A HIGH LEVEL APPROACH TO THE
FAULT TOLERANT EXECUTION OF
AI EXPERT SYSTEMS

DANIEL P. MIRANKER

MARCH 1988 AI88-71

* This work was supported in part by the Office of Naval Research under contract #N00014-86-K-0763.
A High Level Language Approach to the
Fault Tolerant Execution of
AI Expert Systems

Daniel P. Miranker
Taylor Hall 2.214
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712
1/12/88

Abstract

To perform load balancing certain algorithms for the parallel execution of expert systems written in production rule form require the replication of both the program and the data. Thus an intrinsic requirement of fault tolerant systems, the replication of hardware and software components, is automatically satisfied by the parallel implementation of these systems. This paper describes how these algorithms may be extended to create a fault tolerant expert system interpreter.

The fault-tolerant model includes graceful degradation. Certain properties of forward chaining rule systems allow redundant processors to be exploited simultaneously as spares and as additional processors contributing to performance. Further, increased processor utilization may permit software based fault detection mechanisms to run without detracting from overall performance. In a sense the introduction of fault-tolerance into a parallel production system interpreter may be free.

This research was supported in part by the Office of Naval Research under contract N00014-86-K-0763.
# Table of Contents

1. Introduction 1

1.1. Production Systems 1

2. The Parallel Execution of Production Systems 2

2.1. The Full-Distribution Algorithm 2

2.2. Problems with the Full-Distribution Algorithm 3

2.2.1. The Locality Problem 3

2.2.2. The Production System Hot Spot Problem 4

2.3. Load Balancing in Light of the Hot Spot Problem: Copy and Constraining 4

3. Application of Full Distribution with Copy and Constraining to Fault Tolerance 5

3.1. Fault Recovery 5

3.2. Fault Detection 6

3.3. More Opportunity for Load Balancing 6

4. Current Research 7
List of Figures

Figure 1-1: An OPSS Production System Fragment 2
Figure 2-1: Abstract Algorithm Illustrating the Full-<Distribution Algorithm 3
Figure 3-1: Illustration of the Loading Requirements for Fault Recovery 6
Figure 3-2: Illustration of Hash Bucket Duplication With and Without Duplicated Hardware 7
1. Introduction

The field of expert systems, programs that are able to capture the knowledge of highly trained specialists, has matured in Artificial Intelligence (AI) research laboratories and has become an area of growing commercial interest. A report by Arthur D. Little Inc. predicts an annual market for AI-Expert Systems of more than three billion dollars by the year 1990 [Alpert 86]. Expert systems are often implemented using production systems. IBM, AT&T and DEC have built expert systems using the production system format [Griesmer 84, Vesonder et al. 83, McDermott 81]. Certain applications of expert systems require continuing and correct operation of the expert system despite the failure of one or more of the components in the host computer. Examples of such applications include process control, decision systems in autonomous vehicles, and many defense related applications.

Fault tolerant computer systems often require the presence of redundant components to take the place of those that fail as well as redundant representation of the computer program. Parallel computers, by definition, contain multiple copies of each of their constituent functional units. Further, to perform load balancing the most promising parallel production system algorithms distribute multiple copies of the expert system among the computer’s processing elements, (PEs). Thus, the implementation of production systems on parallel computers automatically satisfies a basic requirement of fault tolerant computing.

An ongoing project is investigating the extension of these algorithms to provide continued and correct execution of expert systems despite the presence of multiple hardware faults. A unique feature of the approach is fault tolerance will be created strictly by means that are capturable in software. Many fault tolerant computers include a large quantity of specialized circuitry and redundant hardware components whose only contribution to the system is providing fault detection. The approach in this project makes no assumptions about the target computer, except that it is a distributed-memory, parallel computer supporting asynchronous message passing. Further, all connected, fault-free processors in the system will actively contribute to the computation allowing the speed of the computation to degrade gracefully as faults are introduced. Last, due to increased processor utilization, a fault tolerant implementation of a production system interpreter may perform at nearly the same speed as the non-fault tolerant version on the same piece of hardware.

1.1. Production Systems

In general, a production system [Newell 73, Rychener 76, McDermott and Forgy 78] is defined by a set of rules, or productions, that form the production memory (PM), together with a database of current assertions, called the working memory (WM). Each production has two parts, the left-hand side (LHS) and the right-hand side (RHS). The left-hand side contains a conjunction of pattern elements that are matched against the working memory. The right-hand side contains directives that update the working memory and directives affecting external side effects such as reading or writing an I/O channel. Working memory is updated by adding or removing facts.

In operation, a production system interpreter repeatedly executes the following cycle of operations:

1. Match. An instantiation is an ordered subset of WM elements satisfying a rule’s LHS. For each rule, compare the LHS against the current WM. Enumerate all possible instantiations to form the conflict set.

2. Select. From the conflict set choose a subset of instantiations according to a predefined procedure. In practice select chooses a single instantiation from the conflict set on the basis of the recency of the matched data in the WM, as well as syntactic discrimination.


Production systems form the basis of many approaches to building expert systems. OPS5 [Forgy 81] is in the public domain and has emerged as a widely used production system language. Figure 1-1 contains an example of a rule in
OPS5 syntax. Working memory elements in OPS5 are represented by an initial constant denoting a record type or class name. The constant is followed by a list of attribute value pairs. Attribute names are prefixed with a carrot. The order of the appearance of the attribute value pairs has no significance.

The LHS of the rules are formed by a conjunction of pattern elements. Pattern elements are also called condition elements. Each condition element is composed of an initial class definition followed by a list of attribute pattern pairs. A pattern may be a constant, or a pattern variable, preceded by a relational predicate. The lack of an explicit predicate denotes an equality test. The first condition element containing a particular pattern variable must use an equality test. During the match phase the value substituted for the pattern variable is called the binding of the variable.

\[(p \text{ categorize-job-sizes} \text{ ;rule name} \text{ (message \text{ ^job <x> ^size <y> ^status new}) ;condition element} \text{ ;<x>, <y> are pattern variables} \text{ (job-definition ^size <y> ^size-def medium) ;condition element} \text{ -->} \text{ (make job ^job-name <x> ^job-size <y>)} \text{ })\]

This rule says if
there is a working memory element in the system representing
a message about a new job,
and the job's size matches the definition for a medium size job
then
create a new working memory element tagging as being in the
medium size category.

Figure 1-1: An OPS5 Production System Fragment

2. The Parallel Execution of Production Systems

2.1. The Full-Distribution Algorithm
There are a variety of algorithmic issues leading to a large space of possible parallel algorithms for the execution of production systems [Stolfo 84, Miranker 86]. We will consider a variant of the Full-Distribution Algorithm for distributed-memory parallel computers. Each rule of a production system may be independently matched against the working memory. The Full-Distribution Algorithm stipulates that each PE gets a copy of the rule matcher, a disjoint subset of the rules and a copy of the working memory relevant\(^1\) to those rules. The relevance tests filter the working memory and reduce the number of WM elements stored in each processor. Not all working memory elements have to be represented in all processors.

In the Full-Distribution Algorithm each processor performs the match component of the production system cycle independently of the others and computes a subset of the conflict set. Each processor then selects from its local subset of the conflict set a winning instantiation. When all the processors have completed these local steps, the processors communicate their winning instantiations among themselves to perform a global select operation. The

\(^1\)A working memory element is relevant to a rule if it satisfies some simple and easily computable pattern constraints. In OPS5 the constants and repeated pattern variables within a condition form the intracondition pattern constraints [Forgy 82].
act phase then retrieves the RHS of the winning rule and broadcasts the changes to the working memory to all of the PEs. The synchronization forced by the select and act phase keeps the distributed versions of the working memory consistent. The PEs locally compute the relevance of the changes to working memory. The production system cycle is closed by the initiation of a new match phase. An abstract description of the Full-Distribution Algorithm appears in Figure 2-1.

An advantage of the Full-Distribution Algorithm is the minimal communication required among the PEs. The only communication is the broadcast of working memory changes and the winning instantiation from each PE. It may be concluded from the study of several large OPS5 programs that the total amount of data communicated per cycle averages less than 1.5K bytes [Gupta 84].

1. Initialize: Load the local match and select code into each PE in the machine. Distribute a few distinct rules to each PE. Set CHANGES to the initial working memory elements (WMEs).

2. Repeat the following:

3. Act: For each WME in CHANGES do:
   a. Broadcast the WME to all the PEs.
   b. Each PE determines the relevance of the WME to the locally stored rules. If the WME is relevant, then store the WME for further processing during the match phase.
   c. end do;

4. Match: Each PE must execute the local match code and compute the conflict set of instantiations for its set of rules.

5. Select:
   a. Each PE determines the best instantiation in its local conflict set.
   b. The PEs communicate among themselves to determine globally the best instantiation.

6. Instantiate: Set CHANGES to the set of working memory updates determined by the RHS of the winning instantiation.

7. end Repeat;

Figure 2-1: Abstract Algorithm Illustrating the Full-Distribution Algorithm

2.2. Problems with the Full-Distribution Algorithm

2.2.1. The Locality Problem

OPS5 production system programs have a locality property similar to the working set property exhibited by conventional computer programs on serial computers; no matter how large an OPS5 program is, only a small subset of the rules are interesting\(^2\) at any given instant. Call the interesting subset of rules the working set. The number of partitions into which the system may be divided without severely under utilizing processors is proportional to the size of the working set.

The precise definition and average size of the working set depends upon the low level matching algorithm used compute the conflict set and the particular rule system [Miranker 87a, Gupta 86]. The working set sizes have been measured and range from 2 to 30 rules. Since good match algorithms quickly focus on the rules in the working set the opportunity for rule level parallelism may be naively perceived to be quite small.

\(^2\)A rule is interesting if the match algorithm searches for consistent variable bindings between condition elements.
2.2.2. The Production System Hot Spot Problem
More serious than the locality problem is a hot spot problem that prevents even a modest amount of parallelism from being used constructively. Generally, within a working set of rules, one rule will take a much longer time to match than the other rules in the set. The single culprit rule can take so much longer that it will completely dominate the processing requirements of a given cycle. Gupta, in several large OPS5 programs executed using the RETE match [Gupta 86], has measured a fivefold discrepancy, on average, between the match time of the slowest rule in a working set and the mean matching time of the rules in a working set. For the systems studied in [Miranker 86] using the TREAT match the discrepancy was slightly worse.

2.3. Load Balancing in Light of the Hot Spot Problem: Copy and Constrain
A method called copy and constrain [Stolfo et al. 85] has been proposed to solve hot spot problem. The scheme is best introduced by simple example. Consider the stylized rule

\[ P_1 (C_1 \ldots C_n \rightarrow A_1 \ldots A_m), \]

where the \( C_i \) (i=1,...,n) are condition elements, and the \( A_j \) (j=1,...,m) are actions. If we interpret the working memory, (WM), as some large universal relation, or set of tuples, then we may view the constants in each condition element \( C_i \) as the parameters for a relational selection [Ullman 82]. Let \( R_i \) represent the relation formed by the select operators for \( C_i \).

The set of instantiations of \( P_1 \) is then the multiway join of the relations \( R_i \). The local memory requirements and execution time to match \( P_1 \) are bounded by and may achieve the size of the full Cartesian product of the individual relations \( R_i \).

Suppose for concreteness that \( C_2 \) is a relational selection of a large number of physical objects, represented by the OPS5-style pattern:

\[ (\text{PHYSICAL-OBJECT} \ ^\text{Name}<x> \ ^\text{Color}<y> \ ^\text{Shape}<z>), \]

and that the domain of the Color attribute of relation \( R_2 \) is \{RED, GREEN\}, that is physical objects are either RED or GREEN. To speed up the match of rule \( P_1 \), we split the the rule \( P_1 \) into two copies, each to be loaded in to different PEs, each with additional patterns constraining each rule’s search to objects of each color. Thus we construct two new condition elements:

\[ C'_2 \text{ (PHYSICAL-OBJECT} \ ^\text{Name}<x> \ ^\text{Color RED} \ ^\text{Shape}<z>), \]
\[ C''_2 \text{ (PHYSICAL-OBJECT} \ ^\text{Name}<x> \ ^\text{Color GREEN} \ ^\text{Shape}<z>), \]

and two new rules:

\[ P'_1 (C_1 \ldots C_n \rightarrow A_1 \ldots A_m) \]
\[ P''_1 (C'_1 \ldots C'_n \rightarrow A_1 \ldots A_m), \]

If \( PE_1 \) and \( PE''_1 \) are assigned to distinct PEs, they may be matched in parallel and the set of instantiations of \( P_1 \) is exactly the disjoint union of the instantiations of \( P'_1 \) and \( P''_1 \).

In the best case the filtering mechanism will force half the tuples selected by the \( C_2 \) to reside in each of \( PE' \) and \( PE'' \), and the processing time required to match \( P_1 \) will decrease by half. Local processing requirements and storage of working memory decrease significantly as well. In the worst case all the tuples selected by \( C_2 \) reside in only one of the two PEs and there will be no improvement. If more PEs are available, the scheme can be applied repeatedly, producing many copies of rules each constrained to match a smaller range of distinct working memory elements. Thus, with many PEs available, it is possible to reduce the inter-PE variation in processing times, balancing the execution load over the entire system and increasing overall performance dramatically [Pasik and Stolfo 87].

Several variations on the technique can be used when the range domain of attribute value is unknown. Only one,

\footnote{These select operations are part of the relevancy filters.}
hash partitioning, is discussed here. The interested reader may see [Stolfo et al. 85]. Suppose that in the above example, the domain of the Color field is not (RED, GREEN), but is some (possibly infinite) domain D. A hash function is then introduced as constraining pattern into each copy. Given an easily computable function hash function, h.

\[ h: D \rightarrow \{1, \ldots, k\} \]

We can split \( C_2 \) into k copies

\[ C'_i = (\text{PHYSICAL-OBJECT} \ ^\text{Name} <x> \ ^\text{Color} h(<y>) = i \ ^\text{Shape} <z>, i=1, \ldots, k) \]

and substitute them into \( P_1 \) forming k copies of \( P_1 \) and assign each of the k copies to a separate PE. In the best case the processing time for the match phase of \( P_1 \) is divided by the number of partitions. Thus, the scheme is similar in many respects to hash partitioning tuples in a relational database when the relations exceed the size of main memory. In databases the data is partitioned so it will fit into memory. Each partition is processed sequentially. Hence hash partitioning is applied to produce many queries each operated on in parallel.

3. Application of Full Distribution with Copy and Constrain to Fault Tolerance
Two key aspects of a fault tolerant system are the detection of a fault and the recovery from a fault.

3.1. Fault Recovery
The following scheme describes an extension of the Full-Distribution Algorithm that allows fault recovery. To assure hardware redundancy consider a distributed-memory parallel computer with 2 PEs. Assume that each rule in the system contains at least one condition element that contains a partitioned attribute. Also assume that the rules may be divided into two equal sized subsets, A and B, such that there is a mapping, G, from A to B with the following properties:

- G is one-to-one and onto.
- Given, \( P_1 \) in A, \( P_2 \) in B, and \( G(P_1) = P_2 \), \( P_1 \) and \( P_2 \) each contain a condition element of the same class and having a pattern variable in the attribute that serves as a partitioning key.

To create a singly fault tolerant system, place the rules from set A with hash value field odd and the rules from set B with hash value field even in PE1. Place the remainder in PE2.

The mapping G and the copying algorithm assure that all rules in the system are partitioned and that two full copies of the working memory are made. Each PE will contain a full set of the working memory and a full set of the rules. In a two PE computer it is obvious each PE must contain a full copy of the system. In computers with many PEs it is expensive and unnecessary to have a full copy in every PE. It is sufficient that two copies are distributed to the computer and that a single failure does not lose both copies of a particular element. This loading algorithm assures that each copy of a rule and each copy of a working memory element is loaded into a different PE.

Though the rules are replicated each rule copy contains additional information in it constraining its sensitivity and workload. Until there is a fault the two processor system will act as a parallel processor. The extra constraining information sensitizes each rule copy to a subset of the working memory. Upon a fault all information, both program and data, is present in each PE. Thus, only the state of the faulty PE is lost. To recompute the lost state, the interpreter resident in the working PE must complement the least significant bit of the hash constants in its copy of the rules and recompute the lost portion of the conflict set. On subsequent production system cycles the good PE must ignore the hash partitioning and execute the entire computation. The performance degrades to that of a uniprocessor.
3.3. More Opportunity for Load Balancing

The copy and constrain method does not increase processor utilization to 100%. Also, due to the locality problem most of hash buckets will be empty most of the time. As a result, given a parallel computer, rather than replicate sufficient hardware for fault detection and making full identical copies of the production system it may be better to assure that each hash bucket is replicated and each copy of the replicated bucket is processed by a separate PE. It is possible that the wasted idle processing cycles in the parallel implementation may be used to perform the redundant computation required for the fault detection algorithm. If so, the fault tolerant version would execute in the same time as the parallel version.

This is clarified with a picture, (Figure 3-2). Suppose all the rules may be represented by mapping them to a divided
line and that the intervals represent hash buckets. The first picture illustrates an approach typical of many fault tolerant systems. Two full copies of both the hardware and the software mirror each other’s activities exactly. The second picture shows each individual hash bucket mapped to different PEs within the original piece of hardware. A good allocation program may be able to assign the additional copies to buckets in a way that will have little impact on the performance of the system.

![Diagram](image)

**Figure 3-2:** Illustration of Hash Bucket Duplication With and Without Duplicated Hardware

4. Current Research

A goal of this project is to create a fault tolerant system independent of the target computer. As a first step a portable C based OPS5 compiler is being written and should be complete at the time of this publication. The compiler incorporates recent ideas on the compilation and optimization of rule systems [Miranker 87b]. The compiler's initial target is an Intel hyper-cube. The fault recovery aspect as described above is straightforward. Once implemented an empirical study of the degradation in performance as faults are introduced will be conducted.

The most interesting and unresolved area of this project is the issue of fault detection by the software. On a given cycle a PE may not contribute any instantiations to the conflict set. Thus there may be no data communicated by that PE during the the select phase. The voting algorithm must be robust to adversarial and possibly incomplete information.
References

[Alpert 86] Alpert, B.
No Street Smarts?, Are Investors Flunking the AI Test.
Barron's :16, 8/4/86.

Fault Tolerance Principles and Practice.
Prentice Hall, 1981.

[Dahbura 87] Dahbura, A. T., Sabnani, K.K. and King, L.L.
The Comparison Approach to Multiprocessor Fault Diagnosis.

[Forgy 81] Forgy C. L.

[Forgy 82] Forgy C. L.
Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem.

YES/MVS: a continuous real time expert system.

[Gupta 84] Gupta, A.
Measurements on Production Systems.

[Gupta 86] Gupta, A.
Parallelism in Production Systems.

[McDermott 81] McDermott J.
R1: The Formative Years.

Production System Conflict Resolution Strategies.
Pattern-directed Inference Systems.

[Miranker 86] Miranker, Daniel P.
TREAT: A New and Efficient Matching Algorithm for AI Production Systems.

[Miranker 87a] Miranker, Daniel P.

[Miranker 87b] Miranker, Daniel P.
[Newell 73] Newell, A.
Production Systems: Models of Control Structures.
Visual Information Processing.

[Pasik and Stolfo 87] Pasik, A. and Stolfo, S.J.
Improving Production System Performance on Parallel Architectures by Creating Constrained Copies of Rules.

[Rychener 76] Rychener, M.
Production Systems as a Programming Language for Artificial Intelligence.
Ph.D. Thesis.

[Stolfo 84] Stolfo S. J.
Five Parallel Algorithms for Production System Execution on the DADO Machine,

[Stolfo et al. 85] Stolfo, S.J., Miranker, D.P. and Mills, R.
A Simple Preprocessing Scheme to Extract and Load Balance Implicit Parallelism in the Concurrent Match of Production Rules.

[Ullman 82] Ullman, J.D.
Principles of Database Systems.

ACE: An Expert System for Telephone Cable Maintenance.