1990

Parallelization of Goal-Driven, Production Systems on Hypercube Machines in a C Environment.

Rajendra Kumar Shrivastava
Louisiana State University and Agricultural & Mechanical College

Order Number 9104172

Parallelization of goal-driven, production systems on hypercube machines in a C environment

Shrivastava, Rajendra Kumar, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Parallelization of Goal Driven, Production Systems On Hypercube Machines in a C Environment.

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by

Rajendra K. Shrivastava

B. Sc., Fergusson College, Poona University, India, 1981
M. Sc., Department of Physics, Poona University, India, 1983

MAY 1990

Acknowledgments

I have to express my deepest respects and gratitude to my advisor, Dr. S. S. Iyengar. This research was done under his guidance where he helped me with his understanding of data structures, parallel and sequential algorithms, production systems and all issues related to their implementations.

Along with my advisor, I have to thank my committee members, Doris Carver, Bush Jones, Donald Kraft and Ganesar Chanmugam for their guidance, support and advice during the course of this research. Dr. Kraft, the chairman of the department encouraged and guided me along with the research and dissertation development. Dr. Carver and Dr. Jones were an invaluable source of information in the fields of compilers and programming languages. Dr. Chanmugam my minor professor helped me fulfill the requirements for the Ph.D. degree in my selected minor field of physics.

I would have to express my gratitude to all my family members for their love, support and faith in me; without their well wishes and encouragements I would never have reached this stage in my life. Finally, I have to express my gratitude to all my teachers from my high school days at Loyola High School in Poona during the 1970's until my graduate studies during the 1980's at Louisiana State University.

This research project was funded in part by a DOE-ORNL grant to Dr. Iyengar under contract number DE-AC05-840R2140, with Oak Ridge National Laboratory operated by Martin Marietta, to whom I am indebted for having provided me with this research opportunity.

Table of Contents

List of Figures

List of Tables

Abstract

Production systems are widely used in artificial intelligence to capture the notion of expertise in modeling expert systems. Production systems are computationally intensive programs spending most of the execution time in their *MATCH* or *recognise* phase. Efforts have been made by the research in this dissertation to minimize the production system's execution time by optimizing the MATCH phase. Goal oriented deterministic production systems are commonly used for robotics applications and formed the main class of *production systems* that were studied in this dissertation..

The main motivation for the research was to provide a better MATCH algorithm and use the multiprocessing capabilities of existing parallel computer hardware. The dissertation realizes these goals by transforming a traditional production system's scalar equivalence operations into C arithmetic hashing function to generate an indexing variable for the *switch-case* construct of the C language. Partitioning of the *working memory* into homogeneous blocks and distributing *production memory* over the multiprocessors enhanced the MIMD operation of the production system.

A scheme is formulated and implemented to identify a few key *condition elements* that may be used as an indexing variable and reduce the the number of condition elements used in the MATCH phase. The complete translation from OPS5 code to C and the implementation scheme is presented in this dissertation. Various issues regarding the distribution of the *inference engine* over the multiprocessor environment and other related synchronization topics for distributed systems are covered in the dissertation. A detailed description of the parallel computer's simulator is also pro-

vided in the dissertation.

The dissertation identifies other research topics and problems related to parallelization of production systems, the most significant being the ability to incorporate *LEARNING* in production systems by using one or all of the idle processors that are waiting for the active processor to complete it's activities.

Chapter One

Introduction

1.1. Problem Definition

A *production system* is a programming model that provides a scheme for representing knowledge, expertise and a means for expressing problem solutions. They are rule based programs that rely on knowledge and reasoning to perform a task in a specified field. This mode of program execution, restricted within a specific field, commonly leads the users to call them *expert systems*, since both the program and the human expert have knowledge and reasoning capabilities that are restricted to a specific domain. An expert system belongs to the production system class of programs and successfully deals with problems for which clear algorithmic solutions do not exist.

The structural property of a specialist is to solve a problem by searching the problem state space. The state space may be defined [25, 59] as a composition of:

- States, a collection of features that define a situation;
- Operators, that can transform one state into another; and
- Evaluate function, that rates each state in the problem space.

The search, or recognise, phase in production systems is the core Artificial Intelligence concept that requires both *knowledge* and *reasoning*, which distinguishes it from a naive search problem. The program uses a set of operators based on its knowledge and expertise to modify the state in a manner such that the expert is able to

determine a successful path from the initial state to the final goal in the state space and propose a solution to the problem at hand. The most common form of such programs are rule based production systems.

Traditional rule based programs have been judged by the amount of knowledge that a system embodies and little attention was paid towards efficient search algorithms; this has led to certain programs that spend up to 90% of their execution time in the search phase. Iyengar and his research group at LSU, have been studying various means of optimizing the search scheme and other aspects of rule based systems that merit consideration, with the overall goal being that of having an autonomous robot capable of operating in an hostile environment. The group's research is geared towards providing an environment for machines with limited human like reasoning and thinking features. The HERMIES-IIB robot at Oak Ridge National Laboratory will have multiple rule based systems, where each system is restricted to its domain like navigation, vision, sonar ranging, transducer sensing, I/O communications and etc. This dissertation is a report on the research that outlines a scheme for utilizing current day technology in parallel computers as a host architecture for goal driven expert systems.

Most of the rule based programs that are associated with a robot's activity are goal driven programs; hence, a parallelizing framework for goal driven systems has been the motivation for this dissertation. The OPS5 language based '*Monkey and Banana*' program was selected as a typical goal driven system, where the monkey is similar to a robot and has the capabilities to grasp, hold and move a light object from position A to position B. The final goal of the program is to move the monkey to a

destination such that there exists a a ladder at that point and the monkey is holding nothing in its hands and is able to climb the ladder and grasp the bananas.

1.2. Organization of the Dissertation

Each chapter has a brief introduction to the topics covered therein. Related research work has been mentioned with their contributions and their limitations; this dissertation has either resolved the limitations or identified them as future research work.

Chapter two is a detailed introduction to production systems. This chapter identifies the various modules of a production system and there activities. Different types of problems that have been successfully solved by production systems are discussed. Since a variety of problems have been solved, there exist different problem solving strategies that may be employed by the system; all of which are discussed in this chapter. Iyengar and others [40, 41, 65, 66] have identified a special class of production systems related to real time computing and classified them as *Asynchronous Production Systems*. These *APS* programs have the capacity to respond to changes in the problem state space that are not caused by the set of operators but by dynamic external events.

Chapter three lists some production system languages and discusses their advantages and their limitations regarding knowledge representation and reasoning capabilities in a programming environment. OPS5 language is selected to illustrate knowledge representation aspect of a production system in the form of *production rules* and human expertise in the *inference engine*. Examples that are considered for

parallelization and used to illustrate production system features are based on OPS5 written source codes.

Chapter four summarises the initial research work in the field of production systems. Forgy's RETE network was the first major contribution towards *MATCH* optimization. This chapter discusses the basis for most existing research concerning *recognise act* algorithms in all production systems. The RETE algorithm is discussed in detail since it is the the algorithm that OPS5 implements for production systems.

Chapter five is used to define and illustrate some standard paradigms for parallel computation of software processes. All modules of the production system are evaluated for parallelization purposes and justifications are made for a faster and more efficient *MATCH* phase. The available set of parallel computers for production systems studied in this chapter include the PSM, DADO and Hypercube machines. A significant part of the chapter analyses Gupta's PSM model for the internode and intranode parallelism in RETE and Miranker's TREAT algorithm for the DADO class of machines. The limitations of existing production systems as real time applications for robotics are identified, and these limitations provide the basis for the research reported in this dissertation.

Chapter six is the core of the entire dissertation. This chapter provides an in depth discussion and illustration for the complete translation and transformation of a traditional OPS5 goal driven program to an indexed, C language based, parallel program that is implemented on a hypercube simulator. The transformation scheme has a sequence of intermediate forms that are highlighted. The translation of OPS5 constants to C integers for indexing purposes is the major contribution of this dissertation

in the field of search optimization.

Chapter seven deals with the parallel *inference engine*, the stack like data structure used for maintaining the current goal in a goal driven production system. This chapter also discusses the hypercube simulator's Multiple Instruction Multiple Data features, like the software support and hardware components, that are being simulated. The chapter discusses the synchronization protocol and the other development features provided by the simulator for the cube's post performance diagnostics, obtained by processing a trace file.

Chapter eight provides an analytical study of the research and a review of results. This chapter also identifies some related areas specific to production systems, as well as general parallel computations, that merit further work and research in the future.

Appendices A through G are provided to list the application software in the OPS5 source code, trace files, C definitions and functions and the final C transformed code using an arithmetic indexing scheme. The trace files form an integral part of the parallelization process.

1.3. Contributions of this Research

The main contributions of the research reported in this dissertation are listed below and summarized in figure 1.1.

[1]. Identify the limitations of existing production systems for both sequential and parallel implementations.

[2]. Use shared memory MIMD based hypercube machines for medium grained

parallelization of production systems.

[3]. Propose development of high performance data structures for production systems in a C programming environment. The main aspect is a stack oriented data structure for the homogeneous, partitioned, *working memory* for all currently active goals in a goal driven production system.

[4]. Provide a transformation structure for representing data attribute values, from character strings to numeric constants, enhancing the computational efficiency.

[5]. Provide an algorithm to determine a set of key attributes from the data that is subsequently used in a hashing function for uniquely identifying a problem state space.

[6]. Provide a distributed, tuned *inference engine* for parallel goal driven production systems. The *MATCH* phase, as implemented with features stated in steps 3,4,5 and 6, is the major contribution of this research.

[7]. Provide a distributed synchronization and communicating protocol, in the hypercube programming environment, for parallel execution of production systems.

Chapter Two

Production Systems and Problem Solving Strategies

2.1. Introduction

Production systems are the basis of many rule based expert systems [21, 43, 51, 57, 59]. Rule based systems like expert and production systems are computer programs that emulate the search behavior of human experts in solving a problem. These are not procedural programs, but data driven programs, that are sensitive to current data and future modifications on the data. These data are also termed as the problem state; any changes in the state are carried out by the production system operators like, modification of some existing data or the creation of new data. The flow of program execution is controlled by the order in which the states are generated by the data in the working memory.

Production systems are capable of representing knowledge, expertise, and problem state. They also have a set of state evaluation functions and operators for modifying the state that make production systems simpler than other models of conventional computation, for a certain class of problems. Production systems have been widely used in implementing expert systems, such as the DENDRAL [13, 46] system for inferring chemical structures from spectrometry data, MYCIN [69] system for diagnosis and therapy of bacterial infections of the blood, PROSPECTOR [18] for mineral exploration, the R1 [49] system for configuring DEC VAX computer systems, and the HEARSAY-III [21, 22] production system for understanding speech.

Production systems defined as consisting of three main modules,which they are: a. production memory, a collection of production rules that represents the program knowledge; b. working memory, a collection of data elements that describe the current state of the system, and c. inference engine, a module that provides the decision making mechanism to the program. Each production rule has two parts, the left hand side or LHS, consists of assertions that have to be satisfied by the elements currently in working memory, and the right hand side or RHS, consists of actions that need to be performed and may involve some modification to the data or providing conclusions to the user. The search phase in production systems evaluates the data with respect to its knowledge, and then the system performs the RHS actions of the selected rule in what has traditionally been called the *recognise act*, or *match act*, cycle.

Real time production systems like R1 and HEARSAY-III [22,21] are computationally intensive in the MATCH phase. Some have an extremely large rule base forming the production memory, as in case of R1 production system, which has to consider all the possible configurations for the VAX computer. Others may have an extremely large working memory, as in case of the HEARSAY-III production system for understanding speech. For HEARSAY-III, program data in the working memory represent the entire spectrum of sound frequencies with different modulations that may have been generated by the speech pattern that is being evaluated by the production system.

Depending on the type of problem that production systems solve they are classified into either goal directed forward chaining systems or data directed backward chaining cognitive systems. Robotics applications, path routing and navigation

systems are goal directed systems, where the system tries to achieve a primary goal by modifying states in a sequence that generates the goal state. MYCIN, R1 and DENDRAL are data directed cognitive systems that evaluate the data and try to identify a sequence of problem states that would generate a state corresponding to the current data.

2.1.1. Recognise Act Cycle

The basis for production systems is the MATCH ACT or the recognise act cycle [8, 11, 63, 82] where the production system will recognise all the production rules that have a successful MATCH by iteratively evaluating the production rules over the working memory. The act phase is the action to be executed by the production rule that has been selected by the conflict resolution phase within the inference engine of a production system. This cycle is repeated until no production rules are true or the required results have been obtained.

2.2. Problem Solving Strategies

Production systems are query based programs that are similar to large database application programs; but unlike the database programs, production systems are expected to evaluate the program's initial response to the query and proceed to determine the correct sequence of queries that need to be answered if the original query does have an answer. Production systems have reasoning capabilities in the recognise act phase and may have multiple sub queries that are active at any given instance. database programs have to be provided one query at a time to perform the MATCH

phase only and cannot spawn subqueries that are active concurrently.

There are two approaches that have traditionally been implemented as problem solving strategies in production systems; they are forward chaining and backward chaining. Any production system may use a combination of the two strategies or use either one exclusively , depending upon the application at hand. The deduction or reasoning amongst both problem solving strategies may be based either on deterministic or probabilistic principles at any given production rule.

2.2.1. Forward Chaining

This form of problem solving tries to achieve the specified goal from the initial states of the working memory. For a given a set of initial conditions in working memory, the recognise act cycle will determine if the LHS of any production rule is satisfied and accordingly carry out the required actions. If the goal is not satisfied, then the system based on some pre determined heuristic will determine the the rule to be executed from amongst the currently satisfied production rules, execute that rule, thereby modifying the current working memory and proceed with it's effort of trying to satisfy the required goal.

The strategy is to determine all the goals that will be achieved by the current conditions and see if any of these goals match the required goal. This cycle is carried out till either the goal is satisfied or the production system determines that the goal cannot be satisfied. Figure 2.1 illustrates a problem that would require a forward chaining strategy to answer the question that is posed as the required goal from the initial conditions.

Initial-States

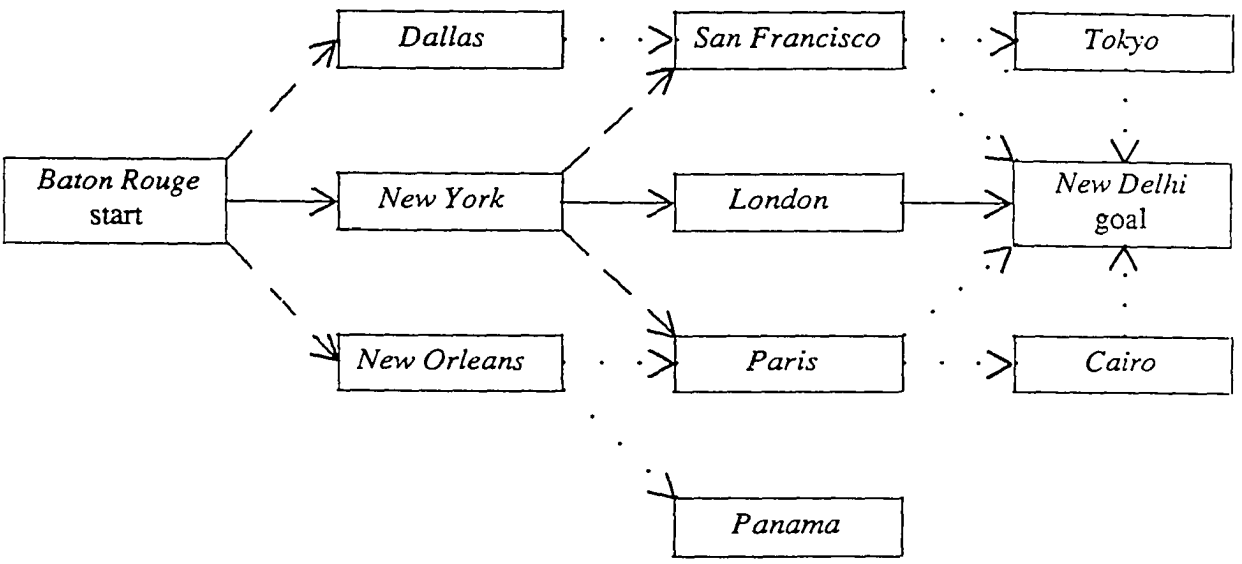
1] Today is Wednesday.
2] Time is 11.45 A.M.
3] Place is Baton Rouge Airport.

Required-Goal

To reach New Delhi, India, as soon as possible.

Algorithm

From Baton Rouge, trace a shortest path to New Delhi.



1. Solid arrow is the selected path. 2. Dashed arrow represents possible path.
3. Dotted arrow represents knowledge, not currently required.

Figure 2.1: Problem statement that requires Forward Chaining.

The production system has to determine a sequence of steps from the initial conditions to satisfy the final goal. This form of problem solving has also been called a

bottom up strategy, where the initial set of conditions form the terminal node of a search tree and the goal forms the root node. Some of production systems that are written in LISP may incorporate backtracking from a currently unsatisfiable goal, while the others written in CLIPS and OPS5 do not support that facility.

2.2.2. Backward Chaining

Given an initial set of conditions and a specified goal, the production system will start at the required goal and try to determine any production rule whose RHS matches the goal and the LHS is satisfied by the initial state. If no such rule is found, then the conditions in the LHS are used as new intermediate conditions and a sequence of production rules is determined such that when they are executed the specified goal may be achieved by the initial conditions. Similar to the case of forward chaining if the goal may not be satisfied directly from the initial conditions by one production rule, some intermediate working memory state will be considered by the production system as new conditions that do achieve the goal and iteratively work its way back to the original states, if possible.

The strategy is to determine the conditions that will generate the specified goal and see if these conditions match the initial state. Figure 2.2 illustrates a problem that would require a backward chaining strategy to answer the question that is posed as the initial conditions that might have led to the specified goal.

Initial-States

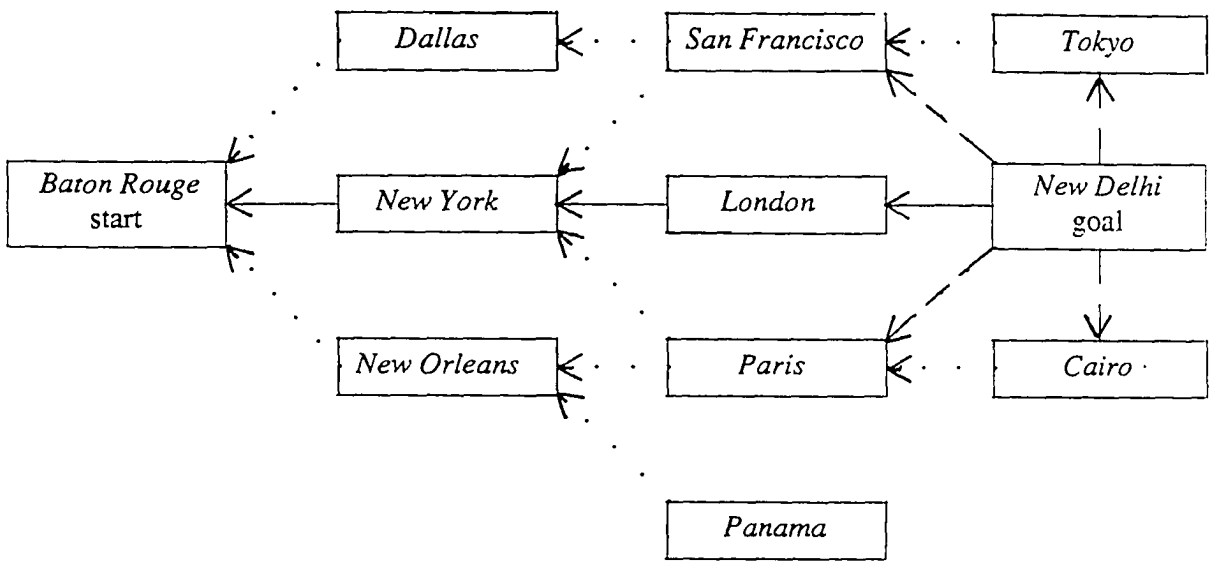
1] Today is Wednesday.
2] Time is 11.45 A.M.
3] Place is Baton Rouge Airport.

Required-Goal

To reach New Delhi, India, as soon as possible.

Algorithm

From New Delhi, trace back a shortest path to Baton Rouge.



1. Solid arrow is the selected path. 2. Dashed arrow represents possible path.
3. Dotted arrow represents knowledge, not currently required.

Figure 2.2: Problem statement that requires Backward Chaining.

Backward chaining strategy presumes that the goal has been attained, the purpose of the program is to determine which conditions caused the goal to occur. The

Backward chaining strategies presume that the goal has been attained, so the purpose of the program is to determine which conditions caused the goal to occur. The production system will try to determine those production rules whose RHS satisfies the condition of being in New delhi and see if the rule's LHS could be satisfied by the initial conditions. If this cannot be done immediately, then some intermediate problem states and goals will be generated by the system. This form of problem solving has also been called top-down or goal directed problem solving and need not necessarily incorporate backtracking. The production system is working it's way back from the goal to see if the initial conditions are sufficient to generate a sequence of actions that would lead the program to the final goal.

2.2.3. Taxonomy on Reasoning

The expertise of any intelligent system is determined by its reasoning capabilities. Artificial intelligence has identified three main forms of reasoning that any expert employs to arrive at conclusions. All three forms of reasoning are incorporated as a knowledge base in the form of production rules. The three types of inferencing are default, deterministic and probabilistic reasonings.

Default Reasoning

Several artificial intelligent systems have included simple mechanisms to allow a kind of reasoning called reasoning by default. Default reasoning is usually accompanied with exceptions, and the two are extremely useful in knowledge representation.

Consider an example of knowledge representation for birds. By default, it is gen-

eraly presumed that birds can fly unless explicitly stated as an exception. The attributes for describing the object type bird would be *(name = aa, wings = bb, living = cc, fly = dd)* where *aa* is the type of bird and *dd* is presumed to be true by default. An illustration of a bird that can fly by default, is the eagle, represented as *(name = Eagle, wings = Yes, living = Yes)* . A bird, with a representation that explicitly state's the bird's flightlessness, is the penguin, *(name = Penguin, wings = Yes, living = Yes, fly = No)*. For the examples stated above the *default* and *exceptions* could be associated with *wings, living* attributes too, since a wingless bird cannot fly and the same is true for a dead bird.

Defaults and exceptions are not easily incorporated into production system programs; they are AI concepts that are extremely difficult to encode into a computer program.

Deterministic Reasoning

In this mode of reasoning there is no ambiguity during the MATCH phase of the production rules. The working memory will either match a given production rule or it will not, though at any instance a multiple number of rules may be found to satisfy the MATCH constraints. Deterministic reasoning may be used in either forward chaining or backward chaining strategies, it provides a simple form of representing production rules.

IF expression has *symbol = '*'* and *argument1 = 1*

THEN expression is evaluated to *value = argument2*.

Figure 2.3: A production rule for mathematical symbolic expression.

Figure 2.3 illustrates a production rule that implements deterministic reasoning to determine the conditional part of a rule. The basis for the rule is determining the type of operator and the numeric value of the first argument in the LHS condition elements before the RHS can execute and evaluate the expression.

Probabilistic Reasoning

Certain applications may have working memory and production memory that have inherent uncertainties associated with them, as in MYCIN, PROSPECTOR and other cognitive programs. These production systems employ statistical principles to compute the conditional probabilities along *Baye's* theorem to resolve the problem of real world randomness or lack of complete information in the problem domain. In such production systems those rules that have the greatest probability of being correct are selected as the production rule to be executed. Figure 2.4 illustrates an expanded form of figure 2.2 and is an example from the MYCIN production system.

IF this rule has *attenuation = 0.8* and the given patient has
abdominal pains = many with *certainty = 0.6*
and *blood sugar = low* with *certainty = 0.4*
and *previous appointments > 0* with *certainty = 1.0*

THEN conclude *diagnosis = hypoglycemia* with *certainty = (0.4 * 0.8)*

Figure 2.4: A production rule for medical diagnosis from MYCIN.

In the MYCIN program, each production rule has a finite positive uncertainty based on the condition elements forming the LHS. Overall certainty of the conclusion in the RHS, in terms of probability, is the value (0.32) which is computed as follows *(MIN[certainties of LHS elements] * attenuation)* which is *(MIN[0.6, 0.4, 1.0] * 0.8)* as indicated by the expression *(0.4 * 0.8)* in RHS of the rule in figure 2.4.

The probabilistic class of reasoning has not been implemented in the indexed form of prallel production systems since there is an inherent ambiguity associated with each rule and the purpose of the indexing scheme is to provide an unique rule for a problem state.

2.3. Production Systems

The main thrust of this research is to identify a more efficient form of knowledge representation using C preprocessors '#define' macros as integers. It is illustrated in appendix C for the monkey and banana program. These integers are subsequently used in an arithmetic hashing function to emulate the deterministic reasoning of a forward

chaining, goal directed production system. The following section provides a brief discussion on the various components of a production system.

A production system has a rule base that emulates knowledge or long term memory and is represented by the production memory. The knowledge is expected to remain static over the period of program's execution and may be modified only by those production systems that have the capability to operate in a learning mode.

Data for program execution is provided as a short term memory is the working memory and it is required until the problem at hand has been resolved. The contents of this memory are dynamic and get updated as a consequence of the RHS actions of selected production rules . The expertise is provided by the conflict resolution phase of the Production System which sets them apart from pure database applications.

2.3.1. Components of Production Systems

The main components of a production system are production memory, working memory, inference engine and may be represented as in figure 2.5. The purpose of the components is as follows.

- *production memory : This is an unordered collection of IF THEN (production rules) statements. It is the knowledge of the expert within a specific domain. The knowledge is expected to be static over the program's execution; however the knowledge may be updated at certain break points within the execution cycle if the production system is capable of executing in a self learning mode. Figures 2.3 and 2.4 illustrate production rules that an integral part of the production memory.*

- *working memory* : A database on which the production rules are operated. This data is dynamic and represents the current problem scenario being tackled by the production system. The changes are a direct consequence of the actions specified in the RHS of the selected rule. It is a collection of instantiations of condition elements and known as memory elements that have been created and or modified till now.

- *inference engine* : That part of a production system which embodies the programmers expertise in it's decision making mechanism. The inference engine has the task of selecting one production rule to be executed from a set of production rules which have been found to *match the contents of working memory. The selected rule is then executed by the ACT module of the inference engine. Section 2.4 has a more detailed explanation for this part of the production system.*

2.3.2. Representation of Production Systems

A simple production system may be graphically illustrated as in figure 2.5 along with the flow of program execution and control. There is an inherent iterativeness associated with all production systems, specially so in the MATCH ACT cycle as illustrated by the feedback loop from ACT to MATCH in figure 2.5.

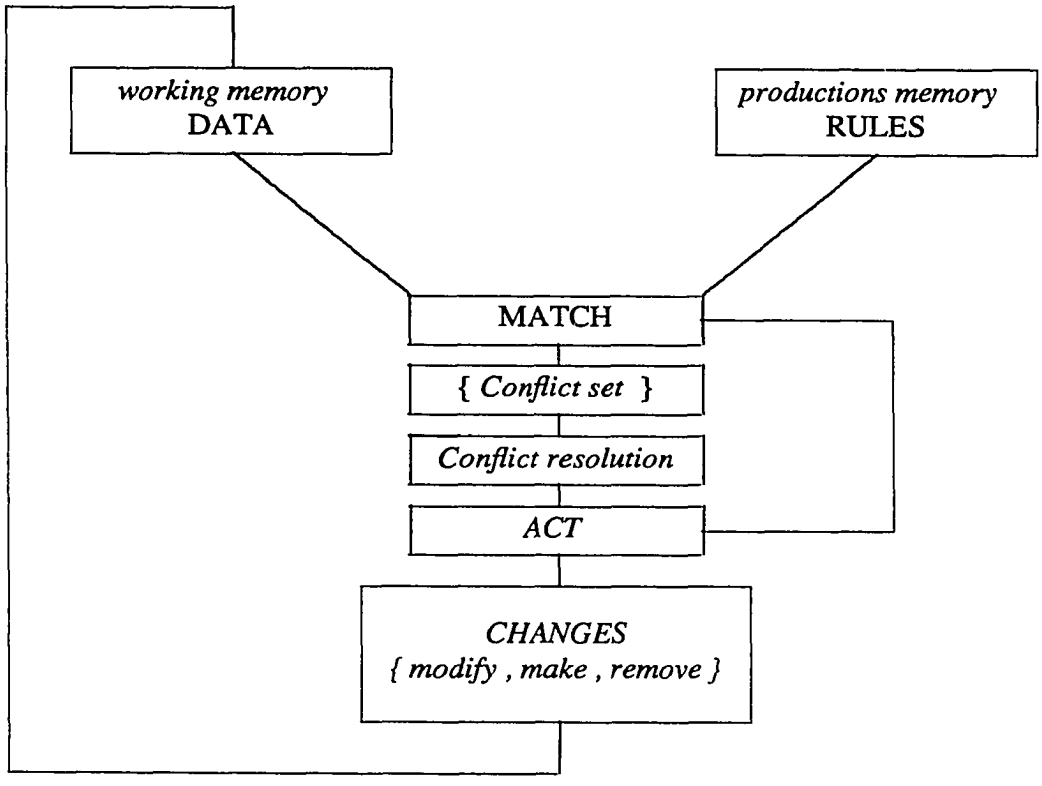


Figure 2.5: Production System representation.

The production system is run under the supervision of a CONTROL program which contains the decision making mechanism in the inference engine. The different phases in the life cycle of a production system are:

[1]. *Match :* This is an incremental equivalence operation specified by the LHS of a production rule on the working memory contents. It iteratively evaluate the LHS of the production rules on current working memory and forms the conflict set of production rules that are satisfied. The conflict set is a collection of ordered pairs known as instantiations of the type

⟨ Production Rule , List of elements matched by it's LHS ⟩.

[2]. *Conflict Resolution :* On the basis of some criterion [priority, probability, recency, ...], select one production rule from the set formed by [1]. If the conflict set is empty then halt the system.

[3]. *Act :* The CONTROL module perform's the action specified by the RHS of the selected production rule. The RHS may be a conclusion/deduction or any of the actions { modify, make, remove } that affect the working memory.

[4]. While *working memory* is not empty GO TO 1.

2.4. The main Modules of Production Systems

As illustrated in figure 2.5 any production system has three main modules, MATCH, INFERENCE ENGINE and ACT, which are active through the program's execution lifetime.

2.4.1. MATCH Module

This module is computationally intensive and a time consuming phase of all production systems. It is directly proportional to the size of production memory and working memory. The rule base or knowledge is represented as a collection of production rules which are of the type

IF < circumstances > THEN <modify data or conclude something> .

The set of circumstances are called the LHS or antecedents of a rule and the action or consequence form the RHS of a rule. Each LHS of a production rule is a collection of condition elements while the RHS is either an action to be undertaken on the working memory or draw some conclusions. All production rules always will have to include a LHS that is a set of conditions or antecedents and a RHS that is set of consequences. A set of circumstances depicting a particular scenario for the Monkey and Banana [8] production system program may be stated as follows

If *there exists an active goal to hold an object named <O>;* **and**

If *there exists an object <O>, such that the weight is light, it is not on the ceiling and it is at a location <P>;* **and**

If *the monkey is not at <P>*

then *create a new active goal such that the monkey will be at the location <P>.*

The above stated logical statement may be translated into the OPS5 form of a production rules as illustrated in figure 2.6

```
(p holds::obj-notceil:at-monkey
   (goal ^status active ^type holds ^obj-name <o> )
   (phys-obj ^name <o> ^weight light ^at <p> ^on <> ceiling )
   (monkey ^at <> <p> )
-->
   (make  goal ^status active ^type at ^obj-name nil  ^to <p> ))
```

Figure 2.6: Illustration of a deterministic production rule in OPS5.

It should be noted that the production rule does not have any probability factor hence it is completely deterministic. The conflict set is generated by the MATCH or recognise phases explained in section 2.3.2.

```
with goal[i] do       /*** for each goal in the working memory ***/
begin
    if((goal[i].status = active)  and  ( goal[i].type = holds))
   then                /*** for each phys_obj in the working memory ***/
       if ((goal[i].obj-nam = phys_obj[j].name)  and  (light  = phys_obj[j].weight)
       and  (ceiling <> phys_obj[j].on)  and  (monkey.at <> phys_obj[j].at))
       then
         begin    /*** RHS add new element of type goal to working memory   ***/
             goal[i+1].status := active;
           goal[i+1].type   := at ;
           goal[i+1].obj-name := nil;
           goal[i+1].to    := phys_obj[j].at;
         end if
 end with
```

Figure 2.7: Pascal like form of OPS5 rule in figure 2.6.

The above representation is an efficient form of the OPS5 rule since we have used a partitioned representation for the working memory into homogeneous arrays of record types goal, phys_obj, monkey. In OPS5 implementation the working memory would simply be an array of heterogeneous record types corresponding to the production system's condition elements.

Illustrating the production system's iterative nature with the OPS5 representation in figure 2.6, the source of the the computationally intensive search becomes clearer in the Pascal language like construct for the selected production rule in figure 2.7. Since OPS5 is inherently iterative care has to be taken in translating the OPS5 production rule; figure 2.7 incorporates the iteration as condition loops executing over different elements of working memory.

2.4.2. INFERENCE ENGINE Module

The expertize is in the conflict resolution phase of the program which has to select one rule from the conflict set for execution. The selection process is the inference engine and is based on some predefined, application dependent strategy and may be any one or a combination of the following schemes.

- *RECENCY*: Fire that rule whose pattern match the most recently modified or created data. OPS5 implements this form of resolution and is applicable to the first condition element in the LHS.
- *SIZE*: Fire that rule whose pattern [LHS] is more specific than the others. The rule that has a pattern which forms the superset of other patterns for the condition elements.

- *STATISTICS*: In a non deterministic system like diagnostics, exploration or cognitive modeling, fire that rule which has the highest probability of being true.

This thesis research work is related to parallelizing OPS5 based production systems and figure 2.8 illustrates some production rules where the conflict resolution may have more than one rule to select from.

Since the OPS5 production rules for the Monkey and Banana problem are completely deterministic the conflict resolution is simplified to the following

- If more than one production rule form the conflict set then using recency as the selection criterion, the rule with a higher value of the index *i* in *goal[i]* would be selected with respect to figure 2.7.
- For production rules with the same goal elements, the next criterion for selection would be the next condition element. On the basis of figure 2.6 this translates to *j* value for phys_obj[j], *this is repeated successively until one production rule has been selected.*
- In case the conflict is still not resolved then the last criterion would be selecting from amongst the still active rules, one rule such that, it forms a superset of LHS patterns.

Figure 2.8 illustrates production rule holds::obj-ceil that has a LHS which forms a superset over rule holds::obj-notceil which in turn forms a superset of patterns over the rule holds::obj-notceil:at-monkey.

```
(p holds::obj-notceil:at-monkey
   (goal ^status active ^type holds ^obj-name <o> )
   (phys-obj ^name <o> ^weight light ^at <p> ^on <> ceiling )
   (monkey ^at <> <p> )
-->
      RHS)

(p holds::obj-notceil
      {(goal ^status active ^type holds ^obj-name <o> ) <goal>}
      {(phys-obj ^name <o> ^weight light ^at <p> ^on <> ceiling) <object>}
      {(monkey ^at <p> ^holds nil ^on floor)  <monkey>}
-     (phys-obj ^on <o>)
-->
      RHS)

(p holds::obj-ceil
      {(goal ^status active ^type holds ^obj-name <o1> ) <goal>}
      {(phys-obj ^name <o1> ^weight light ^at <p> ^on ceiling ) <object1>}
      {(phys-obj ^name ladder ^at <p> ^on floor ) <object2>}
      {(monkey ^on ladder ^holds nil )  <monkey>}
-     (phys-obj ^on <o1>)
-->
      RHS)
```

Figure 2.8: A set of production rules with superset LHS patterns.

2.4.3. ACT Module

The purpose of this module is to carry out the actions specified in the RHS of the selected rule. The active actions always involve some form of modification to the working memory. The modification may be in the form of creating new elements, changing attribute values of existing elements or deleting existing elements. The passive form of RHS actions involve simply informing the user about current status and conclusions.

It is the Act module that really separates production systems from other data driven programs, since this module has the capabilities of setting up new goals and modifying the program environment and letting the MATCH ACT cycle repeat all over again till a terminal condition has been reached. Section 3.4.2 provides more details regarding the RHS action of rules.

2.5. Asynchronous Production Systems

Asynchronous Production Systems or APS as they are generally called were conceived, developed and implemented by Iyengar and others in the research group [40, 41, 65, 66] at Louisiana State University for the HERMIES-IIB robot at CESAR, Oak Ridge National Laboratory. HERMIES-IIB is an acronym for Hostile Environment Robotic Machine Intelligence Experiment, series IIB and CESAR is an acronym of Center for Engineering Systems Advanced Research.

Conventional production systems are data driven programs that execute the recognise act cycle till completion where the data is modified only by the RHS actions of production rules. APS are a version of production systems that enable external events to modify the data in working memory. This ability of APS to recognise the occurrence of external events facilitated the development of real time expert systems which retain the convenience of traditional rule based system's knowledge base representation. Unlike conventional systems, the contents of working memory are not the only information that is available to the APS during the recognise act cycle.

Asynchronous Production Systems use the knowledge base representation of traditional production systems but incorporate a new data structure for representing

external memory and it has a higher precedence over the data in working memory at execution time. The need for recognizing external events is apparent for applications in the fields of Autonomous Robots, Process Control [12, 56], Medical Monitoring, Navigation in Unknown Terrains [9, 38] and etc. All these programs require a rule base for representing knowledge and also should be able to respond to a real time unknown or unexpected external event.

HERMIES-IIB is a robot [9] that is designed to operate in an unknown and hostile environment. It has a set of SONARS to detect moving objects, where the motion is with respect to the Robot and a few transducers to detect fire and other external climatic conditions. The production system that is controlling HERMIES-IIB has the task of determining a safe navigational trajectory for the robot. The path to be generated should avoid all obstacles and also external hazards of fire.

2.5.1. APS Data for HERMIES-IIB Robot

The rule based system for this robot has to take into account data for the external memory that is being generated by the sensors and transducers and the conventional data in working memory that is generated as a consequence of RHS Actions of executed rules. Figure 2.9 illustrates this partition of data for any Asynchronous Production System.

WORKING MEMORY DATA DEFINITIONS

(goal ^status ^type ^start ^destination)
(robot ^position)
(exception ^status ^type ^priority)

EXTERNAL MEMORY DATA DEFINITIONS

(fire-transducer ^smoke ^temp)
(sonar-infra-red ^distance ^height ^width)
(sonar-acoustic ^distance ^height ^width)
(sonar-ultra-sonic ^distance ^height ^width)

Figure 2.9: A set of DATA elements for HERMIES-IIB APS.

The working memory data is used for the navigational aspect of the program which computes the shortest path from an initial point goal ^start A to the final destination goal ^destination B using intermediate positions X for the robot as robot ^position X. Any modification to this data has to originate as an action of the APS itself and this represents the conventional part of the system.

The external memory data is used for the asynchronous exception handling aspect of the program which reacts immediately to stimuli from any external event that HERMIES-IIB is capable of detecting. There is an associated rule base for detection of each external event in production memory as will be described in section 2.5.2. Figure 2.10 illustrates the conceptual layout of an Asynchronous Production System.

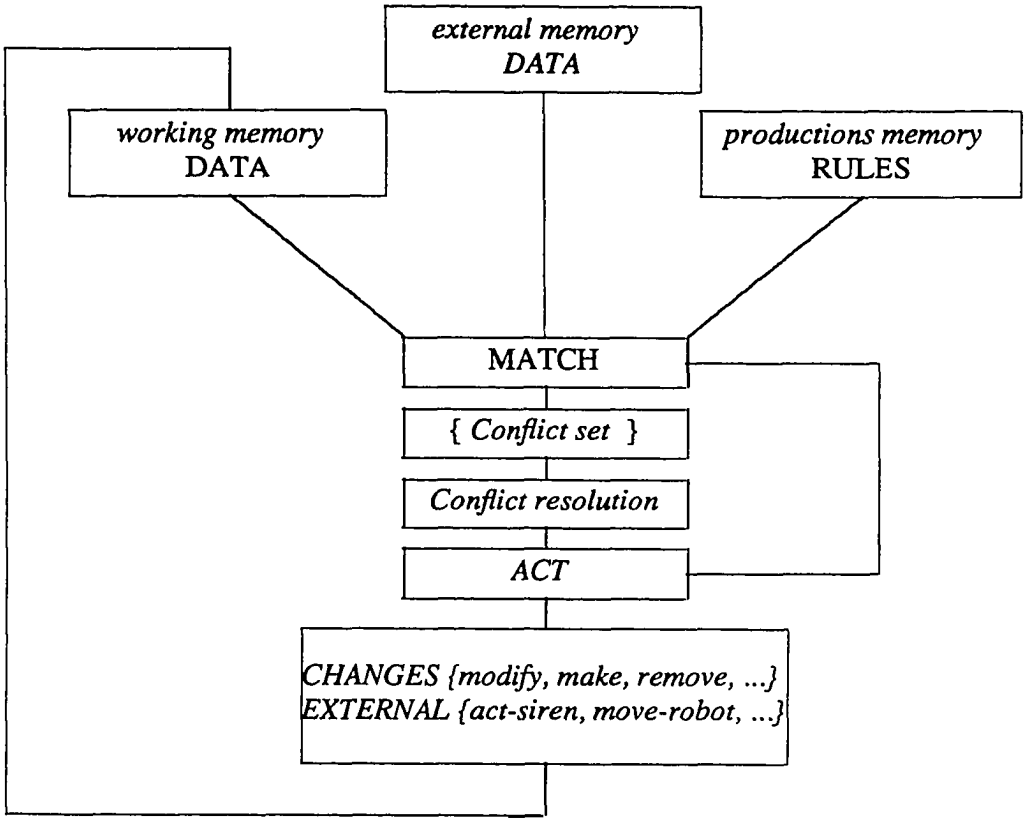


Figure 2.10: Asynchronous Production System representation.

2.5.2. APS Production Rules for HERMIES-IIB Robot

Real time systems should recognise and react to external events and also be able to execute the traditional sequential rule based data driven programs. HERMIES-IIB is an autonomous robot [9, 65, 66] that is capable of navigating in an unknown hostile terrain. The traditional aspect of the HERMIES-IIB program deals with the navigation of the ROBOT. The real time stimuli is provided by the robot's vision, sonar and other transducer sensors. Figure 2.11 illustrates a few production rules that are required for an autonomous robot to be able to react to asynchronous external events.

Rule error checks if the Robot is in a position X such that there exists no goal data element that has the same start value X. The LHS of rule error is the same as that of any other rule of a traditional production system rule that uses the production memory and the working memory to determine whether the rule is satisfied or not. If error is satisfied and executed the RHS consists of actions that are restricted to the ACT module illustrated in figure 2.5.

Rule navigate checks if the Robot is in a position X such that there exists a goal data element that has the same start value X and a destination Y different from X. LHS of rule navigate, is the same as any other production system rule, and the matching is in between the working memory and the production memory. The RHS of rule navigate differs from other rules since the action move-robot invokes the external procedure that actually moves the robot to a new location and is not available to all production systems in the ACT module as illustrated in figure 2.5.

```
---------------------------------------------------------------------------------

A]   CONVENTIONAL RULE WITHOUT EXTERNAL ACTION
  (p error
       {(goal ^status active ^type navigate ^start <s> ^destination <d>) <goal>}
       {(robot ^position <><s>) <robot>}
-->
       (write (crlf)" Robot is not at Starting Position."(crlf))
       (modify <goal> ^status satisfied))

B]   CONVENTIONAL RULE WITH EXTERNAL ACTION
  (p navigate
       {(goal ^status active ^type navigate ^start <s> ^destination <d>) <goal>}
       {(robot ^position <s>) <robot>}
-->
       (move-robot(<s>, <d>))
       (modify <goal> ^status satisfied))

C]   ASYNCHRONOUS EVENT RULE WITH EXTERNAL ACTION
  (p fire-emergency
       {(exception ^status active ^type fire-alarm) <exception>}
       {(fire-transducer ^smoke present ^temp high) <fire-transducer>}
-->
       (activate-siren)
       (modify <exception> ^status satisfied))

---------------------------------------------------------------------------------
```

Figure 2.11: A set of production rules for HERMIES-IIB APS.

Rule fire-emergency checks if the Robot is in a position such that there is a fire in its immediate vicinity. The fire is detected by the existence of a data element exception ^status active ^type fire-alarm that has been placed in the working memory by the robot as a consequence of a hardware interrupt. The same interrupt also placed the other information in the external memory for data element *fire-transducer ^smoke present ^temp high*. The LHS matching process is no longer between working memory and production memory but in between production memory and the external memory.

The RHS action of fire-emergency invokes an external procedure as illustrated in figure 2.10 and figure 2.11, which also is a special APS feature that is not available in traditional production systems.

2.6. APS Execution

APS are rule based production systems that are capable of reacting to external events. There are two main features of any APS that separates them from traditional production systems. Both the features are related to the production rules and involve the MATCH, ACT modules which either need to consider the external memory as data for the LHS of a rule or external procedures as RHS actions.

MATCH If working memory **has a data element for the class exception and**

If external memory is not empty

then match the external memory with the production memory

else match the working memory to production memory.

Unless an external event has been detected by the APS and the external memory updated by the sensors, it proceeds like any other conventional production system executing the recognise act cycle over the production memory and the working memory.

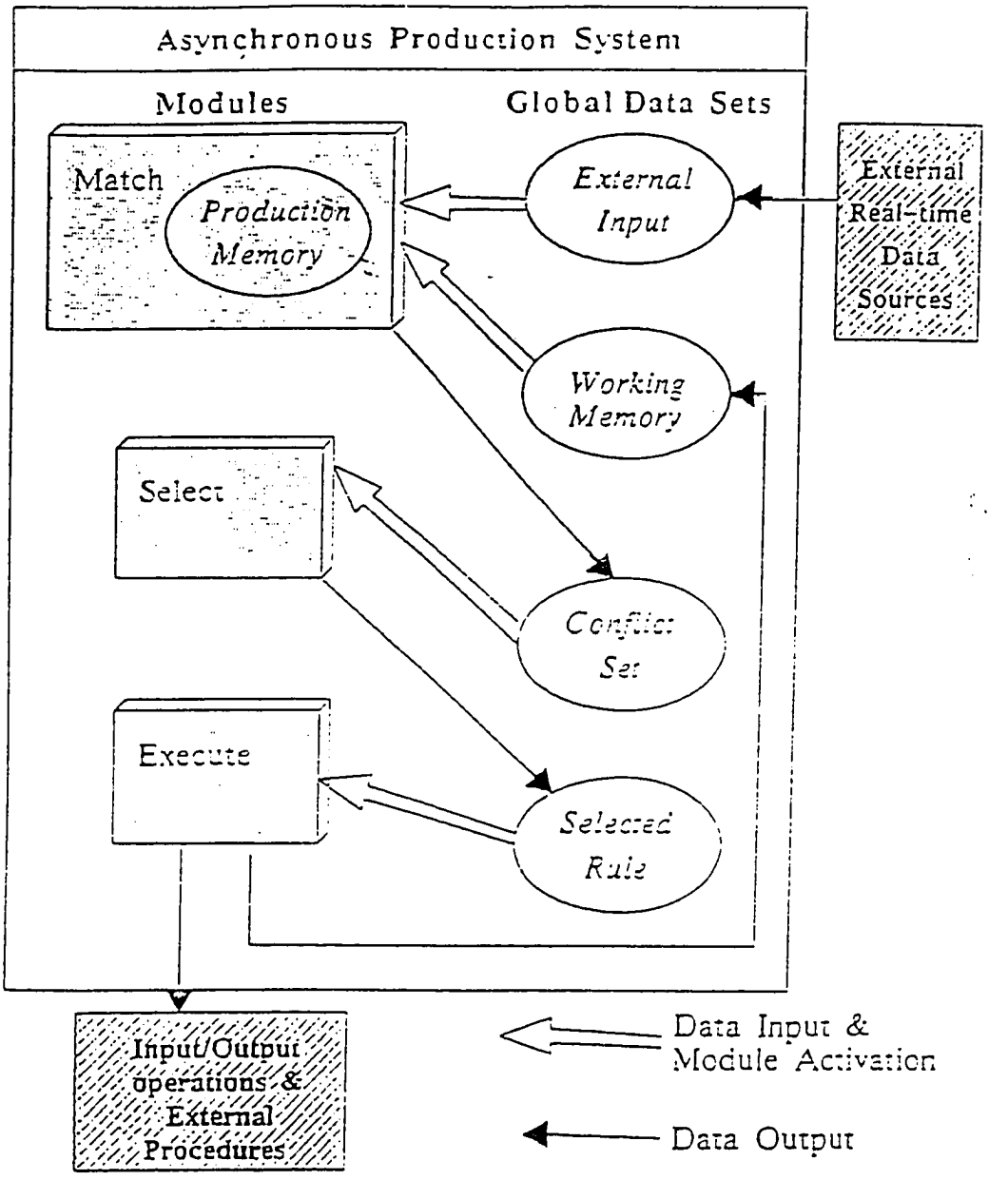


Figure 2.12: Execution Mechanism of the *APS*.

Adopted with permission from Iyengar [40] and others.

ACT **If** external event is detected by transducer and system is *interrupted*

then suspend current activity

and the system state on a STACK like data structure

and transfer control back to the MATCH **module**

and execute the selected rules till external memory is empty

then retrieve saved state and resume regular execution.

The modifications to data may be either to the working memory as in rule error or to the external memory as in rule fire-emergency of figure 2.11. The RHS of an APS rule may invoke external procedures that perform dedicated tasks like *move-robot, activate-siren,.. The ACT module may initiate and carry out its action concurrent to the MATCH module. This module also has the hardware interrupt routines for saving state that are executed as a consequence of an external event.*

The state saving mechanism of the ACT module is similar to the Operating Systems version of a Multiprogramming Environment with preemption of active processes. A significant amount of overhead and system software is required to support this feature but the use of efficient hard wired and microcoded software makes this approach feasible in a real life application. Since sections of APS may run concurrently HERMIES-IIB possess a MIMD (Multiple Instruction Multiple Data) architecture for processing data unlike traditional production systems which are strictly sequential and execute on a single CPU architecture.

APS identified the need for processing real time dynamic data that is required by some production systems applications and also provided a means to process this external data without losing the inherent advantages of traditional rule based systems. The

second advantage of APS is the concurrent mode of operations associated with certain modules which lends APS type of production systems to implementations on today's parallel architectures.

Chapter Three

Features of Production System Programming Languages

3.1. Introduction to Programming Languages

Production Systems programs are rule based programs whose execution flow is determined by the program data. They are controlled by the interaction between knowledge based production memory and the current state in working memory. AI programming languages provide a basic knowledge representation and problem solving methodology that can be applied to solving any rule based problem. A few well known AI languages are list based LISP [48], logic programming oriented Prolog [81] and the LISP interpreted OPS5 [8] language.

LISP is a language whose primary data structure is a list and uses the property list feature as a means of representing facts and knowledge. It lacks in duplicating the inference engine features. Prolog is used mainly to prove theorems and to establish relations amongst objects; data representation is easy and backtracking with cut allows for decomposition of a problem into subproblems and a more efficient search of the problem state space. Prolog uses backward chaining and does not provide the conflict resolution phase that is essential for expert systems.

OPS5 is a programming language that facilitates the creation of an environment for rule based systems, OPS5 written implementations includes large expert systems like McDermott's R1 [49] program for configuring VAX computer, Kahn and McDermott's MUD [42] program for terrain analysis and Kim, Mcdermott and

Siewiorek's TALIB [50] program for Integrated Circuit design.

OPS5 is a LISP based, interpreted, production system language, that is commercially available from Carnegie Mellon University and Digital Equipment Corporation. OPS5 is a language that is used widely to write production system programs and uses elements as the most primitive data type unit for constructing the production system. The working memory is composed of a set of elements that have been defined and instantiated with specific values. OPS5 production rules use the definition of elements to form the LHS as a conjunction of particular conditions. The LHS of a rule is a template generated by common variable binding amongst the condition elements and may have a large set of partial matches from the working memory at any instance, during the entire life cycle of a production system.

The main advantages of OPS5 are it's stability with respect to the production memory and working memory, and efficiency with respect to the inference engine that is based on recency of the elements that match any condition element. The main disadvantages are it's lack of user interface and a limited form of data structures for constructing user defined complex datatypes.

3.2. Elements

The building block of OPS5 language is an *element*. Elements are classified within the program by their names and each element has a set of attributes that may be initialized to numerical values or atomic character strings of length ≥ 1, and by default are presumed to be *nil*. The definitions of elements are used to construct data for the current scenario as working memory elements. Element definitions are also

used for the conditional part of a production rule as condition elements in the LHS of a rule.

The definition of an element is a part of the program declaration section, at the beginning of a program, and is to be found following the string *'literalize'*. The definition is bounded by parenthesis pair as delimiters just like any LISP sentence. An OPS5 example for the definition of an element of class *monkey* with 3 attributes is illustrated in figure 3.1 from the Monkey and Banana program. Figure 3.6 illustrates a complete OPS5 program and the definitions of other element types *goal* and *expression* from an OPS5 based, Mathematical Identities program, named MATH.

```
(literalize monkey    ; representing a memory element class monkey
      at              ; attribute representing horizontal location of monkey
      on              ; attribute representing vertical location of monkey
      holds )         ; attribute representing the type of object held by monkey.
```

Figure 3.1: Definition of an element type.

Elements are similar to the record data structure of Pascal programming language, where the element class corresponds to record name and attributes of elements correspond to fields of the record. Unlike a record, element attributes can have atomic values only, which either is a single character string or a numerical value; no numerical arrays and other nested structured data types are allowed in OPS5. A Pascal language like data structure that is an equivalent form of the OPS5 element in figure 3.1 may be represented as in figure 3.2.

```
type
monkey =  record   /* representing a record class monkey */
      at  : array[1..5]of char /* field representing horizontal location of monkey */
      on : array[1..10]of char /* field representing vertical location of monkey */
      holds  : array[1..10]of char /* field representing  name of object held by monkey. */
       end;
```

Figure 3.2: Pascal equivalent definition of an OPS5 element type.

3.3. Working Memory

Definition of an element sets up the template for the element which may be instantiated to any variable in the working memory. The variables bind values to attributes which are preceded by the cap character or circumflex symbol ^, the values are either alphanumeric constants like *nil* , *floor* or any other valid constant.

working memory elements may be initialized by the RHS of an executing rule and may either be created as a new element or *modify* an existing element's attribute values. A working memory instantiation of the OPS5 element monkey, defined in figure 3.1, may be as follows

(monkey ^at 7-7 ^holds nil ^on floor).

The above instantiation of an element states that there exists an element named monkey with the attributes preceded by ^ and their associated values as *^at = 7-7* , *^holds = nil* , *^on = floor* . The element states that '*there is a monkey on the floor at a location 7-7 and it is holding nothing*'. The value *7-7* is treated as a character string and not as a numerical entity.

Table 3.1: Working memory elements.

(phys-obj ^at 9-9 ^name bananas ^on ceiling ^weight light)
(phys-obj ^at 4-3 ^name ladder ^on floor ^weight light)
(phys-obj ^at 7-7 ^name blanket ^weight light)
(monkey ^at 7-7 ^holds blanket ^on couch)
(goal ^type holds ^status active ^obj-name bananas)
(goal ^type holds ^status active ^obj-name ladder)

Table 3.1 illustrates a part of working memory, containing instantiated data elements of class *monkey, phys-obj,goal* for the Monkey and Banana OPS5 program. Figure 3.6 also illustrates a complete working memory for the OPS5 Mathematical Identities program named MATH.

3.4. Production Rules

Rules form the basis for representing knowledge in production systems, since they specify the given set of conditions to be met in the LHS before it may be executed. The RHS is separated from the LHS by the '-->' operator and the complete rule is enclosed within a pair of parenthesis. In OPS5 a rule begins with the string '*(p* ' and it is identified by the name that follows. In chapter two, figure 2.6 illustrates a production rule named holds::obj-notceil:at-monkey. The components of that same rule are illustrated in figure 3.3.

(p holds::obj-notceil:at-monkey ; PR named *holds::obj...*
(goal ^status active ^type holds ^obj-name <o>) ; CE named *goal*
(phys-obj ^name <o> ^weight light ^at <p> ^on <> ceiling) ; CE named *phys-obj*
(monkey ^at <> <p>) ; CE named *monkey*
-->
(make goal ^status active ^type at ^obj-name nil ^to <p>)) ; RHS for *holds::obj..*
PR is the production rule name.
CE are the condition element names and definitions.
LHS of the rule are the conjuncted condition elements and their attribute values.
RHS is the action specified by the production rule *holds::obj-notceil:at-monkey*

Figure 3.3: Components of an OPS5 production rule.

3.4.1. Condition Elements and LHS of Production Rules

The LHS is a conjunction of condition elements which is a set of conditions that consist of particular instantiations and interacting relationships amongst the elements. Condition elements are not a free, self existing entity like working memory elements as in Table 3.1, but simply a template that the working memory has to match before the rule may be considered as a part of the conflict set. The definition of an element monkey, in figure 3.1 may be used to generate one condition element of the LHS for any production rule in the Monkey and Banana program, as in figure 3.3.

Each condition element in a production rule has attributes that are either constants or variables. Binding amongst condition elements is attained by sharing the variables which are enclosed within angeled brackets or constant attributes. In figure 3.3, condition element named *goal* is defined to be *(goal ^status active ^type holds ^obj-name <o>)* ; the constant attributes are *{ ^status active, ^type holds }* and the

variables, enclosed in angeled brackets is *^obj-name <o>*. The rule in figure 3.3 does not have any constant attributes that are interacting amongst the LHS elements; only the variable attributes are used for determining the logical conjunction. An illustration of variable binding is the variable *<p>* for attribute *^at* which is shared by CE *(phys-obj ^at<p>* and CE *monkey ^at <> <p>)*. Referring back to the production rule illustrated in figure 3.3, it can be seen that the interaction is amongst the attributes that share an existing and instantiated variable name or a constant values.

A condition element may make positive or negative assertions about any of it's attributes. Positive assertions are illustrated in figure 3.3 for attribute *goal ^obj-name , phys-obj ^weight ,....* The assertion may be for variable or constants. Negative assertions are always made by placing <>, indicating negation, immediately before the attribute value. In figure 3.3, CE monkey with attribute *^at <> <p>)* is stating that monkey is not at position <p> and it is bound with the attribute *^at* of CE *phys-obj.* Another illustration of a negative conditions may be found in the CE Iphys-obj and the attribute *^at <> ceiling.*

3.4.2. RHS of Production Rules

The RHS of a production rule is composed of a sequence of actions of the format *action parameter1 parameter2 ...* and is enclosed within a pair of brackets following the '-->' operator of a production rule. For all *actions,* parameters have to be existing working memory elements, except for the *action* that is a **write** command.

The main actions that may be invoked by the RHS of a production rule are of the types { **make, modify, remove, bind, write** }. The purpose and effect of each actions

may be explained as follows:

make This is the means of creating new data elements in the working memory. The element assumes all constant and variable values that are bound by the attributes in the RHS. Production rule *holds::obj-notceil:at-monkey* in figure 3.3 has a RHS stating *(make goal ^status active ^type at ^obj-name nil ^to <p>)* and the variable <p> is bound with the same value as <p> in the production rule. The consequence of the RHS action is to add a new instantiated element of class goal, to the working memory.

```
---------------------------------------------------------------------------------

( p recurse
 {(goal ^from-tower     <from>
       ^to-tower       <to>
       ^other-tower    <other>
       ^number-of-disks { > 1 <n> } ) input-goal }
-->
 (write (crlf) "2prod. " (crlf))
 ( remove input-goal)
 (bind <n-minus-one> (compute <n> - 1 ))
 (..)(..)(..)(..))

---------------------------------------------------------------------------------
```

Figure 3.4: RHS with **bind,remove** actions

remove This action serves to delete existing elements from the working memory which may not be required for subsequent processing. Figure 3.4 illustrates a production rule with a RHS that the action of removing an element named *input-goal* from the working memory.

bind Purpose of the bind action is to store results of a computation in a variable, as illustrated in figure 3.4 where the variable is *n-minus-one* and the computation is *<n> - 1* . Bind may also be used to initialise a variable from the standard input file or keyboard as in the RHS statement *(bind <n> (accept))* where the variable *<n>* accepts the integer initialization interactively from the keyboard.

```
(p    time1x
  {(goal  ^type simplify ^object <n>) <goal>}
  {(expression    ^name <n>       ^arg1  1       ^op  *  ^arg2 <x>) <expression>}
-->
 (modify <expression> ^val   <x>    ^arg1  nil    ^op  nil      ^arg2  nil)
      (write (crlf) "CONGRATS  the goal " <n> " is satisfied  ." (crlf)))
```

Figure 3.5: RHS with **write,modify** actions

modify This action changes the attribute values of an existing element. Figure 3.5 is the OPS5 translation of the mathematical statement in figure 2.3 and the effect of the RHS is to modify the attribute *expression ^val* to the same value as the attribute *expression ^arg2*. The effect of this action is to update an existing element.

write This action prints out the value of a variable or a simple character string message. This is the only RHS action that does not require a parameter as is illustrated in figure 3.4 and figure 3.5. Format control is achieved by the *crlf*

for carriage return and line feed or *tabto* which forces the output to start at specified column number.

Source of data for figure 3.4 is a section of production rule named *Recurse* from the "TOWERS_OF_HANOI" production system. Figure 3.5 illustrates a production rule from the "MATH" production system for multiplication by 1. Both the production systems are written in OPS5, implemented and studied in [70, 71] to derive some empirical results as a part of this research.

3.5. Executing OPS5 Programs

Program execution will be illustrated using a simple OPS5 program named MATH that is listed in figure 3.6. A complete OPS5 production system program will have a declaration section for the elements and a set of production rules to carry out the required task. Figure 3.6 illustrates a small and complete program for the simple mathematical identities of addition, subtraction, multiplication and division. The program has two elements and they are of the type goal, expression and one production rule *start-math* to initialize the working memory and four other rules as production memory rules that may algebraically, be stated as follows:

plus0X: **If** $Y = 0 + X$ **then** $Y = X$;

subX0: **If** $Y = X - 0$ **then** $Y = X$;

mult1x: **If** $Y = 1 * X$ **then** $Y = X$;

divX1: **If** $Y = X / 1$ **then** $Y = X$;

```
(literalize   goal
              type          object )

(literalize   expression
              name arg1  op     arg2  val )


(p start-math { (start) init }
-->
(remove init )
(make goal ^type simplify ^object e1) (make expression ^name e1 ^arg1 1 ^op * ^arg2 99 )
(make goal ^type simplify ^object e2) (make expression ^name e2 ^arg1 88 ^op / ^arg2 1 )
(make goal ^type simplify ^object e3) (make expression ^name e3 ^arg1 0 ^op + ^arg2 99 )
(make goal ^type simplify ^object e4) (make expression ^name e4 ^arg1 88 ^op - ^arg2 0 ))

(p    plus0X  {(goal ^type simplify ^object <n>) <goal>}
  {(expression   ^name <n>      ^arg1 0     ^op  + ^arg2 <x>) <expression>}
-->
  (modify <expression> ^val   <x>    ^arg1 nil    ^op nil     ^arg2  nil)
      (write (crlf) "CONGRATS  the goal " <n> " is satisfied." (crlf)))

(p    subX0   {(goal ^type simplify ^object <n>) <goal>}
  {(expression   ^name <n>      ^arg1 <x>     ^op  - ^arg2 0) <expression>}
-->
  (modify <expression> ^val   <x>    ^arg1 nil    ^op nil     ^arg2  nil)
      (write (crlf) "CONGRATS  the goal " <n> " is satisfied." (crlf)))

(p    mult1X   {(goal ^type simplify ^object <n>) <goal>}
  {(expression   ^name <n>      ^arg1 1     ^op  * ^arg2 <x>) <expression>}
-->
  (modify <expression> ^val   <x>    ^arg1 nil    ^op nil     ^arg2  nil)
      (write (crlf) "CONGRATS  the goal " <n> " is satisfied." (crlf)))

(p    divX1  {(goal ^type simplify ^object <n>) <goal>}
  {(expression   ^name <n>      ^arg1 <x>     ^op  / ^arg2 1) <expression>}
-->
  (modify <expression> ^val   <x>    ^arg1 nil    ^op nil     ^arg2  nil)
      (write (crlf) "CONGRATS  the goal " <n> " is satisfied." (crlf)))

(make start)
```

Figure 3.6: Complete OPS5 program illustrating mathematical identities.

To run an OPS5 program, the user has to invoke the OPS5 interpreter, load the program file and then execute the program as illustrated in figure 3.7.

3.5.1. Runtime OPS5 Commands

Commands typed in by the programmer at the top level, to interact with the interpreter have to be enclosed in square brackets or parenthesis and may be any one of the following set of commands, { **load, run, watch, wm, pm, cs, exit, modify, remove, make, write** } . The first six commands are explained in section 3.5.1 while the last four are the same as the RHS actions of a production rule. The last four commands require parameters to be supplied by the program or the user. Program file for MATH in OPS5 has a last line stating *(make start)* which alternatively could have been typed in at the console; same holds true for other RHS actions.

load This command is used to load in an OPS5 program as illustrated in figure 3.7. The filename has to be preceded with an apostrophe symbol to indicate that the name is a literal value and not an OPS5 element. The interpreter will print an asterisk for every production rule that has been successfully read in. If the program file does not have any syntax errors, the interpreter indicates it by typing the letter 't' after all the rules have been read in.

run The user has the option of executing the program in various integer step sizes or of letting it run to completion. If an integer is specified with the **run** command, the interpreter sets a *break* point in the program execution after the specified number of rules have been executed. Figure 3.7 illustrates an

example where the program is halted after 1 rule has executed and also the use of *[run]* to completion.

watch This command is similar to the *trace* feature available in most programming environments. The user may set the interpreter to trace at levels 0, 1, 2 or 3. At level 3 user is informed of the conflict set, working memory changes, name tags of elements that match the rules and name of the production rule that is being fired. Level 3, 2 and 1 provide less information while 0 provides none at all. The default value for this command is 1. All changes to the working memory are indicated by the '<=' operator indicating old data that has either been removed or modified and '=>' indicating new data. OPS5 uses recency to resolve any conflicts which is implemented by a monotonically increasing integer value for any working memory element as illustrated in figure 3.6 for elements 10 and 12 of the type *expression*. The integers are used as names tags associated with the rule that is being executed and identify those elements from the working memory that have satisfied the production rule.

wm,pm While **watch** provides only the changes to the environment, these commands will provide complete information regarding working memory and production memory respectively. Break points have to be set in the program execution by the **run** command before any of these commands may be executed. In this class of commands, the only command that requires a parameter is **pm** and it needs the name of a production rule that the user wishes to see at execution time from the production memory as illustrated in figure 3.7

```
> ops5
-> [load 'math.idn]
[load math.idn]
*****t
-> [pm divX1]
(p divX1
  {(goal ^type simplify ^object <n>) <goal>}
  {(expression ^name <n> ^arg1 <x> ^op / ^arg2 1) <expression>}
 -->
  (modify <expression> ^val <x> ^arg1 nil ^op nil ^arg2 nil)
  (write (crlf) CONGRATS the goal <n> is satisfied. (crlf)))
=>wm: 1: (start)nil
-> [cs] start-math (start-math dominates)
-> [run 1]
1. start-math 1
***break***
<=wm: 1: (start)
=>wm: 3: (goal ^type simplify ^object e1)
=>wm: 5: (goal ^type simplify ^object e2)
=>wm: 7: (goal ^type simplify ^object e3)
=>wm: 9: (goal ^type simplify ^object e4)
=>wm: 4: (expression ^name e1 ^arg1 1 ^op * ^arg2 99)
=>wm: 6: (expression ^name e2 ^arg1 88 ^op / ^arg2 1)
=>wm: 8: (expression ^name e3 ^arg1 0 ^op + ^arg2 99)
=>wm: 10: (expression ^name e4 ^arg1 88 ^op - ^arg2 0)nil
-> [cs] mult1X divX1 plus0X subX0 (subX0 dominates)
-> [run 1]
2. subX0 9 10
CONGRATS the goal e4 is satisfied.
***break***
-> [wm]
<=wm: 10: (expression ^name e4 ^arg1 88 ^op - ^arg2 0)nil
=>wm: 12: (expression ^name e4 ^val 88)nil
-> [cs] mult1X divX1 plus0X (plus0X dominates)
-> [run]
3. plus0X 7 8
CONGRATS the goal e3 is satisfied.
4. divX1 5 6
CONGRATS the goal e2 is satisfied.
5. mult1X 3 4
CONGRATS the goal e1 is satisfied.
end -- no production true
 5 productions (26 // 38 nodes)
 5 firings (18 rhs actions)
 7 mean working memory size (8 maximum)
 2 mean conflict set size (4 maximum)
 5 mean token memory size (8 maximum)
-> [exit]
```

Figure 3.7: Tracing the execution of an OPS5 program.

for the rule *divX1*. The command **wm** gives the current contents of the working memory along with the recency associated with each element.

In figure 3.7, the recency associated with each element is defined as integers preceding the elements and represents the temporal order in which they were created or modified. The recency ordering is increasing but there are counts that will be missing as illustrated in figure 3.7 for counts *1 (start)* to *3 (goal ...)* and again for *10 (expression ...)* to *12 (expression ...)*; this is explained by the RETE network implementation that OPS5 uses, in chapter four. OPS5 allows for multiple instances of the same data element to be existing in the working memory and there may be an application that requires multiple copies of data with same values but separate recencies.

cs Command **cs** is used to see the contents of conflict set which contains all the currently satisfied production rules and also name of the next production rule that has been selected by the inference engine to be executed. The consequence of executing **cs** command, for the MATH program, is the interpreter displaying conflict set and selected production rule which are represented as *mult1X, divX1, plus0X,* and *(plus0X dominates)* respectively. This is a snapshot of the inference engine results after execution of rule *subX0* in figure 3.7 and the first break point in the program.

exit This command terminates the OPS5 execution session, and will return the programmer to the system level. In case of an execution error due to wrong input or a typographic error in the interactive mode, the interpreter provides Ctrl-d to get back to the top level in the interpreter.

The production system program performance is tabulated at the end of execution. The program diagnostics will provide numerical information regarding program size in terms of the number of rules forming program production memory or the number of rules, which is a static quantity and other dynamic entities like the number of RHS actions, average and maximum size of working memory and conflict set and etc. as illustrated in figure 3.7. This information is extremely helpful for parallelizing any production system as will be illustrated in subsequent chapters.

Chapter Four

Sequential Execution of Production Systems

4.1. Introduction

We have discussed that production systems will cycle through the recognise act phase till the problem has been resolved or the inference engine module of the production system determines that the given problem is unsolvable. The MATCH phase of a production system is its most computationally intense feature and some estimates have placed the time spent during this matching of working memory elements with the production rules in the at 90% [24] of the total execution life. Since a fast recognise act cycle is the basis of real time efficient productions systems, several researchers have worked on the problem of improving the MATCH phase. The search mechanism has been partially improved by Forgy [24] in his RETE algorithm by implementing an indexed tree for the production memory and retaining information about all partial matches too. The MATCH phase was parallelized in different manners by Miranker, Stolfo et.al. [53, 74, 75, 76] and Gupta et. al. [29, 30, 31, 32, 33] in their respective research works.

The RETE algorithm is an approach that computes the conflict set without having to iteratively evaluate each production rule over the working memory during the MATCH phase of an executing production system. The algorithm generates and maintains an *n-ary* tree like indexed network as illustrated in figure 4.1 for a partial subset of production rules from the Monkey and Banana program. In a RETE

network n is the number of condition elements in the production system and determines the number of descendents to the *ROOT* node.

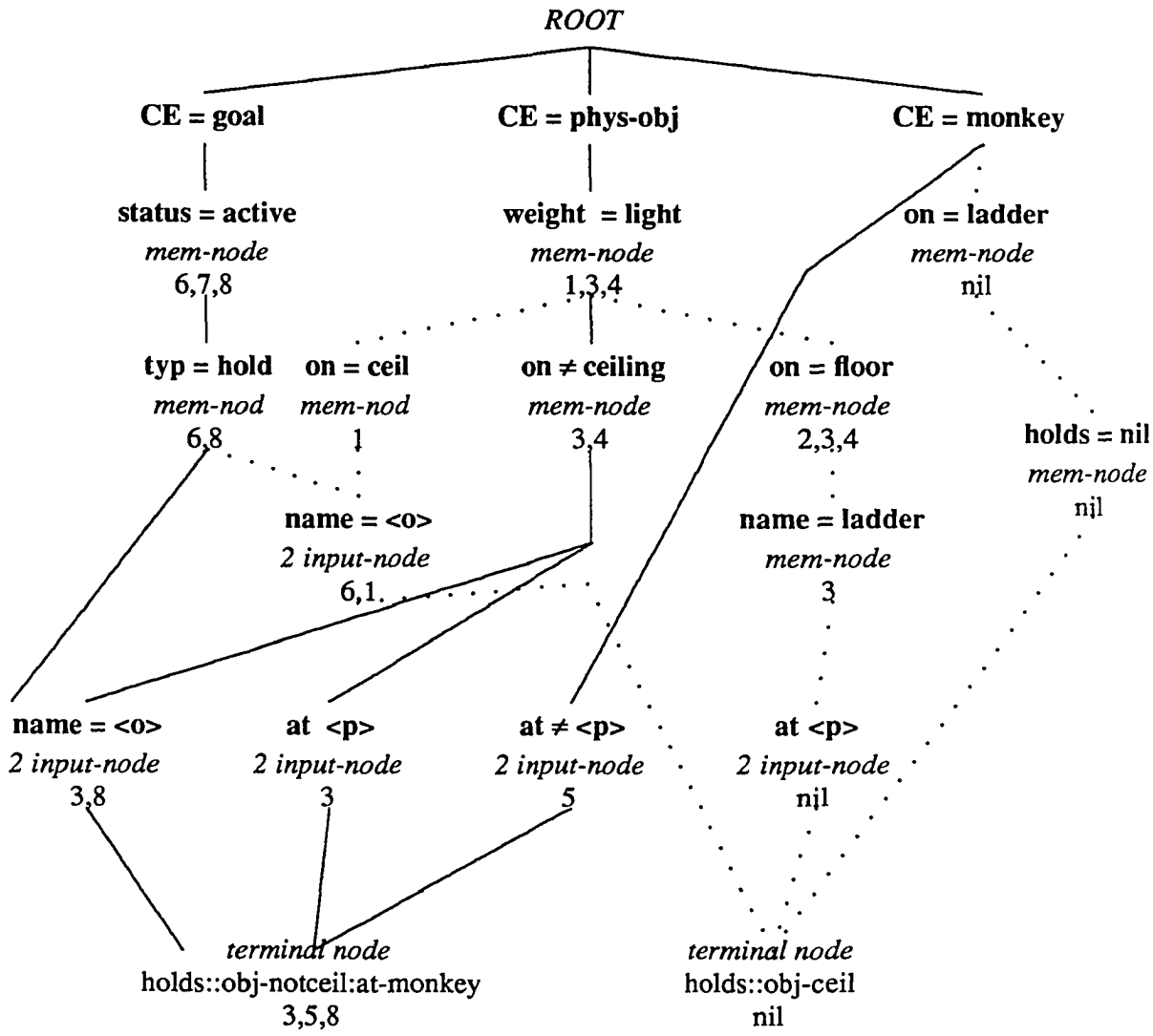


Figure 4.1: RETE network configuration.

The network illustrated in figure 4.1 is for two OPS5 production rules "holds::obj-ceil" and "holds::obj-notceil:at-monkey", that are indicated as terminal nodes to the tree. The data for the network and composing the working memory elements are listed in table 4.1. Since only two rules were considered for the illustration, figure 4.1 is a small subset of the actual RETE network that would be generated by OPS5 while executing the program.

Table 4.1: Data in working memory for figure 4.1.

1	(phys-obj ^at 9-9 ^name bananas ^on ceiling ^weight light)
2	(phys-obj ^at 7-7 ^name couch ^on floor ^weight heavy)
3	(phys-obj ^at 4-3 ^name ladder ^on floor ^weight light)
4	(phys-obj ^at 7-7 ^name blanket ^weight light)
5	(monkey ^at 7-7 ^holds blanket ^on couch)
6	(goal ^type holds ^status active ^obj-name bananas)
7	(goal ^type at ^status active ^obj-name ladder ^to 9-9)
8	(goal ^type holds ^status active ^obj-name ladder)

Table 4.1 is a snapshot of the complete working memory during an actual execution of the program using **wm** command and is containing data elements 3,5 and 8 that MATCH all the conditions in the LHS of production rule holds::obj-notceil:at-monkey. The network in figure 4.1 illustrates partial matches of memory elements to condition elements too, specially for antecedents of the rule holds::obj-ceil, which is found to be unsuccessful during the recognise phase. To trace a successful MATCH of

a production rule the data has to flow from the ROOT node and end at any terminal node; in figure 4.1 the successful path to rule holds::obj-notceil:at-monkey is indicated by a solid line whereas the dotted line represents an unsuccessful path to the rule holds::obj-ceil. Source of data is [8, 70, 71] and is based on the OPS5 version of the monkey and banana production system.

4.2. RETE Network Implementation

A RETE network translates the LHS of a rule into an indexed tree like structure and provides a means for incorporating all the condition element features of a LHS that were discussed in section 3.4.1.

```
---------------------------------------------------------------------------------------

(p holds::obj-notceil:at-monkey
   {(goal ^status active ^type holds ^obj-name <o> ) <goal>}              LHS
   {(phys-obj ^name <o> ^weight light ^at <p> ^on <> ceiling ) <object1>}
   {(monkey ^at <> <p> )    <monkey>}
-->
   (make  goal ^status active ^type at ^obj-name nil  ^to <p> ))     RHS

(p holds::obj-ceil
   {(goal ^status active ^type holds ^obj-name <o1> ) <goal>}       LHS
   {(phys-obj ^name <o1> ^weight light ^at <p> ^on ceiling ) <object1>}
   {(phys-obj ^name ladder ^at <p> ^on floor ) <object2>}
   {(monkey ^on ladder ^holds nil )  <monkey>}
-  (phys-obj ^on <o1>)
-->
   (write (crlf) "Grab " <o1> (crlf))
   (modify <monkey> ^holds <o1> )
   (modify <object1> ^on nil)
   (modify <goal> ^status satisfied ))                              RHS

---------------------------------------------------------------------------------------
```

Figure 4.2: Production rules to construct RETE network in figure 4.1.

Figure 4.1 is a RETE representation for the LHS of the rules illustrated in figure 4.2 from the Monkey and Banana production system. The two rules in figure 4.2 will be used to illustrate the principles behind the RETE network in figure 4.1. The RETE network is constructed or compiled by the following sequence of steps.

- With each production rule, determine the intra-element features for LHS pattern of an element and build a linear sequence of nodes representing that condition element as illustrated in table 4.2 and 4.3 for the rules in figure 4.2.

Table 4.2: Intra-element features of rule holds::obj-notceil:at-monkey.

#	goal	phys-obj	monkey
1	^status active	^name <o>	^at <><p>
2	^type holds	^weight light	
3	^obj-name <o>	^at <p>	
4		^on <> ceiling	

Table 4.3: Intra-element features of rule holds::obj-ceil.

#	goal	phys-obj	phys-obj	monkey
1	^status active	^name <o1>	^name ladder	^on ladder
2	^type holds	^weight light		^holds nil
3	^obj-name <o1>	^at <p>	^at <p>	
4		^on ceiling	^on floor	

These intra-element features correspond to representation of individual condition elements within a production rule. The common intra-element features within a rule for different condition elements will be fused in the inter-element stage. The network is finally constructed by sharing the common features amongst production rules.

- Inter element features are those attributes values that are shared by more than one condition element. With each production rule, determine the inter-element features and build two input nodes, where the inputs are either joining two paths as linear sequences or passing the output of the previous nodes to a third pattern. Table 4.4 and table 4.5 identify these features for the two rules in figure 4.2.

Table 4.4: Inter-element features of rule holds::obj-ceil.

#	goal	phys-obj	phys-obj	monkey
1	^obj-name <o1>	^name <o1>		
2		^at <p>	^at <p>	
3			^name ladder	^on ladder

Table 4.5: Inter-element features of rule holds::obj-notceil:at-monkey.

#	goal	phys-obj	monkey
1	^obj-name <o>	^name <o>	
2		^at <p>	^at <><p>

The inter-element features are the attributes that determine the conjunction of conditions which have to be met by two or more working memory elements. The binding amongst elements may involve atomic values as in the case of *phys-obj ^name ladder* and *monkey ^on ladder* for the rule *holds::obj-ceil* in table 4.5 or variables as is illustrated in table 4.4 for elements *goal ^obj-name <o>* and *phys-obj ^name <o>* for the rule *holds::obj-notceil:at-monkey.*

- With each production rule determine the name of the rule and attach a special node to the last of the two-input nodes in the tree which represents the termination of that particular LHS pattern. The number of terminal nodes in any RETE network will always be the same as the number of OPS5 rules in that production system. Number of rules in a production systems is the first program performance measure that is available to the user at the end of an OPS5 program execution session, as illustrated in figure 3.7, for a program MATH with 5 production rules.
- When two LHSs of a production rule need some identical nodes, the pattern compiler shares part of the network rather than building duplicate nodes. In figure 4.1 the common nodes amongst the two production rules are { **CE = goal, status = active, typ = hold** }, { **CE = phys-obj, weight = light**} and the sharing of the element { **CE = monkey**} right at the top of the network and is illustrated in figure 4.3.

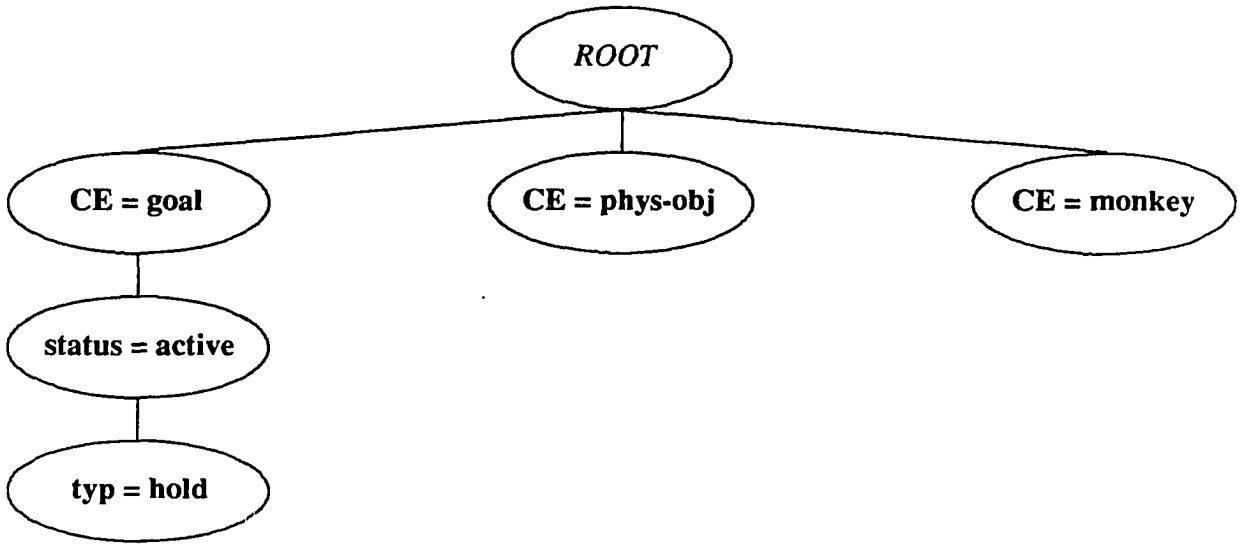


Figure 4.3: RETE network sharing common nodes amongst rules.

Following this algorithm of sharing nodes, OPS5 reduced the actual number of LHS nodes from 38 to 26 by implementing the RETE algorithm for the production memory, as illustrated in figure 3.7 for the MATH production system. This process of sharing nodes reduces the actual number of comparisons that need to be done in the MATCH phase of a system and makes RETE a more efficient form the pattern matching algorithm.

4.2.1. Types of Nodes in RETE Network

The RETE algorithm translates a production memory into a tree like structure that simplifies the MATCH phase. Disregarding the *ROOT* node, any RETE network

has a maximum of four different types of nodes [24,31] as illustrated in figure 4.1, which are {*Constant-test nodes, Memory-nodes, Two-input nodes, Terminal-nodes*}.

1. *Constant-test nodes:* Nodes where the attributes are tested for having constant values. For example *on = ladder , type = holds ...* in figure 4.1. These nodes are always a subset of the intra-element features, that were determined by the algorithm as illustrated in tables 4.2 and 4.3.
2. *Memory-nodes:* Nodes that retain the results of the match from the test node directly preceding it, denoted as *mem-node* the RETE network of figure 4.1 and are also known as the α-*memory* nodes. The set of elements that satisfy the test node *status = active* is {6,7,8}, as illustrated in figure 4.1. Table 4.1 associates the element's recency tags with their respective definitions in the working memory.
3. *Two-input nodes:* Nodes that test for variable bindings amongst the LHS condition elements of a production rule. These are also known as β-*memory* nodes. The network maintains common nodes for those attributes of condition elements that are shared amongst more than one condition element as intra-element features or production rule and are always variables. The associated *two-input node* will have entries for a successful match of an attribute variable that is shared amongst the condition elements and satisfy all the left and right antecedents of that node in the network. For example in figure 4.1, one such node that tests the variable value of the attribute **name = <O>** for **CE = goal** and **CE = phys-obj** results in {3,8} being stored at the corresponding *two-input node* where *memory element 8*

satisfies all the left antecedent for **CE = goal** and *memory element 3* satisfies all the left antecedent for **CE = phys-obj.** A network may have multiple two-input nodes that check for the same attribute values but have entirely different antecedents; this is illustrated in figure 4.1 for the node **name = <o>** with different right antecedents for the attribute test node **on = ceil** and **on ≠ ceil.**

4. *Terminal-nodes:* The RETE network has one special node for each rule in the system. A successful match for a rule would be detected by the flowing of a set of tokens to the terminal node, where the tokens represent the working memory elements that satisfy the LHS of that particular rule. The rule holds::obj-notceil:at-monkey is satisfied by elements 3,5 and 8, on the basis of working memory in table 4.1 as illustrated in the network of figure 4.1.

4.3. Program Execution in RETE Network

The RETE network has a static structure for the production memory and is constructed at compilation time. After the RETE network has been formed, processing of the working memory to generate the conflict set is done by passing *tokens* through the network. The dynamic part of the RETE network is the α-memory and β-memory where information regarding the partial matches of elements is maintained in the tree.

tokens Description of each working memory element is passed to the RETE network in the form of a token which consists of a tag and a list of data elements. The tag is either ' + ' indicating the addition of an element to the working memory or ' - ' indicating deletion of an element from the work-

ing memory. The RHS action of production rule named *holds::obj-notceil:at-monkey --> (make goal ^status active ^type at ^obj-name nil ^to <p>)* in figure 4.2 would create a token

<+ (goal ^status active ^type at ^obj-name nil ^to 7-7)>.

Any modification to an element in the working memory causes two tokens to be sent to the RETE network one for deleting the old form of the element and one for creating the new form of the element. The RHS action of production rule named *holds::obj-ceil --> (modify <goal> ^status satisfied)* in figure 4.2 would create a pair of tokens *<- (goal ^status active ^type at ^obj-name nil ^to 7-7)>. <+ (goal ^status satisfied ^type at ^obj-name nil ^to 7-7)>.*

This would explain the missing recency values to working memory elements in figure 3.7 for the execution of MATH program in an OPS5 environment.

class In the RETE network, copies of the token are distributed to all successors of the ROOT node in the network which constitute the various element classes in the production system. These nodes have one input each and each node will test one feature of the intra-element pattern. The token passes on to the child nodes if the match was a success. The child node may either be a one input or a two input node or to both but all the nodes have to be successors to the initial node.

memory The information about pattern matching is stored at the two input nodes. The memory stores copies of the token arriving at that node until both the

left and right tokens have arrived and then a new token is created with the same tag as the token which was the last to arrive at that node. A ' - ' tag causes the deletion of that data element from the local memory of that node while a ' + ' tag causes a creation of that data element at that node. This mode of local memory implementation enables the user to access information about partial pattern matching too.

terminal The terminal node decides whether an *instantiation* of a successful match is to be added or deleted to the conflict set depending on the value of the tag attached to the token.

4.4. Evaluation of RETE

4.4.1. Advantages

The RETE algorithm removes the necessity of iterating over the set of production rules by using a tree-structured sorting network which indexes onto the patterns found in the LHS of the production rules. Any change in the working memory is transformed into tokens which are distributed via the ROOT through the network. This results in two type of memory that the algorithm has to maintain. The first α-*memory* is that part of the network which retains the results from the constant-test nodes and β-*memory* [29, 53] retains results from the two-input nodes.

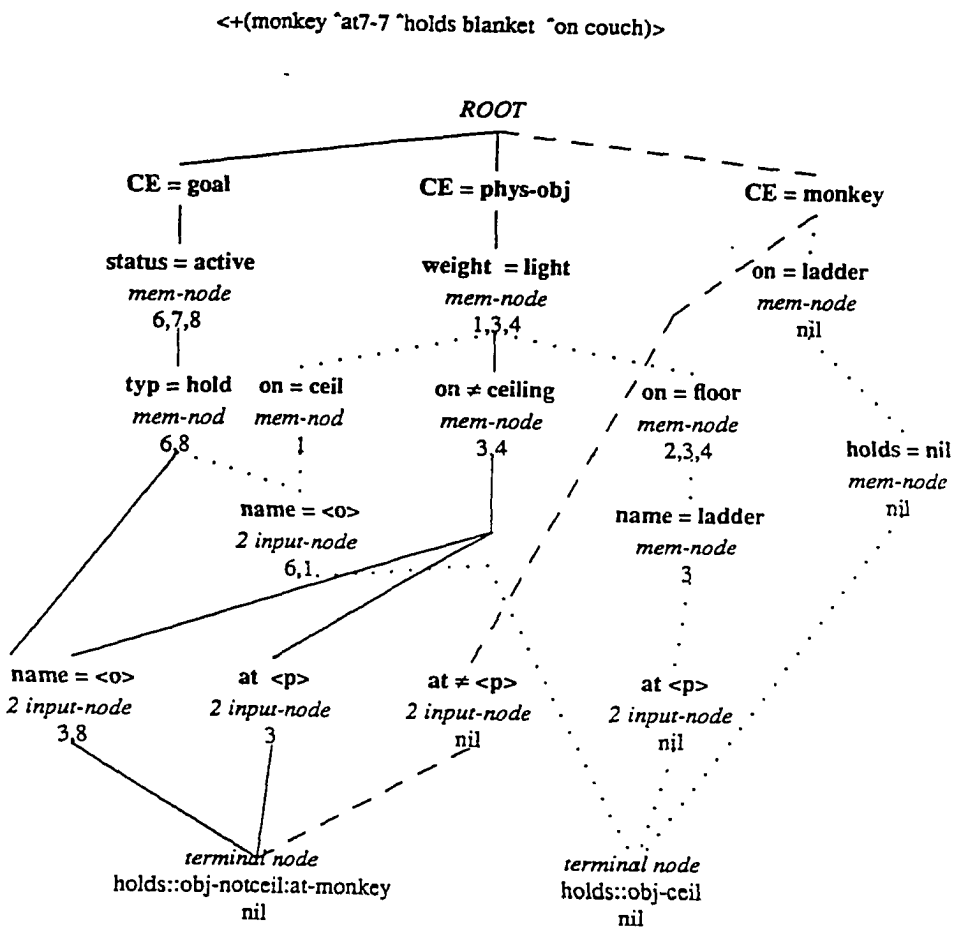


Figure 4.4: Token flow in a RETE network.

RETE minimized the matching phase of the elements in the elements by con-

structing the search network and maintaining state of partial matches in α-*memory* and β-*memory*, wherein space is traded for sake of speed. Figure 4.4 illustrates an instance where the addition of a token to the network uses the partial state information that is stored in the network to recompute the conflict set. The token <+(monkey ^at7-7 ^holds blanket ^on couch)> will flow through the tree and using the partial matches already stored in the memory nodes, thereby resulting in the final form of the network as illustrated in figure 4.1.

4.4.2 Disadvantages

There are two major limitations in executing traditional production systems written in OPS5 language that implement RETE algorithm. The first deals with the time spent by the production system as a consequence of the act phase where the working memory is modified. The disadvantage is apparent during the deletion or modification of an element wherein the partial matches in the network have to be rewound [53] which may lead to memory contention and communication problems in a parallel environment. Consider a RHS action of the type *(modify goal ^status satisfied)* for condition element 8 working on the RETE network in figure 4.1. This action would mean deleting the entry for element 8 in all the descendents of the α-*memory node* **status = active** ; therefore, on an average, time gained in saving partial matches is lost in deleting the partial matches within the network.

The second problem associated with production systems is the combinatorial explosion of the β-memory . Constant test nodes are relatively few but the two-input nodes are many and there may be multiple instances of the same node for different

antecedents. Consider the node **name = <o>** in figures 4.1 and 4.4, which may be logically represented as

For elements $a, b \mid \forall\ a \in$ *LEFT ANTECEDENT*, $\forall\ b \in$ *RIGHT ANTECEDENT*

if a^name = b^name then store a, b in associated two input node.

This form of variable binding could potentially lead to a massive β-*memory* since RETE saves all partial matches and the same condition test may have different LEFT antecedents or RIGHT antecedents or both antecedents may be different. A good illustration from figure 4.1 is the duplication of the two-input node, **name = <o>** for two separate RIGHT antecedent conditions involving the tests **on = ceiling** and **on ≠ ceiling** , but sharing the LEFT antecedent. When the complete network is formed for all the production rules, there is a large amount of redundant information that is stored in the network for determining a single successful match.

These limitations of the RETE algorithm are the major research issues in subsequent work by Gupta et. al. at the Carnegie Mellon University, Miranker et. al. at the Columbia University and Iyengar, Shrivastava et. al. at the Louisiana State University. The solution to a faster match was an indexing algorithm that is not based on a data flow architecture but high performance data structures and a novel knowledge representation scheme for goal driven, forward chaining production systems.

Chapter Five

Parallelism in Production Systems

5.1. Parallelism in General

Iyengar et. al. have been involved in the area of parallel algorithms, search optimization techniques and data [3, 10, 16, 54, 55, 39, 64] structures for quite some time. The ability to generalize a technique to exploit maximal parallelism in the execution of production systems is the focus of this research. This chapter identifies the various aspects of parallelism in general and for production systems in particular. Subsequent chapters in this dissertation formulate general theories for parallel computation of any rule based system.

Parallelism is defined to be the existence of some modules within a program such that they are capable of executing independently of each other and also concurrently with consistent and correct results. Modern day hardware is available with multiple CPU computers, therefore a significant proportion of todays research is geared towards parallel versions or translations of existing implementations.

Operating Systems theory provides a set of conditions that need to be satisfied by any two modules before they may execute concurrently. The three conditions are known as *Bernstein Conditions* [6] must be satisfied for two successive modules S_1 and S_2 to be executed concurrently:

[1] $R(S_1) \cap W(S_2) = \varnothing$

[2] $W(S_1) \cap R(S_2) = \varnothing$

[3] $W(S_1) \cap W(S_2) = \varnothing$

Where $R(S_1)$ is the read set of S_1 and consists of all the variables that are read by the module. Similarly $W(S_2)$ is the write set and is the set of variables that are written to by the module. Bernstein's conditions states that the intersection of these two sets should be a NULL set.

Example 1 We consider a concurrent program to find the maximal value of X, Y, Z, and R.

```
1    parbegin
2        A := max (X, Y);
3        B := max (Z, R);
4    parend
5    C := max (A, B);
```

Since statements 2 and 3 are disjoint (independent), we have no difficulty understanding the program. The statements 2 and 3 are executed in parallel.

Example 2

Consider the following program segment that reads information from two different peripheral devices (such as tapes) and then writes information into a third peripheral device.

```
i:   read (a);
```

```
j:    read (b);
k:    c := a + b;
l:    write (c);
```

Suppose we want to execute some of these statements concurrently. Clearly, the statement *k* cannot be executed before both reads have completed. Similarly, *l* cannot be executed before the new value of *c* has been computed. On the other hand, the statements *i* and *j* could be executed concurrently.

To allow the concurrent execution of the two read statements, this program could be rewritten as follows.

```
parbegin
        read (a);
        read (b);
parend
c := a + b;
write (c);
```

5.2. Parallelism in Production Systems

Material discussed till now has identified that a production has three main active modules which are *MATCH, INFERENCE ENGINE, ACT*. In this chapter an attempt is made at exploiting any and all aspects of parallelisms in the modules. The MATCH phase has inherent parallelism built in its iterative nature of the condition matching with the data. This was amply illustrated in the Pascal like form of the OPS5 written

production rule in figures 2.6 and 2.7. owever all three modules exhibit some levels of parallelism and this chapter is devoted to identifying the various levels and aspects of parallelism that any program designer may be able to exploit in production systems.

5.2.1. Parallelism in MATCH phase

The MATCH module is computationally the most expensive hence most research activities are related to improving this aspect of production systems. Table 5.1 provides a graphic illustration to the above statement and on an average it has been conjectured by Forgy [24] that this matching of working memory elements with the production rules is approximately at 90% of the total execution life. This is found to be true for those production systems that have either a large production memory or a large working memory.

Table 5.1: Sample computational statistics on some production systems.

production system	match	select	act
waltz	86%	1%	13%
mapper	53%	1%	46%
monkey	59%	< 1%	41%
mesgen	76%	10%	14%
mud	94%	<1%	5%

The MATCH phase may be considered to be a form of the *decomposable search* [5, 16] class of problems. These class of problems may exploit the parallel architectures to implement the search in parallel on a static search space that is decomposed

into elementary segments. The membership problem for a set is a typical decomposable search problem where the set may be partitioned into smaller segments. Production systems fall in a class known as almost decomposable search problems since the search space is not static. The search space is static over the period of one cycle only during the recognise act cycle.

Analytical studies have indicated two forms of parallelisms in the MATCH phase and are known as *rule* or *associative* parallelism.

RULE: Rule parallelism is based on a parallel match of all the rules to a single data element. This is a form of MISD, an acronym for Multiple Instruction Single Data, mode of operation for a production system during the match phase. This form of parallelism is incorporated in the RETE network as is illustrated in figure 4.1, even though RETE itself is not implemented on a parallel architecture due to the communications bottlenecks in a tree.

ASSO: Associative parallelism is the DADO implementation of TREAT algorithm where one rule is associatively matched with all the data elements stored at the lower level of the DADO tree machine as is illustrated in figure 5.2. This is a form of SIMD, an acronym for Single Instruction Multiple Data, mode of operation for the DADO subtree with the PE containing the rule and initiating the broadcast on the communication bus as the ROOT node for the subtree.

Both *rule* and *associative* parallelism are based on fine grained architecture and are not feasible on medium grained architectures like the MIMD hypercube comput-

ers. Hypercubes or any other medium grained parallel machine will have to implement a cluster and partitioning algorithm for the production memory and working memory.

5.2.2. Parallelism in Conflict Resolution phase

As compared to the MATCH phase, production systems spend a small portion of time in the conflict set resolution phase, as is illustrated in figure 5.1. Consider the conflict set that is generated by the OPS5 MATH program in figure 5.7 { plus0X:1,2 subX0:9,10 mult1X:5,6 divX1:7,8 }, which is a relatively small set. The small size of the conflict set, is the main reason for a lack of research in parallelizing the conflict set since the selection of a rule from the set is relatively easy and is either based on recency for OPS5 deterministic systems, or some degree of probabilistic measures for non deterministic production systems as were discussed in chapter 2.

5.2.3. Parallelism in ACT phase

Traditional production systems have rules with RHS actions that affect working memory contents as illustrated in figure 4.2 and figure 3.6. A partitioning of the working memory, in homogeneous blocks of data, makes it easier to carry out the modifications to the data and does not require a parallel implementation of the ACT module.

APS [65, 66] identified RHS actions that may be clustered as groups of Input/Output operations, modifications to working memory, modifications to external memory and actions specified by the rule to be executed as external routines.

```
(p error
     LHS
-->
     (write (crlf)" Robot is not at Starting Position."(crlf))
     (modify <goal> ^status satisfied))

(p navigate
     LHS
-->
     (move-robot(<s>, <d>))
     (modify <goal> ^status satisfied))

(p fire-emergency
     LHS
-->
     (activate-siren)
     (modify <exception> ^status satisfied))
```

Figure 5.1: RHS Actions of some production rules for HERMIES-IIB *APS*.

Referring back to the sample set of production rules for *APS* based program for the *HERMIES-IIB* robot in figure 2.11, the parallel modules for the RHS of the rules are illustrated in figure 5.1 and may be defined as follows :

[1] *Input/Output* module to carry out the operations like *{read,accept, write}* for communicating with the user and external world.

[2] working memory module to modify the changes to conventional data as specified in rule *navigate, error.*

[3] *External memory* module to modify the changes to external data as specified in rule *fire-emergency.*

[4] *External routines* module to carry out the physical actions specified by the rule like *{move-robot, activate-siren}* in rules*navigate, fire-emergency* respectively.

5.3. Related Research Work

Including the research reported in this dissertation, there have been four major contributions to the filed of optimizing production system performance in the MATCH phase.

1. Forgy et. al. at Carnegie Mellon University developed the RETE match algorithm and it has been studied in this dissertation as a sequential production system with an efficient MATCH algorithm.
2. Gupta et. al. at Carnegie Mellon University developed the *Production System Machine* also known as the PSM *shared memory* computer, for parallel execution of production systems. Gupta carried out an extensive analysis of various parallelism aspects in all phases of production systems and proposed an architecture scheme for exploiting these parallelism features.
3. Miranker et. al. at Columbia University used the DADO architecture and fine grained parallelism with special hardware connection components to exploit the parallelism in the MATCH phase. Miranker disregarded the RETE network as a parallelisation model and proposed special high structuring of the data for efficiency in speed and memory storage.
4. Iyengar et. al. at Louisiana State University, have proposed an approach based on medium grained parallelism and exploiting the available hypercube architecture as

a computational model. This group also identified the *APS* class of production systems for realtime and robotics applications. This dissertation states some of the results obtained by this research group in the field of MATCH optimization.

5.4. Parallel Architectures for Production Systems

There are three types of parallel architectures, RISC based shared memory MIMD with *hardware task scheduler* at CMU for the PSM, DADO at Columbia and shared memory hypercubes that may be used for parallel execution of production systems.

5.4.1. The PSM Machine

This is a special purpose machine simulated at the Carnegie Mellon University with a multiprocessor and shared memory approach. The machine based on the research by Gupta et. al., exploits *node* and *intranode* parallelism and requires a special *hardware task scheduler* feature, to obtain the maximum concurrent execution from the RETE network. The machine executes the RETE algorithm in parallel with a central *task scheduler* for dynamic load balancing techniques. The RETE network node has the capability of initiating an activity and enqueuing it on the globally shared task scheduler. The PSM machine is a fine grained multiple processor computer that emulates a parallel data flow architecture for the RETE MATCH algorithm. The PSM machine was simulated for the research done by Gupta et. al. and is reportedly under construction at CMU.

5.4.2. The DADO Machine

DADO is a small grain, highly parallel tree-structured machine [52, 76, 77, 78] designed to implement rule based systems at Columbia University by Stolfo and colleagues. The machine is a microprocessor based architecture, interconnected to form a completely balanced binary tree. Each processing element or PE as it is commonly called, has its own RAM memory of the order 20K. The processor for the DADO and DADO2 prototypes is the Intel 8751 chip, an 8 bit microprocessor. The chip has a 4K EPROM memory and 4 parallel, 8 bit ports for I/O that facilitate a tree like connection. Each PE has a special IO switch chip for high speed communications and connects it to the processor's parent, left child and right child. This connection of nodes does not hold true for the ROOT and terminal nodes of the DADO machine.

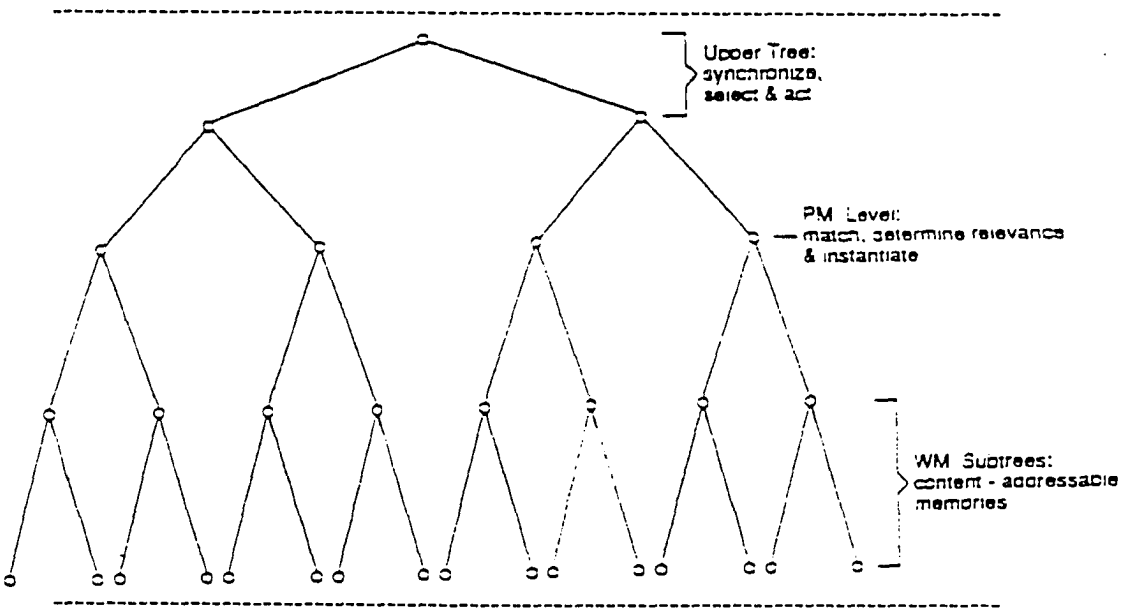


Figure 5.2: Processing Elements as DADO machine components.

The DADO machine is constructed as a multi layered binary balanced tree, the tree may have thousands of nodes and at each node is a processing element of type

illustrated in figure 5.2. The ROOT processor is a special processor, called the HOST, that controls the operations of the entire DADO machine. Each PE within the tree has capabilities of operating in the Single Instruction Multiple Data or Multiple Instruction Multiple Data mode, henceforth referred to as SIMD and MIMD respectively. Each PE has its own data in both modes.

In SIMD mode the PE will execute instructions that are placed on the communication bus by some ancestor PE node. In MIMD mode, the node effectively disconnects itself from the tree and enables that PE to execute any routine that is available in the local memory. Such a PE in MIMD mode may however broadcast instructions to its descendants which will be executing in the SIMD mode. Figure 5.2 illustrates a DADO machine that may be used for executing rule based programs.

Figure 5.3: DADO machine structure for production systems.

The DADO machine uses three levels to implement parallel production systems. The top level is used by the inference engine and for synchronization purposes. The second level is the production memory level. Each PE at this level contains a fraction of the original set of rules and the fraction is determined by the PE memory size. The number of PEs used at the second level is 2^N for some $N \geq 2$, N is determined by the size of the production memory. The third layer of nodes is for the partitioned form of working memory and other actions that may need to be carried out on the data; the number of PEs it has is 2^P for some $P > N$, P is determined by the size of the working memory. The pattern matching is done by the third level since they have the capabilities of accessing *contents addressable memories*, similar to those used in dynamic address generation from logical addresses in an environment using paging and/or segmentation. Processing elements at the second level in conjunction with their subtrees are associatively used to perform the recognise phase in a MIMD manner across the tree but SIMD mode within a given subtree associated with a PE at the second level.

Justification for using the DADO architecture for production systems is the conjecture that future VLSI technology may permit many PEs to be placed on a single chip since it is based on a binary tree design.

5.4.3. Hypercube Machines

These are medium grained MIMD computers with some degree of shared memory and message passing capabilities. Each node processor is a powerful minicomputer and has a significant amount of local memory for data processing or software execution. The motivation and justifications for using hypercubes is provided

in subsequent chapters and illustrates particular aspects of production systems that the architecture is well suited for.

5.5. Motivation for TREAT Algorithm

Anoop Gupta, at Carnegie Mellon University, tried to implement RETE [29] on the DADO architecture and concluded that large scale small grained parallelism is not a sound approach to solve the MATCH phase in parallel. One of the main disadvantages of the RETE network is that it cannot be easily implemented in a parallel environment without leading to communications problems amongst the various nodes of the RETE tree. These problems are encountered on the PSM machine too and get compounded with the synchronization and load balancing for dynamic, parallel execution of production systems.

Miranker and Stolfo developed a new algorithm for production systems, the TREAT algorithm [53] which was implemented on the small grain, microprocessor based, balanced binary tree like, parallel machines DADO [75, 76] and DADO2, at Columbia University. TREAT was found to perform with a good degree of success for a certain class of production systems that have a large proportion of working memory as compared to production memory.

While RETE algorithm was efficient in saving information regarding partial matches of data, Miranker found that a lot of information in the β-*memory* of a RETE network was duplicated at lower levels and at no stage did the network retain information regarding the conflict set. TREAT introduced the concept of *conflict set support* wherein the the system retains complete information of the conflict set across the

recognise act cycles. The conflict set is modified only if the RHS actions on the data effects any one of those rules that are included in the conflict set.

Including the conflict set support, TREAT identifies four different information states that are required during the life cycle of a production system. The other three information states that TREAT retains are *condition membership, memory support* and condition relationship. This division and classification of the intermediate knowledge states is exploited by TREAT as illustrated in section 5.2

5.6. TREAT Algorithm

The basis for this algorithm is that production systems are analogous to relational database systems where the memory elements of a production system may be thought of as tuples of some universal relationship in a relational database and the LHS of a rule as a query in a relational database language. production systems differ from relational database systems in the sense that database systems compute one query at a time on very large databases while production systems compute, as many queries as there are rules over a slowly changing modestly sized database.

```
---------------------------------------------------------------------------
(p    plus0X  {(goal ^type simplify ^object <n>) <goal>}
  {(expression  ^name <n>     ^arg1 0     ^op + ^arg2 <x>) <expression>}

(Join(Select(Class goal, type simplify))
  (Select(Class expression, arg1 0, op +)))
---------------------------------------------------------------------------
```

Figure 5.4: Database query form of the LHS for an OPS5 production rule.

Using some of the database terminology and the OPS5 program for mathematical identities, MATH, in figure 3.6, we can identify four different types of information states or knowledge that exists during the execution phase of any production system. Figure 5.5 represents the complete state information of the MATH program after execution of rule "start-math".

1: (start)nil
-> [cs] start-math (start-math dominates)
-> [run 1]
1. start-math 1
****break****
-> [wm]
3: (goal ^type simplify ^object e1)
5: (goal ^type simplify ^object e2)
7: (goal ^type simplify ^object e3)
9: (goal ^type simplify ^object e4)
*4: (expression ^name e1 ^arg1 1 ^op * ^arg2 99)*
6: (expression ^name e2 ^arg1 88 ^op / ^arg2 1)
8: (expression ^name e3 ^arg1 0 ^op + ^arg2 99)
10: (expression ^name e4 ^arg1 88 ^op - ^arg2 0)nil
-> [cs] mult1X divX1 plus0X subX0 (subX0 dominates)
-> [run 1]
2. subX0 9 10
CONGRATS the goal e4 is satisfied.
****break****

Figure 5.5: Status of production system MATH after firing rule *start-math.*

Unlike RETE algorithm's network, TREAT does not keep track of partial matches of working memory elements to condition elements. TREAT Algorithm exploits the conflict set support for its state saving strategy during the program execution. The four states of information that the algorithm requires are

[1]. *Condition Membership :* Associated with each condition element in the production system is a running count indicating the number of working memory elements that match the condition element pattern. A match algorithm that uses the condition membership may ignore those rules that are inactive. A rule is active when any of its positive condition elements are satisfied and have a membership value that is greater than 0. Table 5.2 illustrates memberships for the elements in column headed **members** on the basis of figure 5.5 as data.

Table 5.2: TREAT based *condition membership* and *memory support.*

condition element	**members**	**memory**
start-math { (start) init }	0	NULL
plus0X (goal ^type simplify ^object <n>)	1	7
plus0X (expression ^name <n> ^arg1 0 ^op + ^arg2 <x>)	1	8
subX0 (goal ^type simplify ^object <n>)	1	9
subX0 expression ^name <n> ^arg1 <x> ^op - ^arg2 0)	1	10
mult1X (goal ^type simplify ^object <n>)	1	3
mult1X (expression ^name <n> ^arg1 1 ^op * ^arg2 <x>)	1	4
divX1 (goal ^type simplify ^object <n>)	1	5
divX1 (expression ^name <n> ^arg1 <x> ^op / ^arg2 1)	1	6

[2]. *Memory Support :* An indexing scheme indicates precisely which subset of the working memory partially matches each of the condition element. By database analogy, memory support explicitly maintain a representation of the relations resulting from the select operations. RETE calls this representation the α-*memory*. Table 5.2 illustrates the information for the indexing in column headed **memory** on the basis of figure 5.5 as data.

[3]. *Condition Relationship :* Provides knowledge about the interaction of condition elements within a production rule. By analogy this corresponds to explicitly maintaining the intermediate results of a multiway join. This knowledge is available to the inference engine and used in the recognise act cycle to select one rule that will be executed.

[4]. *Conflict Set Support :* The conflict set is explicitly retined across production system cycles. By doing so , it is possible to limit the search for new instantiations to those instantiations that contain newly asserted working memory elements. Referring back to figure 5.5, the set is consisting of *{ mult1X, divX1, plus0X, subX0}* and the selected rule is denoted by the statement *(subX0 dominates)*. The RHS action of rule *subX0* as stated in figure 3.6 does not affect any working memory element that is relevant to the rest of conflict set hence the inference engine knows that a new recognise act cycle is not required, the new rule to be selected is from the old conflict set that TREAT had retained.

5.7. States Exploitation by TREAT

In section 5.2, it was shown that TREAT algorithm retains four different types of informations which are *condition membership, memory support, conflict support* and condition relationship. The manner in which TREAT exploits these intermediate states are:

[A]. **Condition Membership:** This is used to partition the working memory into homogeneous classes of data. It is also used to determine the inactive rules at any given instance. It is similar to the child nodes of the RETE network from the

ROOT node which effectively partitioned the data flow.

[B]. **Memory Support:** This is a pointer based referencing method, any PE is capable of indexing onto an elements in the working memory. This indexing value is used as key for the associative MATCH operation in the DADO machine at the third level.

[C]. **Condition Relationship:** This is used at each PE of the second level that is used to store a part of the production memory. This aspect of the TREAT algorithm is strictly dependent on the application program at hand.

[D]. **Conflict Set Support:** If the RHS action of a rule is to add an element to the WM, then the Conflict Set remains the same except for the instantiations that contain the new WM element. Additions to the WM may be used as seeds to initiate a constrained search for new instantiations. Deletions are processed by examining the conflict set directly and removing any instantiations that contain a deleted WM.

[E]. **Negated Conditions** Special consideration has to be given for negated condition elements in TREAT. Two cases have to be considered and they are the addition or deletion of WM elements that partially match or negated condition elements. The effect of negative condition elements matching on conflict set is similar to the effect of positive elements as in [D] except for their action. If an element in the working memory is found to match the negated condition element, then the inference engine will delete that particular rule from the conflict set and the converse holds true for a failure of match with negated conditions.

Condition Membership and Memory Support Exploitation

condition element	CE-num	index	element value
start-math	*0*	*NULL*	*{ (start) init }*
plus0X goal	*1*	*7*	*(goal ^type simplify ^object e3)*
plus0X expr	*2*	*8*	*(expression ^name e3 ^arg1 0 ^op + ^arg2 99)*
subX0 goal	*3*	*9*	*(goal ^type simplify ^object e4)*
subX0 expr	*4*	*10*	*(expression ^name e4 ^arg1 88 ^op - ^arg2 0)*
mult1X goal	*5*	*3*	*(goal ^type simplify ^object e1)*
mult1X expr	*6*	*4*	*(expression ^name e4 ^arg1 1 ^op * ^arg2 99)*
divX1 goal	*7*	*5*	*(goal ^type simplify ^object e2)*
divX1 expr	*8*	*6*	*(expression ^name e2 ^arg1 88 ^op / ^arg2 1)*

Number of active rules is 4

Old-Mem = { init } and New-Delete-Mem = { init }

Old-Mem = indexed values { 1 } and New-Delete-Mem = indexed values { 1 }

New-Add-Mem = indexed values { 3,4,5,6,7,8,9,10 }

Conflict-Set-Support = rule-name:CE-num { start-math:0 }

Figure 5.6: TREAT equivalent of states after *start* in figure 5.5.

Condition Membership and Memory Support Exploitation

condition element	CE-num	index	element value
start-math	*0*	*NULL*	*{ (start) init }*
plus0X goal	*1*	*7*	*(goal ^type simplify ^object e3)*
plus0X expr	*2*	*8*	*(expression ^name e3 ^arg1 0 ^op + ^arg2 99)*
subX0 goal	*3*	*NULL*	*{ (goal ^type simplify ^object e4) }*
subX0 expr	*4*	*NULL*	*{ (expression ^name e4 ^arg1 88 ^op - ^arg2 0) }*
mult1X goal	*5*	*3*	*(goal ^type simplify ^object e1)*
mult1X expr	*6*	*4*	*(expression ^name e4 ^arg1 1 ^op * ^arg2 99)*
divX1 goal	*7*	*5*	*(goal ^type simplify ^object e2)*
divX1 expr	*8*	*6*	*(expression ^name e2 ^arg1 88 ^op / ^arg2 1)*

Number of active rules is 3

Old-Mem = index { 3,4,5,6,7,8,9,10 } and New-Delete-Mem = index { 9,10 }

New-Add-Mem = { (^name e4 ^val 88 ^arg1 nil ^op nil ^arg2 nil) }

New-Add-Mem = indexed value { 11 } and not matching any Condition Element.

Conflict-Set-Support = rule-name:CE-num { plus0X:1,2 subX0:9,10 mult1X:5,6 divX1:7,8 }

Figure 5.7: TREAT equivalent of states after *subX0* in figure 5.5.

Exploitation of the condition membership is done by using the condition element's number denoted as 'CE-num' in figure 5.6. CE-num is an index into a data structure that has all the information regarding the membership and addresses of working memory elements with their index values that satisfy this condition element.

The manner in which TREAT exploits this is clear in the tables of figure 5.6 and figure 5.7 and is of the format **CE-num = XXX, Membership = YYY, INDEX = AAA, BBB, CCC ...** , where CE-num for 'plus0X expr' = 2, Membership = 1, Index= 8, value = (expression ^name e3 ^arg1 0 ^op + ^arg2 99). In the example XXX is an integer indexing onto the element type, YYY is a positive integer and represents how many working memory elements are of the type indexed by CE-num and AAA, BBB, CCC, ... are their respective addresses in memory in the form of index values.

For *memory support*, TREAT retains only the α-memories, containing information of a successful match of condition elements for a given TOKEN in RETE at the **constant test nodes**. Unlike RETE network, TREAT does not retain any information of variable bindings in the form of β-memories but uses the condition relationship with the partitioned memories to determine a MATCH. The working memory is broken into three partitions *old-mem, new-add-mem, new-delete-mem* as illustrated in figure 5.6 and figure 5.7 for the information in figure 5.5 so as to limit the scope of MATCH phase to*new-add-mem* only.

5.7.1. Program Execution by TREAT

TREAT used CE-num to partition the production memory into homogeneous element classes and also partitioned working memory into three dynamic components. The effects of this form of partitioning are clear in figure 5.7 after the rule *sub0X* has been selected and fired with respect to the state information represented in figure 5.6 prior to execution of the rule.

The *old-mem* contains index values of matched elements that have already been processed by the production system, which is the set { 3,4,5,6,7,8,9,10 }. These data items will not be considered again during the recognise act cycle, thereby restricting the scope of the MATCH phase to new elements only. The new data is in two groups, *new-add-mem, new-delete-mem* which are created as a consequence of the RHS of the selected and executed rule. Figure 5.7 illustrates the consequence of the rule subX0 and the new memory partitions are { 11 } and { 9,10 } respectively. All these data partitions get updated by the TREAT algorithm after the execution of a production rule.

The conflict set state will immediately process the new-delete-mem and drop all the instantiations that have the index values from the new-del-mem partition. In figure 5.7 the memory is { 9,10 } and the conflict set has only one instantiation which uses these indexed values of data to match their LHS and the rule is *subX0:9,10*. The rule gets deleted from the conflict set since the data that supports the LHS of the rule have been deleted. The contents of new-add-mem in figure 5.7 do not cause that activation of any new rule, neither does it cause any active rule to complete the MATCH process since it does not match any condition element of the program MATH.

The condition relationship is used whenever the size of old-mem changes and all the related rules are examined so that the set of active rules may be accordingly updated. In figure 5.7 the number of active rules changed from 4 to 3 since the RHS of subX0 deleted the indexed working memory elements { 9, 10 } that were required to satisfy that particular rule. Conversely if the old-mem size increases, the number of active rules might get modified on the basis of the information available in the condition membership.

5.8. Conclusions on TREAT

RETE minimized the matching phase of the elements in the elements by constructing the search network and maintaining state of partial matches. The disadvantage is that the state information has to be updated after the deletion of an element which may lead to contention/communication problems in a parallel environment. RETE performs a lot of computation of the inactive rules to maintain the state information unlike TREAT which uses the condition support state information.

The TREAT algorithm implements a more efficient scheme to implement the variable-handling in the match phase of the production systems as compared to the RETE algorithm and does not have to maintain the β-memory used in RETE hence it is almost twice as fast as RETE [31, 52] in running some benchmark OPS5 production systems.

Since the join or MATCH operation is commutative as well as associative, the search for consistent variables in the α-memory may be considered in a static order based on lexical ordering of condition elements, or in a indexed order.

Figure 5.8 illustrates the associative and commutative nature of the MATCH phase for equivalent forms of the same rule *plus0x* from the MATH program.

```
(p    plus0X  {(goal ^type simplify ^object <n>) <goal>}
  {(expression  ^name <n>  ^arg1 0  ^op + ^arg2 <x>) <expression>}

(p    plus0X  {(goal ^object <n> ^type simplify) <goal>}
  {(expression  ^arg1 0  ^name <n>  ^arg2 <x>  ^op + <expression>}

(p    plus0X  {(expression ^name <n>  ^arg1 0 ^op + ^arg2 <x>) <expression>}
  {(goal ^type simplify ^object <n>) <goal>}
```

Figure 5.8: Commutative and Associative nature of MATCH for plus0X.

Irrespective of the mode of implementing MATCH, changed α-memory is considered first, thereby constraining the search because the changes are generally of a small order with respect to the size of entire working memory. DADO when constructed with PE having associative memories uses the index values, as keys to the contents addressable memories, to determine the addresses of the elements and the result of MATCH.

TREAT was found to perform with a good degree of success for a certain class of production systems that have a large proportion of working memory as compared to production memory. TREAT's performance was directly proportional to the the communication time required by the PE that takes the longest to finish it's processing. The processing elements at the production memory level required larger memories therefore the tree is not a true homogeneous architecture. TREAT did not propose a scheme for the partitioning of the production memory onto the node processors.

Table 5.3: Brief review of related research.

	RETE (Forgy)	TREAT (Miranker,Stolfo)	PSM (Gupta,Forgy)
Basis	Indexed Search Tree & Dataflow Architecture	Special Datastructures & Hardware for Communications	RETE, Shared Mem. MIMD & Task Scheduling Structure
Advantages	1. Reduced the number of match conditions. 2. Simplified Match by retaining partial match. 3. Simple representation for negative logic.	1. Reduced redundant info. 2. Retained conflict set. 3. Special datastructures and algo. for parallel implem.	1. Dynamic load balancing. 2. Global task queue with (1), for better CPU utilisation. 3. Inter and intra node actions are parallelised.
Disadvantages	1. Too much memory for saving all partial matchs. 2. Time consumed in unwinding for del/mod action. 3. Bottlenecks in data flow for parallel implementations.	1. Large *working memory* is required at lower level of tree. 2. Requires a full binary tree of microprocessors. 3. Set of active rules is entire *production memory* for goal driven *production systems*.	1. Mapping rules to nodes. 2. Time is lost in synchronising shared mem. 3. Bottlenecks at global task scheduler. 4. Overhead for dynamic load balancing.

Large scale parallelism is not suitable for production systems since at any given time modifications to the working memory affects [30] only a small number of production rules and this leaves most of the PE in the DADO tree idle for a significant amount of time. If more than one active rule is present at any PE, the set of rules are processed in a strictly sequential manner by the PE.

5.9. Summary

Table 5.3 summarises the advantages and disadvantages of the sequential RETE algorithm as proposed by Forgy, the parallel TREAT algorithm as proposed by Miranker et. al. and the parallel RETE algorithm implemented on the PSM simulator by Gupta et. al. In the following chapters this dissertation outlines an approach towards parallelising goal driven, forward chaining production systems that eliminates the disadvantages listed in table 5.3 and provides for a faster MATCH algorithm.

Chapter Six

Parallelized and Indexed Transformation of OPS5 Production Rules

The principal contribution of this research is encapsulated in this chapter. Robotics application programs are real time rule based programs and the response time for the recogine act cycle is critical; this time constraint was the motivation of this research for a new and faster MATCH algorithm. The main features of the new proposed MATCH algorithm are

1. A translation scheme is proposed for transforming character strings to a set of unique integers.
2. An indexing scheme is provided to identify a few key data elements that are used in an arithmetic hashing function.
3. The evaluation process of the recognise module is an arithmetic operation and not an incremental equivalence scalar operation on character strings.

This chapter provides a solution to the *goal-directed* forward chaining rule based programs that are used to program and control autonomous machines. Though the solution is specific to robotics applications in OPS5 source code, the principles may be applied to parallelising the MATCH phase in an indexed manner and the C programming environment for any rule based system that employs forward chaining as a problem solving strategy.

6.1. Motivation for using C Programming Language

OPS5 has inherent drawbacks with respect to user defined data structures and input output operations. The C programming language provides a rich set of data structures and I/O features that make it a programmers choice for systems design and development.

OPS5 is a LISP based interpreted language, whose implementation requires a scalar equivalence of two attributes to confirm a valid MATCH during the recognise act cycle. The research outlined in this theses provides a means of transforming the scalar equivalence MATCH operation into an *indexed* arithmetic operation using the **switch-case** construct for the MATCH phase, which is easily encoded using features provided in C.

Most of todays parallel computers, including the INTEL iPSC Hypercube series [36] and the Hypercube Simulator [7, 19, 20] at Oak Ridge National Laboratory provide a C programming environment. These existing resources were a major consideration in the research work since it had to be considered for implementation on the HERMIES-IIB Robot. This chapter provides an outline to parallelize the MATCH phase as a static distribution of the production memory over the N nodes of the hypercube, with appropriate synchronization techniques for coordinating the activities of all nodes into a true MIMD machine. The working memory is also partitioned and made available to all the nodes of the hypercube as a shared memory block.

---

start

START PASS I
input OPS5-PRODUCTION-SYSTEM
CE ->NULL; /* the set *CE* of Condition Element types used in the program; */
TEMP initialized to 0; /* TEMP is a distribution table of *CE* in each rule). */
S_{CE}->NULL; /* From *CE* and *TEMP* determine a set S_{CE} of Condition Elements such that
a. the members of the set do not have multiple instantiations within a Production Rule.
b. the entire set is present in most of the Production Rules. */
while *not eof do*
1) add to *CE*, the name following *(literalize* ; /* scanning the definitions of elements; */
2) generate *TEMP*; /* scanning the production rules, looking for *CE* elements; */
end do;
3) from *TEMP* and *CE* /* generating S_{CE} ; */
i] if CE_i occurs once within most production rules add CE_i to S_{CE} ;
ii] reduce S_{CE} to a subset S_{CE} which occurs in most production rules.
END PASS I

START PASS II
ATR_{ind}->NULL; /* set of attributes ATR_{ind} that need a minimum Variable Binding; */
while *not eof and with* S_{CE} *do*
4) within most production rules, if A_i is an attribute of S_{CE} and not instantiated as an independent variable add A_i to ATR_{ind} ;
end do;
END PASS II

START IMPLEMENTATION
V_i->NULL; /* set of vector values for every Production Rule using ATR_{ind} attributes; */
while *not eof and with* ATR_{ind} *do*
5) for each firing of a production rule
i] generate the vector value V_i prior to executing the RHS and
ii] identify the rule it is associated with;
6) increment i;
end do;
7) distribute $\lceil \frac{V_{max}}{N} \rceil$ indexed rules per node;
END IMPLEMENTATION

end ;

Figure 6.1: Simplified version of the algorithm.

6.2. Parallelizing Algorithm

We have seen that the time spent in MATCH phase is directly proportional to the sizes of production memory and *working memory*; and is the most time consuming feature of production systems. The algorithm as illustrated in figure 6.1, has taken two approaches to reduce the time spent by production systems in the MATCH phase. They are :

a. The working memory is partitioned [37,51,53,71] into blocks of homogeneous data thereby reducing the time spent by the program in iterative matching of a rule's condition element to data. This partitioning is feasible in a C programming environment and not in OPS5 which treats working memory as one collection of heterogeneous elements.

b. The MATCH algorithm also incorporates an indexing scheme [70,71] which reduces the number of condition elements that are used in a production rule for determining the conflict set. The indexing scheme is based on the contents of the current working memory for a few key elements. These elements are used by an arithmetic hashing function to generate an index value which is used in the C **switch(index), case ...:** statements, rather than carry out the scalar operations of the type

IF((elem1.attribute1 = XXX) and (elem2.attribute2 = YYY) ....THEN(RHS.....)) as is done in OPS5 to determine the matchability of rules with working memory.

6.2.1. Identifying Elements

Elements are the basis of all production systems. As stated in chapter 1 this

dissertation is based on research efforts to parallelize the program environment for production systems. OPS5 using RETE network did not differentiate between the classes of data and element types. TREAT algorithm on the other hand, as implemented on the DADO machine, carried out the partitioning of data and rules to an extreme. The philosophy used in this research is to utilize the medium grained parallelisms of hypercube architecture and cluster the production memory and working memory components of a production system, on the basis of condition elements.

Starting with an OPS5 production system, we may identify the set of condition elements *CE* by following the scheme outlined in *PASS I* of the algorithm illustrated in figure 6.1. The elements of set *CE* are determined from the definition/declaration of OPS5 elements [8] within the program; this is analogous to the data definition and variable declaration block of any structured program. Set *CE* is the same size as the number of elements used by the production system that is being analyzed for parallel implementation.

The identification process for any element in an OPS5 program is based on a lexical analyzer [2] like approach. OPS5 is highly structured, using parentheses pairs as delimiters. Figure 6.2 illustrates the algorithm, data structures and initializations that would be required to generate the set *CE* which is subsequently used by other parallelizing modules.

The parallelizing algorithm uses the OPS5 language definitions and delimiters with the character/string processing capabilities of C to process any input data file and construct the set *CE* of elements for that OPS5 program file.

```
start /* Lexical Analyzer Like Module. */

input file OPS5-PRODUCTION-SYSTEM /* Data for the Module. */

CE ->NULL; /* the set CE of Condition Element types used in the program. */

open file to READ
while not eof do
        str1 = getstring(file);
                if str1 == "( " do
                        str2 = getstring(file);
                                if str2 = "literalize"
                                        str3 = getstring(file);
                                 add to set CE, the string str3;
                end if-do;
        skip input file till closing parentheses ")";
end while-do;
```

Figure 6.2: Algorithm to identify elements in CE.

Table 6.1: Set of elements CE.

Condition Elements	Monkey and Banana	Math	HERMIES-IIB APS
CE_1	Goal	Goal	Goal
CE_2	Monkey	Expression	Robot
CE_3	Phys-Obj		Exception
CE_4			Sonar1
CE_5			Fire-Transducer

Table 6.1 illustrates the output that is generated by the algorithm in figure 6.2. The set CE is generated for production systems *Monkey and Banana, Math, HERMIES-IIB* that have been defined and discussed elsewhere in this dissertation

report.

6.2.2. Partitioning Production and Working Memory

The set *CE* is used to partition the working memory into homogeneous blocks of data in a C environment. During the MATCH phase the system is restricted to appropriate data block and not the entire working memory as has been the case in traditional systems. The number of data blocks will be the same as the number of elements in the set *CE* and all will be available as a block of shared memory to all the node processors in a parallel environment.

The basis for partitioning the production memory is the manner in which the dominant or controlling condition element is used to determine the outcome of the MATCH phase in any rule. The OPS5 production systems *Monkey and Banana, HERMIES-IIB APS, MATH* are all driven by element goal and the LHS is constructed as *(p XXX (goal ^type <> ^status <> ^.....) (YYY ...)*

For a straightforward parallelization of the production systems onto a hypercube of **P** nodes the production memory may be partitioned in the following manner. Given **P** number of node processors for the hypercube and $\mathbf{R_T}$ number of rules for type **T** where *T* is the set of controlling attribute values for that production memory, then each node processor in the cube has a maximum of $\lceil \frac{\mathbf{R_T}}{\mathbf{P}} \rceil$ rules per BLOCK for each class type **T** of production rules .

In a simple parallel implementation of production systems on the hypercube, rules are clustered by BLOCKS of *goal.type* so that the ' C ' statement ' SWITCH (WM_GOAL[*cur-gol].type*]) ' will activate the pertinent BLOCK or subset of

production rules during the MATCH phase.

6.3. Indexed Production Memory

Other researchers in their attempt to improve the time complexity of the MATCH phase have proposed rewriting of production rules with a reduction in the number of condition elements in the LHS while increasing the overall number production rules [23, 27, 58], thereby reducing the exponent C value but increasing the production memory size P in the time complexity to determine a successful match for traditional systems $O(P \times W^C)$, which will be discussed in detail in chapter 10. The scheme proposed by this theses research transforms the scalar MATCH operations into an arithmetic function for evaluating the *vector* value used in conjunction with the **switch-case** construct of C and indexes onto the arithmetic value as a form of hash function.

Table 6.2: *TEMP* table showing distribution of *CE* elements.

RULE #	CONDITION ELEMENT and INSTANTIATIONS.
1	*goal:1 , monkey:1 , phys-obj:0*
2	*goal:1 , monkey:1 , phys-obj:0*
3	*goal:1 , monkey:1 , phys-obj:1*
4	*goal:1 , monkey:1 , phys-obj:1*
5	*goal:1 , monkey:1 , phys-obj:1*
6	*goal:1 , monkey:1 , phys-obj:1*
7	*goal:1 , monkey:1 , phys-obj:1*
8	*goal:1 , monkey:1 , phys-obj:0*
9	*goal:1 , monkey:1 , phys-obj:3*
10	*goal:1 , monkey:1 , phys-obj:1*
11	*goal:1 , monkey:0 , phys-obj:2*
12	*goal:1 , monkey:1 , phys-obj:1*
13	*goal:1 , monkey:1 , phys-obj:1*
14	*goal:1 , monkey:1 , phys-obj:1*
15	*goal:1 , monkey:1 , phys-obj:1*
16	*goal:1 , monkey:1 , phys-obj:1*
17	*goal:1 , monkey:1 , phys-obj:1*
18	*goal:1 , monkey:1 , phys-obj:1*
19	*goal:1 , monkey:1 , phys-obj:1*
20	*goal:1 , monkey:0 , phys-obj:1*
22	*goal:1 , monkey:1 , phys-obj:1*
23	*goal:1 , monkey:1 , phys-obj:0*
24	*goal:1 , monkey:1 , phys-obj:0*

The algorithm to transform a production system into the indexed form has three phases.

1. Shared Elements: The parallelizing algorithm uses *CE* in determining the subset S_{CE} which contains the shared elements that occur in most production rules.

2. Indexing Attributes: The subset S_{CE} will subsequently be used in the algorithm to determine a set of indexing attributes ATR_{ind} for the indexed MATCH phase.

3. Numerical Values: The transformation from scalar entities to integers for arithmetic operations and generate the vector values at runtime.

6.3.1. Shared Condition Elements.

The purpose of this phase of the algorithm is to determine a set of shared elements S_{CE} from CE such that

- The members of the set do not have multiple instantiations within any production rule of the system.
- The entire set is present in most of the production rules.

The set S_{CE} is constructed using the set CE from figure 6.2 and $TEMP$ table from figure 6.1 that has a frequency table like representation of the production system and its component rules. Table 6.2 shows the frequency distribution of elements within each production rule for the Monkey and Banana production system program. Elements are represented by names *{ goal, monkey, phys-obj}* from table 6.1, preceding the colon symbol ':' and the instantiations per rule { 0,1,2,3 } are are shown by the integer following the symbol ':'. The algorithm to generate S_{CE} may be stated as illustrated in figure 6.3.

---

start /* Lexical Analyzer Like Module. */

input CE, TEMP; /* Data for the Module. */
S_{CE}->NULL;

while *CE , TEMP* not empty do
    if CE_i occurs once within most production rules add CE_i to S_{CE};
end while-do;

while S_{CE} , TEMP *not empty do*
    reduce S_{CE} *to a subset* S_{CE} *which occurs in most production rules;*
end while-do.

Figure 6.3: Algorithm to identify elements in S_{CE}.

From the three condition elements *{goal, monkey, physical-object}* in *CE* a set of condition elements S_{CE} is to be selected such that they fulfill the following conditions.

a] The set occurs in a significant proportion of the production rules. This enables the programmer to cover a large section of the production memory. Based on this criterion alone, we may select all three condition elements *goal, monkey* and *physical object*, as is clear in in table 6.2.

b]. The members of the set do not have multiple instantiations in any given rule. This ensures the deterministic nature of the decision-vector and reduces the computation required to determine the vector. Using this constraint we have to eliminate the condition element *physical-object* which ranges in occurrence from 0 to 3 in any given production rule and are restricted to the elements *{ goal , monkey }* to form the set S_{CE}.

6.3.2. Determining Index Attributes

The set S_{CE} consisting of elements *{goal, monkey}* are the condition elements that will be considered for determining the indexing attributes ATR_{ind}. The criterion for forming ATR_{ind} is stated in step 4 of algorithm in figure 6.1. and explained in detail in figure 6.4.

---

input file, S_{CE} / Data for the Module. */*

ATR_{ind}->NULL; / Set of attributes used in the program for indexing. */*

```
open file to READ
while not eof do
    str1 = getstring(file);

    if str1 == "(p " do
       str2 = getstring(file);
       if str2 ∈ S_CE /* "goal" or "monkey" */
         str3 = getstring(file);
         str4 = getstring(file);
         if str4 is an OPS5 constant
            add to set ATR_ind, the string str3;
         if str4 is bound to variable value for other element ∈ S_CE
            add to set ATR_ind, the string str3;
   end if-do;

  skip input file till closing parentheses ")";

end while-do;
```

Figure 6.4: Algorithm to identify elements in ATR_{ind}.

The set ATR_{ind} is composed of those attributes that are either instantiated as a constants or else instantiated as dependent variables. These attributes will be used to gen-

erate the indexing vector value and the selected set ATR_{ind} is *{ goal ^type, goal ^object-name, monkey ^on, monkey ^holds }*.

The justifications [70, 71] for that particular selection of ATR_{ind} as *{ goal ^type, goal ^object-name, monkey ^on, monkey ^holds }*, are based on examples from figure 6.5 illustrating a small subset of production rules for the Monkey and Banana problem.

i. *goal ^type* is an attribute that is initialized to a constant in all production rules.

ii. *goal ^obj-name* is instantiated either as the dependent variable from the element *(phys-obj ^name <o>* or a constant/nil value as illustrated in the rule *on::floor* of figure 6.5.

iii. *monkey ^on* and *monkey ^holds* are either instantiated as constants as in rule *on::floor* of figure 6.5 or bound to the variable *goal ^obj-name* as in rule *holds::obj-notceil* of figure 6.5.

iv. *goal ^status* was not selected since it has just one constant value *active* in all the production rules; it is not used as vector component but as the condition controlling the *while* loop as shown in figure 6.7 for the implementation of the vector indexed MATCH in C code.

v. *monkey ^at* was not selected since it is mainly used in a small subset of production rules that have the attribute *goal ^type at*; secondly as a variable it is dependent on the attribute *phys-obj ^at* bound by the variable *phys-obj ^name<o>* which was not a member of the set S_{CE}. The rules *at::monkey, holds::obj-notceil* of figure 6.5 respectively illustrate the points at hand to be considered in generating the set of

attributes ATR_{ind}.

```
(p on::floor
        {(goal ^status active ^type on ^obj-name floor ) <goal>}
        {(monkey ^on <> floor)  <monkey>}
-->
        RHS)

(p holds::obj-ceil
        {(goal ^status active ^type holds ^obj-name <o1> ) <goal>}
        {(phys-obj ^name <o1> ^weight light ^at <p> ^on ceiling ) <object1>}
        {(phys-obj ^name ladder ^at <p> ^on floor ) <object2>}
        {(monkey ^on ladder ^holds nil )  <monkey>}
-       (phys-obj ^on <o1>)
-->
        RHS)

(p holds::obj-notceil
        {(goal ^status active ^type holds ^obj-name <o> ) <goal>}
        {(phys-obj ^name <o> ^weight light ^at <p> ^on <> ceiling) <object>}
        {(monkey ^at <p> ^holds nil ^on floor)  <monkey>}
-       (phys-obj ^on <o>)
-->
        RHS)

(p at::monkey
        {(goal ^status active ^type at ^obj-name nil ^to <p1>) <goal>}
        {(monkey ^at <> <p1> ^holds nil ^on floor)  <monkey>}
-->
        RHS)
```

Figure 6.5: Sample set of OPS5 rules.

6.3.2. Index Attributes as Integer Values

This phase of the algorithm needs the following steps to be executed after having identified the set of indexing attributes ATR_{ind}.

i. Execute the OPS5 program using all possible test case scenarios. Keep a trace of the execution along with all relevant information, as illustrated in figure 6.6. The main things to be retained are the initial working memory [70, 71], the various rules that get executed and the final outcome for that particular scenario.

ii. The OPS5 constants for the attributes $\in ATR_{ind}$ are transformed into integers for the C version of production systems by the following 6 steps.

OPS5 constants are transformed into integers by the following process. The integer values associated with the OPS5 attribute constants are selected such that

1. Identify OPS5 constants. {X} = set of all instantiations for attributes, $ATR_{ind} \in$ {X};

   {X} = {NIL, ON, HOLD, AT, COCH, LGHT, ACTV, LADR, FLOR, BANS, CEIL, BLNK, HEVY, SATS}

2. Identify size of the set as lim indicating the number of OPS5 constants that have to be transformed; lim = sizeof({X}) --> lim = 13.

3. Select a set of integers { INT } such that sizeof({INT}) = sizeof({X})

   and {$\forall$ a, b $\in$ *INT* | integer a $\neq$ ba, ab $\neq$ ba, and a $\neq$ bb } where ab is prefix 'a' and 'b' not their product.

```
> ops5
-> [load 'mab]
[load mab]
**********************************************************************t
-> [make g1]
nil
-> [run 2]

1. start1 1
2. test::general:start 3
***break***
-> [wm]

3: (testcase  ^type general  ^name start)
4: (phys-obj  ^at 9-9  ^name bananas  ^on ceiling  ^weight light)
5: (phys-obj  ^at 7-7  ^name couch  ^on floor  ^weight heavy)
6: (phys-obj  ^at 4-3  ^name ladder  ^on floor  ^weight light)
7: (phys-obj  ^at 7-7  ^name blanket  ^weight light)
8: (monkey  ^at 7-7  ^holds blanket  ^on couch)
9: (goal  ^type holds  ^status active  ^obj-name bananas)nil

-> [cs]
holds::obj-ceil:on:at-obj
imp-oss
(holds::obj-ceil:on:at-obj dominates)
-> [run]

3. holds::obj-ceil:on:at-obj 9 4 6
 .....
 .....

18. congrats 3 60
CONGRATULATIONS the goals are satisfied .

end -- explicit halt
-> [exit]
Goodbye
```

Figure 6.6: A sample run of an OPS5 program keeping track of *WM* and *CS*.

4. An example of a set of integers that do not satisfy the constraints stated above is *{ INT - FALSE } = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13}* is false for a = 11, b = 1 with respect to the condition a ≠ bb.

5. A valid set of integers that may be used for the current application are *{ INT } = {10, 21, 22, 23, 30, 40, 48, 31, 32, 33, 34, 45, 49}.*

6. Associate the integers to OPS5 constants. { X<->INT | X_i is value of an attribute in the set ATR_{ind}; INT_i is an unique integer} as illustrated in table 6.3.

Table 6.3: Integers ∈ *INT* and their associated OPS5 constant values for ATR_{ind}.

10 → NIL	31→ LADR
21 → ON	32→ FLOR
22 → HOLD	33→ BANS
23 → AT	34→ CEIL
30 → COCH	35→ BLNK
40 → LGHT	45→ HEVY
48 → ACTV	49→ SATS

Table 6.3 is a LEGEND of integers in set *INT* and their associated OPS5 constant values for ATR_{ind} which will be used as components to generate the index VECTOR values *indx* for the Monkey and Banana OPS5 production system program.

Table 6.4: Vector values V_i for ATR_{ind} based on table 6.3.

	GOAL		MONKEY		
CASE	TYPE	NAME	ON	HOLDS	VECTOR
1	ON	LADR	FLOR	NIL	21313210
2	ON	LADR	FLOR	LADR	21313231
3	ON	LADR	LADR	NIL	21313110
4	ON	FLOR	LADR	BLNK	21323135
5	ON	FLOR	COCH	BLNK	21323035
6	ON	FLOR	LADR	NIL	21323110
7	ON	FLOR	COCH	NIL	21323010
8	ON	FLOR	FLOR	NIL	21323210
9	ON	FLOR	COCH	BANS	21323133
10	HOLD	NIL	FLOR	LADR	22103231
11	HOLD	NIL	FLOR	BLNK	22103235
12	HOLD	NIL	FLOR	NIL	22103210
13	HOLD	LADR	LADR	NIL	22313110
14	HOLD	LADR	FLOR	NIL	22313210
15	HOLD	LADR	LADR	BLNK	22313135
16	HOLD	LADR	COCH	BLNK	22313035
17	HOLD	LADR	FLOR	BLNK	22313235
18	HOLD	BANS	LADR	NIL	22333110
19	HOLD	BANS	FLOR	NIL	22333210
20	HOLD	BANS	COCH	BLNK	22333035
21	HOLD	BANS	LADR	BLNK	22333135
22	HOLD	BANS	FLOR	LADR	22333231
23	HOLD	BANS	FLOR	BANS	22333233
24	AT	NIL	FLOR	NIL	23103210
25	AT	NIL	COCH	BLNK	23103035
26	AT	NIL	COCH	NIL	23103010
27	AT	LADR	FLOR	LADR	23313231
28	AT	LADR	COCH	BLNK	23313035
29	AT	LADR	LADR	BLNK	23313135
30	AT	LADR	FLOR	NIL	23313210
31	AT	BANS	FLOR	BANS	23333233
32	AT	BANS	COCH	BANS	23333033
33	AT	NIL	FLOR	BLNK	23103235

6.4. Generating Vector Values

After identifying the set attributes ATR_{ind} and their associated integer values, each rule in OPS5 is translated into a simple C version. At this point no effort is made to optimize the C code except for the partitioning of the working memory on the basis of *CE* in table 6.1.

The only modification is the generation of the vector value as an integer variable indx by the RHS of a rule. The indx value is the first action executed by the RHS of a rule and is generated as an arithmetic expression which is then used to identify the working memory conditions that are required to satisfy the LHS of that particular rule. The integer indx is subsequently used as the **switch** variable in the **C case** statement for the hypercube implementation of the production system.

*indx = (goal ^type <X>)*1000000 + (goal ^object-name <Y>)*10000*
*+ (monkey ^on <A>)*100 + (monkey ^holds <B>)*

If ATR_{ind} has N attributes then the n^{th} component is multiplied by $10^{2(n-1)}$ to form the vector where $n \rightarrow 1..N$. An indx value of 22333233 would break down into the OPS5 equivalent form of the two elements contributing to the ATR_{ind}, as the following instances in the working memory.

(goal ^type 22=HOLD ^object-name 33=BANS), (monkey ^on 32=FLOR ^holds 33=BANS)

The multiplying base value 10^2 was derived at by the following constraints;

$\forall\ a, b \in INT \mid (base \times a) + b \neq (base \times b) + a$. We are trying to identify a base value

that will retain the uniqueness of the working memory representation; simple algebra gives us a rule of thumb to determine the value of *base* as, $base = 10^i$ where i is the lowest positive integer for the condition (10^i > maximum value of the set *INT*) to hold true.

For the Monkey and Banana production system, which was translated into C code as a part of this research, it is found that max(*INT*) is 49 from table 6.3 and since 100 > 49 therefore the base value is 100 for this implementation.

Table 6.4 lists all the vector values generated at run time for each production rule by all the possible test cases for the Monkey and Banana program using the component values from table 6.3 and the indexing formula for indx described in the preceding section.

---

1. ORIGINAL OPS5 CODE

```
(p holds::obj-notceil:at-monkey
      (goal ^status active ^type holds ^obj-name <o> )
      (phys-obj ^name <o> ^weight light ^at <p> ^on <> ceiling )
      (monkey ^at <> <p> )
--> RHS
```

2. INTERMEDIATE C CODE GENERATING THE VECTOR VALUES

```
switch (WM_gol[GL_cur]->type) {
case HOLDS :
     {  for(i=0;i<=OB_cur;i++)
        if(WM_gol[GL_cur]->nam == WM_obj[i]->nam) gol_nam_loc = i;
     if ((WM_mon->at != WM_obj[gol_nam_loc]->at) &&
           (WM_obj[gol_nam_loc]->wght == LGHT) &&
           ( WM_obj[gol_nam_loc]->on != CEIL)) {
       indx = WM_mon->on*100 + WM_mon->holds +
           WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 ;
       coun++;
         arr[coun] = indx ;
        printf(" INDEX IS %d at %d in %d ",indx,mynode(),coun);
        RHS   }
     break;
   } ...../* end of case */
} ...../* end of switch */
```

3. FINAL C CODE INDEXING ON THE VECTOR VALUE 'indx'

```
switch (indx) {
  case 21313210 :  {
    for(i=0;i<=OB_cur;i++)
       if(WM_gol[GL_cur]->nam == WM_obj[i]->nam) gol_nam_loc = i;
       RHS ;
       indx = WM_mon->on*100 + WM_mon->holds +
       WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 ; }
    break;
   } ...../* end of case */
} ...../* end of switch */
```

Figure 6.7: Phases in translating an OPS5 production rule into an indexed C code.

6.5. Transforming OPS5 to Indexed C code

The intermediate C program is executed to generate the set V_i corresponding to the column containing VECTOR entries in table 6.4.

```
switch (WM_gol[GL_cur]->type) {
case ON :
   {
      if((WM_gol[GL_cur]->nam == FLOR) && (WM_mon->on == FLOR))
      {
         indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000
                     + WM_mon->on*100 + WM_mon->holds;
         coun++;
         arr[coun] = indx ;
         printf(" INDEX IS %d at %d in %d ",indx,mynode(),coun);
         RHS
      }
 ...../* end of case */
   }
 ...../* end of switch */
}
```

Figure 6.8: Intermediate C code for a rule generating it's vector value.

Each vector value is associated with the firing of a production rule, and corresponds to a particular instantiation of the working memory elements, *monkey* and *goal*. The association is brought out by the *printf, arr[coun]* statements in figure 6.9 and comparing them to the OPS5 execution in figure 6.6. The indx values are used for indexing and replace the condition elements used in S_{CE} within each production rule, which is associated with that vector value as illustrated in figure 6.9 and is the actual indexed translation for the OPS5 rules in figure 6.5.

```
while (WM_gol[GL_cur]->stat == ACTV) {
  switch (indx) {
   case 22313210 :
      {
      for(i=0;i<=OB_cur;i++)
          if(WM_gol[GL_cur]->nam == WM_obj[i]->nam) gol_nam_loc = i;
      if (WM_mon->at != WM_obj[gol_nam_loc]->at) {
         RHS ACTION
         indx = WM_mon->on*100 + WM_mon->holds +
         WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 ; }
           else  {   RHS ACTION
      }
      break;

   case 22313035 :
      {
      for(i=0;i<=OB_cur;i++)
           if(WM_gol[GL_cur]->nam == WM_obj[i]->nam) gol_nam_loc = i;
         RHS ACTION
      }
      break ;

   case 22333110 :
      {
      for(i=0;i<=OB_cur;i++)
          if(WM_gol[GL_cur]->nam == WM_obj[i]->nam) gol_nam_loc = i;
         RHS ACTION
      }
      break;

   case 23103210 :
      {
         RHS ACTION
      }
      break;
      default ;
  } /* end switch */
} /* end while */
```

Figure 6.9: Indexed MATCH implementation in C code.

The process of Translating and Transforming a set of production rules as illustrated in figure 6.7, always requires the following sequence of steps:

1] For an existing OPS5 program, run the program for all possible test cases and for each such run retain a copy of the trace file. The trace file should provide the correct sequence of executing rules and the data in the working memory to match the rule. This step is illustrated in figure 3.7.

2] For a given OPS5 trace file and program, identify the set of indexing attributes ATR_{ind} and transform the production rules to the intermediate form as illustrated in figures 6.8 and 6.7. This generates the table of vectors listed in table 6.4.

3] Rewrite the intermediate production rules and replace the LHS of the rule by the corresponding vector value for the indexing by the indx variable of the program that is generated as consequence of the RHS action. The computation of the indx variable is a compact and integer form of the working memory for the selected attributes ATR_{ind} of the elements in S_{CE}.

The production rule *on::floor* in figure 6.5 generates vector values corresponding to *{21323135, 21323035, 21323110, 21323010, 21323210, 21323133}* that are all used to perform the same LHS matching as that in the OPS5 rule.

The production rule *holds::obj-ceil* in figure 6.5 generates vector values corresponding to *{22313110, 22333110}* that are all used to perform the same LHS matching as that in the OPS5 rule.

The production rule *holds::obj-notceil* in figure 6.5 generates vector values

corresponding to *{22313210, 22333210}* that are all used to perform the same LHS matching that in the OPS5 rule. All the above mentioned vectors illustrate a case of multiple instances for the same OPS5 rule; this is the inherent drawback of the indexing scheme which requires a tradeoff in space for the sake of speed.

The production rule *at::monkey* in figure 6.5 generates vector values corresponding to *{23103210}* which is used to perform the same LHS matching that in the OPS5 rule.

Certain production rules that require variable bindings for attributes other than those belonging to the set ATR_{ind} have to be implemented as loops iterating over the partitioned homogeneous working memory. The rule is selected by the indexing value of indx but the loop is still required to bind the variables as in the case of rule *holds::obj-notceil:at-monkey* illustrated in figure 6.7. The integral indx value 22313210 confirms the LHS matching of rule *holds::obj-notceil:at-monkey* but the data element for *phys-obj* has to be identified from *WM_obj[]* such that it has the same name as the physical object in the current goal.

6.6. Distributing the Indexed Production Rules

An indexed parallel implementation of the production systems onto a hypercube of **P** nodes with a production memory of size $\mathbf{V}_{max}$ corresponding to the maximum number of vector entries in table 6.4, may be partitioned into blocks of similar size. The size of production memory at each node processor in the cube is $\lceil \frac{\mathbf{V}_{max}}{\mathbf{P}} \rceil$ rules per node processor.

Unlike the simple non-indexed parallel implementation where the rules were distributed on the basis of a dominant condition element's attribute, the indexed implementation may be distributed in a completely random manner since the cost of accessing any of the rules is a constant.

Chapter Seven

Transformation and Parallelization of Inference Engine

The emphasis in this chapter is to incorporate the indexed C, parallel MATCH along with the control mechanism and the inference engine structure, into a production systems framework. The main control issue is regarding the mutually exclusive mode of updating working memory in shared memory; hence any operator that modifies the data has to treat the working memory as a Critical Section problem. Current day technology provides an efficient hardware solution to the Critical Section problem by non interruptible atomic instructions which are more efficient than the software solution based on *locks, semaphores* and etc.

Synchronization of concurrent programs on parallel computers is always a major implementation problem; that problem has been tackled in this chapter for the parallelized production systems program on hypercube machines. The synchronization problem had to grant shared memory write access to one processor for updating the contents as specified by the production system operator and invoke all nodes after the update has been successfully carried out. The atomic instruction capability is not currently available on the hypercube machines therefore the synchronizations and communications are based on software principles only; these features may be upgraded to the more efficient hardware atomic instructions as and when they become available from the manufacturer.

A hypercube computer, based on the Cosmic Cube [68], is a lossely coupled MIMD computer machine with 2^n processors and the number of neighbors of any

node is provided by the degree, n, of the cube. The address of a node is determined by a binary number as illustrated in figure 7.1 and the neighbors to any node are the n processors such that there is exactly one binary bit that is not the same in their respective address patterns. The figure illustrates a cube of dimension 3 with 8 processors and the set of neighbors for node 0 are {1,2,4} with binary address patterns being 000, 001, 010 and 100 respectively. All node processors are connected to a HOST processor that is not shown in diagram 7.1 but is clear in figure 8.6.

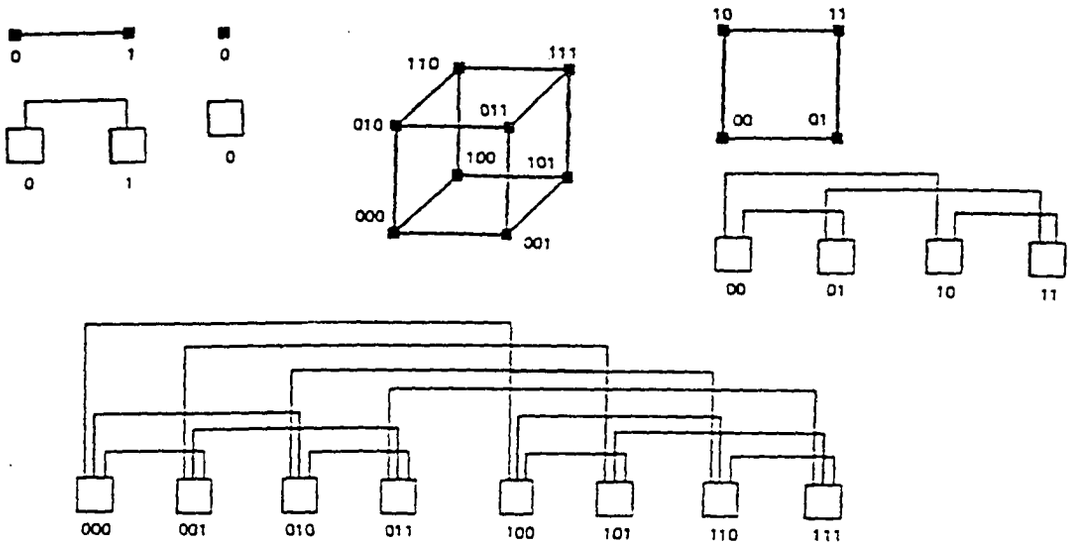


Figure 7.1: Hypercube configurations for degrees 0,1,2 and 3.

```
9. holds::obj:holds 11 6 21
-> [cs ] {at::obj:holds,holds::nil,,imposs (4 occurrences)}(holds::nil dominates)
3: (testcase  ^type general  ^name start)
4: (phys-obj  ^at 9-9  ^name bananas  ^on ceiling  ^weight light)
5: (phys-obj  ^at 7-7  ^name couch  ^on floor  ^weight heavy)
6: (phys-obj  ^at 4-3  ^name ladder  ^on floor  ^weight light)
19: (phys-obj  ^at 4-3  ^name blanket  ^weight light)
21: (monkey  ^at 4-3  ^holds blanket  ^on floor)
9: (goal  ^type holds  ^status active  ^obj-name bananas) ??????????????????
10: (goal  ^type at  ^status active  ^obj-name ladder  ^to 9-9)
11: (goal  ^type holds  ^status active  ^obj-name ladder)
17: (goal  ^type on  ^status satisfied  ^obj-name floor)
23: (goal  ^type at  ^status satisfied  ^to 4-3)
24: (goal  ^type holds  ^status active)nil **********************

10. holds::nil 24 19 21 Drop  blanket
-> [cs]{imposs (3 occurrences), at::obj:holds ,holds::obj-notceil}(holds::obj-notceil dominates)
3: (testcase  ^type general  ^name start)
4: (phys-obj  ^at 9-9  ^name bananas  ^on ceiling  ^weight light)
5: (phys-obj  ^at 7-7  ^name couch  ^on floor  ^weight heavy)
6: (phys-obj  ^at 4-3  ^name ladder  ^on floor  ^weight light)
30: (phys-obj  ^at 4-3  ^name blanket  ^on floor  ^weight light)
28: (monkey  ^at 4-3  ^on floor)
9: (goal  ^type holds  ^status active  ^obj-name bananas)
10: (goal  ^type at  ^status active  ^obj-name ladder  ^to 9-9)
11: (goal  ^type holds  ^status active  ^obj-name ladder)**********************
17: (goal  ^type on  ^status satisfied  ^obj-name floor)
23: (goal  ^type at  ^status satisfied  ^to 4-3)
26: (goal  ^type holds  ^status satisfied)nil

11. holds::obj-notceil 11 6 28 Grab  ladder
-> [cs] {imposs(2 occurrences),holds::obj-ceil:on:at-obj,at::obj} (at::obj dominates)
3: (testcase  ^type general  ^name start)
4: (phys-obj  ^at 9-9  ^name bananas  ^on ceiling  ^weight light)
5: (phys-obj  ^at 7-7  ^name couch  ^on floor  ^weight heavy)
30: (phys-obj  ^at 4-3  ^name blanket  ^on floor  ^weight light)
32: (phys-obj  ^at 4-3  ^name ladder  ^weight light)
34: (monkey  ^at 4-3  ^holds ladder  ^on floor)
9: (goal  ^type holds  ^status active  ^obj-name bananas)
10: (goal  ^type at  ^status active  ^obj-name ladder  ^to 9-9)**********************
17: (goal  ^type on  ^status satisfied  ^obj-name floor)
23: (goal  ^type at  ^status satisfied  ^to 4-3)
26: (goal  ^type holds  ^status satisfied)
36: (goal  ^type holds  ^status satisfied  ^obj-name ladder)nil
```

Figure 7.2: Recency based Production Rule selections in OPS5.

7.1. Inference Engine Activities

All production systems consist of the MATCH, Conflict Resolution and Act modules that carry out the recognise act cycle. The emphasis is on identifying aspects of inference engine that need to be translated into the C Code and then considered for parallelization on the hypercube environment. The two main activities may be summarized as:

- *Conflict Resolution:* Based on *recency* select a production rule from the conflict set formed by the MATCH module; if empty halt the production system.
- *Act:* This module executes the RHS of a selected production rule. The RHS may be a conclusion/deduction or any of the operators *{ modify, make, remove }* that affect the working memory and start MATCH. For *APS* class of production systems, the RHS may involve invocation of some external modules.

7.2. Transformation to C Code

The advantages and principles behind partitioning of the the working memory into blocks of homogeneous data have been discussed. All partitions are stored in the shared memory of the cube and available to the node processors in a MIMD mode. The production memory is also partitioned and distributed over the cube as indexed C code, where maximum rules per node is $\lceil \frac{V_{max}}{N} \rceil$.

7.2.1. C Coded Conflict Resolution

Figure 7.2 is a vivid illustration about the internal workings of any OPS5 production systems. The information is generated by an actual execution [8, 70, 71] of the Monkey and Banana OPS5 program. The noticeable aspects from that piece of OPS5 program execution tracefile are

[1]. The program is based on forward chaining problem solving strategy. Primary goal is working memory element # 9 highlighted with a sequence of question marks '???' in figure 7.2.

[2]. At any given instance more than one production rules are found to be true and form the conflict set as illustrated by the various sets following the OPS5 interpreter command 'cs'.

[3]. The dominating rule or the rule to be selected by the conflict resolution phase is always the rule that is satisfied by the most recent data element of the class goal and whose attribute status has value active as illustrated in the figure with the characters '*****'. The data element is of the type *goal ^status active.*

```
---------------------------------------------------------------------------
while (WM_gol[GL_cur]->stat == ACTV) {  /* If current goal is active. */
  switch (indx) {                 /* Stack pointer is GL_cur, for current goal. */
   case 22313210 :      {       RHS ACTIONS;      }
     break;
    .....
    .....
  } /* end switch*/
} /* end while */
---------------------------------------------------------------------------
```

Figure 7.3: Stack based Production Rule selections in indexed C code.

The above three observations lead to a stack oriented data structure for the partitioned memory containing the data elements of type goal as illustrated in figure 7.3. The MATCH phase will only consider the latest *goal status ^active* rule hence the conflict set resolution phase is trivialized, and the inference engine will determine whether the goals were satisfied or not after the RHS action of a successful rule.

The RHS actions of the selected rule has the task of updating the global variable *GL_cur*, modifying the data in working memory and also generating the indexing value for the variable indx all of which are in the shared memory of the system and have to be modified in a mutually exclusive mode.

7.2.2. C Coded ACT

The RHS actions of the parallelized and indexed C coded production systems have to execute the following sequence of steps as a RHS ACT algorithm which is illustrated in the implemented version of the Monkey and Banana production system in figure 7.4.

[1]. Set the lock for exclusive access to the data elements.

[2]. Perform variable binding for non indexed data elements.

[3]. Wait for other node processors to report a failure for their domain of the MATCH phase.

[4]. If RHS action creates a new element by the OPS5 *make* command then the C code needs to push the element on the appropriate stack and increment the pointer to the top of that stack.

```
---------------------------------------------------------------------------------

while ((GL_cur > -1) && (WM_gol[GL_cur]->stat == ACTV)){ /* If current goal exists and active
  switch (indx) {
     case 22103231 :
       {
[1]       lock(&actlck);
[2]       for(i=0;i<=OB_cur;i++) if(WM_mon->holds == WM_obj[i]->nam)
                      mon_obj_loc = i;
[3]       while(NO_mat < 2) tsleep(5);
          WM_obj[mon_obj_loc]->on = FLOR;
          WM_mon->holds = NIL;
[5]       WM_gol[GL_cur]->stat = SATS;
[5]       GL_cur--;
[6]       indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000
                + WM_mon->on*100 + WM_mon->holds;
          printf("%s %s 0,msg1[7],msg2[WM_obj[mon_obj_loc]->nam]);
          if(NO_mat == 2) {
[7]         unlock(&actlck);
[8]         NO_mat = -2;
            flag1 = 99;
[9]         evpost(&actrlse);
[9]         evclear(&actrlse);    }
       }        break;

     case 22333231  :
       {
          lock(&actlck);
          while(NO_mat < 2) tsleep(5);
[4]       WM_gol[GL_cur + 1] = GolCr(ACTV,HOLD,NIL,NIL);
          indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000
                + WM_mon->on*100 + WM_mon->holds;

      /**** SAME AS ABOVE RULE       ****/

       }        break;
  } /* end switch*/
[10] /* inference engine code */
} /* end while */
---------------------------------------------------------------------------------
```

Figure 7.4: Parallelized RHS actions in indexed C code.

[5]. If the RHS action deletes an element then tha C code has to emulate a pop operation on the stack.

[6]. Evaluate the indx variable, used in new recognise act cycle.

[7]. Release the lock for mutual exclusive access mode to data.

[8]. Initialise all variables required by the inference engine module.

[9]. Post and Clear the eventflags, enabling other nodes to start MATCH.

[10]. Pass control to the inference engine module for both a successful and a failure in the MATCH module.

7.2.3. C Coded Inference Engine

The inference engine has to determine when to break the recognise act cycle, check the status of the system, synchronize all activities and reactivate the cycle. The five steps mentioned below are encoded in the distributed version of the parallelized C coded Monkey and Banana program in Appendix E and highlighted in figure 7.5.

[1]. Determine if a successful match for a production rule is found or not. This is indicated by setting a global variable *NO_mat* = *-2* and local variable *flag1* = *99*.

```
TASK
ruls1()
{
   int i,j,k,flag1;
   while ((GL_cur > -1) &&(WM_gol[GL_cur]->stat == ACTV)) {
[1]    flag1 = -99;
       switch (indx) {
       case 21323135 :
         {
           lock(&actlck);
           while(NO_mat < 2) tsleep(5);
           WM_mon->on = FLOR;  WM_gol[GL_cur]->stat = SATS;   GL_cur--;
           indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000
                        + WM_mon->on*100 + WM_mon->holds;
           printf("%s 0,msg1[3]);
           if(NO_mat == 2) {
             unlock(&actlck);
[1]            NO_mat = -2;       flag1 = 99;
             evpost(&actrlse);   evclear(&actrlse);
             }
         }     break;
       default :    tsleep(1);  }

[2]    lock(&match);     NO_mat++;      unlock(&match);

[3]    if((flag1 < 0) && (NO_mat > -1) && (NO_mat != 3)) evwait(&actrlse);

[4]    if ((NO_mat == 3) || (GL_cur == -1)){
         lock(&match);    NO_mat++;    unlock(&match);
[4] i.    if((GL_cur > -1) && (WM_gol[GL_cur]->stat == ACTV))
           printf("%s %s %s0,msg1[1],msg2[WM_gol[GL_cur]->nam],msg1[2]);
         else {
[4] ii.       printf("%s 0,msg1[0]);  }
[5]      texit(gettid()); }
   }
}
```

Figure 7.5: Parallelized Inference Engine in indexed C code.

[2]. If a successful match is found, the node processor with the success will reset the global variable *NO_mat* to -1 at the end of the *switch(indx) { ... }* loop and repeat the recognise act cycle.

[3]. If a match is not found by the current processor node and some other nodes have not completed the MATCH then the node waits for an event to be posted.

[4]. If all nodes have completed the MATCH phase without a successful match, then there are two possibilities:

i. All the rules are *satisfied* and no goal is *active*

ii. The current active goal is not attainable.

[5]. Irrespective of the system status in step [4], report the status and halt the system.

7.3. Configuration of Distributed Production on Hypercube

The dissertation has provided a framework for parallelizing an OPS5 production system into a C encoded, hypercube program. This process required some fundamental algorithm [1, 16, 34, 72] design principles that have to be considered in any translation scheme from sequential code to a parallel program. The dissertation research considered various parallel architectures as a possible implementation for the indexed C code; the types of computers that were studied fell in the tightly coupled and loosely coupled [61] class of multiprocessors. Both classes of computers could be used in either SIMD or MIMD modes of operations.

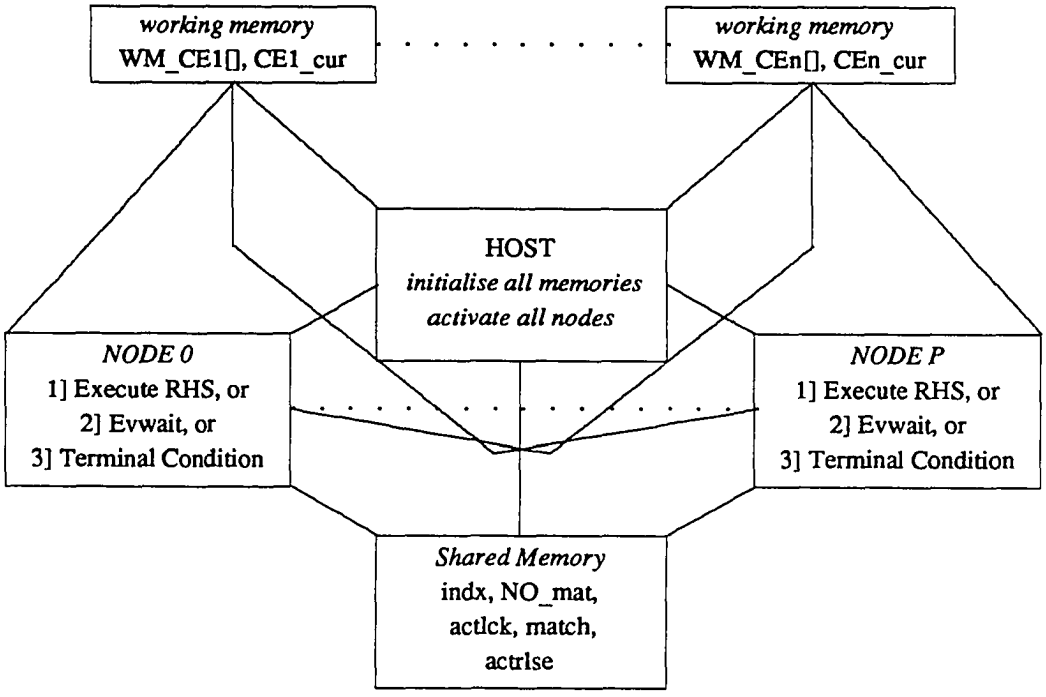


Figure 7.6: Distributed Production System representation.

- Number of nodes is $P + 1$ and identified as nodes 0 .. P.
- Number of homogeneous partitioned working memory blocks is n where n is the number of elements in the production system.
- Shared Memory keeps track of the global variables required by the distributed inference engine module.

Tightly coupled systems have one block of common memory that is available to all node processors via switching networks; none of the individual processors have any local memory. More than one processor may be active at any given instance in the shared memory block and the switching network eliminates the bottlenecks that exist in any shared communications bus in a multiprocessor environment. Some of the prominent tightly coupled multiprocessor parallel computers are the HEP machine [15, 35, 44, 73] produced by Denelcor Incorporated at Denver, the Ultracompute computer [28] at NYU, the C.mmp machine at [26, 47] CMU and the Multimax machine produced by ENCORE.

Loosely coupled architecture provides a small proportion of local memory to each node processor and the entire system is divided into clusters. Remote memory access by any processor is achieved by routing a request through a KMAP (Map Controller) processor. Some prominent loosely coupled machines are the *CM** [78, 79] machine at CMU and the *BBN™* [4] machine based on the butterfly architecture.

The parallel production system is mapped onto a hypercube simulator as illustrated in figure 7.6 and executed in a C environment on the VAX-11/780 computer which is controlled by the BSD 4.3 UNIX operating system. The hypercube simulator used for the dissertation research is based on Brook's [7] multitasking kernel for C and Fortran programs and is used to emulate the INTEL Corporation's iPSC/2 [36] parallel Scientific computer using message passing and shared memory available to the cube via the HOST program or machine. Details about the simulator may be found in reference numbers [19,20] that were made available to the Computer Science Department at LSU, courtesy of the Oak Ridge National Laboratory.

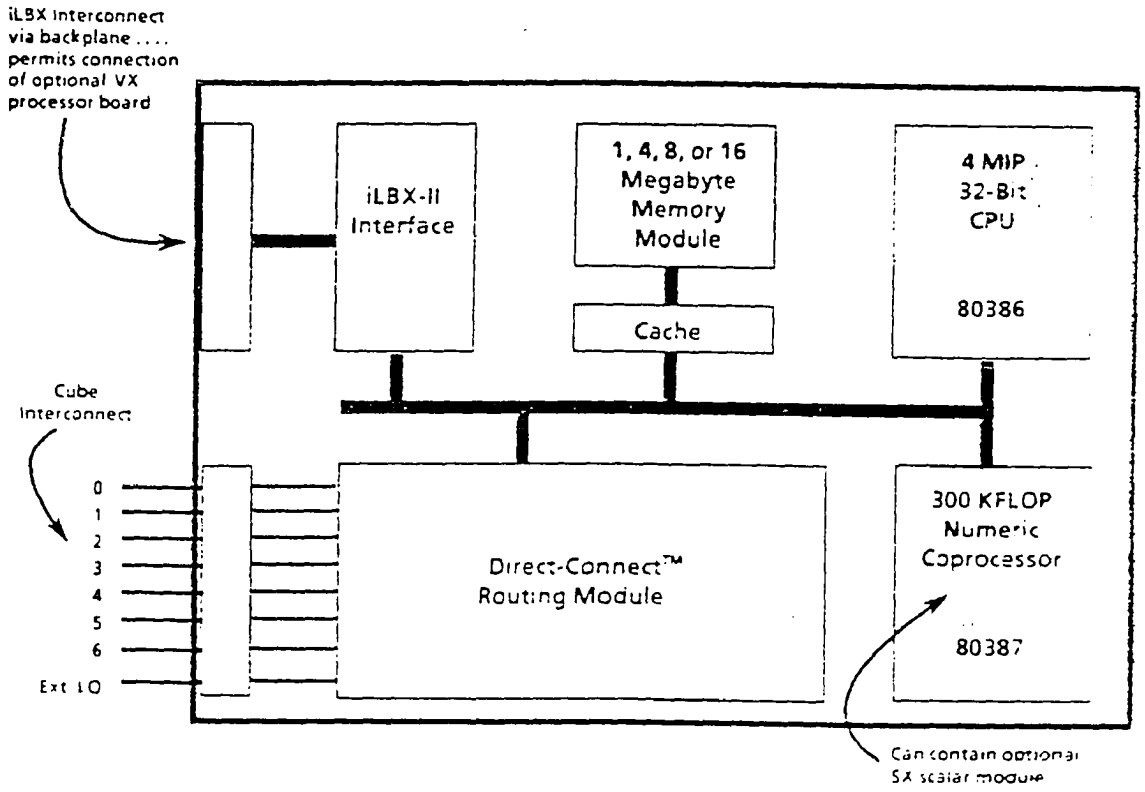


Figure 7.7: iPSC/2 Node Processor

7.3.1. HOST Processor

The HOST processor in a true iPSC/2 hypercube is an Intel Multibus-based System 310AP microcomputer that is connected to each node by an Ethernet communication channel. The HOST is provided as an user interface [36] to the cube for program development and system diagnostics. The HOST is run under the XENIX operating system.

The message passing hypercube simulator used for this research and emulated the HOST as a TASK with task identity number -1, and it is connected to all other node processors. The node processors exist as a set of concurrent TASK in the UNIX environment. The purpose of the HOST program or *task0* is to initialise all variables and simulator parameters for message propagation delays and load the appropriate user software on the nodes processors to make the cube a true MIMD machine.

7.3.2. NODE Processor

The iPSC/2 hypercube is a machine with 2^d nodes and is of the dimension d that denotes the number of nearest neighbors to any given node. Current limitations on hardware restrict the value of d to be less than 8. Figure 7.7 depicts a typical node processor and the CPU is a 32 bit Intel 80386 microcomputer. This 32 bit processor provides a medium grained parallel implementations unlike the small grained programming required by a DADO2 machine that is implemented as a complete binary tree of smaller microprocessors with local memory. The main components of the node processor are

[1]. **Processor :** A 32 bit machine that can execute LISP, FORTRAN , C and Assembly programs at up to 4 MIPS and address up to 16 Megabytes of physical memory.

[2]. **Numeric Coprocessor :** A powerful arithmetic processor, the Intel 8037 chip is capable of 0.3 Megaflops floating point operations at 64 bit precision. Local memory provides some standard algebraic and trigonometric functions. An optional SX Scalar eXtension module provides up to 1.2 Megaflops double precision peak performance. An optional VX Vector eXtension provides up to 6 Megaflops double precision peak performance.

[3]. **Memory and Cache :** Each node has memory modules of up to 16 Megabytes which is mapped through a 64 Kilobyte fast access cache memory providing up to 91% hit rate for some bench mark programs.

[4]. **Communication :** Each node has a dedicated connection routing module for high-speed node to node communication with up to 7 neighbors. Transfer rate is at 2.8 megabytes per second and the I/O is FIFO buffered with buffer size of 1 Kilobyte. A special External I/O channel is provided for fast I/O communications with other peripherals.

[5]. **DMA :** For high speed data transfer in between nodes each node has a DMA controller and an iLBX Intel Local Bus Exchange interface that are provided for nodes with the VX Vector processor for transfer rates of up to 10 Megabytes.

[6]. **Operating System :** Each node has its own multi-tasking operating system called NX/2 Node eXecutive/2, that ensures reliable and optimal

message communications via the direct-connect module. NX/2 also supports a variety of programming languages and some debugging support for program development.

[7]. *Indicators :* Three LEDs Light Emitting Diodes are provided to indicate the activity of various modules and for simple diagnostics. The LED for CPU is green, for Direct-Connect is red and numeric coprocessor computation activity indicated by the amber LED.

7.4. Simulator Software Features

The message passing multiprocessor [7, 19, 20] simulator is capable of providing a local-memory and shared-memory programming environment to the user. The multiple processors are simulated by UNIX tasks that are running concurrently on a single CPU VAX 11/780 computer. The simulator provides task management facilities, message passing facilities, mutual exclusive mode of operation in sections of shared memory and a trace file that can be used for cube performance and debugging diagnostics. The MIMD hypercube machine is emulated by the simulator and it may be reconfigured to suit the requirements of the user.

7.4.1. Task Management

All user programs are classified as a special data structure called TASK as illustrated in appendix D. The main program is *task0* and is used as the HOST program. The set of commands that are used are :

tfork This is used to invoke the node processors in parallel. The nodes are identified as tasks numbers that begin from 0 and are monotonically increasing. The software to be loaded on the node is the piece of subroutine program code following the definition statement in the main program file.

texit This command is used to force an exit from a node and terminate its activity.

twait This is used by the parent process or HOST to terminate after a specified task has terminated.

tsleep This is equivalent to the VMS hibernate feature and is used for synchronising the cube.

7.4.2. Message Passing

The simulator is capable of synchronous and asynchronous message passing in between nodes and the HOST program. The message passing protocol is based on the Caltech Cosmic Cube [68] and large messages are partitioned into smaller packets if required. The packets are of size 8^i where $i \geq 1$ and $i \leq 11$, the largest packet allowed by the simulator is size 16000 bytes.

copen This is used to establish a variable *ci*, of the class CHNL used subsequently as a channel identifier for message passing purposes. The channel is closed by the **cclose** command.

sendw This is a synchronous version of message transmission initiated by any

node. The HOST process has to use the command **sendmsg**. The message is classified by *type,address,size* and the receiver is specified by *node,pid* since multiple processes may be existing at any node. Format of the command is *sendw(ci,type,mesg,size,node,pid)*.

recvw This is a synchronous version of message reception by any node. The HOST process has to use the command **recvwmsg**. The message is classified by *type,address,size* and the receiver is specified by *node,pid* since multiple processes may be existing at any node. Format of the command is *sendw(ci,type,mesg,&size,&node,&pid)*.

status This command is used for checking a specified channel identifier *ci* for asynchronous versions of **send** and **recv** and is used in conjunction with the **flick** command to emulate synchronous activity.

probe This command is used in asynchronous message passing to see if a particular *type* of message is present in a specified *ci*. A **recv** would still be required to retrieve the message.

7.4.3. Shared Memory Access

The mutual exclusive mode of accessing of shared variables is similar to the classical Operating Systems problem of Critical Section [14, 17, 34, 60, 67, 80] with numerous solutions that have been proposed for a single CPU and a multiprocessor environment. The simulator provides features to ensure mutual exclusion and ensure deadlock free mode of operations in the distributed environment.

lock This command is used to lock an unlocked variable of the class LOCK; if the variable is already locked the process is suspended. The **unlock** command release the next process in line for that variable. These two functions are used to emulate the mutual exclusion actions of semaphores.

evwait A process is suspended till an even is posted on a variable of class EVENT. The data type EVENT is in no way associated with the EVENT, COUNTERS and SEQUENCERS of Reed and Kanodia's [62] distributed synchronization algorithm attributes.

evpost All processes that are waiting for that event to be posted are reactivated.

evclear A variable of the type EVENT is cleared and is generally used after an **evpost** command.

7.4.4. Software Support

These are a collection of some useful and general purpose routines and functions that are made available to the simulator programmer for software development and debugging purposes.

aspp This is the Assembler Post Processing mode of compiling the program that is to be executed on the cube and inserts an interrupt after each executable instruction to read the global system clock. The time ordering of events is based on Leslie Lamport's [45] algorithm for a distributed environment.

bld This is an executable file that invokes the appropriate makefiles and linkages for a source code file and generating an executable file with the same

name except for the postfix '.c'.

strace This is used to generate a tracefile, as illustrated in appendix E. The file is used by the **aspp** and **nstats** post processing programs; it may also be used for debugging purposes. The command to end a trace file is **etrace**.

nstats This program accepts a tracefile generated by a program that was compiled in the **aspp** mode and provides runtime performance diagnostics of the cube. Appendix D provides a couple of such diagnostics.

MISC A miscellaneous set of commands that may be required by the programmer; they include **clock** to return the global clock time, **cubedim** for the dimension of the cube, **cube_init** to set up the initial parameters for the simulator, **mynode** for obtaining the node number, **syslog** for inserting marker messages in the tracefile and **gettid** for obtaining the task identity number.

Chapter Eight

Summary and Conclusions

8.1. Problem Analysis

The basis for this research was to provide for an efficient MATCH scheme for production systembased programs, in a parallel environment, exploiting medium scale parallelism of hypercube machines. production system is a rule based, programming model that provides a scheme for representing knowledge, expertise and a means for expressing problem solutions. The evaluation or MATCH function of any rule based program has to perform an *incremental equivalence* operation on the data.

Considering the data in table 4.1 and the rules in figure 4.2 for the Goal directed, forward chaining Monkey and Banana, OPS5 program as a sample representation the analysis for the various search schemes is outlined in subsequent sections. For all the analyses, the parameters are

1. P represents the number of production rules.
2. W represents the average number of data elements in working memory.
3. C represents the average number of condition elements per production rule.

8.1.1. Simple Search

The basis for a simple search is to iterate the rules over the data. For a search scheme that is based on elements the time complexity to complete one iteration for a conflict set is of the order $O(\lceil P \times W^C \rceil)$.

In reality the search or equivalence is always at the attribute levels and the true order of complexity is given by the formula $\sum_{i=1}^{P}[\prod_{j=1}^{C}(CE_{ij}^{K_j})]$. $CE_{ij}^{K_j}$ represents represents the number of attributes the data element possesses in the working memory. Where the condition element is of the j^{th} order, for i^{th} production rule and K_j is the number of attributes in the condition element that need to be bound with the data values.

8.1.2. RETE Search

The basis of the RETE search algorithm was to transform the production rules into an indexed network similar to a dataflow network. In the process of reducing the number of attributes that need to be matched the network introduced state saving mechanisms and thereby increased the number of intermediate results for the evaluate operations. Forgy [24] has proved that worst case time complexity for the RETE network is $O(\lceil W^{2\times C-1} \rceil)$.

8.1.3. TREAT Algorithm

TREAT evolved from the failure of RETE in a parallel environment. TREAT was based on the fine grained parallelism architecture of the DADO machine and its special I/O communicating channels and associative memory. TREAT also introduced the concept of partitioning the working memory and the production memory to operate the production systems in a MIMD mode. The time complexity of the MATCH phase is $O(\lceil AR \times NM \times OM^{C-1} \rceil)$. is affected by the following factors:

Table 8.1: Brief summary of some important results.

	RETE (Forgy)	TREAT (Miranker,Stolfo)	NEW (Shrivastava,Iyengar)
Basis	Indexed Search Tree & Dataflow architecture	Special Datastructures & Hardware for Communications	Shared Memory MIMD & C Language Datastructures
Advantages	1. Reduced the number of match conditions. 2. Simplified Match by retaining partial match. 3. Simple representation for negative logic.	1. Reduced redundant info. 2. Retained conflict set. 3. Special datastructures and algo. for parallel implem.	1. Partitioned working memory. 2. Indexed Match using arithmetic functions. 3. Special datastructures for goal driven prod. systems. 4. Message passing capabilities provide for future expansions.
Disadvantages	1. Too much memory for saving all partial matchs. 2. Time consumed in unwinding after a delete/modify action. 3. Bottlenecks in data flow for parallel implementations.	1. Large *working memory* is req--ired at lower level of tree. 2. Requires a full binary tree of microprocessors. 3. Set of active rules is entire *production memory* for goal driven *production systems*.	1. Recomputes conflict set. 2. Time is lost in synchronising shared mem. 3. Indexing scheme fails for large *working memory* set.
Performance	$O(W^{2C-1})$	$O(R \times W^{C})$	$O(\frac{W'^{C-i}}{N})$

- W is average number of elements in *working memory*
- C is average number of condition elements in the LHS of a *production rule*
- R is the number of affected rules in the TREAT algorithm.
- i is the number of condition elements used for indexing onto the *production rules*.
- W' is average number of elements in *working memory* without the indexed elements.
- N is the number of node processors in the hypercube implementation.

1. *working memory:* Search is initiated by *new-add* and *new-delete* partitions only; since the *old-mem* has already been processed and results are available in the *conflict set.* The new memories are denoted by *NM* and the old memory is denoted by *OM*.
2. *production memory:* Search is restricted to the active set of rules, denoted by *AR* and not the entire *production memory.*

8.2. Conclusions

A brief summary of related research and major contributions of this research is provided in table 8.1. The performance analysis is based on worst case scenario for all algorithms. The research had two major goals

1. Provide an efficient MATCH phase in a parallel environment.
2. Provide a programming shell for production systems on the MIMD architecture using the available resources.

The research goals were successfully achieved and have been reported in this dissertation. Using an indexing scheme for i condition elements and an N processor hypercube environment, the parallel MATCH algorithm results in a time complexity of the order $O(\lceil W'^{C-i}\frac{1}{N}\rceil)$. W' is the partitioned form of working memory and consists of data elements that were not used in the indexing scheme.

This dissertation has identified some parallelisation techniques for production systems and optimised the MATCH phase for, forward chaining goal directed production systems. The optimization was a direct consequence of using C language to

transform the string equivalence operation to integer operations and an integer arithmetic based hashing function that controls the search. Since current day parallel computers are in the development stages most machines have limited software for program development and support; this enforces the use of C programming language, commonly termed as modern day assembly language.

8.3. Future Research

This research was targeted at optimising a class of production systems that are commonly used in robotics and other real time applications. This telescopic approach has led to certain limitations in the current form of the indexed, distributed production system implemented in C on a hypercube with shared memory capabilities. The limitations are identified below and form the core for future research.

1. The character string to integer transformation of attribute values presumes a small set of values since the upper limit on the number of indexed production rules is given by the cross product, $(\lceil S_1 \otimes S_2 \otimes \cdots \otimes S_k \rceil)$, where S_i is the set of values associated with the i^{th} index attribute in the set ATR_{ind}. If any attribute has a large set of values the production memory size increases in a combinatorial explosive manner.

2. Since conflict set is not retained, the program has to recompute for the cognitive class of problem like Algebraic Simplifications, Robotic Range Sensing and Computer Vision.

3. Does not support backward chaining and non goal directed problem solving, hence program has to explicitly keep track of start state.

4. Non deterministic reasoning has not been studied for implementation and needs to be addressed in future.

5. Current version of the distributed program uses *critical section* approach to updating shared memory data; future research should consider atomic instruction type of implementation at hardware level.

6. An important aspect of parallelisation is the ability to amalgamate other production systems as concurrent, cooperating modules in a true autonomous machine. The inference engine will be extremely complex for such a machine.

7. Since any parallel program spends a significant proportion of its time waiting for synchronization primitives to be set, future research should focus on providing a learning capability to some of the node processors not involved in the recognise act cycle. Information exchange could be based on message passing or shared memory, principles.

References

[1] AHO, ALFRED V., HOPCROFT, JOHN E., AND ULLMAN, JEFFREY D., in *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, Readings Massachussetts, 1976.

[2] AHO, ALFRED V., SETHI, R., AND ULLMAN, JEFFREY D., in *COMPILERS Principles , Techniques and Tools* , Addison-Wesley Publishing Company, Readings Massachussetts, 1986.

[3] BASTANI, FAROUKH AND IYENGAR, S. SITHARAMA, "The Impact of Parallel Algorithms on Software Quality," *Proceedings of the Super Computing Conference, Florida*, 1985.

[4] BBN, "Butterfly Parallel Processor Overview," *Version 1, Technical Report*, $ BBN sup TM$ Laboratories Incorporated, Cambridge, MA, December 1985.

[5] BENTLEY, J. L., "Decomposable Searching Problems," *Information Processing Letters*, vol. 8, no. 5, pp. 244-250, 1979.

[6] BERNSTEIN, A. J., "Program Analysis for Parallel Processing," *IEEE transactions on Electronic Computers*, vol. EC-15, no. 5 , pp. 757-762, October 1966.

[7] BROOKS, E. D., "A Multitasking Kernel for the C and Fortran Programming Language," *Technical Report*, no. UCID-20167, Lawrence Livermore National Laboratory, September 1984.

[8] BROWNSTON, L., FARRELL, R., KANT, E., AND MARTIN, N., in *Programming Expert Systems in OPS5*, Addison-Wesley Publishing Company, Readings Massachussetts, 1985.

[9] BURKS, B. L., SAUSSURE, G. DE, WEISBIN, C. R., JONES, J. P., AND HAMEL, W. R., "Autonomous Navigation, Exploration and Recognition using the HERMIES-IIB Robot," *IEEE Expert*, vol. 2, no. 4, pp. 18-31, 1987.

[10] CHANG, HSI AND IYENGAR, S. SITHARAMA, "Efficient Algorithms to Globally Balance a Binary Search Tree," *Communications ACM*, vol. 27, no. 7, pp. 697-702, 1984.

[11] CHARNIAK, E. AND MCDERMOTT, D., in *Introduction to Artificial Intelligence*, Addison-Wesley Publishing Company, Readings Massachussetts, 1985.

[12] CROWLEY, J. L., "Co-ordination of Action and Perception in a Surveillance Robot," *IEEE Expert*, vol. 2, no. 4, pp. 32-43, 1987.

[13] DAVIS, R., BUCHANAN, BRUCE G., AND SHORTLIFFE, EDWARD H., "Production rules as a representation for knowledge-based consultation

program," *Artificial Intelligence*, vol. 8, no. 1, pp. 15-45, 1977.

[14] DEITEL, HARVEY M., in *An Introduction to Operating Systems*, Addison-Wesley Publishing Company, Readings Massachussetts, 1988.

[15] DENELCOR, *HEP/UPX Reference Manual*, Denelcor Incorporated, 1984.

[16] DEY, PRADIP, IYENGAR, S. SITHARAMA, AND BROWN, JOHN T., "Some Issues in the Design of Parallel Search Algorithms ," *Proceedings of the ACM 1987 Conference*, 1987.

[17] DINNING, A., "A Survey of Synchronization Methods for Parallel Computers," *Computer, IEEE* , vol. 22, no. 7, pp. 66-77, July 1987.

[18] DUDA, RICHARD O., GASCHNIG, JOHN G., AND HART, PETER E., "Model Design in the PROSPECTOR consultant system for Mineral Exploration," in *Expert Systems in the Microelectronic Age*, pp. 153-167, Edinburgh University Press, 1980.

[19] DUNIGAN, T. H., "A Message-Passing Multiprocessor Simulator ," *Technical Report*, no. ORNL/TM-9966, Oak Ridge National Laboratory, May 1986.

[20] DUNIGAN, T. H., "A Portable Hypercube Simulator," *Technical Report*, no. ORNL/TM-10410, Oak Ridge National Laboratory, June 1986.

[21] ERMAN, L., HAYES-ROTH, F., LESSER, V., AND REDDY, D., "The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty," *Computing Surveys*, vol. 12, no. 2, June 1980.

[22] ERMAN, L., LONDON, P., AND FICKAS, S., "The Design and an Example Use of Hearsay-III," *Proceedings of the Seventh International Joint Conference on Artificial Intelligence* , vol. 1, pp. 409-415, 1981.

[23] FORGY, C. AND GUPTA, A., "Preliminary Architecture of the CMU Production System Machine," *Proceedings of the $19 sup th$ Annual Hawaii International Conference on System Sciences*, pp. 194-299, 1986.

[24] FORGY, CHARLES L., "RETE : A Fast Algorithm for the Many Pattern / Many Object Pattern-Match Problem," *Artificial Intelligence* , vol. 19 , pp. 17-37, 1982.

[25] FOX, MARK S., "AI and Expert System Myths, Legends, and Facts," *IEEE Expert*, vol. 5, no. 1, pp. 8-20, 1990.

[26] FULLER, S. H. AND OLEINICK, P. N., "Initial Measurements of Parallel Programs in a Multi-Miniprocessor," *Proceedings of $13 sup th$ IEEE Computer Society International Conference*, pp. 358-363, 1976.

[27] GAUDIOT, JEAN-LUC AND SOHN, ANDREW, "Data-Driven Parallel Production Systems," *IEEE Transactions on Software Engineering*, vol. 16, no. 3, pp. 281-293, March 1990.

[28] GOTTLIEB, A., GRISHMAN, R., KRUSKAL, C. P., MCAULIFFE, K. P., RUDOLPH, L., AND SNIR, M., "The NYU Ultracomputer: Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers*, vol. C-32, no. 2, pp. 175-189, 1983.

[29] GUPTA, A., "Implementing OPS5 Production-Systems on DADO," *International Conference on Parallel Processing IEEE*, pp. 83-91, 1984.

[30] GUPTA, A. AND FORGY, C., "Measurements on Production Systems," *CMU-CS-TECHNICAL REPORT*, no. 167, 1983.

[31] GUPTA, A., FORGY, C., AND NEWELL, A., "High-Speed Implementation of Rule-Based Systems," *ACM Transactions on Computer Systems*, vol. 7, no. 2 , pp. 119-146, May 1989.

[32] GUPTA, A., KALP, D., FORGY, C., NEWELL, A., AND TAMBE, M., "Results of Parallel Implementation of OPS5 on the ENCORE Multiprocessor," *CMU-CS-TECHNICAL REPORT*, no. 146, 1987.

[33] GUPTA, A. AND TAMBE, M., "Suitability of Message Passing Computers for Implementing Production-Systems," *National Conference on Artificial Intelligence AAAI*, pp. 687-692, 1988.

[34] HOPCROFT, JOHN E. AND ULLMAN, JEFFREY D., in *Introduction to Automata Theory, Languages, and Computation* , Addison-Wesley Publishing Company, Readings Massachussetts, 1979.

[35] HWANG, K. AND BRIGGS, F. A., in *Computer Architecture and Parallel Processing*, pp. 669-684, McGraw-Hill Book Company, 1984.

[36] INTEL, *Intel iPSC/2 Simulator Manual,* Intel Incorporated, 1987.

[37] ISHIDA, T. AND STOLFO, SALVATORE J., "Towards the Parallel Execution of Rules in Production System Programs," *Proceedings of International Conference on Parallel Processing IEEE*, pp. 568-575, August 1985.

[38] IYENGAR, S. SITHARAMA, JORGENSEN, C. C., RAO, S. V. NAGESVARA, AND WEISBIN, C. R., "Robot Navigation Algorithms using Learned Spatial Graphs," *Robotica*, vol. 4, 1986.

[39] IYENGAR, S. SITHARAMA, RAO, S. V. NAGESVARA, AND KASHYAP, R. L., "Parallel Range Search Algorithm Using Multiple Attribute Trees," *Proceedings of International Conference on Parallel Processing IEEE*, August 1985.

[40] IYENGAR, S. SITHARAMA, WEISBIN, C. R., AND PIN, F., "Asynchronous Production Systems for Controlling Autonomous Mobile Robots," *To appear in, Proceedings of $8 sup {th}$ International Congress of Cybernetics and Systems, New York*, June 11-15, 1990.

[41] IYENGAR, S. SITHARAMA, WEISBIN, C. R., AND PIN, F., "Asynchronous Production Systems for Control of an Autonomous Mobile Robot in Real Time Environment," *To appear in, Journal of Applied Artificial Intelligence.*

[42] KAHN, GARY AND MCDERMOTT, JOHN, "The MUD System," *Proceedings of the First International Joint Conference on Artificial Intelligence, IEEE and AAAI*, December, 1984.

[43] KERAVNOU, E. T. AND JOHNSON, L., in *Competent Expert Systems*, McGraw-Hill Book Company, 1986.

[44] KOWALIK, J. S., in *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, MIT Press, Cambridge, MA, 1985.

[45] LAMPORT, L., "Time, Clocks and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558-565, July 1978.

[46] LINDSAY, R. K., BUCHANAN, BRUCE G., FEIGENBAUM, E. A., AND LEDERBERG, J., "Applications of Artificial Intelligence for Organic Chemistry: The Dendral Project," *Artificial Intelligence*, vol. 11, no. 2, pp. 5-24, 1978.

[47] MASHBURN, H. H., "The C.mmp/Hydra: An architectural Overview," *CMU-CS-TECHNICAL REPORT*, 1979.

[48] MCCARTHY, J., "Recursive Functions of Symbolic Expressions and Their Computation by Machine," *Communications ACM*, vol. 21, no. 12, pp. 184-195, 1978.

[49] MCDERMOTT, J., "R1: A Rule-based Configuration of Computer Systems," *Artificial Intelligence*, vol. 19, pp. 39-88, September 1982.

[50] MCDERMOTT, J., KIM, J., AND SIEWIOREK, D., "Exploiting Domain Knowledge in IC Cell Layout," *Design and Test Magazine of IEEE*, August 1984.

[51] MCDERMOTT, J., NEWELL, A., AND MOORE, J., "The Efficiency of Certain Production System Implementations," in *Pattern-directed Inference Systems*, Academic Press, 1978.

[52] MIRANKER, DANIEL P., "Performance Estimates for the DADO machine : A Comparison of TREAT and RETE ," *Proceedings of International Conference on Fifth Generation Computer Systems*, June 1984.

[53] MIRANKER, DANIEL P., "TREAT : A Better Match Algorithm for AI Production Systems," *AI Architectures*, pp. 42-46, 1986.

[54] MOITRA, A. AND IYENGAR, S. SITHARAMA, "Derivation of Parallel Algorithms from Sequential Programs," *IEEE Transactions on Software Engineering*, vol. SE22, December 1985.

[55]. MOITRA, A. AND IYENGAR, S. SITHARAMA, "A Maximally Parallel Balancing Algorithms for Obtaining Complete Balanced Binary Trees," *IEEE Computers*, vol. C-34, no. 6, June 1985.

[56] MOORE, R. L., HAWKINSON, L. B., KNICKERBOCKER, C. G., AND CHURCHMAN, L. M., "A Real-time Expert System for Process Control," *Proceedings of the First International Joint Conference on Artificial Intelligence, IEEE and AAAI*, December, 1984.

[57] NEWELL, A., "Production Systems: Models of Control Structures," in *Visual Information Processing*, pp. 463-526, Academic Press, 1973.

[58] OFLAZER, KEMAL, "Partitioning in Parallel Processing of Production Systems," *International Conference on Parallel Processing IEEE*, pp. 92-100, 1984.

[59] PARSAYE, KAMRAN, in *Expert Systems for Experts*, John Wiley & Sons, 1988.

[60] PETERSON, JAMES L. AND SILBERSCHATZ, ABRAHAM, in *Operating System Concepts*, Addison-Wesley Publishing Company, Readings Massachussetts, 1988.

[61] QUINN, MICHAEL J., in *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill Series in Supercomputing and Artificial Intelligence, 1987.

[62] REED, DAVID P. AND KANODIA, RAJENDRA K., "Synchronization with Eventcounts and Sequencers," *Communications of the ACM*, vol. 22, no. 2, pp. 115-123, February 1979.

[63] RICH, E., in *Artificial Intelligence*, McGraw-Hill Series in Artificial Intelligence, 1985.

[64] SABHARWAL, ARVIND S., IYENGAR, S. SITHARAMA, SAUSSURE, G. DE, AND WEISBIN, C. R., "Parallelism in Rule-based systems," *Proceedings Applications of Artificial Intelligence VI (SPIE)*, 1988.

[65] SABHARWAL, ARVIND S., IYENGAR, S. SITHARAMA, SAUSSURE, G. DE, WEISBIN, C. R., AND PIN, F., "Asynchronous Production Systems for Real-Time Expert Systems," *Proceedings of AVIGNON 1988: $8 sup {th}$ International Workshop on Expert Systems and their Applications*, 1988.

[66] SABHARWAL, ARVIND S., IYENGAR, S. SITHARAMA, WEISBIN, C. R., AND PIN, F., "Asynchronous Production Systems," *Journal of Knowledge Based Systems*, vol. 2, no. 2, pp. 117-127, June 1989.

[67] SCHNEIDER, FRED B., "Synchronization in Distributed Programs," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 2, April 1982.

[68] SEITZ, CHARLES L., "The Cosmic Cube," *Communications ACM*, vol. 28, no. 1, pp. 22-33, 1985.

[69] SHORTLIFFE, EDWARD H., in *Computer-Based Medical Consultations: MYCIN*, Elsevier, New York, 1976.

[70] SHRIVASTAVA, RAJENDRA K., "Some Empirical Results for the Monkey and Banana Production System on Hypercubes," *Unpublished Tech. Rept. at LSU* , March 1989.

[71] SHRIVASTAVA, RAJENDRA K., IYENGAR, S. SITHARAMA, AND CARVER, DORIS, "An Approach for Parallelizing OPS5 Production Systems with a Faster MATCH Scheme, based on Indexing," *Unpublished Tech. Rept. at LSU* , May 1988.

[72] SILVERMAN, BARRY G., "Distributed Inference and Fusion Algorithms for Real-Time Supervisory Controller Positions," *IEEE transactions on SMC* , vol. 17, no. 2 , 1987.

[73] SMITH, B. J., "A Pipelined Shared Resource MIMD Computer," *Proceedings of International Conference on Parallel Processing IEEE*, pp. 6-8, August 1978.

[74] STOLFO, SALVATORE J., "Five Parallel Algorithms for Production-Systems on the DADO Machine ," *Proceedings National Conference AI-AAAI*, pp. 300-307, June 1984.

[75] STOLFO, SALVATORE J., "Initial Performance of the DADO2 Prototype ," *IEEE Computers Special Magazine on AI Machines* , vol. 20, pp. 75-83, January 1987 .

[76] STOLFO, SALVATORE J. AND MIRANKER, DANIEL P., "DADO : A Parallel Processor for Expert Systems," *Proceedings of International Conference on Parallel Processing IEEE*, pp. 74-82, August 1984 .

[77] STOLFO, SALVATORE J. AND SHAW, D. E., "DADO : A tree structured Machine Architecture for Production Systems," *Proceedings of National Conference on Artificial Intelligence AAAI*, pp. 242-246, August 1982.

[78] STONE, H. S., "Parallel Computers," in *Introduction to Computer Architecture*, p. Chapter 8, Science Research Associates, Chicago, 1980.

[79] SWAN, R. J., BECHTOLSHEIM, A., LAI, K. W., AND OUSTERHOUT, J. K., "The Implementation of the $ Cm sup *$ Multi-microprocessor," *Proceedings of the National Computer Conference*, pp. 645-655, AFIPS Press, Reston, VA, 1977.

[80] TANENBAUM, ANDREW S., in *Operating Systems Design and Implementation*, Prentice-Hall Software Series, 1987.

[81] WARREN, D. H. AND PEREIRA, L. M., "Prolog-The Language and Its Implementation Compared to Lisp," *Proceedings of Symposium on Artificial Intelligence and Programming Languages, SIGPLAN Notices*, vol. 12, no. 8, pp. 74-82, 1977.

[82] WINSTON, PATRICK H., in *Introduction to Artificial Intelligence*, Addison-Wesley Publishing Company, Readings Massachussetts, 1987.

Appendix A

OPS5 Source Code for Monkey and Banana

```
(literalize phys-obj
     name
          at
     weight
     on )
(literalize monkey
     at
     on
     holds )
(literalize goal
     status
     type
     obj-name
     to )
(literalize testcase
     type
     name )



;*************************************;


(p congrats
  (testcase )
  (goal ^status satisfied )
-  (goal ^status active )
-->
  (write (crlf) "CONGRATULATIONS the goals are satisfied ." (crlf))
  (halt) )

(p imposs
  (goal ^status active ^type <g1>)
-->
  (write (crlf) "IMPOSSIBLE the goal" <g1> "cannot be satisfied ." (crlf))
  (halt) )

;*************************************;
```

```
(p on::floor
  {(goal ^status active ^type on ^obj-name floor ) <goal>}
  {(monkey ^on <> floor)  <monkey>}
-->
  (write (crlf) "Jump onto the floor ." (crlf))
  (modify <monkey> ^on floor )
  (modify <goal> ^status satisfied ))


(p on::floor:satisfied
  {(goal ^status active ^type on ^obj-name floor ) <goal>}
  {(monkey ^at <p1> ^on floor)  <monkey>}
-->
  (write (crlf) "Monkey is already on the floor ." (crlf))
  (modify <goal> ^status satisfied ))

;*************************************;

(p on::phys-obj
  {(goal ^status active ^type on ^obj-name <o> ) <goal>}
  {(phys-obj ^name <o> ^at <p> ^on floor) <object>}
  {(monkey ^at <p> ^holds nil ^on <> <o>)  <monkey>}
-->
  (write (crlf) "Climb onto " <o> (crlf))
  (modify <monkey> ^on <o> )
  (modify <goal> ^status satisfied ))


(p on::phys-obj:holds
  {(goal ^status active ^type on ^obj-name <o1> ) <goal>}
  {(phys-obj ^name <o1> ^at <p> ) <object>}
  {(monkey ^at <p> ^holds <> nil)  <monkey>}
-->
  (make goal ^status active ^type holds ^obj-name nil ))

(p on::phys-obj:holds:at-monkey
  (goal ^status active ^type on ^obj-name <o1> )
  (phys-obj ^name <o1> ^at <p> ^on floor)
  (monkey ^at <> <p>)
-->
  (make goal ^status active ^type at ^obj-name nil ^to <p> ))

(p on::phys-obj:satisfied
  {(goal ^status active ^type on ^obj-name <o1> ) <goal>}
  {(phys-obj ^name <o1> ^at <p1> ^on floor) <object>}
  {(monkey ^at <p1> ^on <o1>)  <monkey>}
```

```
-->
  (write (crlf) "Monkey is already on  " <o1> (crlf))
  (modify <goal> ^status satisfied ))


;************************************;


(p holds::nil
  {(goal ^status active ^type holds ^obj-name nil ) <goal>}
  {(phys-obj ^name <o1> ) <object1>}
  {(monkey ^holds {<o1> <> nil }) <monkey>}
-->
  (write (crlf) "Drop " <o1> (crlf))
  (modify <goal> ^status satisfied )
  (modify <monkey> ^holds nil )
  (modify <object1> ^on floor ))


(p holds::nil:satisfied
  {(goal ^status active ^type holds ^obj-name nil ) <goal>}
  {(monkey ^holds nil ^at <p> ^on <q>) <monkey>}
-->
  (write (crlf) "Monkey is holding nothing ." (crlf))
  (modify <goal> ^status satisfied ))


(p holds::obj-ceil
  {(goal ^status active ^type holds ^obj-name <o1> ) <goal>}
  {(phys-obj ^name <o1> ^weight light ^at <p> ^on ceiling ) <object1>}
  {(phys-obj ^name ladder ^at <p> ^on floor ) <object2>}
  {(monkey ^on ladder ^holds nil ) <monkey>}
-  (phys-obj ^on <o1>)
-->
  (write (crlf) "Grab " <o1> (crlf))
  (modify <monkey> ^holds <o1> )
  (modify <object1> ^on nil)
  (modify <goal> ^status satisfied ))


(p holds::obj-ceil:on
  (goal ^status active ^type holds ^obj-name <o1> )
  (phys-obj ^name <o1> ^weight light ^at <p> ^on ceiling )
  (phys-obj ^name ladder ^at <p> ^on floor )
  (monkey ^on <> ladder )
-->
```

```
   (make goal ^status active ^type on ^obj-name ladder ))


(p holds::obj-ceil:on:at-obj
  (goal ^status active ^type holds ^obj-name <o1> )
  (phys-obj ^name <o1> ^weight light ^at <p> ^on ceiling )
  (phys-obj ^name ladder ^at <> <p> )
-->
  (make goal ^status active ^type at ^obj-name ladder ^to <p>))


;************************************;

(p holds::obj-notceil
  {(goal ^status active ^type holds ^obj-name <o> ) <goal>}
  {(phys-obj ^name <o> ^weight light ^at <p> ^on <> ceiling) <object>}
  {(monkey ^at <p> ^holds nil ^on floor)  <monkey>}
-  (phys-obj ^on <o>)
-->
  (write (crlf) "Grab " <o> (crlf))
  (modify <object> ^on nil )
  (modify <monkey> ^holds <o>)
  (modify <goal> ^status satisfied ))


(p holds::obj-notceil:on
  (goal ^status active ^type holds ^obj-name <o1> )
  (phys-obj ^name <o1> ^weight light ^at <p> ^on <> ceiling )
  (monkey ^at <p> ^on <> floor)
-->
  (make goal ^status active ^type on ^obj-name floor ))

(p holds::obj-notceil:at-monkey
  (goal ^status active ^type holds ^obj-name <o1> )
  (phys-obj ^name <o1> ^weight light ^at <p> ^on <> ceiling )
  (monkey ^at <> <p> )
-->
  (make goal ^status active ^type at ^obj-name nil ^to <p> ))


;************************************;


(p holds::obj:holds
  (goal ^status active ^type holds ^obj-name <o1> )
  (phys-obj ^name <o1> ^weight light ^at <p> )
```

```
  (monkey ^at <p> ^holds {<> nil <> <o1>} )
-->
  (make goal ^status active ^type holds  ^obj-name nil ))


(p holds::obj:satisfied
  {(goal ^status active ^type holds ^obj-name <o1> ) <goal>}
  {(phys-obj ^name <o1> ^weight light ^on nil ^at <p> ) <object>}
  {(monkey ^at <p> ^holds <o1>)   <monkey>}
-->
  (write (crlf) "Object " <o1> "is already being held "(crlf))
  (modify <goal> ^status satisfied ))

;*************************************;


(p at::obj
  {(goal ^status active ^type at ^obj-name <o1> ^to <p>) <goal>}
  {(phys-obj ^name <o1> ) <object1>}
  {(monkey ^at <> <p> ^holds <o1> ^on floor)   <monkey>}
-->
  (write (crlf) "Move " <o1> " to " <p> (crlf))
  (modify <object1> ^at <p> )
  (modify <monkey> ^at <p> )
  (modify <goal> ^status satisfied ))


(p at::obj:on-floor
  (goal ^status active ^type at ^obj-name <o1> ^to <p>)
  (phys-obj ^name <o1>  ^at <> <p> )
  (monkey ^on <> floor  ^holds <o1> )
-->
  (make goal ^status active ^type on  ^obj-name floor ))


(p at::obj:holds
  (goal ^status active ^type at ^obj-name <o1> ^to <p>)
  (phys-obj ^name <o1> ^weight light  ^at <> <p> )
  (monkey ^holds <> <o1> )
-->
  (make goal ^status active ^type holds  ^obj-name <o1> ))


(p at::obj:satisfied
```

```
  {(goal ^status active ^type at ^obj-name <o1> ^to <p>) <goal>}
  {(phys-obj ^name <o1> ^weight light ^at <p> ) <object>}
-->
  (write (crlf) "Object " <o1> "is already at " <p> (crlf))
  (modify <goal> ^status satisfied ))

;************************************;

(p at::monkey
  {(goal ^status active ^type at ^obj-name nil ^to <p1>) <goal>}
  {(monkey ^at <> <p1> ^holds nil ^on floor)   <monkey>}
-->
  (write (crlf) "Walk to " <p1> (crlf))
  (modify <monkey> ^at <p1> )
  (modify <goal> ^status satisfied ))


(p at::monkey:obj
  {(goal ^status active ^type at ^obj-name nil ^to <p1>) <goal>}
  {(phys-obj ^name <o1> ) <object1>}
  {(monkey ^at <> <p1> ^holds <o1> ^on floor)   <monkey>}
-->
  (write (crlf) "Walk to " <p1> " Carrying  " <o1> (crlf))
  (modify <object1> ^at <p1> )
  (modify <monkey> ^at <p1> )
  (modify <goal> ^status satisfied ))



(p at::monkey:on
  (goal ^status active ^type at ^obj-name nil ^to <p1>)
  (monkey ^at <> <p1> ^on <> floor)
-->
  (make goal ^status active ^type on ^obj-name floor ))

(p at::monkey:satisfied
  {(goal ^status active ^type at ^obj-name nil ^to <p1>) <goal>}
  {(monkey ^at <> <p1> )   <monkey>}
-->
  (write (crlf) "Monkey is already at " <p1> (crlf))
  (modify <goal> ^status satisfied ))


;************************************;

(p test::general:start
```

```
  (testcase ^type general ^name start )
-->
  (make phys-obj ^name bananas ^weight light ^at 9-9 ^on ceiling)
  (make phys-obj ^name couch  ^weight heavy ^at 7-7 ^on floor )
  (make phys-obj ^name ladder ^weight light ^at 4-3 ^on floor )
  (make phys-obj ^name blanket ^weight light ^at 7-7 )
  (make monkey ^on couch ^at 7-7 ^holds blanket )
  (make goal ^status active ^type holds ^obj-name bananas ))

(p test::general:on-ladder
  (testcase ^type general ^name on-ladder )
-->
  (make monkey ^on ladder ^at 5-5)
  (make phys-obj ^name bananas ^weight light ^at 7-7 ^on ceiling)
  (make phys-obj ^name ladder ^weight light ^at 5-5 ^on floor )
  (make phys-obj ^name blanket ^weight light ^at 5-5 ^on floor )
  (make goal ^status active ^type holds ^obj-name bananas ))



(p test::on:floor
  (testcase ^type on ^name floor )
-->
  (make phys-obj ^name ladder ^weight light ^at 5-7 ^on floor )
  (make monkey ^on ladder ^at 5-7 )
  (make goal ^status active ^type on ^obj-name floor ))


(p test::on:floor:satisfied
  (testcase ^type on ^name floor-satisfied )
-->
  (make phys-obj ^name ladder ^weight light ^at 5-7 ^on floor )
  (make monkey ^on floor ^at 5-7 )
  (make goal ^status active ^type on ^obj-name floor ))

(p test::on:phys-obj
  (testcase ^type on ^name phys-obj)
-->
  (make phys-obj ^name ladder ^weight light ^at 5-5 ^on floor )
  (make monkey ^on floor ^at 5-5 )
  (make goal ^status active ^type on ^obj-name ladder ))

(p test::on:phys-obj:holds
  (testcase ^type on ^name phys-obj-holds)
-->
  (make phys-obj ^name ladder ^weight light ^at 5-5  )
```

```
  (make monkey ^on floor ^at 5-5 ^holds ladder )
  (make goal ^status active ^type on ^obj-name ladder ))


(p test::on:phys-obj:at-monkey
  (testcase ^type on ^name phys-obj:at-monkey )
-->
  (make phys-obj ^name ladder ^weight light ^at 6-5 ^on floor )
  (make monkey ^on floor ^at 3-3 )
  (make goal ^status active ^type on ^obj-name ladder ))


(p test::on:phys-obj:satisfied
  (testcase ^type on ^name phys-obj-satisfied )
-->
  (make phys-obj ^name ladder ^weight light ^at 6-5 ^on floor )
  (make monkey ^on ladder ^at 6-5 )
  (make goal ^status active ^type on ^obj-name ladder ))

;*************************************;


(p test::holds:nil
  (testcase ^type holds  ^name holds-nil )
-->
  (make phys-obj ^name blanket ^weight light ^at 5-5  )
  (make monkey ^on floor ^at 5-5 ^holds blanket )
  (make goal ^status active ^type holds ))


(p test::holds:nil-satisfied
  (testcase ^type holds  ^name nil-satisfied )
-->
  (make monkey ^on floor ^at 5-5 ^holds nil )
  (make goal ^status active ^type holds ))

(p test::holds:obj-ceil
  (testcase ^type holds  ^name obj-ceil:on-floor)
-->
  (make phys-obj ^name ladder ^on floor  ^weight light ^at 5-5  )
  (make phys-obj ^name bananas ^on ceiling  ^weight light ^at 5-5  )
  (make monkey ^on floor ^at 5-5 )
  (make goal ^status active ^type holds ^obj-name bananas))


(p test::holds:obj-ceil:at-obj
```

```
  (testcase ^type holds  ^name obj-ceil-at-obj )
-->
  (make monkey ^on ladder  ^at 5-5  ^holds blanket )
  (make phys-obj ^name ladder ^on floor  ^weight light ^at 5-5  )
  (make phys-obj ^name bananas ^on ceiling  ^weight light ^at 7-7  )
  (make phys-obj ^name blanket ^weight light ^at 5-5  )
  (make goal ^status active ^type holds ^obj-name bananas))


(p test::holds:obj-notceil:on-ladder
  (testcase ^type holds  ^name obj-notceil:on-ladder  )
-->
  (make monkey ^on ladder  ^at 5-5  ^holds nil )
  (make phys-obj ^name ladder ^on floor  ^weight light ^at 5-5  )
  (make goal ^status active ^type holds ^obj-name ladder ))


(p test::holds:obj-notceil:at
  (testcase ^type holds  ^name obj-notceil:at )
-->
  (make monkey ^on floor  ^at 5-5  )
  (make phys-obj ^name ladder ^on floor  ^weight light ^at 7-5  )
  (make goal ^status active ^type holds ^obj-name ladder ))



(p test::holds:obj-satisfied
  (testcase ^type holds  ^name obj-satisfied )
-->
  (make monkey ^on floor  ^at 5-5  ^holds bananas  )
  (make phys-obj ^name bananas  ^weight light ^at 5-5  )
  (make goal ^status active ^type holds ^obj-name bananas  ))


;*********************************;


(p test::at-monkey
  (testcase ^type at  ^name at-monkey  )
-->
  (make monkey ^on floor  ^at 5-7 )
  (make phys-obj ^name ladder ^on floor  ^weight light ^at 7-7 )
  (make goal ^status active ^type at  ^obj-name nil ^to 7-7  ))


(p test::at-monkey:obj
```

```
  (testcase ^type at  ^name at-monkey-obj  )
-->
  (make monkey ^on floor  ^at 5-5 ^holds blanket )
  (make phys-obj ^name blanket ^weight light ^at 5-5 )
  (make phys-obj ^name ladder ^on floor  ^weight light ^at 7-7 )
  (make goal ^status active ^type at  ^obj-name nil ^to 7-7   ))

(p test::at-monkey:on
  (testcase ^type at  ^name at-monkey-on  )
-->
  (make monkey ^on couch  ^at 5-5  )
  (make phys-obj ^name couch ^weight heavy  ^at 5-5 ^on floor)
  (make phys-obj ^name ladder ^on floor ^at 7-7 )
  (make goal ^status active ^type at  ^obj-name nil ^to 7-7   ))


;*************************************;


(p test::at-obj
  (testcase ^type at  ^name at-obj  )
-->
  (make monkey ^on floor ^at 5-5 ^holds ladder )
  (make phys-obj ^name bananas  ^weight light ^at 8-8 ^on ceiling)
  (make phys-obj ^name ladder ^weight light ^at 5-5 )
  (make goal ^status active ^type at  ^obj-name ladder  ^to 8-8  ))


(p test::at-obj:on-floor
  (testcase ^type at  ^name at-obj-on-floor  )
-->
  (make monkey ^on couch ^at 5-5 ^holds bananas )
  (make phys-obj ^name bananas  ^weight light ^at 5-5 )
  (make phys-obj ^name couch ^weight heavy ^on floor  ^at 5-5 )
  (make goal ^status active ^type at  ^obj-name bananas   ^to 7-7 ))


(p test::at-obj:holds
  (testcase ^type at  ^name at-obj-holds )
-->
  (make monkey ^on floor  ^at 5-5 )
  (make phys-obj ^name bananas  ^weight light ^at 8-8 ^on ceiling )
  (make phys-obj ^name ladder  ^weight light ^on floor ^at 7-5 )
  (make goal ^status active ^type at  ^obj-name ladder  ^to 8-8 ))
```

```
;************************************;

(p start1 { (g1) init }
-->
(remove init)
(make testcase ^type general ^name start ))

(p start2 { (g2) init }
-->
(remove init)
(make testcase ^type general ^name on-ladder ))

;************************************;

(p start3 { (o1) init }
-->
(remove init )
(make testcase ^type on ^name floor ) )


(p start4 { (o2) init }
-->
(remove init )
(make testcase ^type on ^name floor-satisfied ) )


(p start5 { (o3) init }
-->
(remove init )
(make testcase ^type on ^name phys-obj) )


(p start6 { (o4) init }
-->
(remove init )
(make testcase ^type on ^name phys-obj-holds) )


(p start7 { (o5) init }
-->
(remove init )

(make testcase ^type on ^name phys-obj:at-monkey ) )
```

```
(p start8 { (o6) init }
-->
(remove init )
(make testcase ^type on ^name phys-obj-satisfied ) )

;************************************;


(p start9 { (h1) init }
-->
(remove init )
(make testcase ^type holds  ^name holds-nil ) )


(p start10 { (h2) init }
-->
(remove init )
(make testcase ^type holds  ^name nil-satisfied ) )

(p start11 { (h3) init }
-->
(remove init )
(make testcase ^type holds  ^name obj-ceil:on-floor) )

(p start12 { (h4) init }
-->
(remove init )
(make testcase ^type holds  ^name obj-ceil-at-obj ) )

(p start13 { (h5) init }
-->
(remove init )
(make testcase ^type holds  ^name obj-notceil:on-ladder  ) )

(p start14 { (h6) init }
-->
(remove init )
(make testcase ^type holds  ^name obj-notceil:at ) )
```

```
(p start15 { (h7) init }
-->
(remove init )
(make testcase ^type holds  ^name obj-satisfied ) )

;************************************;


(p start16 { (a1) init }
-->
(remove init )
(make testcase ^type at  ^name at-monkey  ) )


(p start17 { (a2) init }
-->
(remove init )
(make testcase ^type at  ^name at-monkey-obj  ) )


(p start18 { (a3) init }
-->
(remove init )
(make testcase ^type at  ^name at-monkey-on  ) )


(p start19 { (a4) init }
-->
(remove init )
(make testcase ^type at  ^name at-obj  ) )


(p start20 { (a5) init }
-->
(remove init )
(make testcase ^type at  ^name at-obj-on-floor  ) )


(p start21 { (a6) init }
-->
(remove init )
(make testcase ^type at  ^name at-obj-holds ) )
```

Appendix B

Trace File for execution of OPS5 based

Monkey and Banana Production System

```
Script started on Tue Apr 25 11:31:09 1989
> ops5
-> [load 'mab]
[load mab]
**************************************************************************t
-> [watch 3]
(2 -- conflict set trace not supported)
-> [make g1]
nil
-> [run 1]

1. start1 1
<=wm: 1: (g1)
=>wm: 3: (testcase   ^type general   ^name start)
***break***
-> [cs]

test::general:start
(test::general:start dominates)
-> [run 1]

2. test::general:start 3
=>wm: 4: (phys-obj   ^at 9-9   ^name bananas   ^on ceiling   ^weight light)
=>wm: 5: (phys-obj   ^at 7-7   ^name couch   ^on floor   ^weight heavy)
=>wm: 6: (phys-obj   ^at 4-3   ^name ladder   ^on floor   ^weight light)
=>wm: 7: (phys-obj   ^at 7-7   ^name blanket   ^weight light)
=>wm: 8: (monkey   ^at 7-7   ^holds blanket   ^on couch)
=>wm: 9: (goal   ^type holds   ^status active   ^obj-name bananas)
***break***
-> [cs]

holds::obj-ceil:on:at-obj
imposs
(holds::obj-ceil:on:at-obj dominates)
-> [run 1]

3. holds::obj-ceil:on:at-obj 9 4 6
=>wm: 10: (goal   ^type at   ^status active   ^obj-name ladder   ^to 9-9)
***break***
```

-> [cs]

at::obj:holds
imposs (2 occurrences)
(at::obj:holds dominates)
-> [run 1]

4. at::obj:holds 10 6 8
=>wm: 11: (goal ^type holds ^status active ^obj-name ladder)
break
-> [cs]

holds::obj-notceil:at-monkey
imposs (3 occurrences)
(holds::obj-notceil:at-monkey dominates)
-> [run 1]

5. holds::obj-notceil:at-monkey 11 6 8
=>wm: 12: (goal ^type at ^status active ^to 4-3)
break
-> [cs]

at::monkey:satisfied
at::monkey:on
imposs (4 occurrences)
(at::monkey:on dominates)
-> [run 1]

6. at::monkey:on 12 8
=>wm: 13: (goal ^type on ^status active ^obj-name floor)
break
-> [cs]

at::monkey:satisfied
on::floor
imposs (5 occurrences)
(on::floor dominates)
-> [run 1]

7. on::floor 13 8
Jump onto the floor .

<=wm: 8: (monkey ^at 7-7 ^holds blanket ^on couch)
=>wm: 15: (monkey ^at 7-7 ^holds blanket ^on floor)
<=wm: 13: (goal ^type on ^status active ^obj-name floor)
=>wm: 17: (goal ^type on ^status satisfied ^obj-name floor)

```
***break***
-> [cs]

imposs  (4 occurrences)
at::monkey:satisfied
at::obj:holds
holds::obj-notceil:at-monkey
at::monkey:obj
(at::monkey:obj dominates)
-> [run 1]
17. holds::obj-ceil 9 4 49 52
Grab  bananas

<=wm: 52:  (monkey   ^at 9-9   ^on ladder)
=>wm: 56:  (monkey   ^at 9-9   ^holds bananas   ^on ladder)
<=wm: 4:  (phys-obj   ^at 9-9   ^name bananas   ^on ceiling   ^weight light)
=>wm: 58:  (phys-obj   ^at 9-9   ^name bananas   ^weight light)
<=wm: 9:  (goal   ^type holds   ^status active   ^obj-name bananas)
=>wm: 60:  (goal   ^type holds   ^status satisfied   ^obj-name bananas)
***break***
-> [cs]

congrats       (8 occurrences)
(congrats dominates)
-> [run 1]

18. congrats 3 60
CONGRATULATIONS the goals are satisfied .

end -- explicit halt
 68 productions (221 // 469 nodes)
 18 firings (60 rhs actions)
 10 mean working memory size (14 maximum)
 4 mean conflict set size (8 maximum)
 33 mean token memory size (44 maximum)
nil
-> [cs]

congrats       (7 occurrences)
(congrats dominates)
-> [exit]
> exit
exit

script done on Tue Apr 25 11:37:54 1989
```

Appendix C

Definitions for the C Source in "def.h"

```
#define NIL    10
#define NULL   0
#define GEN    20
#define ON     21
#define HOLD   22
#define AT     23
#define COCH   30
#define LADR   31
#define FLOR   32
#define BANS   33
#define CEIL   34
#define BLNK   35
#define LGHT   40
#define HEVY   41
#define ACTV   48
#define SATS   49
#define T1     51
#define T2     52
#define T3     53
#define T4     54
#define T5     55
#define T6     56
#define T8     58
#define T9     59
#define T10    60
#define T11    61
#define T12    62
#define T13    63
#define T14    64
#define T15    65
#define T16    66
#define T17    67
#define T18    68
#define T19    69
#define T20    70
#define T21    71


#define P 19
```

```
char *msg1[P] = {
      ">>> CONGRATULATIONS the goals are SATISFIED",
[7m
      "Jump onto the floor", "Monkey is already on the floor",
      "Climb onto ", "Monkey is already on", "Drop","Monkey is holding nothing",
      "Grab", "Object ", " is already being held", "Move ", " to ",
      "Object ", " is already at", "Walk to", " carrying",
      "Monkey is already at"  }
;

char *msg2[36] = {
      " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ",
      " ", " ", " ", " ", " ", " ", " ", " ", " "," ON ", " HOLD " , " AT " ,
      " ", " ", " ", " ", " ", " ", " COCH " , " LADR " , " FLOR " ,
      " BANS " , " CEIL ", " BLNK " }
;

typedef struct OBJ {
      int nam ;
      int at ;
      int wght ;
      int on ;
}

typedef struct GOL {
      int stat ;
      int type ;
      int nam ;
      int to;
}
GOL , *PGOL ;
typedef struct MON {
      int at ;
      int on ;
      int holds;
}
MON, *PMON ;

POBJ WM_obj[10];
PGOL WM_gol[10];
PMON WM_mon;

int GL_cur , OB_cur , NO_mat ;
int mon_obj_loc , gol_nam_loc , indx , ladr_loc;
```

```
#define DIM 5

#include <stdio.h>
#include <string.h>
#include <ctype.h>
```

Appendix D

C Functions used by the Monkey and Banana Production System

```
POBJ  ObjCr(nam, at , wght, on , nexto)
int nam, at , wght, on ;
{
      POBJ  aux ;
      char *malloc() ;
      if ( nam == NULL)   return(NULL) ;
      if ((aux = (POBJ) malloc(sizeof(OBJ))) == NULL)
            return(NULL);

      aux->nam = nam;
      aux->at = at ;
      aux->on = on ;
      aux->wght = wght;
      WM_obj[OB_cur + 1] = aux;
      OB_cur++;
      return(aux);
}

PGOL  GolCr(stat , type , nam, to)
int stat , type , nam, to;
{
      PGOL  aux ;
      char *malloc() ;

      if ( nam == NULL)   return(NULL) ;
      if ((aux = (PGOL) malloc(sizeof(GOL))) == NULL)
            return(NULL);

      aux->nam = nam;
      aux->stat = stat ;
      aux->to = to ;
      aux->type = type ;
      WM_gol[GL_cur + 1] = aux;
      GL_cur++;
      return(aux);
}

static  PMON  MonCr( at , on , holds)
int  at , on , holds ;
{
```

```
    PMON aux ;
    char *malloc() ;

    if ((aux = (PMON) malloc(sizeof(MON))) == NULL)
        return(NULL);

    aux->holds = holds;
    aux->at = at ;
    aux->on = on ;
    return(aux);
}



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


POBJ ObjMod(nam, at , wght, on , nexto)
int nam, at , wght, on ;
{
    POBJ aux ;
    aux->nam = nam;
    aux->at = at ;
    aux->on = on ;
    aux->wght = wght;
    WM_obj[OB_cur + 1] = aux;
    return(aux);
}

PGOL GolMod(stat , type , nam, to)
int stat , type , nam, to;
{
    PGOL aux ;

    aux->nam = nam;
    aux->stat = stat ;
    aux->to = to ;
    aux->type = type ;
    WM_gol[GL_cur + 1] = aux;
    return(aux);
}

PMON MonMod( at , on , holds)
int at , on , holds ;
{
    PMON aux ;
```

```
    aux->holds = holds;
    aux->at = at ;
    aux->on = on ;
    return(aux);
}



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




intialise_env()
{
    int start ;
    int obj[][4];
    int gl[5];
    int monkey[3];
    int i, j, k ;
    PGOL p1;

    printf("0HERE ARE 21 TEST CASE SCENARIOS ENCODED AS INTEGERS 51...710);
    printf("0LEASE ENTER AN INTEGER BETWEEN 51 AND 710) ;
    printf("TO SET UP THE INITIAL working memory AND goal0 ) ;
    scanf("%d",&start) ;

    switch ( start ) {
    case 51 :
        {
            WM_obj[OB_cur + 1] = ObjCr(BANS,909,LGHT,CEIL) ;
            WM_obj[OB_cur + 1] = ObjCr(COCH,707,HEVY,FLOR);
            WM_obj[OB_cur + 1] = ObjCr(LADR,403,LGHT,FLOR);
            WM_obj[OB_cur + 1] = ObjCr(BLNK,707,LGHT,FLOR);

            WM_gol[GL_cur + 1] = GolCr(ACTV,HOLD,BANS,NIL);

            WM_mon          = MonCr(707,COCH,BLNK );
           }
        break ;
```

```
case 52 :
    {
        WM_obj[OB_cur + 1] = ObjCr(BANS,707,LGHT,CEIL);
        WM_obj[OB_cur + 1] = ObjCr(LADR,505,LGHT,FLOR);
        WM_obj[OB_cur + 1] = ObjCr(BLNK,505,LGHT,FLOR);

        WM_gol[GL_cur + 1] = GolCr(ACTV,HOLD,BANS,NIL);

        WM_mon          = MonCr(505,LADR,NIL);
    }
    break ;

case 53 :
    {
        WM_obj[OB_cur + 1] = ObjCr(LADR,507,LGHT,FLOR);

        WM_gol[GL_cur + 1] = GolCr(ACTV,ON,FLOR,NIL);

        WM_mon          = MonCr(507,LADR,NIL);
    }
    break ;

case 54 :
    {
        WM_obj[OB_cur + 1] = ObjCr(LADR,507,LGHT,FLOR);

        WM_gol[GL_cur + 1] = GolCr(ACTV,ON,FLOR,NIL);

        WM_mon          = MonCr(507,FLOR,NIL);
    }
    break ;

case 55 :
    {
        WM_obj[OB_cur + 1] = ObjCr(LADR,505,LGHT,FLOR);

        WM_gol[GL_cur + 1] = GolCr(ACTV,ON,LADR,NIL);

        WM_mon          = MonCr(505,FLOR,NIL);
    }
    break ;

case 56 :
    {
        WM_obj[OB_cur + 1] = ObjCr(LADR,505,LGHT,FLOR);
```

```
            WM_gol[GL_cur + 1] = GolCr(ACTV,ON,LADR,NIL);

            WM_mon          = MonCr(505,FLOR,LADR);
        }
        break ;

case 57 :
        {
            WM_obj[OB_cur + 1] = ObjCr(LADR,605,LGHT,FLOR);

            WM_gol[GL_cur + 1] = GolCr(ACTV,ON,LADR,NIL);

            WM_mon          = MonCr(303,FLOR,NIL);
        }
        break ;

case 58 :
        {
            WM_obj[OB_cur + 1] = ObjCr(LADR,605,LGHT,FLOR);

            WM_gol[GL_cur + 1] = GolCr(ACTV,ON,LADR,NIL);

            WM_mon          = MonCr(605,LADR,NIL);
        }
        break ;

case 59 :
        {
            WM_obj[OB_cur + 1] = ObjCr(BLNK,505,LGHT,NIL);

            WM_gol[GL_cur + 1] = GolCr(ACTV,HOLD,NIL,NIL);

            WM_mon          = MonCr(505,FLOR,BLNK);
        }
        break ;

case 60 :
        {
            WM_gol[GL_cur + 1] = GolCr(ACTV,HOLD,NIL,NIL);

            WM_mon          = MonCr(505,FLOR,NIL);
        }
        break ;

case 61 :
        {
```

```
            WM_obj[OB_cur + 1] = ObjCr(BANS,505,LGHT,CEIL);
            WM_obj[OB_cur + 1] = ObjCr(LADR,505,LGHT,FLOR);

            WM_gol[GL_cur + 1] = GolCr(ACTV,HOLD,BANS,NIL);

            WM_mon          = MonCr(505,FLOR,NIL);
        }
        break ;

    case 62 :
        {
            WM_obj[OB_cur + 1] = ObjCr(BANS,707,LGHT,CEIL);
            WM_obj[OB_cur + 1] = ObjCr(LADR,505,LGHT,FLOR);
            WM_obj[OB_cur + 1] = ObjCr(BLNK,505,LGHT,NIL);

            WM_gol[GL_cur + 1] = GolCr(ACTV,HOLD,BANS,NIL);

            WM_mon          = MonCr(505,LADR,BLNK);
        }
        break ;

    case 63 :
        {
            WM_obj[OB_cur + 1] = ObjCr(LADR,505,LGHT,FLOR);

            WM_gol[GL_cur + 1] = GolCr(ACTV,HOLD,LADR,NIL);

            WM_mon          = MonCr(505,LADR,NIL);
        }
        break ;

    case 64 :
        {
            WM_obj[OB_cur + 1] = ObjCr(LADR,705,LGHT,FLOR);

            WM_gol[GL_cur + 1] = GolCr(ACTV,HOLD,LADR,NIL);

            WM_mon          = MonCr(505,FLOR,NIL);
        }
        break ;

    case 65 :
        {
            WM_obj[OB_cur + 1] = ObjCr(BANS,505,LGHT,NIL);

            WM_gol[GL_cur + 1] = GolCr(ACTV,HOLD,BANS,NIL);
```

```
            WM_mon          = MonCr(505,FLOR,BANS);
        }
        break ;

    case 66 :
        {
            WM_obj[OB_cur + 1] = ObjCr(LADR,707,LGHT,FLOR);

            WM_gol[GL_cur + 1] = GolCr(ACTV,AT,NIL,707);

            WM_mon          = MonCr(507,FLOR,NIL);
        }
        break ;

    case 67 :
        {
            WM_obj[OB_cur + 1] = ObjCr(LADR,707,LGHT,FLOR);
            WM_obj[OB_cur + 1] = ObjCr(BLNK,505,LGHT,NIL);

            WM_gol[GL_cur + 1] = GolCr(ACTV,AT,NIL,707);

            WM_mon          = MonCr(505,FLOR,BLNK);
        }
        break ;

    case 68 :
        {
            WM_obj[OB_cur + 1] = ObjCr(LADR,707,LGHT,FLOR);
            WM_obj[OB_cur + 1] = ObjCr(COCH,505,HEVY,FLOR);

            WM_gol[GL_cur + 1] = GolCr(ACTV,AT,NIL,707);

            WM_mon          = MonCr(505,COCH,NIL);
        }
        break ;

    case 69 :
        {
            WM_obj[OB_cur + 1] = ObjCr(LADR,505,LGHT,NIL);
            WM_obj[OB_cur + 1] = ObjCr(BANS,808,LGHT,CEIL);

            WM_gol[GL_cur + 1] = GolCr(ACTV,AT,LADR,808);

            WM_mon          = MonCr(505,FLOR,LADR);
        }
        break ;
```

```
    case 70 :
        {
            WM_obj[OB_cur + 1] = ObjCr(COCH,505,HEVY,FLOR);
            WM_obj[OB_cur + 1] = ObjCr(BANS,505,LGHT,NIL);

            WM_gol[GL_cur + 1] = GolCr(ACTV,AT,BANS,707);

            WM_mon          = MonCr(505,COCH,BANS);
        }
        break ;

    case 71 :
        {
            WM_obj[OB_cur + 1] = ObjCr(LADR,705,LGHT,FLOR);
            WM_obj[OB_cur + 1] = ObjCr(BANS,808,LGHT,CEIL);

            WM_gol[GL_cur + 1] = GolCr(ACTV,AT,LADR,808);

            WM_mon          = MonCr(505,FLOR,NIL);
        }
        break ;

    default :
        printf("0OU ENTERED AN UNACEPTABLE INTEGER TO INITIALIZE0) ;
    }
}
```

Appendix E

C Source code for Monkey and Banana Production System

```
#include <stdio.h>
#include "def.h"
#include "inc.c"
#include "../intel.h"
#define TRACE #define STACK 10000

LOCK actlck = UNLOCKED; LOCK match = UNLOCKED; EVENT actrlse =
CLEARED;

TASK ruls1(); TASK ruls2(); TASK ruls3(); TASK ruls4();

extern intialise_env(); /*********************************/

TASK task0() {
  /* main task */
  int d;
  int type,lth,node,pid;
  int i ;
  PGOL p1;

  strace("mab.trace");
  cube_init(0.1,0.3,0.2,1024);
  d = copen(15);
  GL_cur = -1 ;
  OB_cur = -1 ;
  NO_mat = -1 ;
  WM_gol[GL_cur]->stat == SATS;

  intialise_env();

  indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 +
         WM_mon->on*100 + WM_mon->holds;

  tfork ( ruls1 , STACK ) ;
  tfork ( ruls2 , STACK ) ;
  tfork ( ruls3 , STACK ) ;
  tfork ( ruls4 , STACK ) ;

} /*********************************/
```

```
TASK ruls1() {
  int i,j,k,flag1;
  while ((GL_cur > -1) &&(WM_gol[GL_cur]->stat == ACTV)) {
    flag1 = -99;

    switch (indx) {

    case 21323035 :
      {
        lock(&actlck);
        while(NO_mat < 2) tsleep(5);
        WM_mon->on = FLOR;
        WM_gol[GL_cur]->stat = SATS;
        GL_cur--;
        indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 +
             WM_mon->on*100 + WM_mon->holds;
        printf("%s 0,msg1[3]);
        if(NO_mat == 2) {
          unlock(&actlck);
          NO_mat = -2;
          flag1 = 99;
          evpost(&actrlse);
          evclear(&actrlse);
        }
      }
      break;

    case 22103231 :
      {
        lock(&actlck);
        for(i=0;i<=OB_cur;i++)      if(WM_mon->holds      ==      WM_obj[i]->nam)
mon_obj_loc = i;
        while(NO_mat < 2) tsleep(5);
        WM_obj[mon_obj_loc]->on = FLOR;
        WM_mon->holds = NIL;
        WM_gol[GL_cur]->stat = SATS;
        GL_cur--;
        indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 +
             WM_mon->on*100 + WM_mon->holds;
        printf("%s %s 0,msg1[7],msg2[WM_obj[mon_obj_loc]->nam]);
        if(NO_mat == 2) {
          unlock(&actlck);
          NO_mat = -2;
          flag1 = 99;
          evpost(&actrlse);
          evclear(&actrlse);
```

```
        }
      }
      break;

    case 22333110 :
      {
        lock(&actlck);
        for(i=0;i<=OB_cur;i++)                    if(WM_gol[GL_cur]->nam    ==
WM_obj[i]->nam) gol_nam_loc = i;
        while(NO_mat < 2) tsleep(5);
        WM_obj[gol_nam_loc]->on = NIL;
        WM_mon->holds =  WM_obj[gol_nam_loc]->nam;
        WM_gol[GL_cur]->stat = SATS;
        GL_cur--;
        indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 +
             WM_mon->on*100 + WM_mon->holds;
        printf("%s %s 0,msg1[9],msg2[WM_obj[gol_nam_loc]->nam]);
        if(NO_mat == 2) {
          unlock(&actlck);
          NO_mat = -2;
          flag1 = 99;

          evpost(&actrlse);
          evclear(&actrlse);
        }
      }
      break;


    case 23313231 :
      {

        lock(&actlck);
        for(i=0;i<=OB_cur;i++)                    if(WM_gol[GL_cur]->nam    ==
WM_obj[i]->nam) gol_nam_loc = i;
        while(NO_mat < 2) tsleep(5);
        WM_obj[gol_nam_loc]->at = WM_gol[GL_cur]->to;
        WM_mon->at = WM_gol[GL_cur]->to;
        WM_gol[GL_cur]->stat = SATS;
        printf("%s   %s   %s   %d0,msg1[12],msg2[WM_obj[gol_nam_loc]->nam],
                  msg1[13],WM_gol[GL_cur]->to);
        GL_cur--;
        indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 +
             WM_mon->on*100 + WM_mon->holds;
        if(NO_mat == 2) {
          unlock(&actlck);
```

```
            NO_mat = -2;
            flag1 = 99;

            evpost(&actrlse);
            evclear(&actrlse);
          }
        }
        break;

      default :
        tsleep(1);

      }
      lock(&match);
      NO_mat++;
      unlock(&match);

      if((flag1 < 0) && (NO_mat > -1) && (NO_mat != 3)) evwait(&actrlse);

      if ((NO_mat == 3) || (GL_cur == -1)){
        lock(&match);
        NO_mat++;
        unlock(&match);
        if((GL_cur > -1) && (WM_gol[GL_cur]->stat == ACTV))
          printf("%s %s %s0,msg1[1],msg2[WM_gol[GL_cur]->nam],msg1[2]);
        else {
          printf("%s 0,msg1[0]);
        }
        texit(gettid());
      }
    } } /**********************************/

TASK ruls2() {
  int i,j,k,flag1;
  while ((GL_cur > -1) &&(WM_gol[GL_cur]->stat == ACTV)) {
    flag1 = -99;

    switch (indx) {


    case 22333210 :
      {
        lock(&actlck);


        while(NO_mat < 2) tsleep(5);
```

```
        WM_gol[GL_cur + 1] = GolCr(ACTV,ON,LADR,NIL);
        indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 +
             WM_mon->on*100 + WM_mon->holds;
        if(NO_mat == 2){
          unlock(&actlck);
          NO_mat = -2;
          flag1 = 99;

          evpost(&actrlse);
          evclear(&actrlse);
        }
      }
      break ;

    case 22333035 :
      {
        lock(&actlck);
        for(i=0;i<=OB_cur;i++)                    if(WM_gol[GL_cur]->nam      ==
WM_obj[i]->nam) gol_nam_loc = i;

        while(NO_mat < 2) tsleep(5);
        WM_gol[GL_cur + 1] = GolCr(ACTV,AT,LADR,WM_obj[gol_nam_loc]-
>at);
        indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 +
             WM_mon->on*100 + WM_mon->holds;
        if(NO_mat == 2) {
          unlock(&actlck);
          NO_mat = -2;
          flag1 = 99;
          evpost(&actrlse);
          evclear(&actrlse);
        }
      }
      break ;

    case 22313210 :
      {
        lock(&actlck);
        for(i=0;i<=OB_cur;i++)                    if(WM_gol[GL_cur]->nam      ==
WM_obj[i]->nam) gol_nam_loc = i;
        while(NO_mat < 2) tsleep(5);
        WM_gol[GL_cur]->stat = SATS;
        printf("%s %s 0,msg1[9], msg2[WM_obj[gol_nam_loc]->nam]);
        WM_mon->holds = WM_obj[gol_nam_loc]->nam;
        WM_obj[gol_nam_loc]->on = NIL;
        GL_cur--;
```

```
        indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 +
             WM_mon->on*100 + WM_mon->holds;
        if(NO_mat == 2){
          unlock(&actlck);
          NO_mat = -2;
          flag1 = 99;
          evpost(&actrlse);
          evclear(&actrlse);
        }
      }
      break;

    case 23313035 :
      {
        lock(&actlck);
        for(i=0;i<=OB_cur;i++)                    if(WM_gol[GL_cur]->nam    ==
WM_obj[i]->nam) gol_nam_loc = i;
        while(NO_mat < 2) tsleep(5);
        WM_gol[GL_cur + 1] = GolCr(ACTV,HOLD,WM_obj[gol_nam_loc]-
>nam,NIL);
        indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 +
             WM_mon->on*100 + WM_mon->holds;
        if(NO_mat == 2) {
          unlock(&actlck);
          NO_mat = -2;
          flag1 = 99;

          evpost(&actrlse);
          evclear(&actrlse);
        }
      }
      break;

    default :
      tsleep(1);
    }
    lock(&match);
    NO_mat++;

    unlock(&match);
    if((flag1 < 0) && (NO_mat > -1) && (NO_mat != 3))  evwait(&actrlse);


    if ((NO_mat == 3) || (GL_cur == -1)) {
      lock(&match);
      NO_mat++;
```

```
      unlock(&match);
      if((GL_cur > -1) && (WM_gol[GL_cur]->stat == ACTV))
        printf("%s %s %s0,msg1[1],msg2[WM_gol[GL_cur]->nam],msg1[2]);
      else {
        printf("%s 0,msg1[0]);
      }
      texit(gettid());
    }
  } } /*********************************/

TASK ruls3() {
  int i,j,k,flag1;
  while ((GL_cur > -1) &&(WM_gol[GL_cur]->stat == ACTV)) {
    flag1 = -99;

    switch (indx) {

    case 21313210 :
      {
        lock(&actlck);
        for(i=0;i<=OB_cur;i++)                    if(WM_gol[GL_cur]->nam     ==
WM_obj[i]->nam) gol_nam_loc = i;
        while(NO_mat < 2) tsleep(5);
        WM_gol[GL_cur]->stat = SATS;
        WM_mon->on = WM_obj[gol_nam_loc]->nam;
        printf("%s %s 0,msg1[5],msg2[WM_obj[gol_nam_loc]->nam]);
        GL_cur--;
        indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 +
             WM_mon->on*100 + WM_mon->holds;
        if(NO_mat == 2) {
          unlock(&actlck);
          NO_mat = -2;
          flag1 = 99;

          evpost(&actrlse);
          evclear(&actrlse);
        }
      }
      break;

    case 22313035 :
      {

        lock(&actlck);
        for(i=0;i<=OB_cur;i++)                    if(WM_gol[GL_cur]->nam     ==
WM_obj[i]->nam) gol_nam_loc = i;
```

```
        while(NO_mat < 2) tsleep(5);
        WM_gol[GL_cur + 1] = GolCr(ACTV,AT,NIL,WM_obj[gol_nam_loc]->at);
        indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 +
             WM_mon->on*100 + WM_mon->holds;
        if(NO_mat == 2){
          unlock(&actlck);
          NO_mat = -2;
          flag1 = 99;

          evpost(&actrlse);
          evclear(&actrlse);
        }
      }
      break ;

    case 23103235 :
      {
        lock(&actlck);
        for(i=0;i<=OB_cur;i++)                    if(WM_mon->holds == WM_obj[i]-
>nam) mon_obj_loc = i;
        while(NO_mat < 2) tsleep(5);
        WM_gol[GL_cur]->stat = SATS;
        WM_obj[mon_obj_loc]->at = WM_gol[GL_cur]->to;
        WM_mon->at = WM_gol[GL_cur]->to;
        printf("%s     %d     %s     %s0,msg1[16],WM_gol[GL_cur]->to,msg1[17],
                  msg2[WM_obj[mon_obj_loc]->nam]);
        GL_cur--;
        indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 +
             WM_mon->on*100 + WM_mon->holds;
        if(NO_mat == 2) {
          unlock(&actlck);
          NO_mat = -2;
          flag1 = 99;
          evpost(&actrlse);
          evclear(&actrlse);
        }
      }
      break;

    default :
      tsleep(1);
    }
    lock(&match);
    NO_mat++;

    unlock(&match);
```

```
    if((flag1 < 0) && (NO_mat > -1) && (NO_mat != 3)) evwait(&actrlse);


    if ((NO_mat == 3) || (GL_cur == -1)) {
      lock(&match);
      NO_mat++;
      unlock(&match);
      if((GL_cur > -1) && (WM_gol[GL_cur]->stat == ACTV))
        printf("%s %s %s0,msg1[1],msg2[WM_gol[GL_cur]->nam],msg1[2]);
      else {
        printf("%s 0,msg1[0]);
      }
      texit(gettid());
    }
  } } /*********************************/

TASK ruls4() {
  int i,j,k,flag1;
  while ((GL_cur > -1) &&(WM_gol[GL_cur]->stat == ACTV)) {
    flag1 = -99;

    switch (indx) {

    case 22103235 :
      {
        lock(&actlck);
        for(i=0;i<=OB_cur;i++)                    if(WM_mon->holds  ==  WM_obj[i]-
>nam) mon_obj_loc = i;
        while(NO_mat < 2) tsleep(5);
        WM_obj[mon_obj_loc]->on = FLOR;
        WM_mon->holds = NIL;
        WM_gol[GL_cur]->stat = SATS;
        GL_cur--;
        indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 +
             WM_mon->on*100 + WM_mon->holds;
        printf("%s %s 0,msg1[7],msg2[WM_obj[mon_obj_loc]->nam]);
        if(NO_mat == 2) {
          unlock(&actlck);
          NO_mat = -2;
          flag1 = 99;
          evpost(&actrlse);
          evclear(&actrlse);
        }
      }
      break ;
```

```
case 22333231 :
  {
    lock(&actlck);

    while(NO_mat < 2) tsleep(5);
    WM_gol[GL_cur + 1] = GolCr(ACTV,HOLD,NIL,NIL);
    indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 +
         WM_mon->on*100 + WM_mon->holds;
    if(NO_mat == 2) {
      unlock(&actlck);
      NO_mat = -2;
      flag1 = 99;

      evpost(&actrlse);
      evclear(&actrlse);
    }
  }
  break;

case 22313235 :
  {
    lock(&actlck);

    while(NO_mat < 2) tsleep(5);
    WM_gol[GL_cur + 1] = GolCr(ACTV,HOLD,NIL,NIL);
    indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 +
         WM_mon->on*100 + WM_mon->holds;
    if(NO_mat == 2) {
      unlock(&actlck);
      NO_mat = -2;
      flag1 = 99;

      evpost(&actrlse);
      evclear(&actrlse);
    }
  }
  break;

case 23103035 :
  {

    lock(&actlck);

    while(NO_mat < 2) tsleep(5);
    WM_gol[GL_cur + 1] = GolCr(ACTV,ON,FLOR,NIL);
    indx = WM_gol[GL_cur]->type*1000000 + WM_gol[GL_cur]->nam*10000 +
```

```
            WM_mon->on*100 + WM_mon->holds;
        if(NO_mat == 2) {
          unlock(&actlck);
          NO_mat = -2;
          flag1 = 99;

          evpost(&actrlse);
          evclear(&actrlse);
        }
      }
      break;

    default :
      tsleep(1);
    }
    lock(&match);
    NO_mat++;

    unlock(&match);
    if((flag1 < 0) && (NO_mat > -1) && (NO_mat != 3))  evwait(&actrlse);

    if ((NO_mat == 3) || (GL_cur == -1)) {
      lock(&match);
      NO_mat++;
      unlock(&match);
      if((GL_cur > -1) && (WM_gol[GL_cur]->stat == ACTV))
        printf("%s %s %s0,msg1[1],msg2[WM_gol[GL_cur]->nam],msg1[2]);
      else {
        printf("%s 0,msg1[0]);
      }
      texit(gettid());
    }
  } }
```

Appendix F

Results of Executing the C coded Production System

This file has the cube performance diagnostics, generated by the **nstats** routine on the tracefiles after the source program was executed in the **aspp** mode. The various parameters and their respective formulae are :

1. *Nodal Utilization :* $\frac{\sum_{k=0}^{P-1} utiliz_k}{P}$, where k represents node number for P nodes starting from 0.

2. *Nodal+host Utilization :* $\frac{\sum_{k=0}^{P-1} utiliz_k + utiliz_{HOST}}{P+1}$, where k represents node number for P nodes starting from 0.

3. *Gross Utilization :* $\frac{(\sum_{k=0}^{P-1} busy_k + busy_{HOST})\times 100}{(MAX\,[end_i]-MIN\,[start_i])+(P+1)}$, where k represents node number for P nodes starting from 0, and i includes the HOST and the P nodes.

4. *Sends :* The number of messages sent by that processor or HOST.

5. *Recvs :* The number of messages received by that processor or HOST.

6. *Lth :* The length of the message in units of bytes, and rounded off the next highest factor of 8.

7. *Hops :* The hamming distance between any 2 processors that initiated a message transfer; it represents the number of bits that are set '1' when the binary values of the processor identity numbers are exclusive orred by the $\oplus$ operator. The hop

from HOST to any processor is always '-1'.

[1]. 'C' coded Monkey and Banana, without indexing, with partitioned working memory, with flags and synchronizing variables and executing on a single node.

```
> fina

THERE ARE 21 TEST CASE SCENARIOS ENCODED AS INTEGERS 51...71

PLEASE ENTER AN INTEGER BETWEEN 51 AND 71
TO SET UP THE INITIAL working memory AND goal

51
Jump onto the floor
Walk to 403 carrying BLNK
Drop BLNK
Grab LADR
Move LADR to 909
Drop LADR
Climb onto LADR
Grab BANS
>>> CONGRATULATIONS the goals are SATISFIED
> nstats mab.trace
node    start     end    duration   busy     utiliz   sends  recvs
HOST        4     202      198      198      100%       0      0
  0       201    2743     2542     2542      100%       0      0

Nodal utilization 100%  Nodal+host utilization 100% sends 0  recvs 0
Gross utilization  50%

Total messages     0      0 bytes
lth   count          bytes
Floating exception (core dumped)
```

[2]. 'C' coded Monkey and Banana, without indexing, with partitioned working memory, without flags and synchronizing variables and executing on a single node.

```
> fina2

THERE ARE 21 TEST CASE SCENARIOS ENCODED AS INTEGERS 51...71

PLEASE ENTER AN INTEGER BETWEEN 51 AND 71
TO SET UP THE INITIAL working memory AND goal
```

```
51
Jump onto the floor
Walk to 403 carrying BLNK
Drop BLNK
Grab LADR
Move LADR to 909
Drop LADR
Climb onto LADR
Grab BANS
>>> CONGRATULATIONS the goals are SATISFIED
> nstats mab.trace
node    start     end    duration   busy     utiliz   sends  recvs
HOST        4     214      210      210      100%       0     0
  0       213    1526     1313     1313      100%       0     0

Nodal utilization 100%  Nodal+host utilization 100% sends 0  recvs 0
Gross utilization  50%

Total messages     0       0 bytes
lth   count          bytes
Floating exception (core dumped)
```

[3]. 'C' coded Monkey and Banana, with indexing, with partitioned working memory, with flags and synchronizing variables and executing on 4 separate nodes.

```
> finc

THERE ARE 21 TEST CASE SCENARIOS ENCODED AS INTEGERS 51...71

PLEASE ENTER AN INTEGER BETWEEN 51 AND 71
TO SET UP THE INITIAL working memory AND goal

51
Jump onto the floor
Walk to 403 carrying BLNK
Drop BLNK
Grab LADR
Move LADR to 909
Drop LADR
Climb onto LADR
Grab BANS
>>> CONGRATULATIONS the goals are SATISFIED
> nstats mab.trace
node    start     end    duration   busy     utiliz   sends  recvs
HOST        4     224      220      220      100%       0     0
```

```
 0    214    1880    1666    271    16%    0    0
 1    217    1880    1663    182    11%    0    0
 2    220    1880    1660    185    11%    0    0
 3    223    1880    1657    165    10%    0    0

Nodal utilization 12% Nodal+host utilization 30% sends 0 recvs 0
Gross utilization 11%

Total messages    0    0 bytes
lth  count        bytes
Floating exception (core dumped)
>
```

[4]. 'C' coded distributed parallel program for Matrix Transpose without any shared memory, entire solution is based on message passing.

```
> trans
MATRIX IS
 1 2 3 4 5 2 3 5 6
 7 8 9 0 1 2 9 2 3
 1 0 4 3 3 1 2 2 0
 1 1 0 3 4 3 2 1 5
 1 2 3 4 5 2 3 5 6
 7 8 9 0 1 2 9 2 3
 1 0 4 3 3 1 2 2 0
 1 1 0 3 4 3 2 1 5
 3 4 5 6 7 8 9 0 1

TRANSPOSE IS
 1 7 1 1 1 7 1 1 3
 2 8 0 1 2 8 0 1 4
 3 9 4 0 3 9 4 0 5
 4 0 3 3 4 0 3 3 6
 5 1 3 4 5 1 3 4 7
 2 2 1 3 2 2 1 3 8
 3 9 2 2 3 9 2 2 9
 5 2 2 1 5 2 2 1 0
 6 3 0 5 6 3 0 5 1
>
> nstats trans.trace
node    start    end    duration    busy    utiliz    sends    recvs
HOST      4     3388      3384      2400      71%       9        9
 0     1076     1710      634       133      21%       1        1
 1     1092     1726      634       133      21%       1        1
 2     1108     1742      634       133      21%       1        1
```

3	1124	1758	634	133	21%	1	1
4	1140	1774	634	133	21%	1	1
5	1156	1790	634	133	21%	1	1
6	1172	1806	634	133	21%	1	1
7	1188	1822	634	133	21%	1	1
8	1204	1838	634	133	21%	1	1

Nodal utilization 21% Nodal+host utilization 26% sends 18 recvs 18
Gross utilization 11%

Total messages 18 360 bytes

lth	count		bytes	
8	9	50%	36	10%
16	0	0%	0	0%
32	0	0%	0	0%
64	9	50%	324	90%
128	0	0%	0	0%
256	0	0%	0	0%
512	0	0%	0	0%
1024	0	0%	0	0%
2048	0	0%	0	0%
4096	0	0%	0	0%
8192	0	0%	0	0%
16000	0	0%	0	0%

hops	count		bytes	
-1	18	100%	360	100%
0	0	0%	0	0%
1	0	0%	0	0%
2	0	0%	0	0%
3	0	0%	0	0%
4	0	0%	0	0%
5	0	0%	0	0%
6	0	0%	0	0%

> exit

Appendix G

Trace file for Executing the Indexed C version of the

Production System on the Hypercube Simulator

```
Script started on Mon Jun 12 16:39:01 1989
> cat mab.trace
strace  tid  0 clock 4 running 0
cnt    1      clock  5
tfork  tid  0 clock 214 taddr 11558  stack 10000 waking 1
cnt    2      clock  215
tfork  tid  0 clock 217 taddr 16812  stack 10000 waking 2
cnt    3      clock  218
tfork  tid  0 clock 220 taddr 21632  stack 10000 waking 3
cnt    4      clock  221
tfork  tid  0 clock 223 taddr 25938  stack 10000 waking 4
cnt    5      clock  224
evpost tid 0 clock 224 addr 0150520 was CLEARED
texit  tid  0 clock 224 status 0
cnt    4      clock  224
lock   tid  2 clock 235 laddr 48356 was UNLOCKED
tsleep tid  1 clock 237 ticks 1
cnt    3      clock  237
timer clock 239 waking 1
cnt    4      clock  239
tsleep tid  3 clock 241 ticks 1
cnt    3      clock  241
timer clock 242 waking 3
cnt    4      clock  242
lock   tid  1 clock 242 laddr 48360 was UNLOCKED
lock   tid  3 clock 245 laddr 48360 was LOCKED blocking 3
cnt    3      clock  245
unlock  tid  1 clock 245 laddr 48360 was LOCKED waking 3
cnt    4      clock  246
tsleep tid  4 clock 246 ticks 1
cnt    3      clock  246
timer clock 247 waking 4
cnt    4      clock  247
unlock  tid  3 clock 248 laddr 48360 was LOCKED
lock   tid  4 clock 250 laddr 48360 was UNLOCKED
evwait tid 1 clock 253 addr 0136354 was CLEARED blocking 1
cnt    3      clock  253
unlock  tid  4 clock 253 laddr 48360 was LOCKED
```

```
evwait tid 3 clock 256 addr 0136354 was CLEARED blocking 3
cnt   2     clock  256
evwait tid 4 clock 261 addr 0136354 was CLEARED blocking 4
cnt   1     clock  261
cnt   1     clock  1614
unlock  tid  3 clock 1659 laddr 48356 was LOCKED
evpost tid 3 clock 1663 addr 0136354 was CLEARED waking 2 4 1
cnt   4     clock  1664
evclear tid 3 clock 1665 addr 0136354 was POSTED
lock    tid  3 clock 1668 laddr 48360 was UNLOCKED
unlock  tid  3 clock 1671 laddr 48360 was LOCKED
lock    tid  1 clock 1685 laddr 48356 was UNLOCKED
tsleep tid  4 clock 1688 ticks 1
cnt   3     clock  1688
tsleep tid  2 clock 1688 ticks 1
cnt   2     clock  1688
timer clock 1689 waking 4
timer clock 1689 waking 2
cnt   4     clock  1689
lock    tid  2 clock 1692 laddr 48360 was UNLOCKED
lock    tid  4 clock 1692 laddr 48360 was LOCKED blocking 4
cnt   3     clock  1692
unlock  tid  2 clock 1695 laddr 48360 was LOCKED waking 4
cnt   4     clock  1696
tsleep tid  3 clock 1696 ticks 1
cnt   3     clock  1696
timer clock 1698 waking 3
cnt   4     clock  1698
unlock  tid  4 clock 1698 laddr 48360 was LOCKED
lock    tid  3 clock 1701 laddr 48360 was UNLOCKED
evwait tid 2 clock 1703 addr 0136354 was CLEARED blocking 2
cnt   3     clock  1703
unlock  tid  3 clock 1704 laddr 48360 was LOCKED
evwait tid 4 clock 1706 addr 0136354 was CLEARED blocking 4
cnt   2     clock  1706
evwait tid 3 clock 1712 addr 0136354 was CLEARED blocking 3
cnt   1     clock  1712
unlock  tid  1 clock 1766 laddr 48356 was LOCKED
evpost tid 1 clock 1770 addr 0136354 was CLEARED waking 3 4 2
cnt   4     clock  1771
evclear tid 1 clock 1772 addr 0136354 was POSTED
lock    tid  1 clock 1775 laddr 48360 was UNLOCKED
lock    tid  3 clock 1776 laddr 48360 was LOCKED blocking 3
cnt   3     clock  1776
lock    tid  4 clock 1776 laddr 48360 was LOCKED blocking 4
cnt   2     clock  1776
```

```
lock   tid  2 clock 1776 laddr 48360 was LOCKED blocking 2
cnt    1      clock  1776
unlock  tid  1 clock 1778 laddr 48360 was LOCKED waking 3
cnt    2      clock  1779
unlock  tid  3 clock 1781 laddr 48360 was LOCKED waking 4
cnt    3      clock  1782
unlock  tid  4 clock 1784 laddr 48360 was LOCKED waking 2
cnt    4      clock  1785
lock   tid  1 clock 1786 laddr 48360 was LOCKED blocking 1
cnt    3      clock  1786
unlock  tid  2 clock 1787 laddr 48360 was LOCKED waking 1
cnt    4      clock  1788
evpost tid 3 clock 1789 addr 0150534 was CLEARED
texit   tid  3 clock 1789 status 3

script done on  Mon Jun 12 16:42:13 1989
```

Vita

Name: Rajendra K. Shrivastava

E-mail: INTERNET: rajendra@csvax.lsu.edu

Personal: 29 years old, male, Indian citizen.

Education: Ph. D., in COMPUTER SCIENCE , (Spring-1990), Louisiana State University; Minor field being Physics.

DISSERTATION TITLE: "Parallelization of Goal Driven, Production Systems On Hypercube Machines in a C Environment"; Major Professor being Dr. S. S. Iyengar.

M. S., in Physics, May-1983, Dept. of Physics, Poona University, India.

B. S., in Physics, May-1981, Fergusson College, Poona University, India.

Publications:

The following papers are in progress for refereed international journals and conferences.

1. An Efficient Algorithm for Executing Goal Driven, Forward Chaining Production Systems In Parallel.
2. An Approach for Parallelizing OPS5 Production Systems with a Faster Match Scheme, based on Indexing.

3. Some Empirical Results for Parallel Production Systems.

5. A Distributed Control Structure for the Production System Inference Engine.

6. Asynchronous Production Systems for Controlling Autonomous Mobile Robots.

To be presented at the 8th International Congress of Cybernetics and Systems, New York, June 12, 1990 *on behalf of Dr. Iyengar, Dr. Weisbin, Dr. Pin and the Robotics Research Group at LSU.*

Academic Work Experience/Courses Taught or Graded:

June 1988-Pres. Faculty
Department of Computer Science,
University of New Orleans.

CSCI 4401: Operating Systems.

CSCI 3401: Macros and VMS-Systems Services, Programming.

CSCI 2450: Introduction to VAX-Assembly and Machine Organization.

CSCI 1201: Introduction to Fortran Programming.

CSCI 1000: Computer Literacy, Architecture, DOS, BASIC, Dbase-III and Lotus-123. I was responsible for the CSCI 3401 and CSCI 4401 courses during the ACM accreditation visit at UNO during Fall 1989. The evaluation was successful.

Jan. 1986-1988 Teaching/Graduate Assistant,
Department of Computer Science,
Louisiana State University.

CSC 7002: Operating Systems & Computer Architecture; Graded [Spring '88].

CSC 7001: Programming Languages and Compilers; Graded [Fall '87].

CSC 1248: Pascal Programming; Taught [Fall '86 & Spring '87].

CSC 2263: Numerical Methods; Taught [Spring '86].

June 1985- December 1985 Teaching Assistant,
Department of Physics,
Louisiana State University.

Phy 2008: Physics Lab: Taught [Fall '85].

Programming & Systems Experience:

Languages: OPS5, C, LISP, PROLOG, PASCAL, FORTRAN, ADA.

Systems: BSD 4.3 Unix on VAX 11/780

System V Unix on 3B15

VM and TSO on IBM 3084

FPS 264 Scientific Computers and Tessaract Cube

Intel iPSC/2 Hypercube (Simulator)

SAS/SASGRAPHS on the IBM 3084

DOS on IBM Compatibles

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Rajendra K. Shrivastava

Major Field: Computer Science

Title of Dissertation: "Parallelization of Goal Driven, Production Systems on Hypercube Machines in a C Environment"

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

May 3, 1990