying that certain alternatives need be explored only if other alternatives have failed.

An assertion or a procedure that matches a goal in a node causes a new node to be generated as a successor node to the node that contains the goal. The new node consists of all goals in the parent node where the selected goal is deleted and replaced by the body associated with the procedure head and the matching substitution is applied to the new node. In case of a matching assertion, the body is empty and the new goal node has one less problem to be solved. When an empty node is generated, the problem has been solved.

Executing a problem as outlined above leads to the generation of many nodes, each node of which can be in a partial state of execution. It is in a partial state when all assertion/procedure heads that match a subgoal have not been selected for execution. Thus, there is the option to select many nodes for asynchronous execution.

All possible asynchronous operations may be executed on autonomous machines.

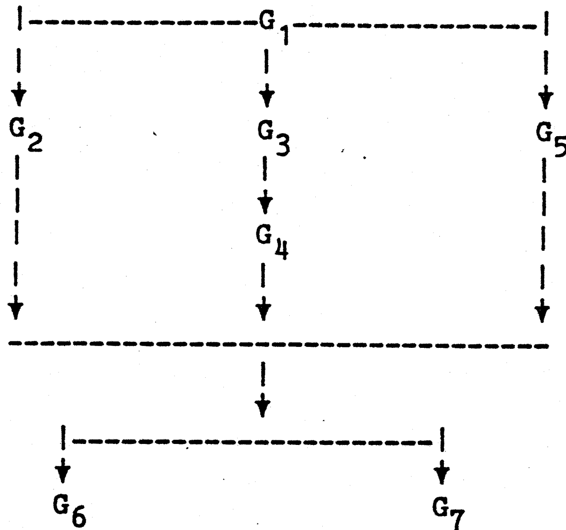
2.3. PRISM Control Facilities and Language

In the previous section we described the possibilities for parallelism in the control structure. Here we specify the support for controlling parallelism in PRISM. PRISM provides the ability to specify for every goal and procedure body a partial order for execution. This partial order expresses the dependencies among the subgoals within a goal (a procedure body may be considered to be a goal). or within alternative procedures for solving the same goal.

The partial order on subgoals in a goal are specified by a notation as explained in the following example.

$$P \leftarrow (G_1,[G_2,(G_3,G_4),G_5],[G_6,G_7]).$$

The procedure head is on the left hand side of the arrow, while the body is the right hand side. The body consists of a set of goals, separated by commas and formed into groups by properly nested pairs of parentheses and brackets. All groups of goals enclosed by parentheses, must be executed in a left-to-right sequence, i.e., the leftmost group in the sequence must be executed and solved before the remaining groups. Groups of goals enclosed in brackets, may be executed independently of other groups in the same set of brackets, i.e., all groups in the bracket may be executed asynchronously. The partial order induced by the above notation is:



The groups formed by G_1; $[G_2,(G_3,G_4),G_5]$; $[G_6,G_7]$, must be executed from left to right since they are enclosed within parentheses, i.e. G_1 must be executed and completed before any other group. Once G_1 is completed, the groups G_2; (G_3,G_4); G_5 may be executed asynchronously since they are enclosed by a bracket. However, since G_3; G_4 are enclosed in parentheses, G_3 must be executed and completed before G_4 is initiated. The next group, $[G_6,G_7]$ cannot be initiated until all groups to its left have been completed, i.e., goals G_1,G_2,G_3,G_4 and G_5. The goals G_6; G_7 may be executed asynchronously.

In the case where no parenthesis or brackets are specified, PRISM assumes a default ordering. This default is user specifiable to be either left-to-right or asynchronous.

The user has the ability to specify a partial-like ordering of procedures with the same procedure name. The user is provided with a notation which permits assigning precedences to procedures. The semantics of the ordering is different than for the ordering of goals. The ordering specified on the procedures is a recommendation on the likelihood of success when the procedure is executed. However, these recommendations may be ignored by the problem solving system which could change the recommended ordering or perform them in parallel. Also provided is a capability to invoke a procedure only if other procedures have been executed and failed. The following notation is used as an example:

1: P <-- G_1,G_2

1: P <-- G_3

2: P <-- G_4,G_5,G_6

*3: P <-- G_7

4: P <-- G_8,G_9

The integers represent the recommended order of execution. The asterisked integers represent a forced ordering. In the above example, the first two procedures (priority=1) may be executed simultaneously. The third procedure (priority=2) is less likely to succeed but may also be executed in parallel with the first two, or even before them if the problem solver so decides. However, the fourth procedure (priority=*3) cannot be executed unless the preceding procedures have been completely executed. A default ordering is provided by PRISM when the procedures are not numbered. The recommended ordering is the sequence in which the procedures were presented to the system.

At the present time, no user facilities are provided for node selection. However, the PRISM problem solver is supplied with several evaluation functions to permit automatic selection of nodes to be expanded.

3. The Problem Solving Machine (PSM)

3.1. The PSM Organization

3.1.1. The Role of the PSM

The Problem Solving Machine (PSM) is the core of the parallel problem solving system. At initiation time, a number of moblets (a moblet is a single ZMOB processing element) are designated as PSMs. The central task of the PSMs is to manage the search space. The complete separation of logic (the problem specification) and control (the strategy of solving the problem) allows a great degree of flexibility while executing the program. Not only can the search strategy be varied dynamically, but due to the inherently nondeterministic nature of logic programs, several mutually exclusive possibilities may be explored simultaneously. The PSMs permit this inherent parallelism to be exploited during the course of solving a problem.

Initially a goal, which represents the problem to be solved, is sent by the VAX to ZMOB and is read by some PSM. This PSM places the goal as the root of a proof tree. A goal is expanded by selecting an atom in the goal and replacing it by the body of a program clause that resolves with it. In this manner a new goal clause, which when solved, solves the original problem, is produced. When an atom is expanded, there may be several program clauses which resolve with it. These represent alternative ways to solve the same problem. These alternative subgoals lead to a branching in the search tree (OR branches).

Thus at any given instant in the problem solution process the search space administered by each PSM consists of a tree of goal clauses. The rest of the search tree is the original goal with which the PSM was initiated. The successor of any clause in this tree is the resolvent obtained by resolving program clauses with some atom in the parent clause. Each leaf node in the tree can be in one of four states:

- the node represents the empty clause
- the node represents a failure node

- the node represents an open goal clause not yet selected for expansion.
- the node represents an active goal clause selected for expansion, but not yet fully expanded.

At any stage the PSM must select an open clause from the search tree, and then select one or more atoms from this clause. This selected atom is then sent to an IDB and/or an EDB for expansion. While the IDB and/or EDB are working on this atom, the PSM can transfer its attention to other nodes in the search tree. An atom sent to the IDB may unify with one or more procedure heads, and all the corresponding bodies are sent back to the PSM which initiated the search, either one at a time or all at once. In the case that more than one procedure body is returned for a given atom, several mutually exclusive subgoal clauses are generated. These mutually exclusive goals can then be solved independently in separate machines.

Thus each PSM has the capability to dynamically send a goal to another PSM machine, if one is available. As with the goal transmitted by the VAX to a PSM, the goal transmitted from one PSM to another becomes a root of a goal tree in the new PSM whose parent is the sending PSM. Each PSM can independently develop and manage the subtrees of the search space generated by the goal node transmitted to the PSM. Each PSM is autonomous except for the knowledge of the parent-child relationship. When a goal assigned to a PSM is completely solved it transmits the solution or failure to its parent PSM. The parent of the PSM to which the original goal was transmitted is the Host (VAX) machine.

3.1.2. Conceptual View of the PSM

This section presents a conceptual view of the program that drives the PSM in terms of the subtasks that compose it and their functional specifications.

The program which drives each PSM is composed of several subtasks. Each subtask operates independently of all other subtasks. These processes do not communicate directly with each other, instead they change global data structures which then may affect another process. This independence makes it possible to consider each process in isolation. This isolation makes the implementation less error prone, and at the same time permits the single PSM process to be split across several machines if the need arises.

The operation of these processes is controlled by a scheduler process which, based on the current state of the global data structures, determines which subtask to invoke next. Thus each process once invoked is allowed to proceed until completion (except in certain special cases, which result in its preemption). Once this process completes, it returns to the scheduler which then applies a decision process to determine which subtask to invoke next. This can be represented by a recursive PROLOG program, of the form:

$$S \leftarrow D_i, P_i, S$$

where S is the scheduler and D_i is a decision process which succeeds if process P_i is to be invoked next, and fails otherwise.

There are six basic processes which compose the problem solver (aside from the scheduler). These are the initialization, input, selection, resolution, output and finalization processes.

The scheduler, once invoked with the initial goal, unconditionally invokes the initialization process which creates all global data structures required by the PSM processes and sets them to their initial values. The scheduler then repeatedly invokes the input, output, selection and resolution processes, by using its decision criteria. This continues until an answer is found or a termination signal is received. Once an answer (or all answers, as the case may be) is found, the finalization process is invoked. If all children PSMs of this PSM have completed already and returned their answers, this PSM transmits its answer to its parent. Otherwise, the finalization process creates a data structure which contains enough information to construct the answers when the children PSMs complete their tasks. Once this data structure is created the PSM is reinitialized and can accept queries.

In this manner PSMs are not kept idle in case they complete before their children do. This also allows a PSM to be its own ancestor if so desired. Thus a cyclic graph of parent-child dependencies may be constructed.

In addition to the six processes mentioned earlier, there are two low level processes which are totally independent of the scheduler and all other processes. These are the mailstop handlers. These processes are interrupt driven and are invoked whenever a message enters (leaves) the input (output) mailstop of the PSM. The input (output) mailstop handler merely places (removes) a message into (from) the input (output) queue and returns to the interrupted process.

The input process understands the message formats of all possible messages that can be received by the PSM. It selects a message from the input queue, decodes it and updates the appropriate global data structure with the information contained in the message.

The output process understands the message formats of all messages that can be sent by the PSM. When invoked with a certain message type, it uses information from the appropriate data structure, and formats this information into the correct message format. This message is then placed into the output queue, ready to be sent out.

The selection process directs the problem solving process by determining which clause, and which atom within the selected clause to operate on next. It is also responsible for the creation of new PSMs.

The resolution process receives the procedure bodies for an atom that has been matched by the IDB and/or EDB. It then inserts a new clause into the proof tree. This new clause consists of the clause from which the atom was selected, with the atom deleted and the procedure body attached in its place. The unifying substitution is then applied to the new clause.

3.2. Control in the PSM

3.2.1. Control Specification Support - Selection Process

The selection procedure determines the control strategy of the system. The user is permitted to specify certain guidelines to direct the selection process. The selection procedure has four main selection functions. These

are: node (clause) selection, atom selection, procedure selection and PSM creation.

Node selection is concerned with choosing a clause, from the search tree, from which an atom is to be slected. Any node which has not been fully expanded, is a candidate for selection. A fully expanded node consists of a clause whose selected atom has been expanded and all leaf nodes descended from the clause are either failure nodes or null clauses. A non-fully expanded node may be either an active or an open node. An active node is one from which one or more atoms have been selected for expansion, but which has not been fully expanded. An open node is one from which no atom has yet been selected for expansion.

Atom selection is concerned with selecting an atom, for expansion, from a selected node in the search tree. There are several system defined and user defined constraints that will affect atom selection.

As defined in section 2, the user has the ability to specify which atoms in a clause may be executed in parallel and which must be done in sequence, i.e. a partial order on the execution of the atoms. These user specified constraints limit the atoms which can be selected at any stage. Only those atoms which do not depend on any other atom or those for which the atoms they depend on have already been solved are candidates for selection.

In addition to these user-defined orderings, there are certain orderings implied by the structure of the node itself. There are two basic ways in which the contents of a clause dictate the ordering on atom selection. These are: dependent atoms, and special predicates.

Two or more atoms in a clause are said to be dependent when they share variables. In this case what is desired is the first (or all) binding(s) which cause the atoms to succeed. This can be accomplished either by processing the atoms in parallel and then intersecting the sets of bindings for the shared variables, or by finding a binding which satisfies one predicate and then substituting it in the others and determining if they succeed with that binding. This can be repeated until some binding succeeds or all are exhausted (nested loops method). In either method a special AND node has to be generated with the dependent atoms as its children and one of the above techniques applied. In this system the nested loops method will be adopted since space limitations make the set of values method infeasible.

The special predicates are a set of language supplied predicates whose semantics dictate that certain other predicates in the clause must be fully solved before these system defined predicates may be invoked, i.e., these predicates induce a partial ordering on the atoms in a clause. These predicates are: write, read, fail, / (the cut operator), not, and the evaluable predicates (e.g., arithmetic operations).

Once these user and system defined constraints have been satisfied, a set of atoms which are candidates for selection will remain. The atom selected from this set will be selected based on user or system supplied heuristics.

Procedure selection is concerned with choosing which procedure body should be given the highest priority when several bodies match an atom which was sent for expansion. This decision is made by the IDB and is not influenced by the PSM.

PSM selection, is concerned with the decision of when to initiate another PSM with a subproblem. Whenever a branching of the search tree is induced by either multiple alternate subproblems (OR-branches) or by independent conjunctive subproblems (AND-branches), this branch becomes a candidate for execution in another PSM machine. The actual process of determining when a new PSM is initiated is discussed in the following section.

3.2.2. PSM Creation

The decision of when to initiate another PSM with a subproblem is not an easy one. If the subproblem is too small, a large amount of overhead would be incurred to solve it. If the subproblem is too large, the parent PSM may complete before the child and remain idle until its children complete. In general it is extremely difficult if not impossible to determine how complex a subproblem is. Thus no attempt is made to determine the complexity of a subproblem, in the initial system. Instead a new PSM is initiated every time a branching of the search tree takes place, and there is a PSM available. At any given instant, there may be several active branches within a PSM, and thus several candidate nodes which may be sent to other machines. In this case all or only some of these nodes may be shipped out. This is determined by the user or by system supplied heuristics.

In order to reduce idle time, machines which have completed their alloted task are permitted to accept fresh queries, as follows. If no further processing can be done then either all possible answers for the goals this PSM was invoked with have been found, or all paths resulted in failure, or all paths local to this PSM have been fully explored and there are some children of this PSM which have not yet completed their work. In the cases where all answers have been found or all paths have failed, this information is transmitted to the parent of this PSM, and the PSM state is restored to one in which a new query can be accepted. In the case where all local paths have been explored and some chidren PSMs are still active, a data structure is constructed which contains enough information to reconstruct the complete answer from the information in this PSM and from the answers from the currently active children PSM. Once this data structure is constructed, the local proof tree is destroyed and the PSM state is restored to one in which a new query can be accepted.

In this manner PSMs are not kept idle in case they complete before their children do. This also allows a PSM to be its own ancestor if so desired. Thus a cyclic graph of parent-child dependencies may be constructed.

3.2.2.1. AND Parallelism

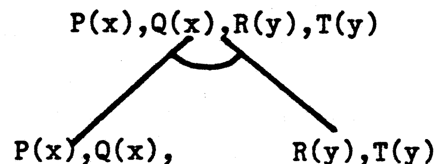
An AND-branch in the search tree can be one of two types. The first type of AND-branch results when there is a conjunction of atoms which do not share variables. This results in a node which has as its children two or more sets of atoms which do not share variables. We shall refer to such an AND-node as an AND_1-node. The second type of AND-branch results when there is a conjunction of atoms which do share variables (dependent atoms). This results in a node which has as its children two or more sets of atoms which do share variables. These children are ordered so that those atoms which bind variables are executed earlier than those atoms which use those variables. Such an

AND-node will be referred to as an AND_2-node.

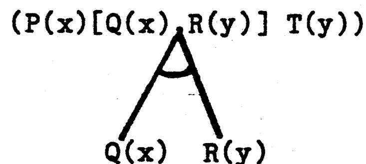
The children of an AND_2-node are currently always executed in the same machine since concurrent execution across machines requires an excessive amount of control and communication to synchronize the producers and consumers of the answers. Thus in the initial system the children of only AND_1-nodes are executed concurrently in separate machines.

We previously defined a clause to be an AND_1-Node if it could be split into two or more sets of atoms that do not share variables.

This definition must be revised when dealing with ordered clauses. For example let <- P(x),Q(x),R(y),T(y) be a clause. According to the definition above we create an AND_1-Node as follows:



Now let <-(P(x),[Q(x),R(y)],T(y)) be an ordered clause. The execution sequence imposed by the order in the clause does not allow a similar split. Thus a split may be performed only on sets of the clause that may be executed in parallel. Therefore the clause above is represented as:



The split would be executed dynamically after P was solved.

3.2.2.2. OR Parallelism

An OR-branch is created in the search tree when there are several matching bodies for a selected atom. These several bodies are alternative ways of solving the selected subgoal and are thus independent. The existence of multiple bodies then results in the formation of an OR-node with each of these bodies as a child.

Since these children are independent of each other they may be executed in separate machines. However those bodies which should not be attempted until some other body fails are not scheduled for execution until the body it depends on has failed.

3.2.3. Handling Negation and the CUT(/) Operator

In addition to imposing an implicit ordering on the atoms within a node, the cut(/) operator and negation both require special treatment in concurrent systems.

3.2.3.1. The CUT(/) Operator

The cut operator is a means of achieving determinism in sequential logic programs. The execution of a cut in the body of a procedure results in all alternatives for the parent node of the node containing the cut to be discarded. However, in sequential execution all alternatives with higher priority than the one containing the cut have already been completely executed before the one containing the cut. Thus the semantics of the cut operator are unclear for concurrent execution, since the alternative containing the cut may be executed concurrently with, or even before alternatives with higher priority. This would lead to an incompatibility between sequential and concurrent execution of the same logic program.

In the interest of preserving compatibility between the concurrent and sequential execution of logic programs containing the cut operator, we have defined the concurrent cut as detailed below.

The presence of the cut operator causes an implicit ordering of atoms within the node containing the cut. The cut operator requires all atoms preceding it in the node to be processed before it. The invocation of this operator results in the following:

(a) all bindings that have been computed for variables in atoms preceding the cut will not be recomputed in the event of a failure of some atom succeeding the cut

(b) all alternative procedure bodies which have lower priority than the body containing the slash are discarded. However all higher priority bodies are still considered, i.e., if procedure P has 3 bodies and if the second (middle priority) body contains a cut then the third body will never be considered if the cut in the second is executed, but the first body will be unaffected.

3.2.3.2. Negation

The NOT meta-predicate defined in most sequential logic programming is an implementation of Negation-by-Failure [Clark 1978]. In this scheme, the negation of an atom is considered to hold if all attempts to prove the atom, fail. This is well defined in the case where all the arguments of the atom are constants. However this is not well defined when one or more of the arguments are unbound variables. This is because the atom could succeed with some particular bindings for the free variables, and fail for some other bindings. Thus the behaviour of the NOT meta-predicate can be anomalous in the case where all argumentds are not constants. The semantics of negation can be extended to handle atoms with variables as arguments by creating a set of bindings for which the atom fails and assuming the negation of the atom holds for precisely this set of bindings. This is how we define negation in PRISM.

The NOT meta-predicate requires that all atoms preceding it in the node must have been solved before it is invoked. The execution of this meta-predicate results in the creation of a special negation node which has as children the predicates which are to be negated. These predicates are solved as if they were positive atoms for whom all answers are desired. When all these predicates have been solved, the sets of values bound to each variable will be complemented with the domain over which they are defined. These

complements will be returned as answers by the negation node.

4. Examples of Control in PRISM

In this section we provide an example of AND-parallelism and an example of OR-parallelism to illustrate some of the capabilities of PRISM.

4.1. AND Parallelism

This example of AND-parallelism provides a preorder traversal of a binary tree.

Let

P(u,z) means that the preorder traversal of a binary tree u is z.
$t(y_1,x,y_2)$ denote a tree whose left branch is y_1, whose root is x and whose right branch is y_2.
$Append(y_1,y_2,z)$ mean that if y_2 is appended to the tail of y_1 the result is z.

We may then write,

1: $P(nil,nil) <--$

*2: $P(t(y_1,x,y_2),x.z) <--$
$([P(y_1,z_1),P(y_2,z_2)],Append(z_1,z_2,z))$.

1: $Append\ (nil,y,y) <--$

*2: $Append\ (x.y,v,x.w) <-- Append\ (y,v,w)$.

Thus, the base case, P(nil,nil) is always tested before the general case. When the preorder clause defined by *2 is executed, the preorder traversal of the left and right branches may be done asynchronously.

Since the preorder traversal of the left branch is independent of the right branch, they may be executed asynchronously in different processors. Each of the sub-branches may again be split to be executed on different machines. Hence, a number of different PSMs can be executing the problem at the same time. Thus, the time to execute the search is proportional to the size of the longest branch, rather than the number of nodes in the tree as in a sequential search.

Even if the problem is executed on a single machine, because one can be searching for matches on many nodes of the search tree, the multiple IDB machines can be working in parallel performing matching operations for each of the nodes in the search tree.

4.2. OR Parallelism

Consider a database problem whose database is shown below (Figure 2).

Extensional (Relational) Database

```
MOTHER(Rita, Sally)
MOTHER(Alice, Beth)
MOTHER(Laura, Christine)

FATHER(Harry,Jack)
FATHER(Harry, George)
FATHER(Jack, Sally)
```

Intensional Database

```
GRANDPARENT(x,z)    <- [Parent(x,y),Parent(y,z)]
PARENT(u,z)         <-  Mother(u,z)
PARENT(u,z)         <-  Father(u,z)

Query:   <- GRANDPARENT(x, Sally).
```

Figure 2.

We shall describe how the problem might be solved in PRISM on the ZMOB system. We use the following abbreviations in the series of figures that follow.

M-MOTHER	R-RITA	H-HARRY
F-FATHER	S-SALLY	G-GEORGE
G-GRANDPARENT	A-ALICE	J-JACK
	B-BETH	

We assume that there are only two moblets assigned to the EDB, one moblet contains the F-table, and the other the M-table. We assume that the IDB is replicated on two moblets and two moblets are allocated to be PSMs.

When the system is to be loaded, the ZMOB executive specifies the moblets allocated to the problem. The PRISM executive is informed of the machines and allocates the EDB, IDB, and PSMs to specific moblets. The data and programs are sent in a burst mode by the PRISM executive, resident in the host machine, to the appropriate moblets. Mask registers are set in the EDB moblets so that they can recognize the encoding for MOTHER and FATHER. The state of the system as would exist on ZMOB is illustrated in Figure 3. Processing of the query shown in Figure 2 is now described.

1. The query is submitted by the user to the PRISM executive which sends a message requesting response from a free PSM. We assume that the PSM on the first moblet encountered responds and that the query is sent over the belt in a burst mode. Figure 4 shows the state of the system at this point.

2. The PSM-1 forms a goal tree, and selects the only atom to be solved. It knows from preprocessing that the "G" predicate resides in an IDB machine. It sends it out to be matched against all procedure heads with

the same name. IDB-1 receives the request and also notes that there is only one response possible. See Figure 5.

3. IDB-1 finds a single match, informs PSM-1 that it has a match and at the request of PSM-1 transmits the body of the procedure and the unifying substitution. See Figure 6.

4. PSM-1 forms a new node (the resolvent clause) and selects the easier of the two subproblems to be solved, namely PARENT(y,S). It determines that the PARENT relation is intensional, and sends a message out to an IDB to be processed. Since IDB-1 is not busy, it accepts the message and now finds two matches. The PSM-1, in the meantime, determines that it has no work to be done since no additional responses are possible for the root node and it must wait for a response.

5. PSM-1 is informed by IDB-1 when it has found all matches for PARENT(y,S). There are two responses. PSM-1 may request that both responses be transmitted (or it may be done one at a time). Assuming both are transmitted, the PSM forms two nodes (OR branches). Since a PSM is available, it transmits one of the two nodes to PSM-2 to be solved. Now, PSM-1 can send out a request for MOTHER(y,S), while PSM-2 can send out a request for FATHER(y,S). These requests are sent out by pattern on the relation name and accepted by different moblets where the two relations are stored.

At this stage, two PSMs are cooperating in the solution of the problem, and two EDB machines are searching for data. The processing continues in a manner similar to the above description, until a solution is arrived at, as shown in Figure 3-9.

The above illustrates how OR parallelism may be handled within PRISM. Both AND and OR parallelism may be executing simultaneously. Each of the PSM machines may be working on a problem at some stage and all IDB and EDB machines may also be executing simultaneously.

5. Summary and Future Work

There have been several proposals to achieve parallelism in logic programming systems (Clark[1981], Hogger[1982], Pereira[1978], van Emden[1976], Wise[1982]). All these schemes, including PRISM provide natural ways of expressing algorithms for execution on conventional distributed architectures.

PRISM provides an implementation of logic programs on a highly parallel architecture. The design exhibits a high degree of modularity and orthogonality. By this we mean that portions of PRISM can be replaced by functionally similar modules with a minimum impact on the system. This provides a flexible tool to experiment with the implementation of various control strategies on diverse architectures. It provides full OR parallelism, partial AND parallelism and permits parallel asynchronous search for assertions and procedures. Parallelism is transparent to the user. We provide a proper interpretation of negation based on negation-by-failure.

The system represents a first approach to developing an experimental tool for the design and implementation of large AI systems. There are many capabilities that need to be added in a second development. Some of these are: co- routining; a full AND-parallelism; user specificable heuristics; typing of

variables; intelligent backtracking for arbitrary execution sequences; non-top-down search methods; and the ability to incorporate lemmas dynamically. These capabilities need to be incorporated into a coherent control language that would permit the user to specify diverse aspects of control to varying depths of detail while a problem is being solved. Some of the issues related to the above developments are explored below.

The fundamental difficulty in distributed problem solving arises from the fact that distributed control has not been cohesively studied and it is hard to achieve effective global solution by distribution of tasks: good local decisions are not necessarily a guarantee for an effective global procedure. Thus our efforts were aimed at the construction of a system that will be able to support various problem solving strategies without paying the price in efficiency of the execution. The main emphasis in our system was directed towards modularity, flexibility and adaptivity. We believe that a paper design is rarely as good as an effective implementation on a real parallel machine, which will enable modifications and enhancements with minimal programming effort, and consequently were admittedly willing to make some compromises in the initial system. Consistent with this philosophy the system components induce a logical network topology, and virtual processors may be added or deleted easily in our system without any changes to the rest of the system. Each set of machines is seen as a single machine to the rest of the system and any modifications and improvements to individual components may be made without effecting the rest of the system. In this section we briefly discuss some of the enhancements to PRISM that are currently under implementation or being investigated.

Database Machines

EDB - Today's databases are far larger than the memory capacity of a few hundred 64K microcomputers. Thus it will be useful to incorporate in our system a set of peripheral devices that will be attached to each EDB(or possibly shared by several EDBs). This will enable both an increased memory capacity and an ability to dynamically reconfigure the database machines in case of an unbalanced demand on one of the EDB machines. Additionally it will be constructive to facilitate basic database operators such as join,projection for efficient data retrieval.

IDB - In the current implementation the set of intensional database axioms (IDB) is replicated over several machines. This philosophy was based on the assumption that the IDB is relatively small and therefore may be stored in one machine. This assumption simplified the communication protocols and the operation of each IDB machine. We are currently developing a scheme in which the IDB is distributed over several machines. Additionaly to achieve effective performance from a highly parallel machine there is a need to control the ratio between the communication and the computation time, that is for a full utilization of a system it is desirable to increase the computation and minimize communication. In the system to date this control exists in the PSMs which may decide to solve a subproblem themselves rather than dispatch it to a different PSM. Similar techniques will be incorporated in the IDB. The IDB machine could perform several atomic resolution steps before returning the bodies and the unifiers to the PSM, thus increasing the ratio between computation and communication involved in a single resolution step. This effect may be attained either by a partial compilation of the the IDB axioms, or by

parameterized execution of the IDB that will perform a number of resolution steps as indicated by the PSM that initiated the query.

PSM - Machines Structure Sharing

Currently there is no sophisticated memory management done in the PSM. The memory management schemes most commonly used in Prolog implementations are copying and structure sharing. The decision not to incorporate structure sharing in PRISM was motivated by two factors. Firstly an implementation of a straight forward structure sharing will result in a tremendous overhead in pointer chaining before each literal(clause) is sent for expansion or a cumbersome and inefficient resolution operation if the structure sharing is done across processors. Secondly, since more than one path in the proof tree is active during execution, a locking mechanism must be incorporated to disallow bindings from two different paths to be applied to variables of the same literal simultaneously.

Parallelism enhancements in the PSM.

The flexible implementation of the selection procedures allows some dynamic exploitation of inherent parallelism in the program. This includes automatic detection of literals that do not share variables,and selecting literals that will maximize the degree of the parallelism in the new subgoal. Consider the goal : <- P(x),Q(x,y),R(y,z). It is quite clear that if Q is an EDB predicate, binding of x and y in Q will result in a new goal with two literals that do not share variables, and therefore maximize the parallelism in the clause. At the moment our system supports only local detection of parallelism, that is parallelism internal to a single clause. We are currently investigating partial compilation techniques to maximize global parallelism, during execution, and global planning strategies to optimize the performance of the system in terms of utilization of the computational power of ZMOB on the one hand and search space pruning on the other.

It is evident that cooperative problem solving must be supported with communication channels between PSMs to minimize some of the redundant search, by sharing partial results, eliminating goals that are logically related (by implication or subsumption) and task sharing.

The notion of a PSM as defined in our system is problem independent,that is each PSM may accept any problem. We are investigating the possibility of an Expert PSM which is dedicated to the solution of a class of problems. This notion will minimize some of the effort spent by the PSM in the selection process by precompiling some or all of the selection procedures.

Our system is based on a goal driven procedure invocation . Some thought has been given to facilitation of data-driven procedural invocation, that will allow effective simulation of data-flow machines.

Before any of the above enhancements are attempted we will need to perform many experiments with PRISM to determine its strengths and weaknesses. We plan to experiment with algorithms by alternatively modifying the logic, the control, and the architecture.

6. Acknowledgements

Work on this effort was supported by AFOSR grant number 82-0303 and by NSF grant number MCS-79-19418.

7. References

Chakravarthy[1982]
Chakravarthy, U.S., Kasif, S., Kohli, M., Minker, J., Cao, D., "Logic Programming on ZMOB: A Highly Parallel Machine", Proc. 1982 International Conference on Parallel Processing, IEEE Press, 1982, New York, pp 347-349.

Clark[1978]
Clark, K.L., "Negation as Failure", in Logic and Databases, H. Gallaire and J. Minker (Eds.), Plenum Press, 1978, New York.

Clark[1981]
Clark, K.L., Gregory, S., "A Relational Language for Parallel Programming", DOC 81/16, Dept. of Computing, Imperial College, 1981, London.

Computer[1982a]
Computer, Vol. 15, No. 2, Special issue on Data Flow Systems, February 1982, IEEE Press, New York.

Computer[1982b]
Computer, Vol. 15, No. 1, Special issue on Highly Parallel Computing, January 1982, IEEE Press, New York.

Eisinger[1982]
Eisinger, N., Kasif, S., Minker, J., "Logic Programming: A Parallel Approach", Technical Report 1128, Dept. of Computer Science, University of Maryland, College Park, 1981.

Hewitt[1977]
Hewitt, C., "Viewing Control Sructures as Patterns of Passing Messages", Artificial Intelligence, Vol. 8, North-Holland Publishing Company, 1977, pp 323-364.

Hogger[1982]
Hogger, C.J., "Concurrent Logic Programming", in Logic Programming, K.L. Clark and S-A. Tarnlund (Eds.), Academic Press, 1982, New York.

Kahn[1977]
Kahn, G., MacQueen, D.B., "Coroutines and Networks of Parallel Processes", Proc. IFIP Congress 77, North Holland, Amsterdam, pp 564-569.

Kornfeld[1979]
Kornfeld, W.A., "Using Parallel Processing for Problem Solving", A.I. Memo No. 561, MIT A.I. Lab, December 1979, Cambridge, MA.

Kowalski[1979]
Kowalski, R.A., "Logic for Problem Solving", Elsevier North Holland Inc., 1979, New York.

Kung[1980]
Kung, H.T., "The Structure of Parallel Algorithms", in Advances in Computers 1980, M.C. Yovits, Ed., Academic Press, 1980, pp 65-112.

Lieberman[1981]
Lieberman, H., "Thinking About Lots of Things At Once Without Getting Confused", A.I. Memo No. 626, MIT A.I. Lab, May 1981, Cambridge, MA.

Minker[1982]
Minker, J., Asper, C., Cao, D., Chakravarthy, U.S., Csoeke-Poeckh, A., Kasif, S., Kohli, M., Piazza, R., "Functional Specification of the ZMOB Parallel Problem Solving System", Technical Note Z-1, Dept. of Computer Science, University of Maryland, College Park, 1982

Pereira[1978]
Pereira, L.M., Monteiro, L.F., "The Semantics of Parallelism and Co-Routining in Logic Programming", Divisao de Informatica, Laboratorio Nacional de Egenhario Civil, December 1978, Lisbon.

Rieger[1980]
Rieger, C., Bane, J., Trigg, R., "ZMOB : A Highly Parallel Multiprocessor", TR-911, Dept. of Computer Science, University of Maryland, May 1980, College Park, Mayland.

Rieger[1981a]
Rieger, C., Trigg, R., Bane, J., "ZMOB : A New Computing Engine for AI", TR-1028, Dept. of Computer Science, University of Maryland, March 1981, College Park, Maryland.

Rieger[1981b]
Rieger, C., "ZMOB : Hardware from a User´s Point of View", TR-1042, Dept. of Computer Science, University of Maryland, April 1981, College Park, Maryland.

van Emden[1976]
van Emden, M.H., Lucena, G.H., de Silva, H.M., "Predicate Logic as a Language for Parallel Programming", Research Report, Dept. of Computer Science, Univ. of Waterloo, Ontario.

Wise[1982]
Wise, M.J., "A Parallel Prolog: The Construction of A Data Driven Model", Proc. 1982 ACM Symposium on LISP and Functional Programming, ACM Press, 1982, New York, pp 56-66.

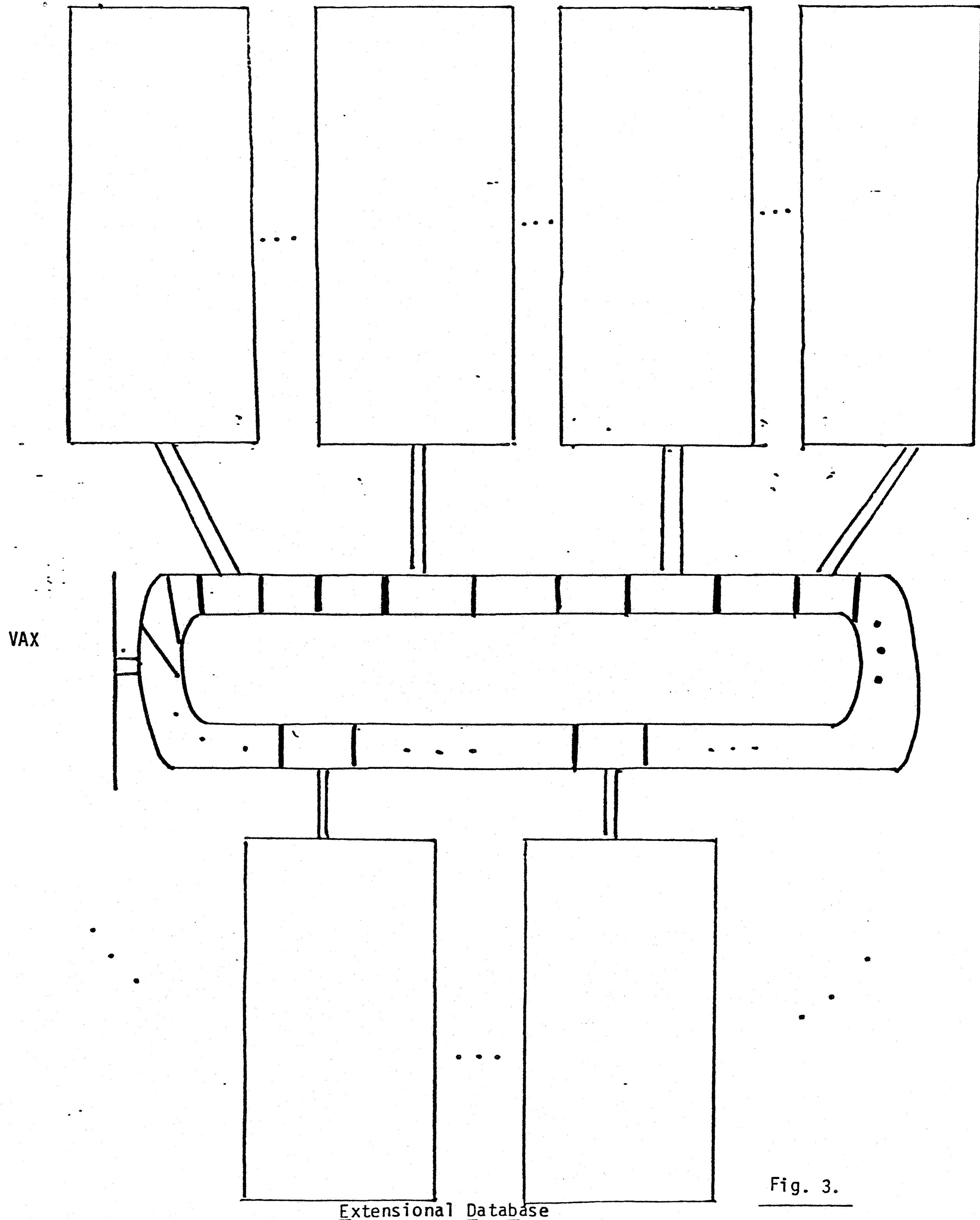


Fig. 3.

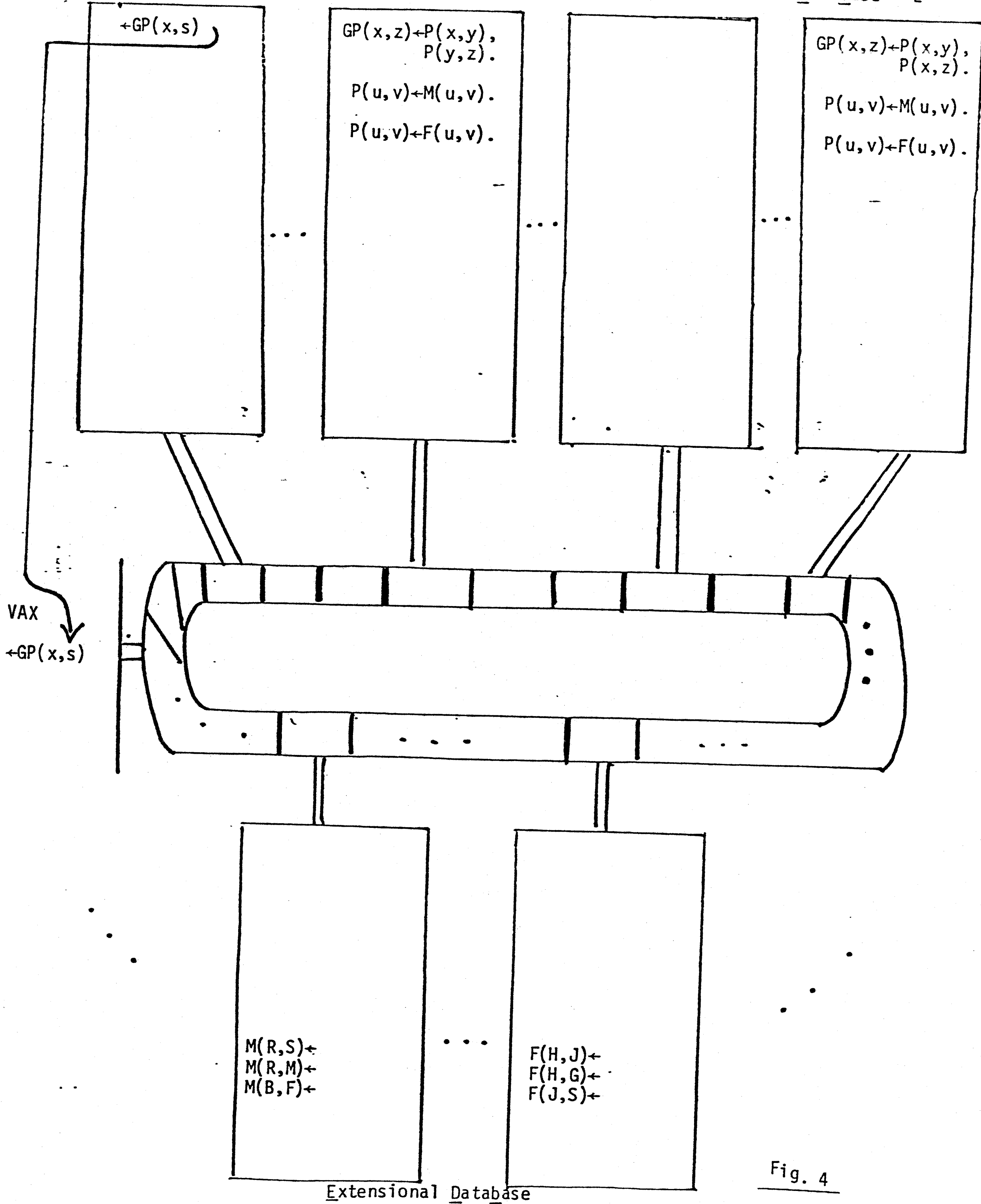


Fig. 4

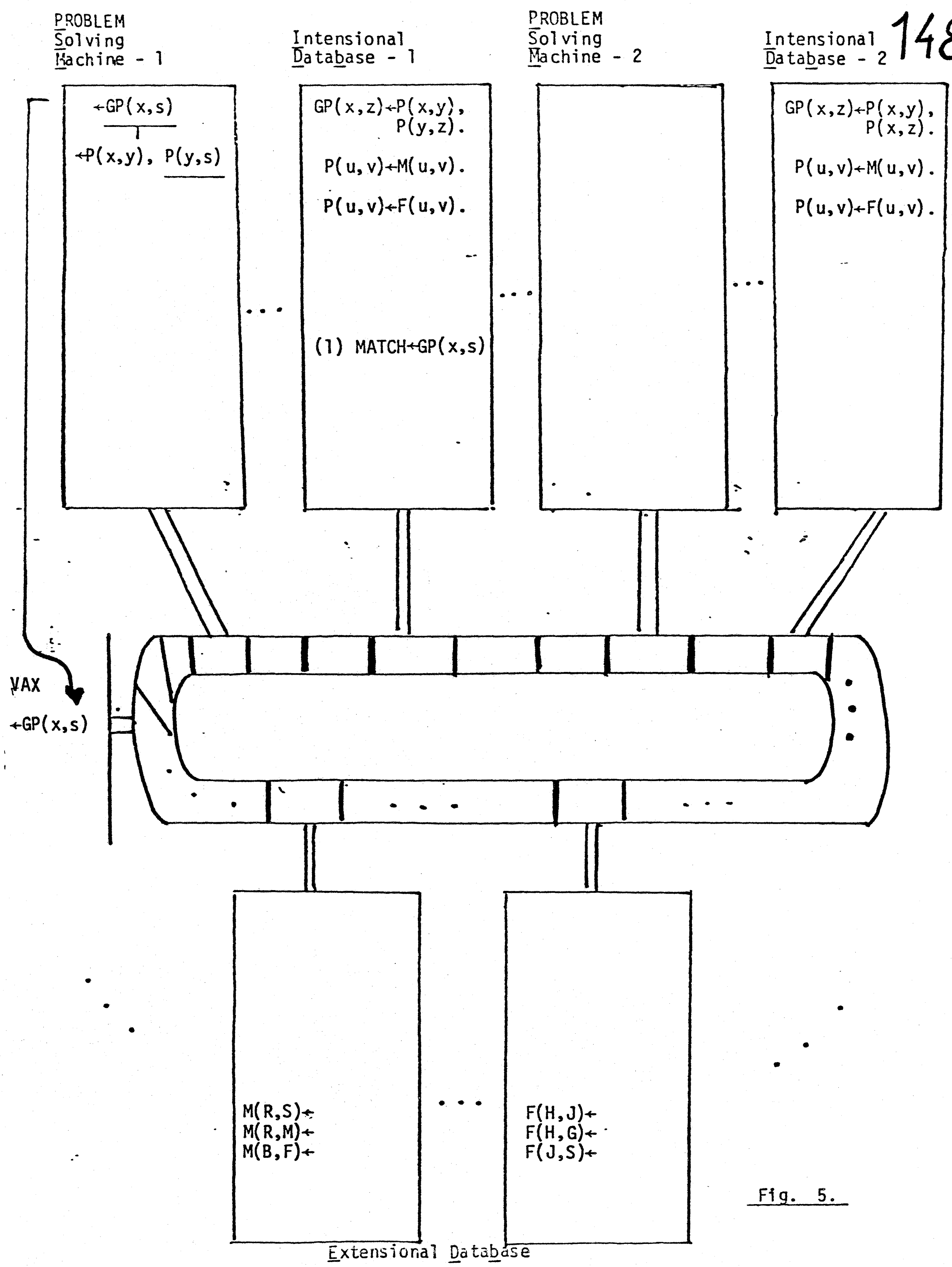


Fig. 5.

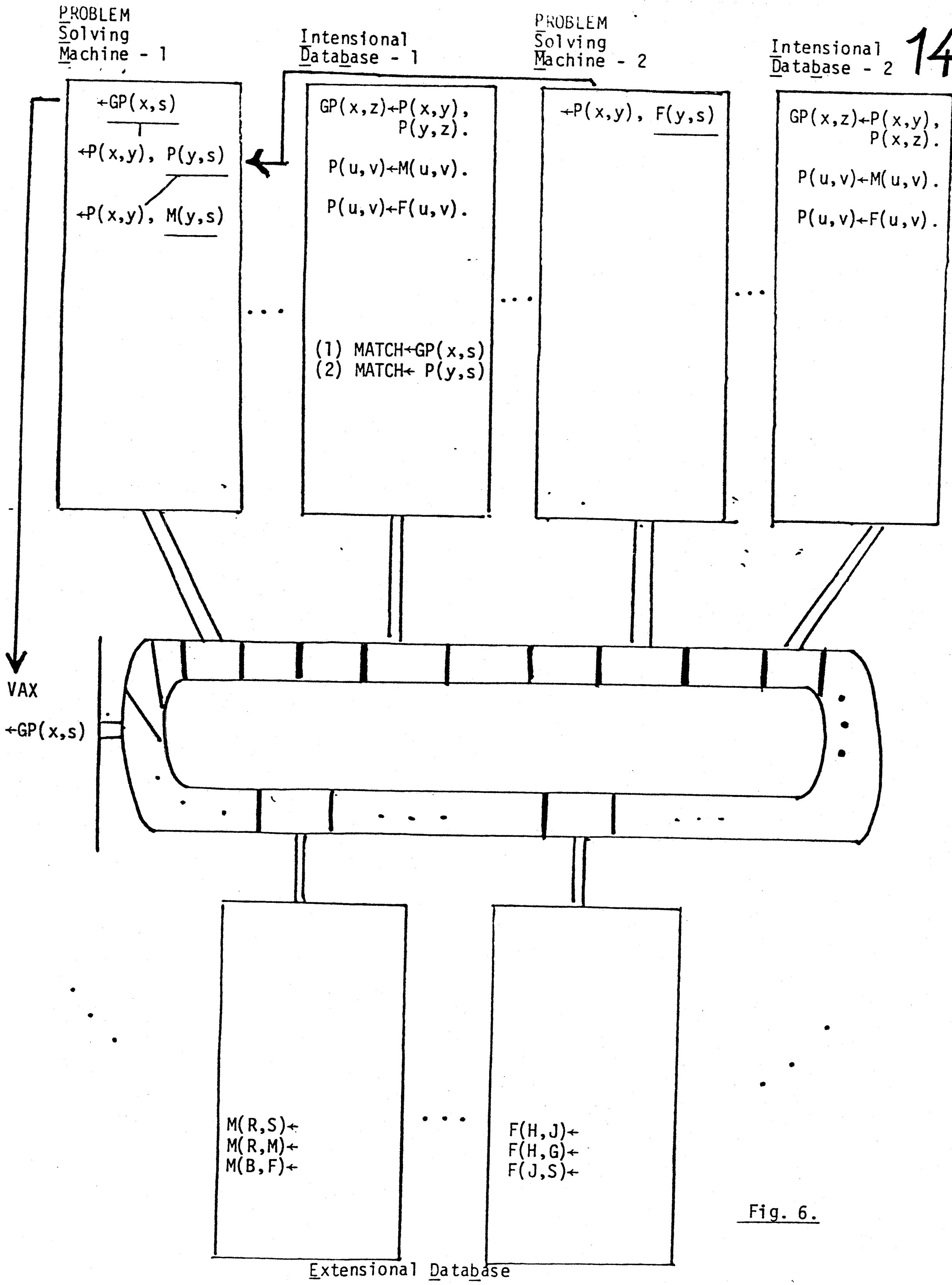


Fig. 6.

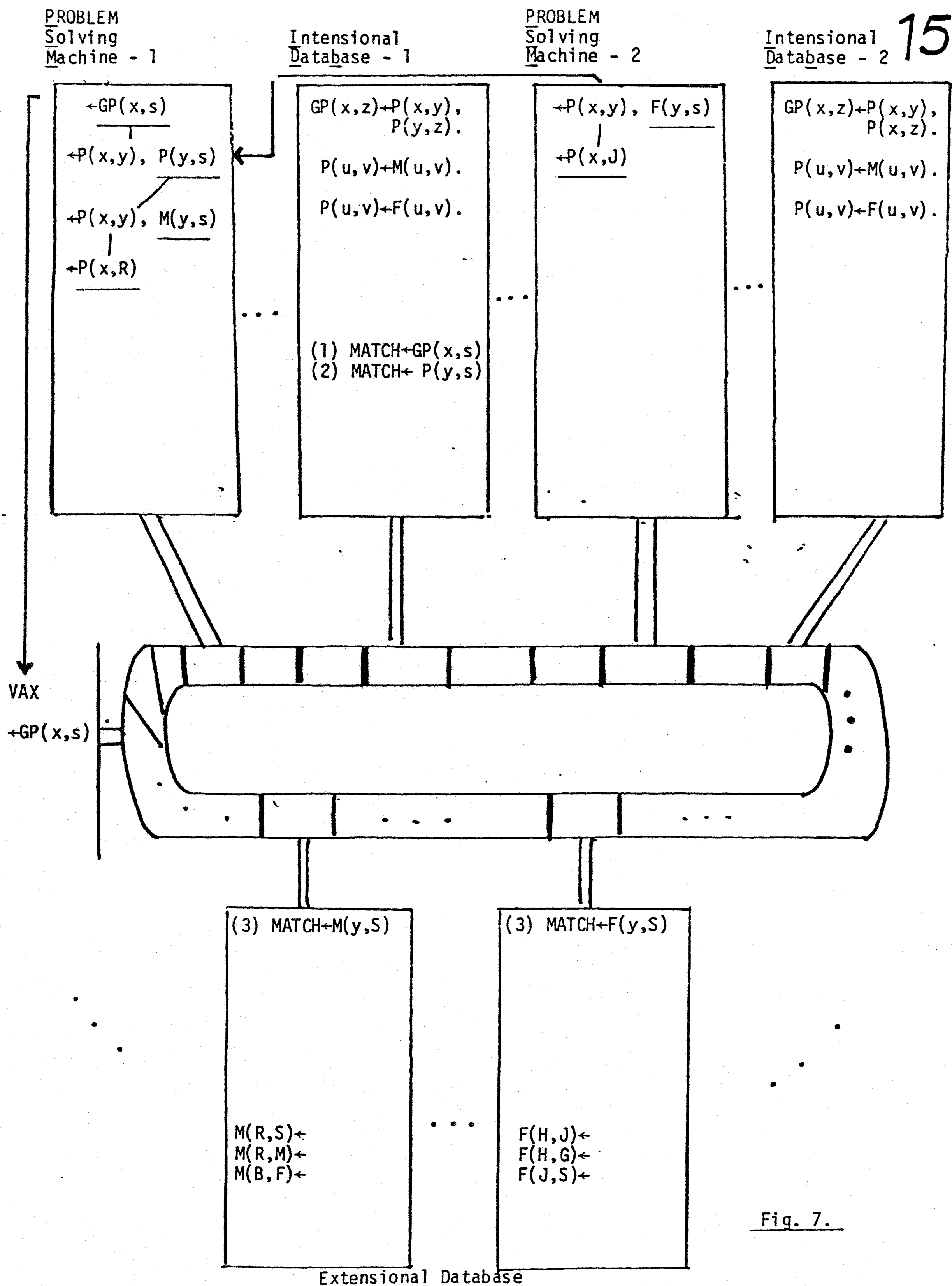


Fig. 7.

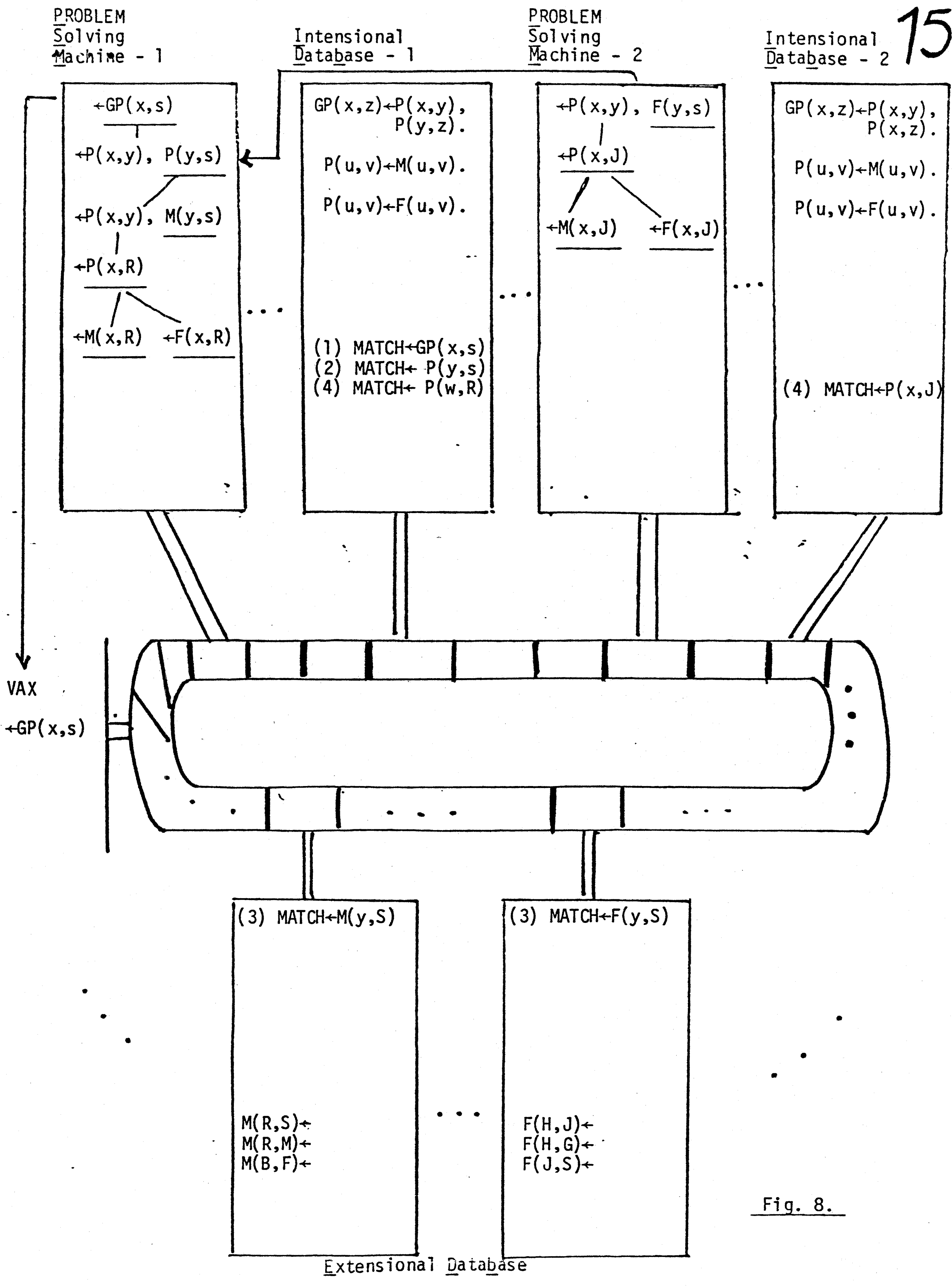


Fig. 8.

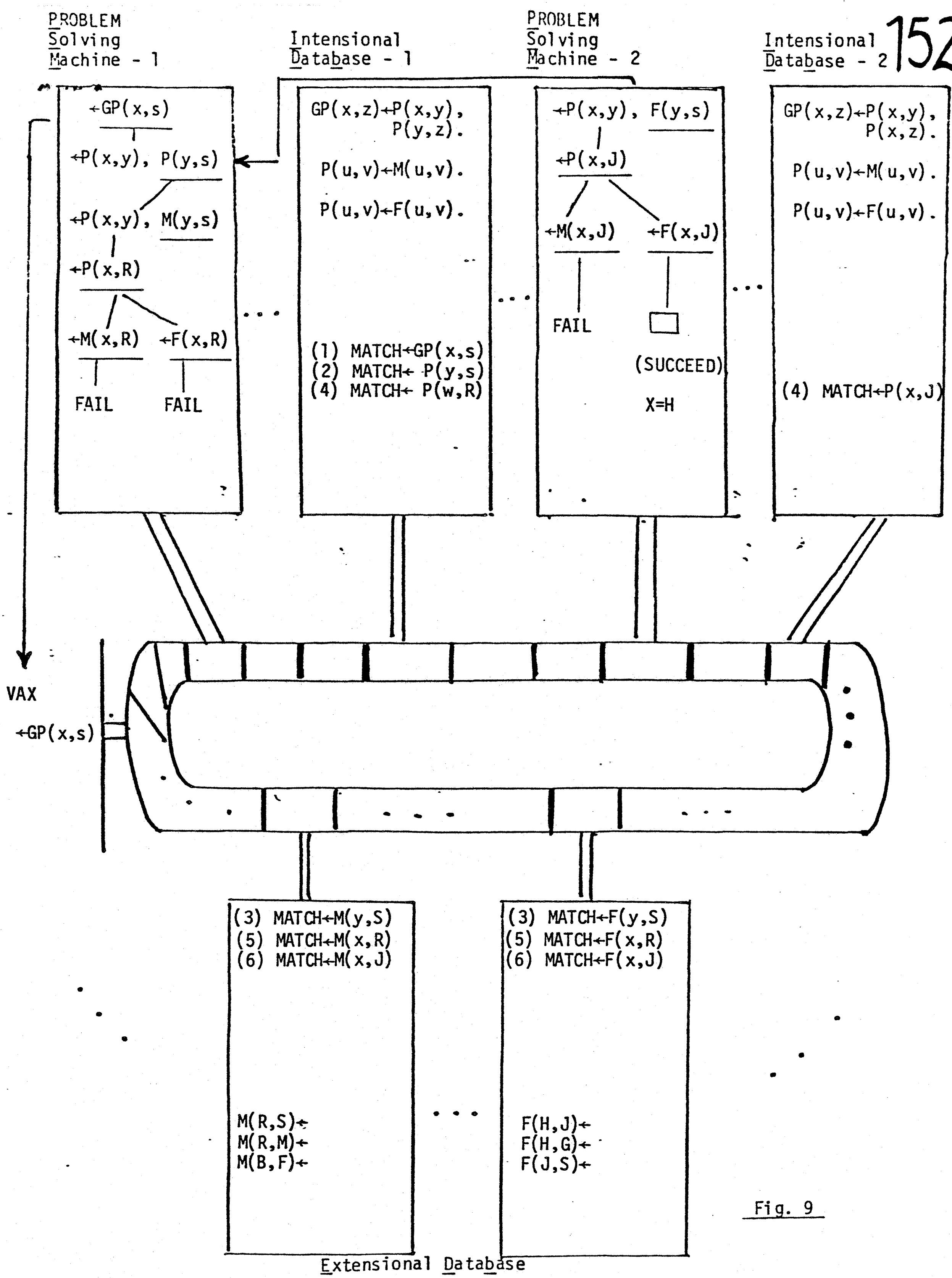


Fig. 9