OBJECT ORIENTED REAL TIME ASYNCHRONOUS PRODUCTION SYSTEMS

By

TOLETY SIVA PERRAJU

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE AWARD OF THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES

SCHOOL OF MATHEMATICS AND COMPUTER/INFORMATION SCIENCES

UNIVERSITY OF HYDERABAD

HYDERABAD 500 134

INDIA

SEPTEMBER 1993

CERTIFICATE

This is to certify that the thesis work entitled OBJECT ORIENTED REAL TIME ASYNCHRONOUS PRODUCTION SYSTEMS being submitted by Tolety Siva Perraju in partial fulfillment of the requirements for the award of the degree of Doctor of Philosophy (Computer Science) of the University of Hyderabad, is a record of bonafide work carried out by him under my supervision.

The matter embodied in this thesis has not been submitted for the award of any other research degree.

Prof. B.E. Prasad,
Supervisor,
Department of Computer and Information Sciences,
University of Hyderabad.

Prof. B.E. Prasad,
Head,
Department of Comp. and Info Sci.,
University of Hyderabad

Prof. V. Kannan,
Dean,
School of MCIS,
University of Hyderabad

DECLARATION

I, Tolety Siva Perraju, here by declare that the thesis work entitled OBJECT ORIENTED REAL TIME ASYNCHRONOUS PRODUCTION SYSTEMS is the bonafide work carried out by me, as per the Ph.D. ordinances of the University. The matter embodied in this thesis has not been submitted for the award of any other research degree to the best of my knowledge.

Tolety Siva Perraju

To my parents

Smt. T. Soudamini and Sri. T.Radhakrishna

and my brothers

T.V.L. Narasimha Murthy and T. Srinivas

ACKNOWLEDGEMENTS

A work of this magnitude is only possible with the help of many people. It is out of deep sense of gratitude to all these people, I am writing these acknowledgements. In all my humbleness, I express my sincere gratitude to my supervisor Prof. B.E.Prasad for his guidance, support and encouragement. But for his untiring efforts, I could not have carried out this voluminous work. Many of the ideas presented in this thesis have originated in my discussions with him. He has enlightened as to me what research is about.

I thank Dr. G.Uma for her advice and encouragement. The quality of the thesis has greatly improved on account of her incisive comments.

My happiness knows no bounds to thank Sri. KVSS Prasad Rao. The idea of developing a real time expert system has germinated in our discussions. It is a great pleasure and privilege working with him. I thank him again for his encouragement to pursue this research work. I express my gratitude to Sri WN Raghupathy, Sri V Nagraj and Sri SP Gupta.

I prostrate with respect before my father Sri. T.Radhakrishna. He is responsible for inculcating in me the mathematical and analytical skills which I possess today.

Many people have helped me in the preparation of this thesis. Foremost among them, stands Smt. P.Umarani. Much credit should go to her, for her immeasurable help in the implementation of what is designed. I sincerely thank her for the same. I thank all the faculty at the University of Hyderabad, Dr.A.S.Reddy, Dr.Arun Agarwal, Dr.P.S.Rao, Prof. A.K. Pujari, Atul Negi for their advice and help. I thank Sri.PVSS Kameswara Rao, Sri.Balakrishna and the staff of the AI Lab and the Department of Computer Science, for their help. I express my thanks to Prof V. Kannan, Dean, School of MCIS.

I would like to thank all my friends, who, over the years had made life so enjoyable. I thank Vivek, MP Reddy, MR Prasad, Kailash, Rajeshwara Rao, Sriramu, Chalam, Chato, GN Rao, JVR Sagar, MS Kishore, Ramanaiah, Vardharaj, GS Raju, Phaninath, J Suryanarayana, Venugopal Reddy, Nalinikumari, Lakshminarayana, Parameswaran, Vishwanath, GV Subba Rao, NV Rao, T.Satya Narayana, Bapi Raju and epsecially CS Murthy and TV Rama Rao. I take this opportunity to express my respects to my close friend Mrs. Anuradha who is no more. I thank all my friends and colleagues, who during the various phases of my career, have helped me a lot to carry on through research. Forgetting to mention their names is not by any intention, but only a slip. I thank all my teachers for the education they kindly bestowed me with. There are many people from whom, I had the privilege to receive help, but not the chance to thank them. I take this opportunity to thank them all.

Last but not the least, I express my deep sense of reverence to my parents and family, who have been a constant source of inspiration and encouragement throughout my educational and other endeavours. My two younger brothers with their love for me have made my life pleasant. I take this opportunity to express my love and affection for them.

Tolety Siva Perraju

Abstract

Real time domains like aerospace, military, and process monitoring require expert systems to possess capabilities for handling continuous streams of data, temporal knowledge and provide reactive response. Traditional expert systems fail to meet these requirements. In this thesis, techniques for temporal data and knowledge representation, continuous reasoning, matching and interference analysis for multiple rule firing are developed. An object oriented real time asynchronous production system architecture based on an extended Petri net model, incorporating the above techniques is proposed.

Data in real time domains are multi faceted. A structured and efficient data management scheme is required to capture the semantics of real time data. Knowledge about temporal events and their relationships has to be represented in real time expert systems. This requires a suitable representation formalism. A unified object oriented data and knowledge representation scheme which can capture these properties is presented. An augmented rule structure is defined to represent knowledge about temporal properties. Three rule types viz. Autonomous rules, Clock Synchronised rules and Spanning rules are defined to represent different kinds of knowledge. Clock synchronised rules represent knowledge about temporal relationships between events. Spanning rules pertain to trends in historical data.

The reactive response behaviour of real time expert systems, depend on the efficiency of the reasoning process. Since data is continuously arriving the latest data values invalidate the earlier data values. This property is used in the design of the REX match algorithm. The evaluation of the premises in the rule base is modelled as Select and Join operations on the working memory. The evaluation of Spanning premises is modelled as the evaluation of an aggregation query on a set of object instances. The *Match process* is designed to accept external data when available and handle the Clock synchronised rules and Spanning rules efficiently.

The match algorithm performs linear number of premise evaluations per attribute update.

Expert system architectures with single rule firing per inference cycle, cannot handle all simultaneously occurring events in the external world. Therefore, multiple rule firing in a single inference cycle has been adopted in REX. In a multiple rule firing model, integrity of working memory has to be ensured. This is achieved through interference analysis. Techniques proposed in literature are compute intense, deadlock prone and specific to OPS5. We have proposed a new technique for interference analysis. This algorithm is deadlock free and less compute intense. The algorithm is not specific to REX and can be applied even to OPS5 like systems. The asynchronous rule firing scheme is presented next. This is followed by discussions on the significant implementation aspects. Finally, a summary of the work and the scope for further work are presented.

Table of Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

1.1 Expert Systems

An expert system is an intelligent program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solution. The knowledge necessary to perform at such a level, plus the inference procedures used, can be thought of as a model of the expertise of the best practitioners of the field [20].

The knowledge of an expert system consists of facts and heuristics. The *facts* are a segment of information, that is agreed upon by experts in a domain and is publicly available. The *heuristics* are empirical pieces of knowledge about good judgement in the domain. Heuristics characterise the expert level decision making process in the field and are mostly private. The performance level of an expert system is primarily a function of the quality and size of the knowledge it possesses.

An expert system consists of

- a knowledge base composed of the domain facts and problem solving heuristics.
- an inference procedure for solving the problem utilising the knowledge base
- a working memory (global database) which acts as a repository for all data including the inputs for the current problem, the present problem state and any other relevant data.

Expert systems differ from conventional software in many important aspects. In expert systems, there is a clear demarcation between the knowledge about the

problem domain and the knowledge for applying the same in solving a problem [18]. The former is encoded in the knowledge base, while the later is implemented as an inference procedure of the expert system. Ideally, it should be possible to change the system, by simple additions and deletions to the knowledge base.

The potential uses of expert systems are many. They can be used to

- design
- monitor
- interpret
- analyze
- diagnose
- explain and
- consult.

Consequently, there are many application domains of expert systems. Some of the domains are

- mission planning
- signal analysis
- command and control
- manufacturing
- image analysis
- software engineering
- electronic warfare and adaptive control
- logistics
- real time monitoring and process control.

1.1.1 Rule Based Systems

The most popular way of representing domain knowledge in an expert system is by using production rules. An expert system using production rules to encode domain knowledge is termed a *rule based system.*

The cognitive strategies of most human experts in complex domains are based on the mental storage and use of a large collection of pattern based rules [50]. Rules are a powerful paradigm for acquiring, organising and utilising expertise. A well chosen rule base may maintain control over otherwise intractable explosion of combinatorial complexity [24].

The problem solving paradigm used in rule based systems is rule chaining. If problem solving is initiated with a set of conditions and moves towards a conclusion, the method is called forward chaining. Production systems are a well known type of rule based systems in which the control structure can be mapped into the forward chaining paradigm. The production system interpreter embodies a mechanism that implements forward reasoning through repeated execution of Match-Select- Act cycles [53, 92]. If the conclusion is known, but the path to the conclusion is not known, then it is necessary to work backwards. This method is termed backward chaining. Forward chaining (data driven reasoning) is used for functions like monitoring, interpretation and analysis. Backward chaining (event driven reasoning) is mostly used for diagnosis and to a limited extent for functions like design and planning.

Rule based systems are used successfully in many domains. To deploy the rule based systems in domains requiring real time performance several methods have been suggested. They are

- Use of faster technology: As fast hardware becomes available, it can be used to speed up production system execution.
- Use of better algorithms: According to Forgy [22], in forward chaining systems the match phase contributes 90% of the run time and is a bottleneck that needs to be improved. TREAT [51] and LEAPS [54] are examples of fast match algorithms.

- Use of better architectures: Architectures like asynchronous production systems [79] are employed.
- Use of innovative inference procedures: The process of firing a single rule in a cycle leads to large number of inference cycles. Multiple rule firings within a single inference cycle can be used to speed up rule based system execution [31, 33, 40, 81].
- Parallelism: Parallelism inherent in a rule based system like match parallelism, act parallelism [27, 83] and also application parallelism [29] are exploited.

1.1.2 **Blackboard** Systems

The blackboard model is a scheme proposed for organising reasoning steps and domain knowledge to construct a solution for such problems. A blackboard model consists of [59]

- Knowledge sources: Knowledge needed to solve the problem is partitioned into knowledge sources which are separate and independent.
- Blackboard: The problem solving state is kept in a global data store called the blackboard. Knowledge sources produce changes in the blackboard. Communication and interaction among knowledge sources is through the blackboard
- Control: The control component decides which knowledge source can be applied when and to what part of the blackboard.

The problem solving behaviour of a system is determined by the knowledge application strategy encoded in the control module. The choice of the appropriate knowledge application strategy is dependent on the characteristics of the application.

1.1.3 **Model Based Systems**

Model based systems are thought to be suitable for diagnosis and design applications. Knowledge for diagnosis and design applications consists of descriptions of hardware

and software components and their functions, causal descriptions of how they achieve their functions, malfunction modes, design plans and links to underlying scientific knowledge [48]. A coherent description of all the above is a model. Model based systems are based on reasoning from first principles.

1.2 Requirements of Real Time Expert Systems

The above three architectures are widely used in implementing knowledge based expert systems. In traditional application domains, the performance of the expert system is measured in subjective terms like correctness of solution and its quality. Response time is secondary and no specific requirement exists. However in real time applications the timing behaviour of the system is a significant parameter. Expert systems that support reasoning with real time data and its temporal properties are called real time expert systems. In real time expert systems, the time at which the solution is produced plays a major role in determining the correctness. Real time expert systems can broadly be classified into monitoring and control systems. A real time monitoring expert system puts to use the incoming data and generates advice at a rate greater than or equal to the rate at which data arrives into the system. A real time control expert system always produces a response at or before the time at which the response is required [42].

Typical real time domains like military, industry, space and medicine are pregnant with a variety of possible applications. These real time application domains possess peculiar characteristics that differentiate them from other expert system application domains. Jakob et.al classified these characteristics along three dimensions - time, complexity and uncertainty. Another dimension that has been added is fault behaviour. The characteristics of the application domains along these dimensions are [34]

- Time

    1. Continuous operations : Plants operate continuously, making vigilance and maintenance operations difficult.

2. Propagation delays : Since the plants are composed of physical systems, delays are inherent to the system. Changes in the system do not occur instantaneously but, do so, over a period of time. This is true for both normal and abnormal states.
3. Evolution : The system is continuously changing and therefore, there is no single discrete steady state but only a continuum of steady states.

- Complexity

1. Structural: Many diverse subsystems are brought together and structural diversity is inherent to the process system. Monitoring as many as 2000 to 3000 signals is not uncommon in such system.
2. Behaviourial : The structural complexity imparts a certain amount of behaviourial complexity too. Even though steady state behaviour of the system can be predicted, it becomes impossible to predict all possible failure modes and their combinations. Failures can combine in unexpected ways and produce disastrous consequences like Three Mile Island case.
3. Operational modes : In different operational modes, like start-up, steady state, shut down and failure modes, the system behaves differently.
4. Multiple models : Different models of the system become relevant in different operational modes. Mathematical models are apt for steady state operations, but could fail to explain system behaviour in start-up, shut down or failure modes. Even in steady state, different aspects of the system may require different models. For example in a nuclear power plant, steady state operations could need at least two models one for the nuclear fission and, another for the generator mechanisms used.

- Uncertainty

1. Data: Data are obtained from sensors. Sensors themselves being physical systems prone to errors and degradations, events like erroneous data or data washouts should always be considered. Further scarcity of sensors due to design flaws, cost constraints and other limitations could also cause non availability of data.

2. Model: The second most important source of uncertainty is the model of the system. A model, determines the behaviour of the given system in a specific bandwidth of operation. It is possible that the present operational state of the system could fall beyond this bandwidth resulting in model failure. Further, the approximations used in the model could also result in uncertainty in the expected system behaviour.

- Fault behaviour

1. Multiple faults : When faults occur in process systems, they normally do not occur in isolation but occur in large numbers. For example, it is reported that in the Three Mile Island accident some 300 lights indicating alarms went on and off in the first few seconds. This clearly taxes the operator's cognitive capability.
2. Propagated or chained faults : Due to tight coupling of various subsystems, faults propagate from one subsystem to another. The common cause for these apparently different faults must be identified and acted upon. Piece meal action independently for each fault can result in more disasters.

The above features of real time application domains require real time expert systems to possess some specific properties. Time is the most important entity, and the way it is handled characterises the system's real time behaviour. Various issues like time representation, representation of temporal data, encoding of temporal knowledge and management of temporal reasoning are to be addressed, by the system architecture. The other essential characteristics are [36]

1. Continuous reasoning: Real time systems are characterised by continuous streams of input data. This implies continuous reasoning. If all data and conclusions are stored, very soon the demands on the storage system will out strip supply. Even if a large memory system is available the response time becomes too large to be acceptable. However the old data and conclusions will become invalid as new data come and their deletion will not hamper system

performance. So, it should be decided when and how data can be discarded to facilitate continuous reasoning.

2. Reactive response behaviour : The process system is dynamic and the expert system should react to the changes in the process system i.e. initiate reasoning immediately upon sensing change.

3. Temporal representation of data : Data in process systems is dynamic and varies with time. Propagation delay and slow state change in process systems necessitate the expert system to study trends, make forecasts and generate early warnings. Therefore a time based representation and storage of data must be available for the expert system to study historical data.

4. Reason about temporal relationships of events : Temporal relationships form an important input for fault diagnosis of process systems.

5. Focus of attention : The expert system should focus on important events in the system rather than divert attention to trivial and frivolous events.

6. Asynchronous arrival of events : Monitoring system in real time domains look for fixed threshold alarms. When process variables exceed these threshold values, alarms are generated. The expert system should accept these asynchronous alarms to take necessary action.

7. Model delayed feedbacks : The expert system should consider occurrence of delayed feedbacks in its model of the process system

8. Handling of hardware and software interrupts : Since the expert system is a part of the total process control computing system, it is expected to handle software and hardware interrupts that can occur during the course of execution.

9. Embedded operation : Early expert systems are non communicative, and are available on a stand alone computer. This is not acceptable in real time systems. Ideally, they should be embedded in the target system itself (e.g. on board computers in space vehicle) or should accept data on line through a network from the process control computers.

10. Simultaneous Handling of Multiple Events : The expert system should be able to handle simultaneously occurring multiple events. These events can be either independent events or connected events in a chain.

11. Guaranteeing response in real time : This pertains to ensuing that either certain deadlines are met by the expert system or quick reaction is initiated in response to inputs. This is an area where least attention is focused in AI research.lt is a difficult goal to achieve, due to the non polynomial algorithms employed in AI systems.

1.3 Goals of Thesis

Though many expert systems are reported for real time applications, most of them, do not meet all the requirements enunciated in Section 1.2. The aim of the thesis is the design and development of an object oriented real time asynchronous production system for dealing with continuous streams of input data, meeting the following requirements.

- Temporal Representation of Data
- Representing and Reasoning about Temporal Relationships
- Modelling Delayed Feedbacks
- Continuous Reasoning
- Simultaneous Handling of Multiple Events
- Interruptible Reasoning
- Embedded Operation
- Reactive Response
- Focusing Attention
- Handling Asynchronous Events

1.4 Thesis Outline

Chapter 2 presents a survey of current techniques and tools in real time expert systems. A review of some real time expert systems is also presented.

Chapter 3 presents an Extended Petri Net formalism to model the knowledge representation, reasoning and knowledge verification aspects in real time expert systems. The architecture of REX based on the Extended Petri Net model is explained with a block schematic diagram.

Chapter 4 presents an object oriented scheme for representing data and knowledge of the real time domain. This representation is particularly tuned for temporal data and knowledge.

Chapter 5 discusses the match algorithm used in the inference engine. The techniques used in the algorithm for meeting specific real time knowledge representation requirements are also discussed. The inference engine uses a multiple rule firing strategy. This requires synchronising interfering rules. Chapter 6 presents a new interference analysis algorithm and the asynchronous rule firing strategy.

The concepts developed in these chapters are used in the implementation of REX. Chapter 7 details the implementation aspects of REX.

Finally in Chapter 8, the conclusions from this work are presented. Directions that can be pursued for enhancement the work are also discussed.

Chapter 2

COMPARATIVE STUDY OF REAL TIME EXPERT SYSTEMS

2.1 Introduction

Real time expert systems axe being implemented in a number of domains like industry, military and space. All real time applications demand execution speed. However, speed alone is not real time. Many more issues are involved and different techniques have been developed to meet them. Some techniques are developed in generic tools for implementing real time expert systems, while some are implemented in specific expert systems. In this chapter, we have attempted a survey of the techniques and tools developed, and some of the real time expert systems implemented.

2.2 Techniques

New data and knowledge representation schemes, efficient algorithms, problem specific inference procedures and architectures, have been proposed to modify expert systems for real time applications.

2.2.1 Data and Knowledge Representation Schemes

Due to the time varying and continuous stream of input data, a suitable representation formalism that supports time operations, temporal reasoning must be available. The data representation scheme must be able to archive time tagged data. Similarly, the knowledge representation scheme must have facilities to represent event based

knowledge and knowledge about temporal relationships. In [12, 64] knowledge representation schemes capable of representing data and knowledge for temporal domains are presented, (details presented in Section 2.3).

2.2.2 Efficient Match Algorithms

The RETE state saving pattern match algorithm is a fast algorithm used in OPS5 [22]. RETE's principle of state saving is based on the concept of temporal redundancy and minimises the number of comparisons of working memory elements. Further, since similar rules compile into similar networks, identical paths on the network can be shared among different rules. But, RETE match algorithm has many disadvantages. The concept of state saving means that a deletion entails the same sequence of operations that occurred during the addition of a working memory element. Hence deletions are expensive. The Beta memories maintained in the RETE are the cartesian product of the two input memories. This beta memory grows exponentially in space resulting in large execution times. YES/RETE is an improvement of RETE which improves network sharing and hence improves network update efficiency [39].

The TREAT algorithm is developed to improve these short comings [51, 53]. TREAT is based on Mc Dermott's conjecture that retesting costs in production system match will be less than the cost of maintaining the state. TREAT is a non state saving algorithm and uses the concept of conflict set support. The non state saving nature of TREAT makes deletions cheaper but addition could be costly. It is empirically demonstrated that TREAT consistently out performs RETE. Miranker et.al. introduced match code compilation to enhance the processing speed of production systems. The TREAT compiler compiles production systems and generates C code that performs the TREAT match [52]. Recent results reported by Miranker show that a compiled version of RETE match out performs a compiled version of TREAT [65]. The discussion about RETE and TREAT is essentially about constants and both algorithms are exponential in time and space. LEAPS is a lazy match algorithm with linear worst case complexity [54]. LEAPS is based on the observation that RETE and TREAT perform eager computing of rule instantiations but fire only

observation that RETE and TREAT perform eager computing of rule instantiations but fire only one instantiation in a cycle. The lazy match algorithm folds the selection strategy into a search for instantiations such that only one instantiation is computed per inference cycle. Uni-RETE is a specialisation of the RETE **for** unique attribute representations. Unique attribute representation is introduced to eliminate combinatorics from match without losing production system functionality [87]. Unique attribute representation implies a single token per beta memory. This makes Uni-RETE match linear in both time and space. A speed up of upto 10 is reported in SOAR production programs using Uni-RETE. Match Box is an incremental matching algorithm. The idea is to precompute a rule's binding space and then have them monitor the working memory for incremental formation of tuple instantiations [66]. Match Box effectively exploits fine grained parallelism. On a massive parallel architecture, Match Box can perform incremental join testing in constant time. If the binding space is much smaller than the tuple space, then Match Box can also be deployed on serial machines. In REX, a new match algorithm based on the properties of real time data and knowledge (details in Chapters 4 and 5) is proposed.

2.2.3 Parallelism for Speedup

Parallelism in match is initially investigated by Gupta in building the Production System Machine (PSM) [27]. The objective of this study is to investigate fine grained parallelism in production systems using RETE match algorithm. Different sources of parallelism like production parallelism, intra-node and inter node parallelism, RHS parallelism are investigated. It is reported that the amount of speed up that can be obtained is limited, due to the small number of affected productions caused by a few number of working memory changes. Another reason cited for the low speed up factors is the large variance in the processing requirements of the affected productions. A bus based architecture is recommended for implementing parallel match algorithms [27].

The applicability of data flow principles to parallel production system match is

the piling up of tokens on the input arc of the root node. The splitting of root node into multiple nodes is proposed to avoid this congestion. Multiple root nodes are based on the principle that a n - attribute working memory element cannot match a m -attribute condition element if m > n.

The improvements in parallel hardware resulted in efficient message passing computers. A match algorithm for message passing computers is developed at CMU. It is estimated that 98% of the RETE match time is spent in two input node activation. This algorithm reduces the computation time of two input node activations. The cross product effect in two input node activations is a major hurdle in achieving higher speed ups in parallel RETE match. PESA-1 is a RETE match algorithm that reduces the cross product effect [38], by partitioning the RETE net at the intra node level. DRETE (Distributed RETE) is a match algorithm implemented on a special purpose machine CUPID. DRETE reduces the cross product effect by a different method. A separate task is created for each element in the beta memories. A task has three ports, two for the incoming input memory nodes and one for output. CUPID is designed as a match processor attached to a host.

The DADO project at Columbia University investigates parallelism on a tree structured machine. This approach is found to provide little speed up because the inherent parallelism in production systems is low and consequently most of the processors are idle. Another reason is that due to the large number of processors, the processors used are simple and have narrow data paths, no cache and low speed which reduces the execution speed, A n -ary tree architecture to exploit coarse grained parallelism in production systems is proposed in the HERMIES robot project [78]. The productions in a program are partitioned into subsets and each production processor receives one such subset. The efficient partitioning of the productions into distinct subsets is the central idea behind this architecture. Some of the partitioning strategies considered in this project are dynamic partitioning, static partitioning, heuristic partitioning and hybrid partitioning. Ing-Ray Chen and Poole present a performance analysis of rule grouping in real time expert systems developed using Activation Framework [7]. A model describing the run time behaviour of expert system reasoning in real time architectures is constructed. Based on a parametric evaluation of this model, they present an optimising rule grouping algorithm with

the Keringhan-Lin heuristic graph partitioning algorithm as its core for k-way partitioning of the rule set. The results of the experimental evaluation show that k-way partitioning is a trade off between processor overheads and rule overheads. A large number of partitions are efficient if there are large number of rules or large number of instructions per rule.

2.2.4 Asynchronous Execution of Inference Engines

Inference Engines are usually sequential processes requiring dedicated processors for their implementation. In real time domains inference procedures must be embedded in the system, and execute along with other processing tasks in the system.

The asynchronous production system architecture is an attempt in this direction. In the APS defined by Sabharawal et. al. for real time expert systems in HERMIES robot [79], the three inference cycle phase - match, select and act are executed asynchronously on three different processors of a shared memory multi processor system. A fourth processor takes external input and changes data in the working memory.Changes in working memory are communicated between the three modules using interrupts. Another variety of asynchronous production systems discount the necessity of synchronisation in the conflict resolution phase of the inference cycle across the different processors. In other words they adopt a *fire when ready* policy [56, 57]. Synchronous production systems quickly reach saturation speed ups while asynchronous systems can continue to exploit linear speed up under increased work loads.

2.2.5 Multiple Rule Firing Systems

The speedup achieved by parallel execution of production systems is constrained by the number of working memory changes in each inference cycle. The concept of firing a single rule in an inference cycle (single state change) in production systems coupled with few actions per rule results in small number of working memory changes. Thus only small speed up factors are obtained by the parallellisation efforts.

Multiple rule firing systems obtain higher speedup factors by allowing more than one rule to be fired concurrently. This increases the number of working memory changes and hence the speed up. The result of multiple rule firings can be different from the results of any sequential firing of those rules. In this case interference is said to occur among multiple rule firing. In order to guarantee a consistent firing environment for a set of rules, it is necessary that interference among rules be detected and interfering rules be inhibited from firing concurrently. Static(compile time) and run time analyses of rules and rule instantiations respectively using data dependency graphs are carried out to detect interfering rules. Static analysis requires large storage space, while run time analysis requires high computation times. Most multiple rule firing systems usually make a space/time trade off using a combination of compile time and run time analyses [39].

Ishida proposed a model of parallel execution that performs interference analysis among rule instantiations using a data dependency graph [31, 33]. The compile time analysis detects interference by utilising all information written in the source programs. Run time analysis of rule instantiations produces more accurate data dependency graphs. Consequently interference analysis at run time allows more concurrency than compile time analysis. A rule selection algorithm based on these interference analyses is also proposed. This multiple rule firing increases the degree of concurrency by a factor of 2 to 9. Ishida later extended this parallel production system model into a distributed production system [32].

Schmolze presents an approach to the problem of guaranteeing serializable behaviour in synchronous production systems that fire rules simultaneously [81]. Two approaches to the serialization problem based on examining data dependency graphs are presented. One examines rule instantiations. This approach allows higher concurrency. The algorithm is $0(n^4)$ complex. A sub optimal $0(n^2)$ complex algorithm is also presented. The second approach examines rules and hence detects lower degrees of concurrency. Both the approaches are similar to Ishida's solutions though the detection strategy in the graphs is different.

Concurrent Rule Execution Language (CREL) is designed to avoid run time overhead by performing extensive compile time analysis [37]. Compile time

analysis based on serializability criteria is used to partition the productions into uniform clusters. Rules in different clusters can be executed concurrently without any further run time checking. It is impossible to determine precisely which rule instantiations within a cluster can be executed concurrently. The problem of run time checking for multiple rule firings within a cluster is a two stage process in CREL. In the first step interference between rule instantiations is detected. This is an $0(n^2)$ problem. To cut the computation in this step, the interference checks are precompiled into C functions and are executed at run time to detect interference. In the second step, the set of instantiations to be fired concurrently are detected. This is again an $O(n^2)$ problem. To reduce computation time at the cost of reduced concurrency a $0(n)$ algorithm is employed. The CREL system is implemented on a Sequent Symmetry shared memory computer.

Kuo and Moldovan describe the implementation of a multiple rule firing system on a hypercube (iPSC/2) [38]. Unlike most other models which deal with only interference problem, this model takes into account the convergence problem also. The convergence problem is concerned with the correctness of the parallel solution. In this model, the original problem is successively divided into smaller problems, one for each context. The convergence problem is addressed at two levels - the context and program level. Each context within the program is analyzed to find out if a conflict resolution step is required within the context. If so the context is called a sequential context or else it is a converging context. Rule instantiations within a converging context can be fired concurrently, while those in the sequential context need to be synchronised. At the program level, the contexts are analyzed for compatibility and reachability. Three different rule firing models are proposed. They are rule dependence model (RDM), single context multiple rule (SCMR) model and multiple context multiple rule (MCMR) model. In RDM only the compatibility problem is addressed without any consideration for contexts. This model is similar to Ishida's multiple rule firing model. The SCMR model extends RDM by considering contexts. In this model rules from a single converging context are allowed to fire concurrently. The MCMR model extends the SCMR models by considering multiple contexts. Rule instantiations from a given context are allowed to fire with instantiations from other contexts, if the context is compatible with and is not present in

the reachable sets of the other contexts being considered. The MCMR models offers the highest degree of concurrency. The number of contexts affects the performance of the models. Hence partitioning of rules into contexts is an important problem to be addressed in these models.

Matsuzawa proposes a parallel execution method for production systems with multiple worlds [49]. A world is defined as the set of working memory instances generated by the last rule firing. Each fired rule instantiation generates one world. A combination of several such worlds is called a virtual world or a merged world. Two worlds generated by rules Rl and R2 can be merged together if different sequences of firing Rl and R2 yield the same results. If the worlds can be merged then the corresponding rules can be fired concurrently. An algorithm is proposed to detect if worlds can be merged by examining rule instantiations and their creation history. This model however does not take into account negated condition elements in the production system.

The multiple rule firing models studied so far, use syntactic knowledge to detect interfering rules and inhibit them from firing concurrently. PARULEL is a parallel rule language which employs user defined meta rules to perform synchronisation in its multiple rule firing model[84]. These meta rules are called 'redaction' rules. Redaction rules eliminate rule instances from the conflict set. Meta rules specify the conditions under which two rule instances conflict and which of the conflicting rule instances need to be deleted from the conflict set. Meta rules can not only deal with data consistency conflicts but can also be used to deal with semantic consistency conflicts. However this concept of 'redaction' leaves the door open for bugs and it is the programmer's responsibility to provide a complete set of 'redaction' rules. ALEXSYS is a financial expert system built using PARULEL.

The concurrent execution of production rules in a database implementation is studied by Raschid et. al.[74]. DBMS serialisability criterion is taken as the correctness criterion for a correct concurrent execution of productions. A protocol based on the two phase locking scheme to ensure serializability is proposed with two extensions to the transaction manager. One extension allows pages of relations to be accessed while the relation itself may be locked. The second extension explores

the concept of nested sub transactions to allow concurrent execution of productions within a single transaction. A simulation test bed developed to study the rule features and database characteristics that influence the performance of concurrent rule execution. Transactions are used as a means of encapsulation of rules in this simulation study. The positive RHS actions result in database inserts while negative action elements results in deletes to database. Experiments show that throughput is inversely proportional to the number of inserts and is proportional to the number of deletes. A more detailed summary of the results is given in [14].

The above rule firing models obtain speed up through the inherent concurrency in the reasoning model ignoring application parallelism in the quest for speed up. SPAM - a production system architecture for computer vision applications is an example of a system achieving speed up by exploiting application parallelism coupled with production system parallelism [29]. The performance results of SPAM point to a possibility of obtaining linear speed ups (a speed up of 12 is obtained with 14 processors) by coupling task parallelism with production parallelism. Swarm is a non deterministic parallel language [39], that provides formal methods for specifying production problems, for coding production systems and for verifying the correctness of coded programs. The Swarm execution cycle is as follows

- non deterministically choose a transaction from the transaction space.
- match sub transactions simultaneously against the tuple space
- execute simultaneously all sub transactions that evaluate to true and make changes to the tuple and transaction spaces.

Niemann [56, 57] proposes a parallel production system that provides appropriate language mechanisms to design serializable concurrent production programs rather than providing a guarantee of serializability. A simple locking scheme for working memory coupled with appropriate language mechanism ensures design of serializable programs. These concepts are implemented in UMass Parallel OPS5 which incorporates parallelism at the rule, act and match levels. This also focuses on aspects of control in parallel rule firing systems. Three different control strategies

serializable programs. These concepts are implemented in UMass Parallel OPS5 which incorporates parallelism at the rule, act and match levels. This also focuses on aspects of control in parallel rule firing systems. Three different control strategies are proposed viz. dynamic scheduling of rule actions to make effective use of processors, heuristic control and algorithmic control [58]. In REX, a polynomial time deadlock free interference analysis technique is proposed and implemented (details in Chapter **6).**

2.2.6 Distributed Problem Solving Systems

The computational demands and the amount of knowledge typically exceed the capabilities of a single intelligent agent. Cooperative distributed problem solving architectures are being proposed to handle such computational requirements [23, 3, 26).

2.2.7 Problem Specific Inference Procedures

Problem specific inference procedures [34, 61, 62] and latest techniques like neural networks [83] are being developed to meet the specific needs of application domains. EXTASE [34] is such a situation assessment system. Inference in EXTASE is performed by traversing the plant tructural graph and creating a hypothesis graph which corresponds to the current system configuration and state. The inference algorithm is explicitly designed to suit a particular situation and is not a generic facility.

2.3 Tools

There are different tools which implement the above techniques to facilitate development of real time expert systems. A survey of the different tools is presented below. In Table 2.1, a summary of the features (discussed in Chapter 1) of the different tools is presented.

LES did not support temporal reasoning and hence diagnosis is performed using current telemetry data without reference to historical data. A later version of LES, provides temporal support facilities like handling frame slot values as a function of time. These values can be referred to in the rule consequents, external procedures and interactive updates to frame base. One major shortcoming in LES is the fixed amount of storage for temporal data. The knowledge engineer needs to define *a priori* the number of data values to be stored for a particular slot. In continuous monitoring systems, it is impossible to decide a priori the amount of history to be considered during inference.

Active databases is another area where constraints in real time applications are being studied. **HiPAC** is an active database project that seeks to perform time constrained management [12]. This project addresses two critical areas in active databases: the handling of time constraints and the avoidance of wasteful polling by use of situation - action rules as an integral part of DBMS condition monitor. A rich knowledge model in HiPAC makes it possible to define timing constraints, situation - action rules and precipitating events. This model stands in contrast to models used in real time knowledge based systems.

Procedural Reasoning System (PRS) diagnoses malfunctions in process control applications in real time [30]. A PRS agent consists of a database of the system's current beliefs about the world, a set of current goals and a library of plans and procedures that describe actions and tests required for achieving these goals. The PRS inference engine performs reasoning and planning by manipulating this database. At any time goals are established and events occur that change the beliefs in the database. These changes triggers Knowledge Areas (A plan is represented as a graphic network called Knowledge Area). One or more of the triggered KA's are selected and placed on the intention structure (agenda). The inference engine selects one of the KA's and executes one step of the plan represented by the KA. This results in the establishment of new goals and events in the database and completion of an inference cycle. PRS does not guarantee any reactive or response times. It is the responsibility of the programmer to program KA's to ensure any reactive or response time requirements. PRS has been applied in real time applications like mobile robot control, air traffic management, handling malfunctions

in the reaction control systems of NASA's space shuttle and diagnosing control failures in a telecommunication network.

Bounded response time is an important consideration in applying rule based systems to real time applications. **MRL** is a real time production system language designed to facilitate accurate analysis of response times while maintaining the flexibility and expressiveness of traditional production languages like OPS5 [90]. MRL uses Rhyme match algorithm. Rhyme is a hybrid of RETE, TREAT and Match Box algorithms. Two expert systems viz. Orbital manouvering and reaction control systems valve and switch calculation expert system (OMS) and space station integrated situation assessment system(ISA) are coded in MRL and analyzed for bounded response times [91].

PAMELA (Pattern Matching Expert System Language) is developed for coding real time expert systems [1]. PAMELA implemented enhancements to the RETE match algorithm to improve processing speed. PAMELA offers interrupt handling facilities that are so essential to tackle real time problems. It handles interrupts at the end of right hand sides, after specific working memory element modifications and at specific points requested by the user. PAMELA is implemented in a language called CHILL and later ported to C. A parallel version of PAMELA is also implemented.

OPS5 does not provide facilities for temporal reasoning necessary for an real time expert system. So a a restricted time map manager **(TMM)** is coupled with an OPS5 like inference engine [6]. TMM is a point based system and manages only the imprecise or unknown future. All past and known times are represented on a blackboard. TMM is used to perform temporal tests and reason about event sequences.

Asynchronous Production Systems **(APS)** is a result of a innovative execution strategy and shell architecture for rapid integration of asynchronous data in production systems, and the provision of real time response to the same data [79]. The APS architecture is similar to forward reasoning rule based systems like OPS5. The APS inference engine comprises of three asynchronous concurrent modules viz.

match, select and execute. A fourth concurrent module is used to gather external input data. Unlike traditional production systems, APS is able to accept sensory input and integrate it in the inference process. APS requires a MIMD machine with a shared memory, shared bus interrupt driven architecture. Expert systems for the HERMIES robot are implemented using APS. APS can continuously monitor input data but does not provide facilities for temporal reasoning.

Concurrent OPS5 (CROPS5) is a production system language designed for real time applications [63]. CROPS5 applications coexist with other real time tasks on the same computing platform. It is specifically designed to be preemptible and priority driven. In contrast to the single data stream of OPS5, CR0PS5 has multiple concurrent prioritised data streams with disjoint sets of productions. CROPS5 reduces the granularity of interruptability from match processing boundary to token processing boundary. This helps in reducing execution time variance. An aircraft collision avoidance system is implemented using CR0PS5. CR0PS5 however does not provide any facilities for temporal knowledge representation and reasoning.

G2 is a commercially available object oriented real time expert system [13]. It offers an object oriented data definition facility for structured definition of system configuration. It offers an IF - THEN type rule definition scheme. A good window oriented developer's interface is available. A data interface facility (called GSI) is available to connect the expert system to the external world. G2 provides both forward and backward chaining. Due to it's ability of revising previous decision based on new information, G2 has real time capabilities.

CAGE and **POLIGON** are concurrent problem solving frameworks [60] are developed for a class of applications involving real time interpretation of continuous streams of inaccurate data using diverse.sources of knowledge. CAGE uses concurrency at the knowledge source level and its target architecture is a shared memory multiprocessor. POLIGON studies concurrency at the blackboard node level and its target architecture is a distributed memory multiprocessor system. These problem solving architectures are appropriate for applications with large data parallelism.

GEST (General Expert System Tool) is a blackboard tool developed by GTRI

(Georgia Technical Research Institute) [76] which facilitates development of cooperating expert systems. GEST supports both forward and backward chaining. It further provides a blackboard architecture. It is used to develop Pilot aid's for real time decision making.

BLOBS is an object oriented blackboard system framework for reasoning in time [96]. BLOBS is designed for applications which require processing continuous streams of input data and reasoning about geometrical and temporal information. BLOBS attempts to amalgamate the best features in object oriented and blackboard systems. MXA is another blackboard system shell for real time applications [86]. It is developed for tactical picture compilation i.e. producing a representation of the environment surrounding a naval task group. Tactical pictures need to be updated every few seconds using sensor information from diverse sources like radar, ESM and intelligence. This task and its associated processing should be carried out in real time. MXA includes both event driven and goal driven reasoning. A blackboard is used as an internal means of communication between the rules and records the hypothesis reached so far and the way different hypotheses interconnect.

MUSE is a tool kit for embedded real time AI [75]. The typical intended applications are operator assistance, monitoring and fault diagnosis in complex electronic and mechanical systems. The basic structure of a MUSE application is a set of separate reasoning modules which communicate by means of shared access to particular databases. The other important features of MUSE are multiple representation languages (production rules, deductive rules, procedures and frames), object oriented approach and knowledge base compilation.

HCVM is a computational model of reflective control of resource bounded problem solving [19]. The key features of HCVM include interruptability, event driven adaptive control, modularity, information sharing and multitasking control. Schemer is used to build a number of real time applications like diagnosis of malfunctioning processes, automated performance management of advanced avionics systems and heuristic control virtual machine among many others.

RT-1 is a small scale coarse grained distributed architecture for real time applications based on the blackboard paradigm [16]. RT-1 focuses on four categories of software innovations to improve real time performance. They are control reasoning, focus of attention, parallelism and algorithm efficacy. RT-1 is implemented on a Symbolics Lisp machine using Flavors and Common Lisp.

Generic Black Board (GBB) provides an efficient pattern matching and retrieval functions. Work in GBB is directed towards improving performance of complex blackboard transactions. Efficiency of knowledge source execution and control issues are not addressed in GBB and are left to the application developer [16].

HOPES (Hierarchically Organised Parallel Expert System) also proposed for real time applications [10], is structured into knowledge sources organised in a hierarchial fashion. A multi level blackboard is introduced as a communication mechanism between different knowledge sources. The hierarchical structuring of knowledge sources allows HOPES to handle complex hierarchically structured applications like continuous signal interpretation. HOPES is implemented in an object oriented fashion. A radar signal processing and interpretation application is implemented in HOPES. Other important features of HOPES are uncertainty management and time management.

Erasmus is a configurable blackboard development system [2]. The main objective in Erasmus development is reconfigurability, and certain performance compromises are made. The chief feature of the knowledge source structures in Erasmus are interruptability, asynchronous knowledge source execution, phases, preconditions and obviation conditions. Erasmus is used to develop a cockpit information management application [35].

Activation Framework (AF) supports implementation of object oriented real time distributed cooperative problem solving on interconnected computers [26]. It is based on the paradigm of expert object communicating by messages in a community of experts. The issues addressed in AF development are concurrency, uncertainty about future events, resource limitations, reasoning about time, focus of attention, timeliness, modularity and distributability . It is implemented in C language. AF

timeliness, modularity and distributability . It is implemented in C language. AF belongs to a separate genre of problems solving systems that are classified as Distributed problem solving systems or Distributed AI (DAI) systems. AGORA [3], MACE [23] are other similar DAI frameworks.

In recent past, neural net based connectionist expert systems are being proposed as a means of achieving higher speedups. Sohn and Gaudiot present a three layer net architecture to model the inference cycle of production systems [83]. The neurons are connected in a ring structure and feed forward to follow the logical model of the inference. Hybrid Symbolic/Connectionist architecture **(HSC-PS)** is proposed to achieve parallel execution of rules [80]. HSC-PS is based on the network structure of the connectionist expert system tool SC-net. The distinctive feature of HSC-PS is its ability to implement variable binding which is not possible in other net based expert systems. Distributed Connectionist Production System **(DCPS)** consists of five spaces: two clause spaces, WM space, rule space and bind space [83]. Clause spaces are responsible for matching WME's. WME's and rules are represented as patterns of activity of neurons or connection weights between the cells. There is no conflict resolution in DCPS. In the current DCPS model a successful match takes nearly 90 seconds and the number of rules is limited.

2.4 Systems

Many real time expert systems are implemented and reported in literature. We present eleven such systems.

L*STAR [42] (Lockheed Satellite Telemetry Analysis in Real Time) is a monitoring system built to aid the HST (Hubble Space Telescope) operator in performing real time monitoring and analysis of data from HST. The system is divided into three processes viz. data management process, inference process (the expert system) and the I/O process. The data management process gathers the telemetry data, pre processes it and transmits the same to the inference and I/O processes. The inference process uses this data along with its to generate advice for the operator. The advice is passed to the I/O process which gives it to the operator through a user friendly

Tools	Feature Numbers										
	1	2	3	4	5	6	7	8	9	10	11
LES[64]	Yes	Yes	Yes	No	No	Yes	No	PS	Yes	No	No
PRS[30]	Yes	Yes	No	PS	PS	Yes	PS	PS	No	No	No
CROPS5[63]	PS	Yes	No	Yes	PS	No	PS	Yes	No	PS	No
PAMELA[1]			No		Yes	No	PS	PS	No	No	No
MRL[90]	No		No		No	No	No	No	No	No	No
APS[79]	No	PS	No	No	Yes	No	Yes	No	No	No	No
GEST[76]	Yes		No			No			No		No
BLOBS[96]	Yes		No			No			Yes		No
MXA [86]			No			No			No		No
MUSE[75]			No			No			No		No
CAGE[60]	Yes		No			No			No	Yes	No
POLIGON[60]	Yes	Yes	No	Yes	PS	No			No	Yes	No
HOPES[10]	Yes	Yes	No	Yes	PS	No	PS	PS	No	Yes	PS
HCVM[19]	Yes	Yes	No	Yes	Yes	No	PS	PS	No	PS	No
RT-1[16]	Yes	Yes	No	Yes	Yes	No	No	PS	No	PS	No
ERASMUS[2]	Yes	Yes	No	Yes	PS	No	PS	PS	No	PS	No
G2[13]	Yes	Yes	No	Yes	Yes	Yes		PS	No	No	No
REX (proposed tool)	Yes	Yes	Yes	Yes	Yes	Yes	PS	PS	Yes	Yes	No

PS: Partial Support

Table 2.1: Comparison of features of Real Time Expert System Tools

Feature No.	Feature
1	continuous reasoning
2	reactive response behaviour
3	represent and reason about events and their temporal relationships
4	Focus of attention
5	Async. arrival of events
6	Model delayed feedbacks
7	Handle Interrupts
8	Embedded operation
9	Temporal rep. of data
10	Handling multiple events
11	Guaranteed RT response

on the same machine, the processes are implemented on three different machines. The inference process uses forward chaining and is coded in C. The facts are represented as attribute\object\value triples. Archival of data based on time stamping is provided.

Wheels [64] involves monitoring and diagnosing problems of the Hubble Space Telescope's four reaction wheel assemblies. To assess the wheels operating status and its health telemetry data is available. Wheels is developed using LES (Lockheed Expert System shell). Wheels uses forward inference to provide control, while backward inference is used to perform diagnosis.

QES [82] is a real time expert system for quality control of steel products in a mill. QES's knowledge base is categorised intro structural knowledge and behaviourial knowledge. Structured knowledge is an object oriented representation of processing steps, connections, steel products variables and defects. The behaviourial knowledge includes predictions and diagnosis. It is represented as rules. These rules are further classified into

- inspection rules which confirm or deny the predictions made by prediction rules and
- feed-forward and feed-backward rules that send suggestions up and down the processing line.

A prototype of QES is implemented in Lisp and G2 standard interface provides the process interface. Since steel mill operations are of long duration and do not change often. Hence, the real time requirements are not as stringent as in applications like telemetry data analysis.

REACTOR [55] assists nuclear reactor operators in the diagnosis of accidents. It monitors a nuclear reactor facility, detects deviations, determines the seriousness and recommends an appropriate response. The knowledge about diagnosis is called event oriented knowledge and is represented as production rules. The reactor configuration is called function oriented knowledge and is used to aid the diagnosis process.

ESCORT (Expert System for Complex Operations in Real Time) is a model based expert system that deals with the problem of cognitive overload experienced by operators in process plants [62]. ESCORT diagnoses the underlying problem that caused the alarm using process models (both structural and functional) and causal reasoning. It is implemented in KEE on a Xerox 1186. It is reported that ESCORT deals with around 500 analog and 2500 digital signals and can provide advice within one second of an alarm occurrence. **EXTASE** is a situation assessment system [34]. It finds cause of alarm signals occurring on the heating furnace of a distillation column. Extase's knowledge base is designed to explain unit functioning based on known relationships . A causal graph (C-graph) is used to represent permanent process information. To diagnose it builds and validates a search space of hypothesis (a H- graph). The H-graph contains temporary information related to the problem and represents the current status of the reasoning process.

IPCS (Intelligent Process Control System) uses hierarchial fault propagation models, structural and functional models to perform control, monitoring and fault diagnosis in a process control system [61].IPCS is used to build a diagnostic system for a cogenerator plant in Japan. The IPCS reasoning algorithm consists of two parts viz. the fault component identification algorithm and the inter level migration process. The fault component identification algorithm is responsible for identifying a faulty component in the context of a process fault model. The inter level migration process migrates the fault identification algorithm to lower levels of the fault hierarchy and refines the fault diagnosis. IPCS accurately diagnoses single fault occurrences, but fails in accurate diagnosis of non interacting and interacting multiple faults. It is prone to diagnose such faults as single independent faults.

Integrated Operator Advisor (OA) is a prototype knowledge based system built for the operation and safety maintenance of nuclear power plants [5]. Safety threats are organised in a safety function hierarchy. Each node in the hierarchy represents a safety goal that is lost if the corresponding safety function cannot be maintained. OA detects safety threats by monitoring the threat identifying conditions stored in the nodes of the hierarchy. If a match between the conditions and the current sensor values is obtained the corresponding safety threat is established. OA stores safety and operational procedures in an integrated procedure hierarchy and

chooses a relevant safety procedure from this hierarchy for execution. It does not provide any general purpose temporal representation facilities and these need to be explicitly compiled into the procedures. OA is tested on a set of realistic simulated scenarios and a comprehensive testing is being undertaken.

Mimic [17] is another model based approach to fault diagnosis. It maintains a set of candidate models since a given behaviour might be caused by one of the several faults. Each model represents a possible condition of the system including its state and faults. Two tasks in Mimic maintain the model. One is the tracking task, which keeps the model in step with the observations of the physical systems. If a discrepancy between the observed behaviour and the model occurs, the second task called the diagnosis task is invoked. The diagnosis task detects the fault and injects the same in the model, so that predictions will continue to be in step with the observations. Diagnosis in Mimic is assisted by semi- quantitative simulation. Semi-quantitative simulation uses both qualitative and numerical simulation (a combination of QSim and a numerical model). Mimic refutes or confirms a hypothesis by tracking the process system. If it cannot refute/confirm the hypothesis, the tracking set keeps growing. In practical terms it requires an adequate number of well placed sensors. This could be a difficult proposition in large process control systems. If a catastrophe or an unusual combination of faults occur the simulation model will be at loss and cannot respond in real time. Like the earlier model based systems, this approach is ideal for slowly evolving systems but not systems with a fast dynamic response.

REACT (Rapid Expert Assessment to Counter Threats) is developed to aid pilots in determining appropriate threat response strategies during combat situations [76]. It is also being used to monitor the state of aircraft's on board systems in order to off load the pilot. REACT is built around the paradigm of cooperating expert systems in a blackboard environment. Knowledge is organised as one or more frames. Each knowledge source has an associated fact and rule bases. The rules operate on the frames. REACT is developed on GEST (General Expert System Tool) developed by GTRI (Georgia Technical Research Institute).

TRICERO is a signal understanding system in the area of air defense. It consists of three expert systems (ELINT, COMINT and ELINT - COMINT correlation) [93]. The three expert systems in tandem perform analysis and correlation of signals. A blackboard architecture is selected as the means of structuring the solution by the three expert systems. In addition to dealing with the issues of building distributed cooperating expert systems, TRICERO addresses issues like control in blackboards, knowledge application strategies, strategies for dealing with time and methods for dealing with stale data and conclusions.

2.5 Summary

In this chapter, a survey of the different techniques, tools and systems is presented. Though all the techniques and tools surveyed are not necessarily about real time expert systems, they touch upon related aspects of knowledge representation, speed up and problem solving organisations. It can be observed that no system meets all the criteria defined for a real time expert system. Each system focuses on a specific subset of the requirements which are salient to the domain under investigation. In REX (our proposed tool), we provide essential features of real time expert systems(refer Table 2.1). REX is designed to be a continuous reasoning reactive real time expert system. It's characteristic features are its data and knowledge representation scheme capable of handling temporal properties, it's polynomial time match algorithm and asynchronous multiple rule firing model. In the following chapters, we present the detailed design of REX.

Chapter 3

ARCHITECTURE

3.1 Introduction

The most important activity in building an expert system, is the development of a complete and consistent domain knowledge and an efficient reasoning strategy. A formal model which can represent various aspects of building an expert systems, like knowledge representation, its verification and reasoning, is a Petri Net. A formal theory helps in abstracting and representing about a multitude of tasks in a complex activity.

Petri nets are formal models that are simple, yet powerful to represent complex systems with concurrent and interacting components. A systems approach is possible [44]. With Petri nets, it is possible to integrate different aspects of building expert systems, viz. knowledge representation, knowledge base verification and reasoning in a single formal model. The static structure of the Petri Net can be used for modelling knowledge representation and verification aspects. The dynamic aspects of Petri Nets can be used to model the reasoning process in expert systems.

A Petri Net is a bipartite directed graph, with places and transitions as nodes and directed arcs connecting places to transitions and vice versa. A formal definition is in [72]. Petri nets provide the basic formalism to model rule bases and capture all crucial aspects of representation and verification [8, 15, 46, 73].

3.2 The Extended Petri Net Model

The traditional rule models in production systems do not capture real time abstractions like events and temporal relationships. The rule model has been extended to effectively represent knowledge in a real time environment. Four rule types have been defined. They are Autonomous rules, Clock Synchronised rules, Event Spanning rules and Time Spanning rules. The detailed description of the rule types is given in Section 4.3. The elementary Petri Net model cannot model these augmented rule types. An Extended Petri Net model has been defined to model these rule types.

The Extended Petri Net is defined as a nine tuple EPN = <VP, PP, HP, EP, PT, ET, RT, I, O> where

- VP is the set of value places. A *value place(vp)* is a cumulative place [88] containing tokens representing the values of an attribute(parameter).
- PP is the set of premise places. The presence of a 'T' token in a *premise place(pp)* indicates truth of the premise, while a 'F' token indicates otherwise.
- HP is the set of hold places. The *hold place(hp)* represents hold slots in the rules.
- EP is the set of event places. The presence of a token in an *event place(ep)* indicates occurrence of the event.
- PT is the set of premise transitions. A *premise transition(pt)* represents the computation of a premise's truth value.
- ET is the set of event transitions. An *event transition (et)* represents the computation of a event's occurrence.
- RT is the set of rule transitions. A *rule transition(rt)* represents a rule.
- I is the input function which describes the input places of a transition and the input transitions of a place.

- 0 is the output function which describes the output places of a transition and the output transitions of a place.

The Extended Petri Net semantics are summarised in Table 3.1. For example, in the table, the first entry is about the Value Place. The graphic symbol for a *value place* is a circle with a shaded lower chord. The *value place* represents an attribute value. A *value place* can have only a *Rule transition* as its input. It can have either a *Premise Transition* or *Event Transition* as its output transitions. The other entries in the table represent the properties of the remaining places and transitions in an EPN. For a transition enabled by the absence of tokens in its input place, in the graphic representation the input arc is terminated at the transition end with a circle.An attribute can also be updated by an external input, then the corresponding value place will be a source place and has no input transitions. Only value places can be source places in the net. A hold place is a sink place. All transitions in the EPN have associated computational functions. The function associated with a premise transition computes the premise truth value. The function of an event transition determines if the event has occurred. The function associated with a rule transition performs the actions of the rule.

3.2.1 Representing Rules with Extended Petri Nets

The Extended Petri Net model can capture all the rule types denned in REX. For example, consider the following clock synchronised rule C1. The knowledge represented by the rule Cl is: *If the Yaw command of 10° LEFT is issued by the vehicle control system and the yaw feedback of* 10° *LEFT is obtained before the elapse of 2 seconds after the command is issued then it is concluded that the yaw rate in the vehicle is too high* (Section 4.3 gives the detailed semantics of the different rule types). The EPN representation for the rule is given in Figure 3.1.

Cl:

Event: Yaw.and = 10° *LEFT*
Time limit: *2 seconds*
Time operator: *BEFORE*

Place/ Transition	Symbol	Represents	Input type	Output type
VP (value place)	Circle with shaded lower chord	Attribute value	RT	PT/ET
PP (premise place)	Circle	Premise truth value	VP	RT
HP (hold place)	Concentric circle	Hold slot	RT	
EP (event place)	Circle with shaded upper chord	Event	ET	PT
PT (premise transition)	Vertical line	Computation of of truth value	VP	PP
ET (event transition)	Pair of vertical lines	Computation of occurrence	VP	EP
RT (rule transition)	Box	Rule firing	PP	VP/HP

Table 3.1: Semantics of Places and Transitions in EPN

Premise: Yaw.fb = 10° *LEFT*
Hold; [message "yaw rate too high, check up"]

In the graphical representation, the event is represented by the event transition T1. The transition's input place is a value place representing the attribute Yaw.command. It's output place is a event place (P4). The premise is represented by the premise transition T3. The transition's input place is a value place (P3) and output place is a premise place (P6). The rule's time operator is BEFORE. The 'BEFORE' operator inhibits the firing of the rule after the lapse of 2 seconds from the occurrence of the event. This phenomenon is represented by the transition T2. The computational function associated with premise transition T2, counts a time of 2 seconds from the occurrence of the event (occurrence of a token in P4). After the lapse of 2 seconds a 'T' token is placed in P5. A 'F' token is in P5 before the lapse of two seconds. The rule transition 'T4' represents the complete rule. The transition is fired when the premise is satisfied within 2 seconds of the event occurrence. The computational function of T4 represents the rule's actions/hold.

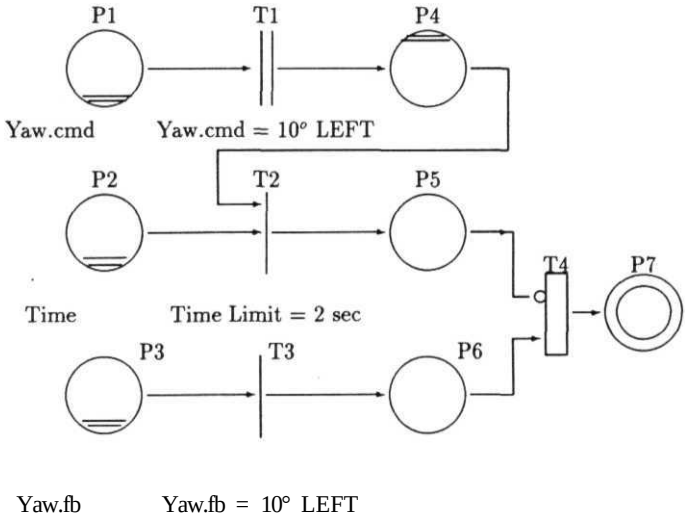


Figure 3.1: EPN Representation for rule Cl

Place id	Place type	I(P)	0(p)	Place semantics

Table 3.2: Structure of Place Property Table

Trans. id	Trans. type	I(t)	0(t)	Trans. semantics	Priority	Rule status

Table 3.3: Structure of Transition Property Table

The graphical EPN representation has been transformed into two tables, suitable for manipulation by programs. The two tables are *Place Property Table(PPT)* and *Transition Property Table(TPT).* The PPT represents the places' properties. The TPT represents the transitions' properties. The dynamics of the Extended Petri Net (transition firings and markings) can be represented as updates in these two tables.

The structure of the Place Property Table is in Table 3.2. The Place-id represents the unique identification number of the place. The Place-type denotes the type of place (VP, PP, EP or HP). Input and output transition lists are the sets of input transitions and output transitions. Place semantics for a value place is the list of values, for a premise place it is a 'T' or 'F' token, for an event place and hold place it is the presence or absence of a token.

The structure of the Transition Property Table is in Table 3.3. The Trans, id is the unique transition identifier. The Trans, type is the transition type. I(t) represents the set of input places. 0(t) represents the set of output places. The transition semantics for a premise and event transition is the associated computational function. The semantics are represented by a <function name-[time]>. We assume that the function denoted by <function name> embodies the computation of the truth value of the premise or event. For transitions representing time counters like T2 in Figure 3.1, the function name is T. The transition semantics of a rule transition is also represented by a computational function which performs the actions of the rule. The semantics of a rule with an output hold place are represented by a generic Hold function. Priority is for rule transitions and indicates rule priority. The rule status indicates whether the rule is not enabled, enabled or fired. An inhibitor arc is represented by a tilde preceding the input place in the TPT. The PPT and TPT

Place id	Place type	I(P)	0(p)	Place semantics
P1	VP	ext. input	Tl	
P2	VP	timer	T2	
P3	VP	ext. input	T3	
P4	EP	Tl	T2	
P5	PP	T2	T4	
P6	PP	T3	T4	
P7	VP	T4	NULL	

Table 3.4: Place Property Table of rule Cl

Trans. id	Trans. type	I(t)	0(t)	Trans. semantics	Priority	Rule status
Tl	ET	P1	P4	Evl-func		
T2	PT	P2,P4	P5	T-2		
T3	PT	P3	P6	Prl-func		
T4	RT	~P5,P6	P7	Hold	1	

Table 3.5: Transition Property Table for rule Cl

tables for the net in Figure 3.1 are given in Tables 3.4 and 3.5 respectively.

3.2.2 Forward Reasoning in Extended Petri Nets

The forward reasoning algorithm in production systems is the repeated execution of the following cycle.

- Match :For each rule, determine if the premises are satisfied.
- Select:Choose a rule whose premises are satisfied.
- Act :Execute the actions of the selected rule.

This reasoning algorithm can be modelled as a traversal of the net by firing enabled transitions along the path. An update to a value place enables a set of premise transitions. When the premise transition is fired, the associated computational function calculates the truth value of the premise and places a 'T' or 'F' token in its output premise place. A rule transition with appropriate tokens in its input premise places is enabled and can be fired. Firing a rule transition invokes the

associated function which performs the actions of the rule by updating the output value or hold places of the rule transition. The updated value place in turn enables some premise transitions and the cycle continues. Thus forward chaining can be easily modelled with Extended Petri Nets.

Consider the rule Cl. In a scenario, let us assume that the Yaw command of 10° LEFT is issued. In this case, the event of rule Cl is satisfied. Now, the transition Tl is fired, and a token is placed in event place P4. Let us assume that immediately, the Yaw feedback becomes 10° LEFT. Now, premise transition T3 is enabled and fired. A 'T' token is placed in premise place P6. Since the feedback became 10° LEFT almost immediately, let us assume that the 2 second limit has not elapsed. So, a T' token is present in premise place P5. Since a tilde is present on the arc to transition T4, this is an enabling condition for transition T4 (the rule premises and temporal relationships are satisfied). Since the transition is enabled, it is fired and a token is placed in place P7. Real time expert systems need to fire multiple rules concurrently, in order to handle simultaneously occurring events in the external world. The asynchronous dynamic nature of the net model allows us to model concurrent multiple rule firing. In Petri nets, transition firing is instantaneous. However, in actual implementations, the process takes finite time, and uninhibited concurrent firing can lead to race conditions. This results in loss of integrity of the places (working memory). So, a mechanism to protect the integrity of the working memory is required. Such a mechanism is termed as interference analysis in multiple rule firing systems. The reasoning cycle with multiple rule firing will be

- Match :For each rule, determine if the premises are satisfied.
- Select :Choose a set of non-conflicting rules whose premises are satisfied.
- Act :Execute the actions of the selected rules concurrently.

3.2.3 Reasoning Algorithm

The *reasoning algorithm* for the multiple rule firing forward reasoning system is given below.

reason()

Execute the following steps repeatedly.

1. Determine the set of Attributes (value places) that are updated.
2. For each of the updated attributes determine the set of premises and events that have to be evaluated (i.e. find the set of output premise and event transitions for each updated value place).
3. For each of the premises determined in the earlier step, evaluate the premises and determine their truth value (i.e. for each premise transition determined in step 2, evaluate the premise function and place either a "I" or 'F' token in the corresponding output premise places).
4. (a) For each of the events determined in step 2, evaluate the events and determine if they have occurred (i.e for each event transition determined in step 2, evaluate the event and place a token in the corresponding output event place, if the event has occurred).
   (b) For each of the events determined to have occurred in step 4(a), find the list of premises to be evaluated (i.e for each event place in which a token is placed in step 4(a), find the set of output premise transitions).
   (c) For each of the premises determined in step 4(b), evaluate the premises and determine their truth value (i.e. for each premise transition determined in step 4(b), evaluate the transition function and place either a 'T' or 'F' token in the corresponding output premise place).
5. For the premises determined to be true in steps 3 and 4, find the set of rules that are likely to have matched (i.e. for each premise place updated in steps 3 and 4, find the set of output rule transitions).
6. For the rules determined in step 5, find the set of rules, all of whose premises are true (i.e find the set of enabled rule transitions in the net).

7. From the set **of** matched rules determined in step 6, find the set of rules that can be fired while maintaining working memory integrity (i.e. perform interference analysis).

8. Perform the actions of the rules determined in step 7 (i.e. fire the enabled rule transitions and update the corresponding output value and hold places).

}

Steps 1 to 6 of the above algorithm correspond to the Match phase of the forward reasoning inference cycle. Step 7 is the select (interference analysis) phase. Step 8 is the Act phase.

3.3 REX's Architecture

REX is an expert system shell architecture for building real time expert systems reasoning with continuous streams of input data [67, 70]. REX has been designed to meet the requirements discussed in Section 1.2. Data in real time domains is multi faceted and has many properties. Simple data structures cannot capture the semantics of real time data. Further, the expert systems have to reason with historic data. This requires a structured and efficient data management scheme, which are readily available in an object oriented model with its facilities for data abstraction, encapsulation and inheritance. Further, the augmented rules in REX can also be modelled in an object oriented fashion, leading to a uniform treatment of both data and knowledge. Hence, we have used an object oriented data and knowledge model in REX. The REX architecture is designed based on the Extended Petri Net model and the reasoning algorithm presented in the earlier section. The REX architecture is depicted in Figure 3.2. It consists of

- *an external data interface*
- *an object manager which manages the associated data and knowledge store*
- *a common **work** area consisting of*

- *object instances*
- *an Attribute Event Premise (AEP) Index Table*
- *an Attribute Table*
- *a Premise Table*
- *an Event Table and*
- *a Rule Table*

• *a reasoning subsystem consisting of*

- *an acquisition module*
- *an evaluation modules*
- *and a scheduling module*

3.4 External Data Interface

The *External Data Interface* consists of two tasks. They are

1. *External Data Acquisition Task (EDAT)*
2. *Attribute Update Task (AUT)*

External Data Acquisition Task

The *External Data Acquisition Task (EDAT)* accepts preprocessed sensor data from the external world in continuous streams of data packets. The data packet format though varying from application to application has a generic structure shown in Figure 3.3.

The data packet has a fixed format. Each data packet has a time reference, which indicates the time at which the data values in the packet are sensed in the external world. The position of the datum identifies the corresponding attribute. The EDAT will obtain these data packets and store them in the buffer. The *Attribute Update Task* would do further processing on this buffer.

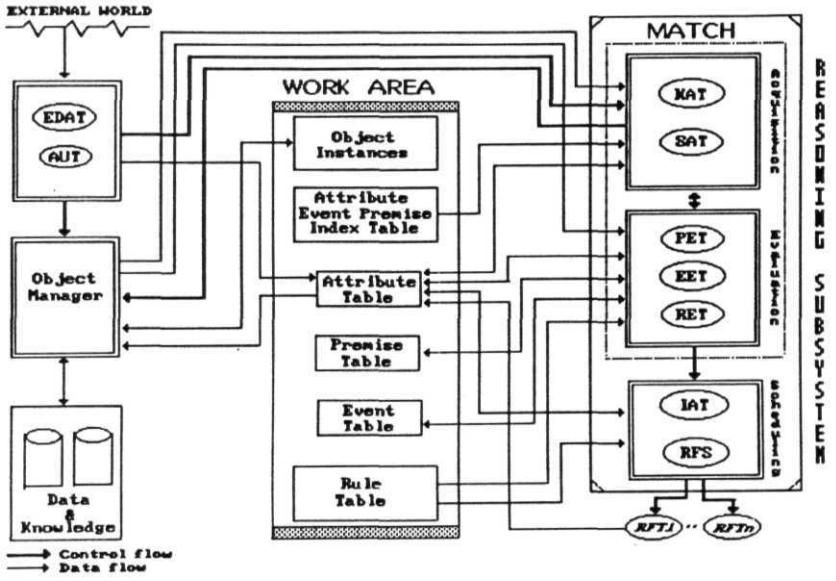


Figure 3.2 : ARCHITECTURE OF REX

EDAT	External Data Acquisition Task
AUT	Attribute Update Task
KAT	Knowledge Acquisition Task
SAT	State Acquisition Task
PET	Premise Evaluation Task
EET	Event Evaluation Task
RET	Rule Evaluation Task
IAT	Interference Analysis Task
RFS	Rule Firing Scheduler
RFT	Rule Firing Task

time reference	attribute value	attribute value		attribute value

Figure 3.3: Generic structure of a data packet

Attribute Update Task

The *Attribute Update Task (AUT)* processes the data placed in the buffer by EDAT. It extracts the data values from the packet and posts them in the Attribute Table of the Work Area. The AUT will then invoke the Object Manager. The Object Manager will form object instances by using the updated Attribute Table. The newly created object instances will be placed in the Object Instances space of the Work Area. The *Reasoning Subsystem* is then triggered. It uses the Attribute Table and Object Instances space in the reasoning process. Thus external world data is available for the *Reasoning Subsystem* through the *External Data Interface.*

The objective of dividing the job of external data interface into EDAT and AUT is to avoid loss of external data. The Attribute Table is a common data area used by the external data interface, object manager and the reasoning subsystem. Hence , synchronisation of access to the Attribute table is necessary. If the external data interface is implemented as a single task, there is a possibility that while waiting for access to the Attribute table, external data arrives and is lost. The EDAT and AUT are organised using the producer- consumer paradigm. The EDAT produces data from the external world and the AUT consumes the same by updating the Attribute Table.

3.5 Object Manager

An unified object oriented paradigm is used to store data and knowledge in REX. The conventional rule structure is augmented to represent knowledge in REX. A taxonomy of rule classes(types) with inheritance relationships has been defined in REX (detailed description is in Section 4.3). All rules in the rule base are instances of one of the rule classes. The *Object Manger* in addition to it data management tasks, is responsible for the rule base management. The *Object Manager* performs

the following tasks

- Schema Evolution
- Creation and Management of Object Instances
- Storage of Object Instances
- Support for predefined methods on Objects
- Work Area Initialisation

The *Object Manager* has to perform the above tasks both for data and rule objects. However, there is no schema evolution for rule objects as its schema is fixed. The *Object Manager* has to provide the necessary functions to create, edit and store rule objects. The *Manager* in addition will provide facilities for rule compilation and integration of the generated code with the *reasoning subsystem.* The *Object Manager* will also retrieve the rule objects during work area initialisation and build the Premise, Event and Rule tables.

3.5.1 Schema Evolution

A schema in an object oriented data model consists of a set of classes bound by a certain hierarchial relationships. The evolution of the schema is through the definition of classes and the hierarchial relationships among the classes. The *Object Manager* provides functions for defining a class, its attributes, its methods, default values, legal values, super classes and subclasses. The entire object schema is defined using these functions.

3.5.2 Creation and Management of Object Instances

The *Object Manager* a unique identifier called the Object Identifier (OID) for each object, at the time of creation. All references to the object are made using the object identifier. The necessary function for creation of new instances is provided in

the *Object Manager*. Further, all objects belonging to **a** particular class are stored in a doubly linked list in the **Object Instances** space. The *Object Manger* handles this list and provides the instances needed to the *Reasoning Subsystem.*

3.5.3 Retrieval of Object Instances

Object instances are usually available in the **Object Instances** space of the Work Area. However, due to space constraints and low frequency of use, some instances may be shifted to the secondary store. The *Object Manger* decides when an object should be shifted to secondary store. So, when required it has to retrieve these instances from the secondary store to the Work Area. Objects are stored in two different formats, one for the main memory and another for the secondary storage. Hence retrieval of objects involves format conversion. The *Object Manager* provides the necessary functions to implement this task.

3.5.4 Secondary Storage Management

The object instances are shifted to the secondary store when they exceed the space constraints of the Work Area. The secondary store also provides the persistency of object instances. It has been built using a B+ indexed file structure. The index is maintained on the Object Identifiers. The *Object Manager* provides the functions for maintaining the secondary store and index. A similar secondary storage for rule objects is also to be managed by the *Object Manager*

3.5.5 Support for Predefined Methods on Objects

A set of methods have been defined for all classes in the taxonomy. These methods perform tasks like attribute update, attribute value retrieval and object display. The *Object Manager* supports these functions.

3.5.6 Work Area Initialisation

The Work Area will consist of all state information and knowledge necessary for the *Reasoning Subsystem.* The *Object Manger* has to initialise this area during system startup phase. The *Object Manger* will build the Attribute Table from the class taxonomy. Similarly, from the rule base, the *Object Manager* will build the Premise Table, Event Table and Rule Table in the Work Area. The Attribute - Event - Premise Index Table is built next. After the Work Area has been initialised by the *Object Manager,* the reasoning process can be initiated.

3.6 Work Area

The Work Area is more than the traditional working memory. It maintains all state information and knowledge indices. It is designed to help build an efficient reasoning subsystem. The Work Area consists of

- **Object Instances Space**
- **Attribute Table**
- **Attribute Event Premise Index Table**
- **Premise Table**
- **Event Table and**
- **Rule Table**

The Place Property Table (PPT) and Transition Property Tables (TPT) of the Extended Petri Net are divided into four different tables viz. Attribute Table, Premise Table, Event Table and Rule Table to design an efficient reasoning system.

3.6.1 Object Instances Space

The Object Instances Space of the Work Area consists of recent object instances in a temporal order. Instances of all classes defined in the object taxonomy of the expert system are stored in this space and are used in evaluating Spanning premises (refer Section 4.3 for details about Spanning Premises).

3.6.2 Attribute Table

The Attribute Table is a repository of current data about all object attributes defined in the object taxonomy. In addition to storing the attribute's current value, it stores other information like valid life span and update flag which is useful for the *Reasoning Subsystem*. Chapter 5 presents the structure of the Attribute Table and its use. The Attribute Table can be viewed as a sort of cache storage for the *Match Process*. The *Acquisition Module* can retrieve all current object attribute values from the Attribute Table without accessing the Object Instances Space through the *Object Manager*.

3.6.3 Attribute Event Premise Index Table

The *Attribute Event Premise (AEP) Index Table* is a static table used by the *Acquisition Module*. The *Match Process* in REX is an incremental process. In every inference cycle, it is sufficient to evaluate only those premises/events, whose truth values/occurrence flags could have changed due to attribute updates. The AEP Index Table maintains, for every attribute the list of premises and events in which the attribute participates. This table is used by KAT of the *Acquisition Module* to determine the premises and events to be evaluated.

3.6.4 Premise Table

The Premise Table contains information about all the premises present in the rule base. It stores information such as truth value and any event dependency(like in a

clock synchronised rule). The truth values are updated by the PET and used by the RET to form the MRS. Details about the table structure and use are in Section **5.3.**

3.6.5 Event Table

The Event Table contains information about the events defined in the rule base. Information such as time limit, time operator (both defined for clock synchronised rules in Chapter 4), event occurrence flag are stored in this table. It also stores the list of premises to be evaluated if the event occurs. This table is used by the *Event Evaluation Task* of the *Evaluation Module.* Details about the table structure and use are in Section 5.3.

3.6.6 Rule Table

The Rule Table contains information about the type of rule (refer Section 4.3), its premises, actions and priority. The information in Rule Table is used by the *Rule Evaluation Task* to form the Matched Rule Set. The *Rule Firing Scheduler* uses this table to retrieve the action codes of the rules and schedule their execution. This table like the AEP Index Table is a static table and is not updated during run time. Details about the table structure and use are in Section 5.3.

3.7 Reasoning Subsystem

Real time expert systems monitoring continuous streams of input data are reactive in nature. So, the reasoning process is data driven. Hence a forward chaining problem solving paradigm is ideal for such applications. The REX's *Reasoning Subsystem* uses the forward chaining with multiple rule firing model to handle simultaneously occurring events in the external world. The *Reasoning Subsystem* executes the following cycle repeatedly.

- Match :For each rule, determine if the premises are satisfied.

- Select :Choose a set of non-conflicting rules whose premises are satisfied.
- Act : Execute the actions of the selected rules concurrently.

The Match phase of the cycle is performed by the *Acquisition* and *Evaluation Modules.* The *Scheduling Module* implements the select and act phases.

3.7.1 Acquisition Module

The *Acquisition Module* is responsible for determining changes in the current system state and the relevant knowledge to be applied. The module consists of two tasks,

- *State Acquisition Task (SAT)* and
- *Knowledge Acquisition Task (KAT).*

State Acquisition Task

The Match process in REX is an incremental process. The process is shown in Figure 3.4. The set of satisfied rules is determined by using the updates to the Attribute Table as the seeds (i.e. initiation points for determining the rules). The *State Acquisition Task (SAT)* has to determine the changes made to the Attribute Table either by the *Attribute Update Task or Rule Firing Tasks.*

Knowledge Acquisition Task

Since, the REX reasoning system is data driven, the current input data external world determines the relevant knowledge to be applied. The *Knowledge Acquisition Task (KAT)* determines this knowledge by using the state information captured by the SAT and the Attribute Event Premise (AEP) Index Table.

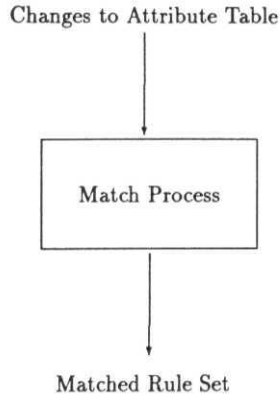


Figure 3.4: Incremental Match

3.7.2 Evaluation Module

The primary function of the *Evaluation Module* is to determine the set of matched rules in the current context. The module interacts with the Acquisition Module during the course of state evaluation. It is composed of three tasks. They are

- *Premise Evaluation Task (PET)*
- *Event Evaluation Task (EET)* and
- *Rule Evaluation Task (RET).*

Premise Evaluation Task

Knowledge in REX is modelled using augmented Premise-Action rules (refer Section 4.3 for details about the knowledge representation scheme). The premises are evaluated by the *Premise Evaluation Task* using the contents of the Attribute Table to determine the applicability of rules. The KAT of the *Acquisition Module* determines the list of premises to be evaluated. The PET executes the corresponding premise evaluation functions to determine the set of satisfied premises (i.e. the set of premises which values are true). REX employs the strategy of executing compiled code rather than interpreting premises. This is done to speed up the evaluation process.

A real time reasoning system, should possess adequate facilities for representing knowledge about system trends, representing events and their temporal relationships. A simple and generic Premise-Action knowledge representation facility cannot capture complex abstractions in real time environments. In REX, two augmented rule representation facilities termed as Clock Synchronised rules and Spanning rules and are provided (details are presented in Section 4.3) for this purpose.

Spanning rules are event based, and they can system trends. The evaluation of spanning premises in these rules is based on the calculation of historical trends in the external environment. The Attribute Table contains only values representing the current state. System state at earlier points of time (i.e. historical data) are

available as object instances in the Object Instances space. The *State Acquisition Task* interacts with the *Object Manager* to retrieve the necessary instances and passes the same to the PET.

Event Evaluation Task

Clock synchronised rules represent event based knowledge. The evaluation of premises in these rules is triggered by the occurrence of a predefined event. The *Event Evaluation Task* determines event occurrences by evaluating events in a manner similar to premise evaluation by PET. For all events, that have occurred, the EET finds out from the Event Table, the set of premises that must be evaluated (these premises are in those clock synchronised rules and spanning rules that are to be triggered in the current context). This list of premises is passed to the PET, which determines the truth value of these premises.

Rule Evaluation Task

The *Rule Evaluation Task* determines the set of matched rules (rule instantiations) and forms the Matched Rule Set. It uses the Rule Table and Premise Table coupled with the set of satisfied premises to form the Matched Rule Set (conflict set in OPS5 terminology) which is then passed to the *Scheduling Module* for further processing.

3.7.3 Scheduling Module

The *Scheduling Module* is responsible for the creation of rule firing tasks and scheduling their execution. As explained earlier, REX adopts an asynchronous multiple rule firing model to enable reasoning about multiple events concurrently. The *Scheduling Module* consists of two tasks,

- *Interference Analysis Task (IAT)* and
- *Rule Firing Scheduler (RFS).*

Interference Analysis Task

Taking a simplistic view, the *Scheduling Module* just needs to create tasks for each of the rule firings and put them on the schedule queue for execution.(There is no need to interpret the rule actions (consequents), as they are also compiled into execution code similar to the premises). However, the fundamental issue involved in multiple rule firing models, is work area consistency. For example, assume that the following two rules are present in the Matched Rule Set. If the *Scheduling Module* simply creates two tasks for the execution of the rule actions concurrently, it leads to inconsistency in the Attribute Table and consequently in further reasoning.

R1:
Premise : $A.a_1 < 30$; $B.b_2 > 40$
Action : $D.d_1 = 100$

R2:
Premise : $D.d_1 < 50$
Action : $A.a_1 = 100$

This inconsistency arises because the actions of one rule invalidate the premises of another rule. If both the rules are fired (action execution) concurrently then there is no equivalent serial execution path. This is termed as *interference* [31] in multiple rule firing models. In order to ensure consistency of the Work Area, the *Scheduling Module* must detect such rules and inhibit them from firing concurrently. This task is known as *interference analysis.* The *Interference Analysis Task (IAT)* performs this function. IAT performs interference analysis on the Matched Rule Set generated by the *Evaluation Module,* and it generates as output a set of rules called the *Eligible Rule Set (ERS).* The rules in ERS possess the property of maintaining Work Area consistency even if fired concurrently.

The task of interference analysis requires comparing the actions of every rule with the premises of every other rule. This results in combinatorial explosion, unless,the number of combinations to be tested is pruned using suitable techniques. Different

techniques and approaches to perform interference analysis are proposed [31, 33, 40, 56, 57, 81], some of which are surveyed in Chapter 2. These techniques require both off-line (compile time) and run time analysis of the rules and are space and compute intense. The trade off usually applied in these techniques is between their execution time and the degree of concurrency. In REX, a new approach to interference analysis is designed and implemented. This technique requires little compile time analysis and is less compute intense than most other techniques. The IAT implements this algorithm and forms the ERS. The design details are presented in Chapter 6.

Rule Firing Scheduler

The *Rule Firing Scheduler (RFS)* creates separate tasks for each of the rules in ERS. The tasks execute asynchronously and make updates to the Attribute Table. The successful completion of all the rule firing tasks signals the completion of a single inference cycle and starts a new cycle. Since, the different rule firing tasks execute asynchronously, without waiting for another rule firing task, we terms our REX as an *asynchronous production system.*

3.8 Rule Firing Tasks

The *Rule Firing Tasks (RFT's)* execute the actions of rules present in the ERS. The RFT's update the Attribute Table. These tasks are shown as being outside the *Reasoning Subsystem* in Figure 3.2. This is because they are a consequence of the *Reasoning Subsystem* performing its function, rather than a part of the subsystem itself.

The *Rule Firing Tasks* on completion should trigger the *Acquisition Module* of the *Reasoning Subsystem* to start the next inference cycle. Similarly, the *Object Manager* should be asked to form new object instances based on the attribute value updates made by the rule firings. If every firing task, triggers the *Acquisition Module* and the *Object Manager*, there would be a plethora of task copies of *Object Manager* and *Acquisition Module* vying to run. Multiple match processes lead to

race conditions and inconsistent results. The *Match Process* should start after all rule firings are complete. In order to ensure this, all RFT's execute a *Set_and_Test* function before triggering the *Acquisition Module.* The *Rule Firing Scheduler* of the *Scheduling Module* while creating the rule firing tasks set the variable RFT_number to the number of tasks being created. The *Set_and_Test* function would use this variable. The pseudo code of this function is

```
Set_and_Test() {
    RFT_number - -;
    if(RFT_number == 0)
        trigger Acquisition Module and Object Manager
}
```

The execution of *Set_and_Test* function by all the RFT's will ensure that the *Match Process* is triggered only after all rule firings are complete.

3.9 Other Factors

The Attribute Table is a common memory area which is referenced by most process. The AUT and the rule firing tasks update the Attribute Table, while all processes reference it. The correctness of the MRS formed by the *Match Process* depends on the integrity of the Attribute Table. Since, multiple processes can update the Attribute Table concurrently, measures to ensure its integrity are necessary. The Attribute Table should not be updated while the *Reasoning Subsystem* is referencing it. A locking scheme is employed to synchronise access by the *External Data Interface* and the *Reasoning Subsystem* to the Attribute Table. This synchronisation is a primary reason for bifurcating the *External Data Interface* into two separate tasks, one for obtaining external world data (EDAT) and another for update the Attribute Table (AUT). The *External Data Interface* is allowed to updating the Attribute Table while rule firings take place. This is because the set of attributes updated by the *External Data Interface* and *Rule Firing Tasks* will be disjoint (rule firings cannot update sensor data).

3.10 Summary

In this chapter, an Extended Petri Net model has been defined. This EPN model can capture the representation , reasoning and verification aspects in real time systems. The REX architecture based on this EPN model is presented. The different tasks and memory structures in REX were explained using a schematic diagram of the architecture. The data and knowledge representation scheme is the subject of the next chapter.

Chapter 4

DATA AND KNOWLEDGE REPRESENTATION SCHEME

4.1 Introduction

Knowledge representation deals with the structures used to represent the knowledge provided by the domain expert or experts [28]. Efficient knowledge representation is the key to overall success of expert systems. Although there is no single structure to represent knowledge in the most effective manner, a number of different approaches are suitable based on the nature of the problem. Some problems may require the use of more than one knowledge representation structure. The commonly used knowledge representation structures are

- production rules
- frame based schemes
- logic schemes.

Production rules are the most commonly used representation for encoding domain knowledge. However, they are inadequate for defining domain objects and static relationships among these objects [21]. Object oriented representations or frame systems can represent such domain knowledge effectively. The class taxonomies in object oriented representations provide the knowledge engineer with a powerful tool for structuring the domain descriptions and integrating the different object types into a unified coherent system model. Thus object oriented representation handles effectively those areas of knowledge representation where production rules are inadequate. The integration of object oriented representations with the production rules to form an hybrid representation can successfully meet the knowledge representation requirements in many domains.

Time plays a crucial role in real time applications. For example, the monitoring system of an aircraft jet engine must keep changes in parameters like thrust, air flow, air temperature to keep track of the engine performance. These changing quantities must be recorded to answer queries about system health, performance degradation and make projections about factors like reliability. Hence it is necessary to preserve data over a period of time to answer such historical queries. Further, as physical systems are slow changing entities, the historical data can be used to produce early warnings based on historical trends. Recently, there is a growing interest in incorporating the time dimension into the data models. Most of this work is based on the relational model. Time is added as a special attribute of the relation. Some models assume non first normal form relations and use tuples to record attribute history. However, it is widely recognised that the relational model is inadequate to capture the semantics of complex entities that occur in real time domains [95]. Object oriented data models with their properties data encapsulation, inheritance and complex attributes can capture the semantics of complex entities. However, these models are static and offer only a snapshot view of the world. Suitable attributes and facets have to be defined to incorporate the time dimension in object oriented data models.

The next section discusses an object oriented data model which incorporates the time dimension. In Section 4.3, an object oriented augmented rule structure and taxonomy has been defined. Section 4.4 presents the criteria for verification of the rule bases. The schemes for rule base verification using the Extended Petri Net model have also been presented in Section 4.5.

4.2 Object Structure and Management

The factors to be considered in choosing an appropriate time representation are [47]

- Primitive time entity: This can be either an instance representation (time point) or an interval representation.
- Time ordering: Linear ordering or a branching time or circular time

- Specification of time structure: Time can be mapped to fundamental types like integers, real numbers or derived types like calender day
- Time boundedness: In point representation, this refers to the time span of the problem and in interval representation it refers to open or closed intervals
- Time metric: The metric for measuring the distance between two time points or intervals

Continuous streams of input data imply that the data values are instantaneous values observed in the domain. The point representation of time leads to a correct representation of data values. Hence the point representation is chosen. The continuous arrival of input data also presents an implicit ordering of all data values available to the system. So, a linear ordering is chosen. Since the rate of arrival of data depends on the combination of response times of the sensor and the monitoring systems, it is difficult to fix points in time at which data will arrive into the system. So, the time points are mapped to the set of real numbers. This enables to record data arriving at arbitrary time points. Further, it is not possible to put a time limit after which data will cease to arrive. Hence, there is no bound on the time points. Every time point will have a predecessor and successor. The time is in seconds, which is the most common metric used in most real time systems.

This ontology is quite suitable for the class of applications targeted at. But it could be unsuitable for a different class of applications where the interval representation of time is appropriate. Object oriented models are extendable. Intervals and associated semantics can be easily defined as specialisation of the basic time point model.

Data obtained in real time systems possesses two distinctive features. They are *time at which the data are obtained and the life span of the data i.e. timeout.* Every data value in the environment has a birth and a life span. For example in a dynamically changing system the fact that a certain pressure is 300 Ksc has little or no meaning. But the fact that the pressure at time 500 sec is 300 Ksc is meaningful and has immense value to the agent. For example if the timeout (i.e. life span) of the pressure is given as 10 sec, it means that the pressure value 300 Ksc has

no meaning after 510 sec unless it is updated by the sensor. So it is essential to represent the time of obtaining a data value in addition to the value. Thus data in a real time system is a triple <*v,t,to*> , where *v* is the data value and *t* is the time at which the value is obtained and *to* is the timeout of the data item. Object oriented models do not provide for explicit representation of these properties of data . So an additional facet called *timeout* is defined for attributes in the classes.

In REX, a base class *OBJECT* has been defined. All object classes are derived from the base class *OBJECT*. It provides the clock service which is so essential to the real time system. In addition to clock service, *OBJECT* provides generic function implementation for access to members of the classes. The structure of *OBJECT* is

```
class OBJECT {
    time;
}
```

A class named **Pressure** with attributes *line-pressure* and *accumulator-pressure* whose timeouts are 2 sec and 4 sec respectively the class definition would look like

```
class Pressure {
    time;
    line_pressure;
    time@line_pressure;
    accumulator_pressure;
    time@accumulator_pressure;
}
```

Two new parameters are created when the class **Pressure** is defined viz. *time@line_pressure* and *time@accumulator_pressure*. The parameter *time@linc_pressure* indicates the valid life time of the attribute *line_pressure*. It is the time after which the value of *line_pressure* is no longer valid. Another policy adopted by REX, and similar to many OODB implementations is that any update to the attributes results in the creation of a new instance. All references to objects of a given class refer to the latest instance created unless explicitly specified otherwise. This is because in a real

time reasoning system a new data value from the sensor invalidates the old data value. However for calculating trends, earlier values are also available.

The update behaviour of objects is brought out in the following example. Let there be an object of class Pressure with object-id *1* at time 0. If the attribute *line-pressure* is updated time *1* to value 450 a new object with a different object identifier *2* is created. The two objects would be as shown below.

```
object-id :1 {
        time : 0
        line_pressure : 200
        time@line_pressure : 2
        accumulator_pressure : 5000
        time@accumulator_pressure : 4
}
object-id :2 {
        time : 1
        line_pressure : 450
        time@line_pressure : 3
        accumulator pressure : 5000
        time@accumulator_pressure : 4
}
```

It is observed that the parameter time@accumulator_pressure remains unchanged since the value of accumulator_pressure is obtained at time *0*. Any references to the object of class Pressure are directed to the object with object-id 2. Whenever the value for accumulator_pressure is requested the value 5000 is returned only if

$$\underline{currenttime} \leq \text{Pressure.time@accumulator\_pressure.}$$

In this way retrieval is allowed only for attributes which still have a valid life time.

A limited number of instances of all classes are maintained in the object Instances space for access by rules. If the number of instances created crosses this limit the *Object Manager* shifts old instances to secondary store. Old object instances are

temporarily brought to *Object Instances space*, if required for trend calculations. Lockheed Expert system shell (LES) [64] provides for storage of temporally varying attribute values within its frame structure. However LES provides for storing only a fixed number of data values.

4.3 Augmented Rule Structure and Taxonomy

Real time knowledge based process monitoring system tracks process variables intelligently so as to detect faults and perform malfunction diagnosis. A majority of the process applications are characterised by large data sets and equally large malfunction hypotheses space.

Usually operators are provided with manuals which define alarms and procedures to isolate and repair faults. These manuals are organised on the basis of symptom - cause relationships and represent the compiled knowledge about the system behaviour. Real time knowledge based systems should make use of this compiled knowledge. Structured representation of knowledge becomes sine qua non for efficient search in a large domain knowledge space. Some of the well known structured techniques are frames, rules, causal models and case based indexing systems. Rules offer an easy and efficient way of representing symptom-cause relationships and associated compiled knowledge. Hence production rules are chosen as the knowledge representation structure.

The domain knowledge in temporal systems has features not present in knowledge of traditional application domains. They are

- knowledge about temporal relationships of events and their consequences
- knowledge, based on trends established in the system dynamics.

The traditional knowledge representation formalisms like rules, semantic nets and frames support representation of knowledge about static snapshots of the domain. They do not provide for representing knowledge that takes into account the dynamic nature of the domain (like temporal spacing of events and event sequences).

A knowledge representation scheme is a surrogate for the real world entities it purports to represent [11]. Further it provides a framework for modelling the problem domain. Inadequate representations result in loss of fidelity of the model. The significant consequence being the errors in conclusions arrived by the reasoning process. This imperfection must be reduced to the extent possible by suitably augmenting the representation scheme. The current rule systems lack facilities to represent the semantics of the application and temporal information. This is also true for other KR schemes like frames and semantics nets. In REX, the rule structure is augmented to represent knowledge about temporal properties and behaviour. Further in consonance with the aim of building an object oriented knowledge based system and treat all entities as objects, rules are treated as objects derived from the base class *OBJECT* with a specialisation hierarchy. The rule taxonomy is shown in Figure 4.1 according to the notation given by G. Booch [4]. There are three types of rules. They are *Autonomous rules, Clock synchronised rules and Spanning rules.* The *clock synchronised rules and spanning rules* are specialisations of the *autonomous rules. Spanning rules* are further categorised as *Event spanning rules and Time spanning rules.*

4.3.1 Autonomous Rules

Autonomous rules are generic rules from which all other rule types are derived. Firing of *autonomous rules* is not attached to any trigger events or enabling conditions, hence the name. If the work area contents match the premises of *autonomous rule,* the rule is scheduled for firing. The structure of an *Autonomous rule* is

Priority

Premises

Actions

Hold

The priority of the rule indicates the importance of the rule in the total rule set. This information is used for scheduling rule firings. The premises are a conjunction

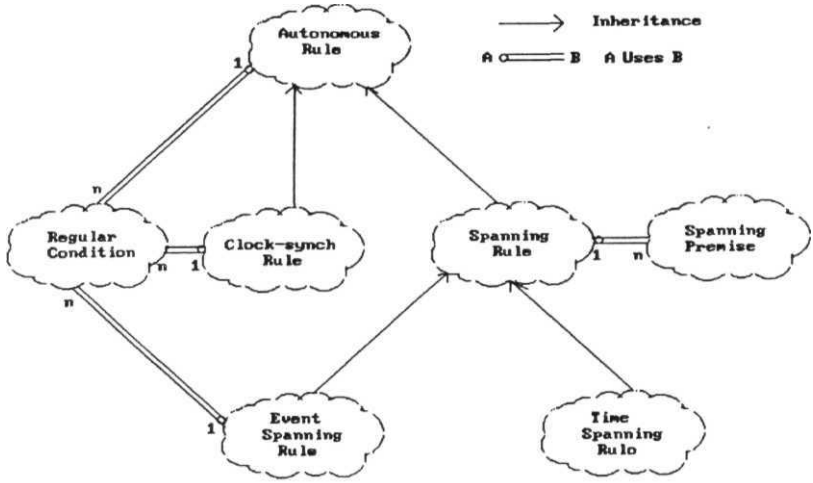


Figure 4.1: Rule Taxonomy in REX

of a set of conditions. The satisfaction of the premises makes the rule eligible for firing. The actions are the consequents of the rule. Actions assert new beliefs about the system state and these assertions are entered in the work area. An example rule is,

Rule A1:

Premise: Pressure.accumulator_pressure > 1000 Ksc

Action: Hydraulic_system. status = OK.

Real time monitoring systems track the process variables to ascertain the normal behaviour of the system under investigation and carry out fault detection by differentiating between normal and abnormal conditions. The Hold slot in the rule structure asserts a fault state of the system under investigation. Hold is an action which asserts a system malfunction and terminate a reasoning thread. Rules with non empty Hold slots have empty Action slots and vice-versa. Usually, a rule with a non empty Hold slot terminates a reasoning thread by establishing a fault state of the system. An example of a rule with a non empty Hold slot is

Rule A2:

Premise: Pressure.line_pressure < 100 Ksc and Yaw_fb > 15°

Hold: [message "System_controllability is negative."]

4.3.2 Clock Synchronised Rules

Physical systems do not react instantaneously to inputs but do so with a certain time lag. For example in an aerospace vehicle the reaction of the fins to the auto pilot control is not immediate but occurs after a definite time lag. The auto pilot of the vehicle, will expect the fins to follow the commands with this time lag. If the expected action of the fins fails to take place, a warning is signalled to the ground control immediately for remedial action. Systems monitoring such physical processes should conduct their investigation of the process by anticipating these time delays. *Autonomous rules* clearly do not cater to these requirements. We propose a specialisation of *autonomous rules* called *clock synchronised rules* to model such

Semantics of AFTER operator		
Time	Match Conditions	
	Met	Not met
Delay over	Fire rule	Ignore
Delay not over	Don't care	

Table 4.1: Semantics of AFTER operator in Clock Synchronised Rules

physical phenomena. The structure of the *clock synchronised rules* is

Event

Time limit

Time operator

Priority

Premises

Actions

Hold

Clock synchronised rules have three additional attributes They are Event, Time limit and Time operator. The Event is a trigger condition which enables the match and firing of the rule based on the value of Time operator. The Time operator can take as values either AFTER or BEFORE.

If the operator is AFTER, the match of the premises is initiated after an amount of time equal to Time limit has elapsed from the occurrence of the Event. The semantics of the AFTER operator are summarized in the Table 4.1.

The type of knowledge that can be represented with a clock synchronised rule using the AFTER operator is given below.

Knowledge : If the pressure valve is *opened and even after 5 seconds the* pressure *is less than 40 Ksc then it can be concluded that the fluid level in the tank is low and pump motor should be switched on.*

Semantics of BEFORE operator		
Time	Match Conditions	
	Met | Not met	
Delay over	Don't care	
Delay not over	Fire rule	Try again

Table 4.2: Semantics of the BEFORE operator in Clock Synchronised Rules

Rule:

Event: Hydraulic_system.Pressure_valve = *OPEN*

Time limit: 5 *seconds*

Time operator: *"AFTER"*

Premise: Hydraulic_system.Pressure < *40 Ksc*

Hold: [message " fluid level low - switch pump motor"]

In physical systems, some events are expected to happen at a given time and at a given pace. For example if an auto pilot issues a yaw command, the vehicle is expected to achieve the desired turn at a desired rate and by the end of a desired period. If the yaw rate is achieved too quickly it may induce undesirable body accelerations. These situations are modelled using the BEFORE operator. The situation is depicted in the Table 4.2.

The following is the knowledge about an aerospace vehicle's responses to it's auto pilot's commands. If the auto pilot has issued a Yaw command, the Yaw feedback is to become equal to the command. In an hypothetical vehicle, let us assume that the Yaw feedback has to become equal to Yaw command 2 seconds after the command is issued but before 6 seconds. This can be represented with two clock synchronised rules, one with the BEFORE operator and another with the AFTER operator. The rules are

Cl:

Event: Yaw.and == +10°

Time limit: *2 seconds*

Time operator; *"BEFORE"*
Premise: Yaw.fb == -10°
Action: Yaw.Controlstatus = Fast

C2:
Event: Yaw.and == +10°
Time limit: *6 seconds*
Time operator: *"AFTER"*
Premise: Yaw.fb ≠ -10°
Hold; Yav.Controlstatus = Lag

4.3.3 Spanning Rules

Early warnings based on historical trends are a necessary feature in reactive systems as preventive action can be contemplated before the alarm actually occurs. This requires the study of historical data before issuing warnings. Knowledge for this kind of reasoning is captured using *Spanning rules. Spanning rules* are derived from *Autonomous rules* and their structure is shown below.

From time
Spanning premises
Priority
Premises
Actions
Hold

The additional attributes of *Spanning rules* are From time and Spanning premises. Spanning premises are conditions ranging over data history to calculate trends like increase, decrease and rate of change. The From time parameter indicates the time over which parameter history is to be obtained to calculate the trends.

Spanning rules can be either triggered by events occurring in the system under investigation or periodically. Two specialisations of *spanning rules* are derived viz. *event spanning rules* and *time spanning rules.*

Event Spanning Rules

Event spanning rules are triggered by an event. The structure of this rule is

From time

Event

Spanning premises

Priority

Premises

Actions

Hold

The knowledge to be modelled using an *event spanning rule* looks like : if hydraulic oil level is below 3000 litres and rate of decrease of pressure is more than 10% in the last 100 seconds then a leak in the pipe system is suspected. This is modelled as an event spanning rule as shown below.

S1:

Event: **Hydraulic_system.OilLevel** < *S000 l*

From Time : *100 seconds*

Spanning Premise: **Decrease(Hydraulic_system.Pressure)** > *10%*

Hold: [message "leak in pipe system suspected, check up"]

Time Spanning Rules

Some critical parameters of physical systems are monitored periodically to observe trends and make conclusions about the status of the physical system. Such situations are modelled by a *Time spanning rule.* The *Time spanning rule* is derived from the *Spanning rule* class. The structure of these rules is

From time

Cycle time

Spanning premises

Priority

Premises

Actions

Hold

The attribute Cycle time specifies the time period in which the rule should be fired once. This rule is used for periodic monitoring as in the example below.

Knowledge: If the bearing temperature of the reaction wheels has increased by 10% in the last 5 seconds then it can be concluded that the bearing lubrication is low.

S2:

Cycle time: *5 seconds*
From Time : *10 seconds*
Spanning Premise: Increase(**Bearing**. temperature) > *10%*
Hold: [message "bearing lubrication low, check **up"]**

The four different rule types defined above enable modelling of knowledge and events peculiar to real time monitoring applications. Rules in the rule base are instances of one of the rule classes. This conceptual view of the rule taxonomy is carried to the storage level by storing rules in an object base similar to object store using the same access methods provided by the object manager. Rules are thus

treated as first class objects with all properties of independent objects.

4.3.4 Condition Elements

The premises, spanning premises, events and actions of rules in REX are formed by conjunction of *Condition elements.* The *condition elements* are further categorised into *regular conditions* and *spanning conditions.*

Regular Conditions

Objects of the *Regular conditions* class form the *premises, events* and *actions. Regular condition* class is a derived class of base class *OBJECT.* The structure is shown below.

LHS

rel-op

RHS

method name

The LHS and RHS are expressions on the classes and attributes in the object taxonomy of REX. The relational-operatorcan be ==,! =,<,>,<=,>=. The condition expressed using a *regular condition* is translated in a C++ method and attached to the object. The evaluation of this method returns true or **false**

Spanning Conditions

Objects of the class *Spanning conditions* form the *Spanning premises* in *Spanning rules.* Again like *regular conditions* the *spanning condition* class is derived from the base class *OBJECT.* The structure is shown below.

Function

Class

Attribute

rel-op

value

The attribute **function** specifies the trend. It could be Decrease, increase or rate of change. The class name, attribute name attributes identify the parameter whose trend is to be calculated. The three functions decrease, increase and rate of change are generic functions implemented for the class *Spanning conditions.*

4.3.5 Hold

Hold asserts a fault state of the physical system under investigation. This is a message to the operator console. The *Hold* action terminates a reasoning thread. Rule with a *Hold* assertion will have empty *Action* slots.

4.3.6 Event

The *Event* in a *Clock synchronised* or a *Spanning* rule specifies the trigger for matching or firing the rule. *Events* are instances of the class *Regular condition* and are usually formed by a conjunction of more than one *condition element.* The evaluation of events is triggered by updates to the Attribute Table.

4.4 Knowledge Base Verification

Hayes-Roth points out that rule based systems lack a suitable verification methodology or a technique for testing the consistency and completeness of a rule set [77]. The verification of rule bases is classified into two categories. They are intra rule and inter rule verifications. In *intra rule verification,* a given rule is verified for completeness and freedom from contradictions. A rule in the knowledge base is defined to be

Complete if all slots in the rule objects are filled with legal entries. An *autonomous rule* should posses legal non empty premise slot and either a non empty *action* or *hold* slots. *Autonomous rule* which possesses non empty *action* and *hold* slots is declared invalid. A *spanning rule* should have non empty *spanning premise* and *from time* slots. The *premises* and *actions* of any rule should be free from contradictions. The contradictions fall into two categories: *value contradicting* and *operator contradicting.* Vaiue *contradicting* premises have identical attributes and relation operators and dissimilar values. *Operator contradicting* premises possess identical attribute value pairs but have complementary operators. The complementary operator table is given below

==	! _
>	<
>=	<=

Two *spanning premises* are function contradicting if they have the same object attribute value pairs but possess contradicting functions. The *decrease* function contradicts the *increase* function and vice versa. Two *actions* are said to be contradicting if they assign different values to the same attribute. All the above verifications can be incorporated in the syntax analysis of the rule base.

Inter rule verification pertains to the complete rule base. Rules identified to be of the following categories are invalid rules [25, 45, 73, 85] - *contradicting, dead end rules, and nonstarter rules.*

Two *autonomous rules* are defined to be contradictory if they either

- possess *identical premises, but contradictory action/hold slots* or
- possess *operator contradicting premises, but identical action/hold slots*
- or possess *identical premises, actions/hold slots, but different priorities.*

An example of contradicting Autonomous rules is given below

Rule A1:

Premise:

Pressure < *100 Ksc and* **Yaw** > 15°

Hold:

[message **System_controllability** is negative.]

Rule A2:

Premise:

Pressure > *100 Ksc and* Yaw > 15°

Hold:

[message **System_controllability** is negative.]

The above two rules possess *operator contradicting premises* and hence are invalid.

Two *Clock synchronised rules* are defined to be contradicting if they

- *satisfy the conditions imposed for contradicting autonomous rules*
- or *possess identical premises, actions, but have operator contradicting events.*

Two *Time spanning rules* are defined to be contradicting if they

- *satisfy the conditions imposed for contradicting autonomous rules*
- or *they have function contradicting spanning premises but identical actions/hold slots.*

Two *Event spanning rules* are defined to be contradicting if they

- *satisfy the conditions imposed for contradicting autonomous rules*
- or *possess identical premises, actions, but operator contradicting events*

A rule is a *dead end rule* if the *actions* of the rule do not participate in the *premises* of at least one other rule. This is because, in REX, it has been assumed

that a reasoning chain terminates in an Hold. If the attributes updated by **a** rule's action do not participate in the premises of another rule, it implies that a reasoning chain has terminated in an Action. So, REX cannot have *dead end rules* though it is a forward chaining system. A rule is a non starter rule if the premises of the rule can never be satisfied. This happens if data that matches the premises is not generated either by the external world or by the actions of another rule.

4.5 Rule Base Verification Using Extended Petri Nets

The criteria for verification of rule bases can be implemented in the Extended Petri Net model in a simple way. The algorithms to carry out inter rule verification using Extended Petri Net representation are given below. [71].

Let us define four functions

- get_next_rule(), which gets the next rule (transition), by suitable table look-up procedure.
- get_next_transition() which gets the next transition
- get_prev_transition() which gets the previous transition and
- Val(rt_i) gives the place, value pairs $\forall p_i \in O(rt_i)$

The procedure for detecting contradictions between two rules R_i and *Rj* (the corresponding rule transitions rt_i and *rtj)* is **contradict**(rt_i, rt_j).

```
contradict(rt_i, rt_j){
contradict = false;
if (I(rti) = I(rt_j)) {
    if( Val(rt_i) ≠ Val(rt_j)) {
        contradict = true;
        return(contradict); } }
```

```
    else if(I(rt_i) ≠ I(rt_j) and Val(rt_i) = Val(rt_j){
            pt_i = get_prev_transition(rt_i)
            ptj = get_prev_transition(rt_j)
            pti in contradictlist(pt_j)
        contradict = true;
        return(contradict);}
    else if(I(rt_i) = I(rt_j) and Val(rt_i) = Val(rt_j)) {
        if( priority(rt_i) ≠ priority(rt_j))
            contradict = true;
            return(contradict); }
    else if(V( p_k € I(rt_i)) if( ~p, € I(rt_j))
            contradict = true;
    return(contradict);
}
```

The following procedure given R_i detects if R_i is a *non starter rule.*

```
nonstarter(rt_i) {
    if ∀(rt_j ∈ rt ∧ rt ≠ rt_j)
    if (I(rt_i) notin O(rt_j) V I(rt_i) ≠ external input
            return(true);
        return(false); }
}
```

The following procedure given R_i detects if R_i is a *deadend rule.*

```
deadend(rt_i) {
    if (O(rt_i).type ≠ HP and
    ∀(rt_j € rt A rt_i ≠ rt_j)
        if O(rt_i) notin I(rt_j)
            return(true);
        return(false); }
```

4.6 Summary

In this chapter, a unified object oriented data and knowledge representation is proposed. The scheme is designed taking into consideration the temporal nature of the real time domains. The data representation supports the concept of data validity through the timeout facet of attributes. The knowledge representation scheme is an object oriented rule base. Rules are treated as objects in an uniform taxonomy with data objects. Various rule types are defined in the taxonomy to represent different situations. Clock synchronised rules represent event based knowledge. Spanning rules represent knowledge about trends in the system. Treating rules as first class objects is a special feature of this representation. This allows uniform storage and retrieval of both data and rules. Further, the extensibility provided by the object oriented representation, helps in extending the rule model to meet emerging needs of the domain. Other shells using an object oriented representation of data, treat rules as separate entities operating on the data objects [9]. Another shell treats rules as methods of an object [43], thus restricting the scope of the rules. Finally the verification criteria for the knowledge base are presented. In the next chapter, the match algorithm used in the REX Reasoning Subsystem is presented.

Chapter 5

MATCH ALGORITHM

5.1 Introduction

The match problem in production systems can be formulated as: *Given a set of rules, the current work area, the current matched rule set and a set of changes to the work area, what is the best way to determine the new matched rule set.* The problem can be formally defined using the database query language primitives. A premise evaluation in a rule is either a selection or a join query. For example, consider the following two premises

Premise 1: $A.a_1 < 30$

Premise 2: $A.a_2 < B.b_1$

Evaluation of Premise 1 is equivalent to a selection on the object instances of A. It can be expressed as

$$\sigma_{A.a_1<30}(A)$$

Evaluation of Premise 2 is equivalent to a join on the object instances of A and B. It can be expressed as

$$A \bowtie_\theta B$$

where θ is the relation $A.a_2 < B.b_1$. The complete evaluation of a rule's premises can be expressed as a join of many selection and join operations like the ones given above. Thus the result of match operation on a rule could be formally defined as

$$P_1 \bowtie P_2 \bowtie P_3 \ldots \bowtie P_i \ldots \bowtie P_n$$

where P_i is a premise and its evaluation is equivalent to either a 'select' or 'join' operation as in the example above.

Hence,the match process can be considered equivalent to query evaluation in database systems. RETE [22] and TREAT [53] are examples of match algorithms that use the principles of query evaluation.

The RETE match algorithm is the most commonly used match algorithm in OPS5 production systems. It employs the principles of incremental match and state saving between match cycles [22]. RETE compiles the left hand sides of production rules into a discrimination network (like a data flow net). Changes to the working memory are provided as input to this network. The RETE net performs a series of tests on each of these working memory elements. These tests performed can be either Select or a Join operations. The results of a selection are stored in memory nodes called a memories and those of a join are stored in β memories. As the working memory changes flow through the net, the different a and β memories along the path are updated. By saving the state in this manner, testing of these working memory elements can be avoided in subsequent match cycles. The terminal nodes of a RETE net signal the successful match of a production. When the processing of working memory elements in the net reaches these terminal nodes, a rule instantiation is entered into the conflict set. Thus, changes in the working memory produce changes in the conflict set. The output from the network is a modified conflict set.

The state information stored in the a memories, β memories and the conflict set reduces the number of comparisons made . This is the main advantage of the RETE net. However, the maintenance of β memories is the major drawback of RETE. A β memory stores the result of a join operation. While processing the working memory elements, many such cartesian products are stored in a beta-memory and each cartesian product may be a part of many beta-memories. This can result in combinatorial increase of β memory sizes. Also, deletion of a working memory element results in the deletion of all the corresponding cartesian products from many beta-memories and requires a large execution time.

The TREAT algorithm had been proposed to overcome these disadvantages of RETE. TREAT is based on Mc Dermott's conjecture that the cost of recomputing the joins will be less than the cost of storing and maintaining the results of such joins [53]. Hence the TREAT algorithm does not maintain β memories. However,

it stores the results of select operations in a memories. TREAT recomputes the join whenever a working memory update occurs. The advantage of TREAT is that there will be no combinatorial explosion in space requirements due to /? memories. Also, due to the absence of β memories, a deletion will be a simple operation. But an addition to the working memory is quite costly, as many joins have to be recomputed. However, empirical results on the performance of RETE and TREAT show that TREAT is faster and thus proves Mc Dermott's conjecture.

The REX match algorithm also adopts the principles of incremental match and limited state saving like TREAT. However, the TREAT algorithm cannot be adopted to the REX rule system, which has been specifically designed for building real time expert systems. During the design, the requirements of the real time application domain have lead to the evolution of new rule structures like Clock synchronised and Spanning rules. The evaluation of spanning premises in REX, is equivalent to an aggregation query on a set of object instances. The TREAT algorithm cannot handle the matching of these augmented rule structures. The REX match algorithm has been proposed to handle these additional requirements. The REX match algorithm and its handling of different rule types is discussed in the rest of this chapter.

5.2 Issues in the Design of REX Match Algorithm

REX was designed for building real time expert systems that deal with continuous streams of input data, where the most recent data values obtained will invalidate the earlier data values. In view of this, any premise evaluation has to be made with the most recent data. Premise evaluation performed with earlier data values will be invalid. The REX match algorithm makes use of this property. With this property, the result of either the 'select' or 'join' operations will only be a single instance, unlike RETE or TREAT, where the result can be more than tuple. This property is similar to the unique attribute property used by the Uni-RETE match algorithm [87]. Thus the REX match is similar a TREAT implementation with the unique attribute property.

The REX rule semantics have specific requirements, which have to be met by

the match algorithm. The three rule types in REX deal with different types of data. Autonomous rules deal with latest values (similar to rules in most other rule based systems). Spanning rules pertain to trends in historical data. Clock synchronised rules deal with data that has to be obtained in future. These three different rule types need to be handled differently. The REX match algorithm has attempted to meet these unique requirements.

5.3 The Work Area Structure

The specifics of a match algorithm depend to a large extent on the rule semantics and the work area organisation. The rule semantics have been defined in Chapter 4. The Work Area has five tables and an `Object` Instances space. The five tables are Attribute Table, Premise Table, Event Table, Rule Table and Attribute Event Premise (AEP) Index Table. The structures of these tables are given in Tables 5.1 to 5.5.

The Attribute Table has information about all the attributes defined in the object taxonomy. The contents of this table are sorted on Attribute name. Every attribute has a unique identification number termed as **Attribute number.** All premises and events refer to an attribute, using this number. **Type** identifies the data type of the attribute. The **attribute@time** entry contains the life span of the attribute value. The **shared request** and **exclusive request** counts are used in interference analysis. The **update flag** is set when the attribute value is updated.

All premises in the rule base are identified by unique **premise id.s.** The **Premise Table** consists of premise data sorted on *premise id.s.* Each entry has a list of **rule id.s** in which the premise participates. The **event id.** indicates if the evaluation of the premise is dependent on the occurrence of the event. The **truth value** of the premise is another entry in the table. The **premise type** states whether the premise is an ordinary premise or a spanning premise.

The **Event Table** has data related to events defined in the rule base. The table is sorted on **event id.s.** A list of *premise id.s* specifies the premises to be evaluated

Att. No.	Att. name	Type	Attribute® time	value	shared request count	exclusive request count	update flag

Table 5.1: Structure of the Attribute Table

Premise id.	List of rule id.'s	Event id.	Truth value	Premise type

Table 5.2: Structure of Premise Table

if the event occurs. The time **operator** entry is for clock synchronised rules. For an event belonging to the spanning rule, the entry is **NIL.** The **time limit** column reflects the time limit specified in the clock synchronised rule to which the event belongs. The 'from time' in event spanning rules and 'cycle time' in time spanning rules are also entered in this column The **event time** entry specifies the time at which the event has occurred. The **service flag** indicates if the rule in which the event participates has been evaluated after the event has occurred. The **occurrence flag** is set when the match process finds that the event has occurred.

The Rule Table consists of rule related data. and is sorted on **rule id.s.** The **priority** entry states the rule priority. The **Read List** is the list of attributes in the rule's premises. The **Write List** is the list of attributes in the rule's actions. The *Premise id.s* list is the list of the rule's premises. The *Action id.* states the rule's actions. The *Event id.s* entry specifies the event number in case of a clock synchronised or spanning rule. The **Hold id.** entry specifies the hold message id. if the rule has an non empty hold slot.

The AEP Index Table is an inverted index from attributes to events and premises. In this table, for each attribute, the list of premises and events in which the attribute participates is stored.

Event id.	List of premise id.s	Time limit	Time operator	Event time	Occurrence flag	Service flag

Table 5.3: Structure of Event Table

Rule id.	Priority	Read List	Write List	Premise id.s	Action id.	Event id.	Hold id.

Table 5.4: Structure of Rule Table

Att. No.	List of Premise id.s	List of Event id.s

Table 5.5: Structure of **AEP** Index Table

5.4 Match Process

The REX match algorithm uses the list of updated attributes as the starting point in finding the true premises. The algorithm determines those premises and events, whose truth values are likely to be changed, from the AEP Index table. It then evaluates these events to determine if they have occurred. The set of premises to be evaluated due to the occurrence of events is determined next. Now the premises are evaluated and the matched rules are determined. The premises that have to be evaluated can be either regular premises or spanning premises. Spanning premises are evaluated by considering many object instances. Thus they would require more computing time than other premises. Since, the match has to meet real time requirements, it cannot not wait for the completion of evaluation of the spanning premises. So, the evaluation of the spanning premises is carried out as a separate task independent of the other steps in the match process. This process can be represented with the following steps

1. From the attribute updates, determine the set of events and premises that have to be evaluated
2. Evaluate the events, if the events are found to have occurred then find the premises that have to be evaluated.
3. Now, evaluate the premises determined in steps 1 and 2. If there are any spanning premises amongst these, they have to be evaluated using a separate process.
4. Using the set of premises determined to be true in the earlier step, find the set of matched rules.

This process is explained below with the help of a demonstrative example.

5.5 A Demonstrative Example

Let us consider an expert system for monitoring a simple hydraulic controller. The controller receives *force demands* along the three axes. It responds within 2 seconds, by manipulating the fluid pressure in the hydraulic lines. If the demand cannot be met within 2 seconds, the controller is deemed to have failed. The *hydraulic tank pressure, the line pressure* and the *return line pressure* and the *force demands* on the controller in the x,y and z directions are monitored continuously. The expert system based on the hydraulic pressures and the force demands in the three axes, determines the controller response in the three axes.

In REX, three classes are defined, to monitor such a controller. They are

```
class Pressure{
    Tank_pressure;
    Line_pressure;
    ReturnLine_pressure;
}

class Force_demands{
    x
    y;
    z;
}

class Status{
    x;
    y;
    z;
}
```

Let the characteristics of the hydraulic controller be defined as follows

1. If the tank pressure is less than 100Ksc, then the controller can meet any force demand in the y-axis only if the demanded force is less than 10N.
2. If there is a force demand of greater than UN in both x and z directions, and if the line pressure is less than 60Ksc within 2 seconds of the demands being made, the controller cannot meet the demands.
3. If the return line pressure is not zero and the tank pressure is less than 80Ksc even after 1 second, then it can be concluded that the force demands cannot be met by the controller.
4. If the line pressure is less than 5 Ksc and the decrease in the tank pressure in the last 10 seconds is greater than 100%, then the force demands cannot be met.

The above characteristics can be modelled with the following rules.

Rule 1:

Premises: $Pressure.Tank\_pressure < 100$; $Force\_demands.y < 10$;
Action: $Status.y = 1$;
Priority: 1;

Rule 2:

Event: $Force\_demands.x > 11$; $Force\_demands.z > 11$;
Premises: $Pressure.line\_pressure < 60$;
Action: $Status.x = 0$; $Status.z = 0$
Time Limit: 2 seconds;
Time Operator: BEFORE;
Priority: 2;

Rule 3:

Event: $Pressure.ReturnLine\_pressure = 0$;

Premises: $Pressure.Tank\_pressure < 80$;
Action: $Status.x = 0$; $Status.y = 0$; $Status.z = 0$;
Time Limit: 1 second;
Time Operator: AFTER;
Priority: 3;

Rule 4:

Event: $Pressure.Line\_pressure < 5$;
Spanning Premise: $\text{Decrease}(Pressure.Tank\_pressure) > 100\%$;
Action: $Status.x = 0$; $Status.y = 0$; $Status.z = 0$;
From Time: 10 seconds;
Priority: 4;

Let the three pressures be 60, 4 and 1 Ksc respectively. Let the force demands made on the controller be 12, 4, and 14N in the x, y and z directions respectively. Now, in the example we will see how the match process determines the current world state and the appropriate rules to be fired.

For the sake of brevity the following surrogates have been used. The Pressure class is represented as A. Similarly the other two classes Force-demands and Status are represented as B and C respectively Now the classes are

Class $A\{a_1, a_2, a_3\}$
Class $B\{b_1, b_2, b_3\}$
Class $C\{c_1, c_2, c_3\}$

The rules would now be

Rule 1:

Premises: $A.a_1 < 100$; $B.b_2 < 10$;
Action: $C.c_2 = 1$;
Priority: 1;

Rule 2:

Event: $B.b_1 > 11$; $B.b_3 > 11$;
Premises: $A.a_2 < 60$;
Action: $C.c_1 = 0$; $C.c_3 = 0$
Time Limit: 2 seconds;
Time Operator: BEFORE;
Priority: 2;

Rule 3:

Event: $A.a_3 != 0$;
Premises: $A.a_1 < 80$;
Action: $C.c_1 = 0$; $C.c_2 = 0$; $C.c_3 = 0$;
Time Limit: 1 second;
Time Operator: AFTER;
Priority: 3;

Rule 4:

Event: $A.a_2 < 5$;
Spanning Premise: $\text{Decrease}(A.a_1) > 100\%$;
Action: $C.c_1 = 0$; $C.c_2 = 0$; $C.c_3 = 0$;
From Time: 10 seconds;
Priority: 4;

The five tables for the above sample rule base are shown in Tables 5.6 to 5.10.

All the attributes defined in the object taxonomy are sorted lexicographically, and entered in the Attribute **Table**. and it is shown in Table 5.6. The rules in the rule base are assigned id.s in the order in which they are entered in the rule base. All premises of the rules in the rule base are assigned unique premise id.s. A rule can have more than one premise, each of which is assigned a different premise id.. For example, rule 1 has two premises $A.a_1 < 100$ and $B.b_2 < 10$. The two premises

Att. No.	Att. name	Type	Attribute® time	value	shared request count	exclusive request count	update flag
1	$A.a_1$	float					
2	$A.a_2$	float					
3	$A.a_3$	float					
4	$A.time$	float					
5	$B.b_1$	float					
6	$B.b_2$	float					
7	$B.b_3$	float					
8	$B.time$	float					
9	$C.c_1$	float					
10	$C.c_2$	float					
11	$C.c_3$	float					
12	$C.time$	float					
13	$Object.time$	float					

Table 5.6: Attribute Table for the sample rule base

would be assigned id.s 1 and 2 respectively. The Premise Table for the four rules is in Table 5.7.

An event in a rule, can have more than one condition element. In such a case, an event is said to have occurred if all conditions in the event are determined to be true at the same time. So, while evaluating an event, it is necessary to evaluate all condition elements at the same time to determine if the event has occurred. Hence, unlike premises, an event is considered as a single indivisible unit and assigned a single id. For example, the event in rule 2 has two condition elements. But only one event id. is shown for the rule. The Event Table for the four rules is given in Table 5.8. The Rule Table entries for the four rules are shown in Table 5.9. In the 'Read List' column, attributes that participate in the rule's event are not listed. This is because, once an event has occurred it has no further role in the premise match or rule firing. The entries in the AEP Index Table for the four rules are shown in Table 5.10.

All the above tables, are generated off line as a one time activity for the rule base. The tables are loaded into the Work Area by the *Object Manager* as a part of its initialisation activity. The tables have to be recomputed, if any modifications are made either to the object taxonomy or the rule base.

Premise id.	List of rule id.s	Event id.	Truth value	Premise type
1	1	Nil		
2	1	Nil		
3	2	1		
4	3	2		
5	4	3		Spanning

Table 5.7: Premise Table for the sample rule base

Event id.	List of premise id.s	Time limit	Time operator	Event time	Occurrence flag	Service flag
1	3	2	BEFORE			
2	4	1	AFTER			
3	5	10	Nil			

Table 5.8: Event Table for the sample rule base

Rule id.	Priority	Read List	Write List	Premise id.s	Action id.	Event id.	Hold id.
1	1	$A.a_1$, $B.b_2$	$C.c_2$	1,2	1	Nil	Nil
2	2	$A.a_2$	$C.c_1, C.c_2$	3	2	1	Nil
3	3	$A.a_1$	$C.c_1$,$C.c_2$,$C.c_3$	4	3	2	Nil
4	4	$A.a_1$	$C.c_1, C.c_2, C.c_3$	5	4	3	Nil

Table 5.9: Rule Table for the sample rule base

Att. No.	List of Premise id.'s	List of Event id.'s
1	1,4,5	
2	3	3
3		2
4		
5		1
6	2	
7		1
8		
9		
10		
11		
12		
13		

Table 5.10: AEP Index Table for the sample rule base

So, for the current external state, the following object instances are created.
A{a_1 = 60; a_2 = 4; a_3 = 1;}
B{b_1 = 12; b_2 = 4; b_3 = 14;}

These object attribute values will be entered in the Attribute Table to reflect the new updates. Now the match is initialised. The 'current time' at this instance is read from the real time clock and stored in the variable '**MatchTime**'. All events which would be determined to have occurred in the ensuing match cycle are defined to have occurred at 'MatchTime'. Also, to ensure a consistent view of the Attribute Table for the match process, the table is locked. Updates by *External Data Interface* will not be allowed until this lock is released after *Interference Analysis.* All attributes whose values have been updated are collected in the set CHANGED.

CHANGED = $\{A.a_1, A.a_2, A.a_3, B.b_1, B.b_2, B.b_3\}$

In the next step, for each attribute in CHANGED, the premises to be evaluated are collected in a set called PREMISES using the AEP Index Table.

PREMISES = {1,2,3,4,5}

Similarly, the events to be evaluated are collected in a set called EVENTS.

EVENTS = {1,2,3}

In the PREMISES set, if any premise is event dependent (i.e. the premise belongs to a spanning rule or a clock synchronised rule), it is removed from PREMISES. This is because the premise need not be evaluated unless the corresponding event has occurred. Accordingly, premises 3, 4, and 5 are removed from the PREMISES set.

PREMISES = {1,2}

The events in the set EVENTS are evaluated first to determine if they have occurred. In this phase, each event in the set EVENTS is evaluated using the Attribute Table. If the event has occurred, the occurrence flag and the service flae of the event are set in the Event Table. The MatchTime is noted as the event

time in the Event Table. In the example, all the three events have occurred. Their occurrence flags and service flags are set and the **'MatchTime'** is entered in the event time column in the Event Table.

Next, from the Event Table, for each event, whose occurrence and service flags are set, the set of premises to be evaluated is determined. In the example for the event 1, the set of premises to be evaluated is {3}. The premise 3 will be added to the PREMISES set if the event's timing condition is satisfied. For event 1, the Time Operator is BEFORE. So the timing condition is

MatchTime — EventTime < EventTimeLimit

Since, in this case, both MatchTime and Event Time are same, the timing condition has been satisfied and the premise 3 will be added to the set PREMISES, which now consists of

PREMISES = {1,2,3}

For event 2, the Premise to be evaluated is {4}. The Time Operator is AFTER, Hence, the timing condition is

MatchTime — EventTime > EventTimeLimit

This timing condition is not satisfied and hence premise 4 is not added to the set PREMISES.

For event 3, the premise to be evaluated is {5}. Since the time operator entry is NIL, there is no timing condition to be satisfied. So, the premise can be added to the set PREMISES. However, premise 5 is a spanning premise (from the Premise Table) and its evaluation has to be carried out using the Object Instances space instead of the Attribute Table. In spanning premise evaluation, data over a period of time has to be evaluated. If the time over which data has to be compared is very large, all necessary instances may not be in the Object Instances space and have to be retrieved from secondary storage. This will take considerable time when compared to the evaluation of other premises. So, the evaluation of spanning premises has to be carried out independent of other premise evaluations. Hence, premise 5 is entered in a different set called SPAN- PREMISES.

Premise id.	List of rule id.'s	Event flag	Truth value	Premise type
1	1	Nil	T	
2	1	Nil	**T**	
3	2	1	**T**	
4	3	2		
5	*A*	3		Spanning

Table 5.11: Premise Table for the sample rule base after Premise Evaluation

SPAN-PREMISES = {5}.

Now, the premises to be evaluated are

PREMISES = {1,2,3}
SPAN-PREMISES = {5}

Each premise in PREMISES is evaluated and its truth value is determined. The evaluation is carried out by using the values in the Attribute Table. In the example, premises 1,2, and 3 evaluate to true. So, the truth values are updated in the Premise Table and the premises are added to the set TRUE- PREMISES. The updated Premise Table is shown in Table 5.11.

TRUE-PREMISES = {1,2,3}

A task called *'SPremise Evaluator'* determines if the premises in the set SPAN-PREMISES are true. *SPremise Evaluator* is a part of the *Premise Evaluation Task* (refer Chapter 3). However, it runs as an independent task and other tasks in the *Match Process* do not wait for its completion. Upon completion, the *'SPremise Evaluator'* adds to TRUE-PREMISES, those spanning premises which have been evaluated to be true. The added spanning premises will then be considered in the subsequent inference cycle.

In the next step, the Rule Evaluation Task determines the rules in which the TRUE-PREMISES participate. The set is RULE- SET.

RULE-SET = {1,2}

For each rule in RULE-SET, it is determined if all the premises of the rule are

true. In the example, the premises of the rules 1 and 2 are true. Hence the rules are added to the MATCHED RULE SET and deleted from the RULE-SET.

RULE-SET = {}

MATCHED RULE SET = {1,2}

The formation of the MATCHED RULE SET(MRS) completes the Match Process and enables interference analysis and rule firing.

The Rule Evaluation Task (RET) does not wait for the SPremise Evaluator to finish its task. Evaluation of spanning premises might take considerable amount of time and waiting for them to complete will affect the real time nature of the system. If the 'SPremise Evaluator[1] completes its task early enough, the RET will be able to consider the true Spanning Premises also. Otherwise they will be considered in the next match cycle. The truth values of premises are maintained across match cycles. This is similar to *Condition Support* in RETE and TREAT algorithms.

The premises of the clock synchronised rule with the AFTER operator will be evaluated in that match cycle in which their timing condition is satisfied.

In the example, only an Event Spanning rule has been considered, whereas a Time Spanning Rule has not been considered. The match of a Time Spanning rule should be carried out once in every time cycle. The completion of cycle time can be modelled as a timer alarm event. This event can be used to trigger the premise evaluation of a Time Spanning rule. Hence the match of a Time Spanning rule is similar to that of an Event Spanning Rule.

>

5.6 REX Match Algorithm

The *Match Process* is started either after the *Attribute Update Task* or after the *Rule Firing Tasks* have updated the attribute values in the Attribute Table. These tasks set the update flag while updating attributes in the Attribute Table. This flag is used subsequently by the *Match Process*. The algorithm for implementing the *Match Process* is given below.

begin algorithm

1. Lock the Attribute Table. Set 'MatchTime' to current time as given by the clock.
2. From the Attribute Table, add to the set CHANGED, the attributes whose update flags are set. Reset the update flags.
3. For each attribute in CHANGED, do
   (a) Add to PREMISES, the list of premises entered against the attribute in AEP Index Table.
   (b) Add to EVENTS, the list of events entered against the attribute in the AEP Index Table.
4. For each element 'premise' in PREMISES, if the event flag of 'premise' in Premise Table is not NIL then delete 'premise' from PREMISES.
5. For each 'event' in EVENTS do
   (a) evaluate the 'event'
   (b) if the 'event' has occurred, set the occurrence and service flags of the 'event' in the Event Table
   (c) set the Event time in the Event Table as equal to 'MatchTime'
6. For each event in the Event Table, do
   (a) if the service flag is set then do
      i. if (Time Operator == BEFORE and *MatchTime* - *EventTime*< *EventTimeLimit)* then add the premise id.'s from the Event Table to the set PREMISES, reset the service flag of the event.

ii. if (Time Operator == AFTER and $MatchTime - EventTime > TimeLimit$)then add the premises id.'s from the Event Table to the set PREMISES, reset the service flag of the event.

iii. if (Time Operator —= NIL) then for each premise id in the premise id list of the event do, if the premise type is spanning add it to SPAN-PREMISES else add it to PREMISES, reset the service flag of the event.

7. If there any members in SPAN-PREMISES trigger the 'SPremise Evaluator' task.(This task will evaluate the set SPAN-PREMISES and add to TRUE-PREMISES those spanning premises which evaluate to true. This is an independent task and other activities in the match process do not wait for its completion.)

8. For each element premise in PREMISES do

   (a) Evaluate the **'premise'**

   (b) Update the truth value in the Premise Table

   (c) If truth value of the **'premise'** is true, add premise to TRUE-PREMISES

   (d) If truth value of the 'premise' is false, and is present in TRUE-PREMISES, remove it from TRUE-PREMISES.

9. For each member of TRUE-PREMISES, add to RULE SET, the list of rules entered against the premise in the premise Table.

10. For each **'rule'** in RULE-SET do

    (a) From the rule table, determine the corresponding list of premises

    (b) If all premises of the rule have a truth value **'T'** then add the rule to MATCHED RULE SET. Delete these premises from TRUE-PREMISES.

end algorithm

5.7 **Interruptability and Synchronisation**

REX is a real time expert system tool. The External Data Interface has to update the Attribute Table whenever external data is available. However the match process would require a consistent view of the Attribute Table. So, while the, match process is in progress, the AUT (of External Data Interface) has to be inhibited from updating the Attribute Table. Hence, the Attribute Table has to be locked during the Match process and Interference Analysis. During rule firing, updates to the Attribute Table by AUT can be allowed without any synchronisation. This is because the AUT posts the data obtained from the external world to the Attribute Table. This data is sensor data and the rule firing tasks cannot update the attributes which represent the sensors. The 'SPremise Evaluator' could take considerable time in evaluating spanning premises. This task can be in progress while the rule firing tasks are updating the Attribute **Table** . The concurrent execution of the 'SPremise Evaluator' and Rule Firing Tables does not affect the integrity of the Match Process. This is because 'SPremise Evaluator' has to consider only objects in the `Object` Instances space and not the Attribute Table. Hence the 'SPremise Evaluator' can be executed concurrently with the Rule Firing Tasks and AUT.

5.8 Number of Evaluations in REX Match

The cost of a match algorithm can be determined by the number of evaluations (both premises and events) that are triggered by a single attribute update [53].

If a single attribute value in the Attribute Table is updated, then the premises and events listed against the attribute in the AEP Index Table have to be evaluated. Let us assume that there are 'p' number of premises and 'e' number of events listed against an attribute in the AEP **Index** Table. All of these premises need not be evaluated. Those premises belonging to clock synchronised rules and spanning rules have to be evaluated only if their events have occurred. Let the fraction of premises dependent on events be *'f'*. Then the number of premises and events to be evaluated

are (l-f)p+e

These 'e' events in turn could lead to some more premises being evaluated. Let the number of such premises be p_e. These premises belong to clock synchronised and spanning rules in which the 'e' events participate. Let the fraction of premises belonging to clock synchronised rules be *'c'*. So, the number of premises of clock synchronised rules would be '$c.p_e$'. The number of premises of spanning rules are '$(1\text{-}c).p_e$'.

Now, consider the clock synchronised rules' premises with the BEFORE operator. If the fraction of clock synchronised rules with BEFORE operator is α, the number of premises in a rule with BEFORE operator is $\alpha.c.p_e$. The number of times a premise belonging to a rule with BEFORE operator is evaluated is proportional to the Attribute Table update frequency and the time limit of that rule. If the update frequency of the Attribute Table is 'u' and the average time limit is 't', then the number of evaluations for these premises is $\alpha.c.u.t.p_e$. The number of evaluation for a clock synchronised rule premises with AFTER operator is 1 (This evaluation is made after the time limit has elapsed). So, the total number of evaluation for premises of rules with AFTER operator is $(1 - \alpha).c.p_e$.

For spanning premises, if 'h' is the average number of history (earlier) object instances considered, then the number of evaluations is $(1 - c).h.p_e$.

So, the total number of evaluations made per single attribute update are

$$(1 - f)p + e + \alpha.c.u.t.p_e + (1 - \alpha).c.p_e + (1 - c).h.p_e$$

$$= (1 - f)P + e + (\alpha.c.u.t + ((1 - \alpha).c + (1 - c).h)p_e$$

The factors '/, c, α, u, t' and 'h' can all be considered as constants for a given rule base. So, the number of evaluations made for single attribute is given by a linear equation in the number of premises and events. Hence, the time required for the match would also be a linear equation in the number of premises and events.

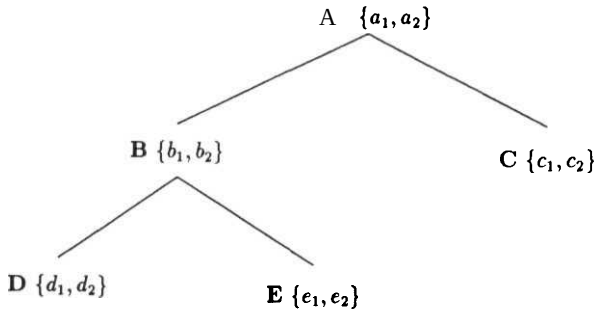


Figure 5.1: Example Object Taxonomy

5.9 Derived Rules

The object oriented nature of data affects the design of the match algorithm. For example consider the object taxonomy in Figure 5.1 and the sample rule given below.

Rule 1:
Premises: $B.b_1 < 3$; $C.c_1 = 4$
Action: $B.a_1 = 0$

In the example rule, the premise $B.b_1 < 3$ can be satisfied by object instances belonging to the three classes B, D and E. The match algorithm has to consider this

object taxonomy while performing the match. This could lead to an inelegant and difficult to maintain match algorithm. In REX, the problem has been overcome by deriving new rules from the given rule and adding them to the rule base. The new rules are

Rule 2:
Premises: $D.b_1 < 3$; $C.c_1 = 4$
Action: $D.a_1 = 0$
Rule 3:
Premises: $E.b_1 < 3$; $C.c_1 = 4$
Action: $E.a_1 = 0$

Rule 2 and 3 are called derived rules. The derived rules though present in the rule base are not visible to the user. Any changes made to the original rule will be reflected in the derived rules. If the original rule is deleted, the derived rules are also deleted. With the additional rules in the rule base it is not necessary to consider the object taxonomy during match process. Thus the match algorithm need not consider the object taxonomy.

5.10 Summary

The REX match algorithm meets the requirements of the three different types of rule semantics. The evaluation of regular premises is similar to a 'select' or 'join' query evaluation in databases. The evaluation of a spanning premise is similar to an aggregation query on a set of object instances. The algorithm maintains the truth values of premises across inference cycles. This is similar to 'condition membership' in TREAT algorithm. The number of evaluations made for a single attribute update is linear. If the REX match algorithm is stripped of the parts evaluating the clock synchronised and spanning rule, then it is similar to TREAT algorithm with unique attribute property. In the next chapter, interference analysis technique used in REX is presented.

Chapter 6

INTERFERENCE ANALYSIS

6.1 Introduction

Reasoning algorithms in forward chaining production systems traditionally employ the concept of a single rule firing per inference cycle. This results in few attribute updates (working memory changes) per inference cycle. Gupta [27] in his study on parallelism in OPS5 production system programs, concludes that the small number of working memory changes per inference cycle leads to smaller speedup factors than expected. Multiple rule firing models are being designed to increase the number of working memory changes per inference cycle and consequently the speed up [31, 33, 37, 38, 39].

A real time expert system has to handle simultaneously occurring disjoint events in the external world. If only one rule is fired in a single inference cycle, then only one or a few related events can be handled per inference cycle. The rest of the events will be ignored and would be considered for firing in subsequent inference cycles. This will lead to loss of the reactive nature of real time expert systems. Hence, it is necessary for a real time expert system to adopt a multiple rule firing model. Multiple rule firing models allow more than one rule to be fired concurrently in a given inference cycle. This will enable a real time expert system to handle simultaneously occurring disjoint events in the external world.

However, the result of the multiple rule firings can be different from the results of any sequential firing. In this case, interference is said to occur among multiple rule firings [31]. For example, consider the following two rules.

R1:

Premise : $A.a_1 < 30$; $B.b_2 > 40$

Action : $D.d_1 = 100$

R2:

Premise : $D.d_1 < 50$

Action : $A.a_1 = 100$

Let us assume that both the rules have matched and a single rule firing policy is adopted. If, Rl is selected and fired, it updates the attribute $D.d_1$. As a consequence, R2 will no longer match and cannot be fired. If, R2 is selected and fired, it will update the attribute $A.a_1$. As a result, Rl will not match and cannot be fired. However, if a multiple rule firing policy is adopted and both Rl and R2 are fired concurrently, then there is no equivalent serial execution. Hence, interference is said to occur between Rl and R2.

In order to guarantee a consistent firing environment for the set of rules to be fired concurrently, it is necessary to detect interference among rules and inhibit interfering rules from firing concurrently. The technique for determining interfering rules is called *Interference Analysis.* Techniques based on data dependency graphs have been proposed to detect interference among rules and rule instantiations [31, 37, 38, 39, 81]. These techniques require large storage space or large computational time [39].

The problem of maintaining working memory consistency in a multiple rule firing environment is similar to the problem of maintaining database consistency in a concurrent transaction environment. In database environments, locking protocols are used to control access to the database by different transactions. Raschid **et.al.** [74] designed a modified two phase locking protocol to implement concurrent rule firing in a database production system environment. However this scheme is prone to deadlocks.

A new interference analysis technique has been developed based on access control of working memory. This scheme is deadlock free and less compute intense than

most other interference analysis techniques [68]. Most of the work in the area of multiple rule firing and interference analysis is carried out using the OPS5 production system. The interference analysis technique in REX, is first developed using OPS5 production system semantics and later adapted to REX. In this chapter, the technique is presented in the context of OPS5. Later, it has been shown how the technique has been adopted to REX.

6.2 OPS5 Production Systems

An OPS5 production system is defined as a set of productions called the production memory (PM), together with a database of assertions, called the working memory (WM). Each production has two parts, the left hand side (LHS) and the right hand side (RHS). The LHS is a conjunction of pattern elements that are matched against the working memory. The pattern elements are called condition elements (CE). The RHS consists of a set of actions. Each action is called an action element (AE). Positive condition elements are satisfied when a matching working memory element (WME) exists. Negative condition elements are satisfied when no matching WME exists. The RHS specifies assertions to be added or deleted to the WM. Action elements specifying addition of assertions are termed positive AE's, while negative AE's specify deletion of assertions from WM [53].

The conventional production system interpreter repeatedly executes the following cycle of operations.

1. Match: For each rule, determine if the LHS matches the working memory environment.
2. Select: Choose exactly one rule instantiation matching the working memory.
3. Act: Execute the actions indicated by the RHS of the selected rule instantiation.

In multiple rule firing systems, the production system interpreter repeatedly executes the following cycle of operations.

1. Match: For each rule, determine if the LHS matches the working memory environment.
2. Select: Choose a set of non-conflicting rule instantiations matching the working memory.
3. Act: Execute the actions indicated by the RHS of the selected rule instantiations concurrently.

In multiple rule firing systems, with the elimination of the conventional conflict resolution strategy, an alternative is required to select a set of non interfering rules which can be fired concurrently. Two rule instantiations are said to be interfering if the working memory elements (WME's) referenced by one rule instantiation are modified by the other rule instantiation.

Interference analysis can be carried out at compile time (off- line) and run time. Compile time analysis is space intensive, while run time analysis is computationally expensive [33, 39]. Though run time analysis yields a higher degree of concurrency, many researchers propose to use a blend of compile time and run time analyses as the later is compute intense. Most compile time and run time analyses techniques use data dependency graphs of production systems to detect interference. In these techniques, a data dependency graph is constructed from the following primitives [33].

1. A Production node (P-node) which represents a set of instantiations, denoted by circles.
2. A working memory node (W-node) which represents a set of WME's, denoted by squares.
3. A directed edge from a P-node to a W-node, which represents the fact that a production node modifies a working memory node. More specifically, the edge labelled '+' (**'-'**) indicates that an instantiation in a W-node is added (deleted) by firing an instantiation in a P-node.
4. A directed edge from a W-node to a P-node, which represents the fact that

a working memory node is referenced by a production node. More specifically, the edge labelled '+'('•') indicates that an instantiation in a W-node is referenced by positive (negative) condition elements of a P- node.

Interference is said to exist between two productions P and Q, if a W-node exists between the P-nodes of P and Q and satisfying any of the following conditions. The W-node is

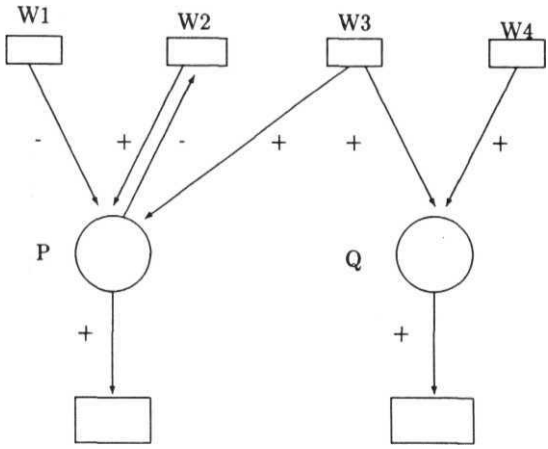
- *C1*: '+' ('-') changed by P and '-' ('+') referenced by Q.
- *C2:* '+' ('-') changed by Q and '-' ('+') referenced by P.
- *C3:* '+' ('-') changed by P and '-' ('+') changed by Q.

The above conditions are called *'paired rule conditions'*. The following is an example of two interering productions. The dependency graph is given in Figure 6.1. This example is taken from [31].

```
P : (p make-possible-trip
    (candidate-city name <x> state New- York)
    -(weather-forecast place <x> date today weather rainy)
—->         (make possible-trip place <x> date today)
    (remove 1)
)

Q : (p make-weather-forecast
    (symptom ânimal frog âction croak place <y>)
    (candidate-city name <y>)
—>          (make weather-forecast place <y> date tomorrow weather rainy)
)
```

The *Interference analysis* using *Paired rule conditions* can be formulated as a join problem. Interference occurs if the result of the join between the positive condition elements of one production and the negative condition elements of another



W1 (weather-forecast place <x> date today weather rainy)
W2 (candidate-city name <x> state New-York)
W3 (candidate-city name <y>)
W4 (symptom animal frog action croak place <y>)
W5 (possible-trip place <x> date today)
W6 (weather-forecast place <y> date tomorrow weather rainy)

Figure 6.1: Dependency Graph for Productions P and Q

production is non empty. Based on this formulation an algorithm to detect interference using the RETE net itself has been proposed [69]. This technique does not need to build any dependency graphs.

The paired rule conditions have to be repeatedly applied to the conflict set for finding a set of compatible rule instantiations. However a different condition called *'All rule condition'* which results in more concurrency is defined as follows[31]:

Let $P_1, P_2, \ldots P_i, P_{i+1}, \ldots, P_{n+1}$, *where* $P_{n+1} = P_1$, *be an acyclic sequence of rule instantiations. Cyclic interference exists in rule instantiations, if, for all i, there exists a working memory element that is '+'('-') changed by* P_i *and is '-'('+') referenced by* P_{i+1}. *Interference occurs between two rule instantiations* P_A *and* P_B, *if any of the following conditions are satisfied*

- B1: *There exists cyclic interference in rule instantiations that include* P_A *and* P_B.
- B2: *There exists a working memory element that is '+'('-') changed by* P_A. *and '-'('+') changed by* P_B

However condition C3 of *'Paired rule conditions'* and condition B2 of *'All Rule Condition'* are not relevant to OPS5 productions systems [31]. These conditions have applicability only in production systems performing set operations.

In multiple rule firing models, serializability is taken as the correctness criteria [37]. Both the 'Paired rule conditions' and 'All rule condition' are defined based on serializability. A set of concurrent rule firings are serializable if there exists some sequential order of the rule instantiations which produces the same results as the concurrent firing. A set of rule firings is serializable if a cycle of dependencies does not exist between the rule instantiations [37]. Schmolze [81] proposes an algorithm to determine rule instantiations which do not conflict. The algorithm is of $O(n^4)$ complexity. He also proposes a sub-optimal algorithm of $O(n^2)$ complexity where 'n' is the number of rule instantiations. This sub- optimal algorithm is similar to the selection strategy proposed by Ishida [31]. In CREL [37], the selection of instantiations is a two step procedure. In the first step interferences between rule

instantiations is determined. This is an $O(n^2)$ problem. To reduce the time in this step, the interference checks are pre compiled into C code. The code is executed for possibly conflicting instantiations and interference if any is determined. In the second step, the set of instantiations to be fired is determined. This is $O(n^2)$ problem. In order to reduce the computation time, a suboptimal algorithm of complexity $O(n)$ is used. The disadvantage with this approach is that some rule instantiations will be unnecessarily inhibited from firing, which would result in less concurrency. However, this run time strategy coupled with optimising transformations of rules is found to be effective. Kuo, Miranker and Browne [37] prove that parallel firing of interfering rules is not serializable without run time checking. So, run time checking of rule instantiations is a must for enhanced parallelism in multiple rule firing systems. Raschid, Sellis and Lin [74] report a scheme for concurrent execution of productions in a database implementation. They ensure serializability of the concurrent execution by implementing a modified two phase locking protocol. Their system does not implement any run time checking of relation tuples and hence it allows less amount of concurrency. Further their scheme is prone to deadlocks.

6.3 Synchronisation through Access Control

The problem of rule synchronisation (i.e. inhibiting interfering rules from firing concurrently) in multiple rule firing models can be viewed as a problem of maintaining working memory consistency when multiple rule instantiations are being fired concurrently. This has a direct analogy to concurrency control in database operations. Hence, the interference problem can be viewed as the problem of maintaining working memory consistency when multiple rule instantiations are being fired concurrently. The interference analysis technique presented here achieves rule synchronisation, wherever necessary, through access control of working memory, using request lists. Rule instantiations to be fired concurrently are selected using the request lists. This algorithm is deadlock free.

The algorithm has used the concepts of alpha memories, strong and weak pattern elements. The definitions of these concepts are given below.

In evaluating a rule system, match algorithms maintain an index structure called an alpha memory, for each CE in the rule system. An alpha memory provides a fast access to those WME's that satisfy each CE independent of the satisfaction of other CE's [52].

A pattern element e_1 is said to be weaker than another pattern element e_2 if

1. there exists at least one attribute in e_2 which is not present in e_1 and
2. for every attribute a_i in e_1 and e_2, a_i possesses either the same value or is assigned to an variable in e_1 and
3. every attribute of e_1 is present in e_2.

For example in the two patterns below
e_1: (candidate-city name <y>)
e_2: (candidate-city name <x> state New- York)

e_1 is the weaker pattern. Alpha memory of pattern e_1 is said to be weaker alpha memory when compared to the alpha memory of e_2. A working memory element 'w' matching a pattern 'e', will match all patterns weaker than 'e'. Hence 'w' will also become a member of alpha memories of all patterns weaker than V.

6.3.1 Access Requests

Rule instantiations can make three types of requests for access to working memory elements and alpha memories. They are

1. shared access request,
2. exclusive access request, and
3. shared-exclusive access request.

The requests are similar to lock requests in database environments. However, unlike a database locking scheme, all requests for access to an entity (working memory element or alpha memory) are accepted and queued in the RqJist (Request list) of the entity. Queuing in the list does not grant access to the entity, but only expresses a possibility of access being granted to the requesting rule instantiation. If a rule instantiation has both a shared request and an exclusive request on the same entity, the two requests are combined into a single shared-exclusive request. A shared request is said to be in conflict with an exclusive or a shared-exclusive request on the same entity. Similarly, an exclusive request is said to be in conflict with a shared or a shared-exclusive request on the same entity

After the match phase, shared access requests are queued in the RqJists of all working memory elements matching the positive condition elements. Shared access requests are also queued in the Rq_lists of the alpha memory of each negated condition element. The addition of a working memory element to an alpha memory of a condition element weaker than the negated condition element, will affect the match of the negated condition element of the rule instantiation. Hence shared access requests are also queued in the Rq_lists of all alpha memories whose pattern elements are weaker than the negated condition element. Similarly, for each rule instantiation, exclusive access requests are queued in the Rq_lists of all working memory elements to be deleted. Exclusive access requests are queued in the Rq_lists of alpha memories of CE's matching the positive action elements of the rule instantiation and alpha memories of CE's weaker than the alpha memories matching the positive action elements.

The select phase of the inference cycle is replaced by the interference analysis algorithm. The input to the algorithm is the conflict set *CS* produced by the match phase and the output is a subset of *CS* called the select list *SL*. The rule instantiations in *SL* are selected such that there is no cycle of interference in *SL*. This ensures that when the rule instantiations in SL are fired concurrently there will be a equivalent serial execution order. The complete algorithm is presented below.

6.3.2 The Interference Analysis Algorithm

The algorithm was developed with the assumption that every working memory element and alpha memory is accompanied by a tag. This tag can be set as either *shared, exclusive or shared exclusive,* depending upon the type of access request. A shared tag indicates that a rule instantiation in *SL* possesses a shared request on the entity. Similarly, an exclusive tag indicated that a rule instantiation in *SL* possesses an exclusive access request on the entity.

The algorithm has been divided into separate phases viz. *'request'* phase, *'select'* phase and *'post firing'* phase. In the *'request'* phase, requests are entered into the RqJists of the necessary working memory elements and alpha memories. Rules to be considered for inclusion in *SL* are assumed to be unmarked. Rules considered by *'select'* phase are marked. In the *'select'* phase, a set of non conflicting rules are selected. In the *'post firing'* phase, house keeping jobs are carried out. The algorithm is presented below.

begin algorithm

Request phase

1. For every rule instantiation r_i in the conflict set CS do steps 2 to 5.
2. For each WME w_j, matching a positive CE 'pCE' of r_i add a shared request on behalf of r_i in the RqJist of w_j.
3. For each negated CE 'nCE' of r,, add shared requests on behalf of r_i in the RqJist's of the alpha memory of nCE and other alpha memories weaker than the alpha memory of nCE.
4. For each negative AE 'nAE' of r_i,, add an exclusive request on behalf of r_i to the Rq_list of the WME to be deleted.
5. For each positive action element 'pAE' of r_i, add an exclusive request on behalf of r_i to the RqJist's of the alpha memories of negated conditions elements, matching or weaker than pAE.

Select phase

1. While there exists an unmarked rule in *CS* do steps 2 to 5.
2. Select an unmarked rule instantiation r_i in the conflict set *CS.*
3. For r_i do steps 4 to 6.
4. Add r_i to the select list *SL,* if r_i satisfies at least one of the following three conditions.
   (a) all the working memory elements and alpha memories requested by r_i are not tagged.
   (b) each working memory element or alpha memory requested by r_i in shared mode is not tagged in exclusive mode.
   (c) each working memory element or alpha memory requested by r_i in exclusive mode is not tagged in shared mode.
5. If r_i is entered into the select list *SL,* then for each entity(working memory element or alpha memory) in whose Rq_list r_i is a member do,
   (a) if the entity has no tag, then tag the entity with the request made by r,.
   (b) if the entity is already tagged and the tag is conflicting with the request made by r_i, then make the entity's tag 'shared-exclusive'.
   (c) if the entity is already tagged and the tag is same as the request made by r_i, then proceed to next entity.
   (d) if the entity has a 'shared-exclusive' tag, then proceed to next entity.
6. Mark r_i as 'considered' and go to step 1.

Act phase (of inference cycle)

1. fire all rules in *SL* concurrently.

Post firing phase

1. For every rule instantiation r_i fired, remove all requests made by r_i in the respective Rq_lists. Also, remove the tags of all working memory elements and alpha memories marked by r_i.

end algorithm

The *'post firing' phase* has to be carried out after all rules in the select list *SL* are fired.

6.3.3 Example

The selection of rule instantiations by the *'select'* phase of the algorithm is demonstrated with the help of an example. It is assumed that the conditions elements in each production are numbered sequentially from CEl to CEn where 'n' is the number of condition elements in the production. Consider the following productions

P1: (P one-one-out-P1
(stage reduce-candidates)
(junction junction-ID <x> line-ID-1 <1- ID>)
(junction junction-ID {<y> <> <x>} line-ID-1 <1-ID>)
(labelling-candidate junction-ID <x> line-1 out)
-(labelling-candidate junction-ID <y> line-1 in)
—>
(remove 4)
)

P2: (P two-two-minus-P2
(stage reduce-candidates)
(junction junction-ID <x> line-ID-2 <1- ID>)
(junction junction-ID {<y> <> <x>} line-ID-2 <1-ID>)
(labelling-candidate junction-ID <x> line-2 -)
- (labelling-candidate junction-ID <y> line-2 -)
—>

(remove 4)

)

Consider the working memory to be

- W1: (stage reduce-candidates)
- W2: (junction junction-ID J1 line-ID-1 11 line-ID-2 12)
- W3: (junction junction-ID J2 line-ID-1 11 line-ID-2 13)
- W4: (labelling-candidate junction-ID Jl line-1 out line-2 -)
- W5: (junction junction-ID J3 line-ID-1 11 line-ID-2 12)
- W6: (junction junction-ID J4 line-ID-1 11 line-ID-2 13)
- W7: (labelling-candidate junction-ID J3 line-1 out line-2 -)

With this state of the working memory there are four instantiations created two for each rule; They are

- I1: {P1, W1,W2,W3,W4}
- I2: {P2, W1,W2,W3,W4}
- I3: {P1, W1,W5,W6,W7}
- I4: {P2, W1,W5,W6,W7}

The requests entered by the four instantiations after the match phase in RqJists of different entities are shown below.

- RqJist(Wl) : (I1,s), (I2,s), (I3,s), (I4,s)
- RqJist(W2) : (I1,s), (I2,s)
- RqJist(W3) : (I1,s), (I2,s)

- RqJist(W4) : (ll,se), (I2,se)
- RqJist(W5) : (I3,s), (14,s)
- RqJist(W6) : (I3,s), (I4,S)
- RqJist(W7) : (I3,se), (I4,se)
- Rq_list(Alpha memory of CE5 of P1) : (I1,s), (I3,s)
- RqJist(Alpha memory of CE5 of P2) : (I2,s), (I4,s)

Assume that in the *'select'* phase of the algorithm, instantiation I1 is considered first. I1 is entered into the select list *SL*. I1 now tags working memory elements Wl, W2, W3, W4 and the alpha memory of CE5 of PL The tags would be

- Wl : s
- W2 : s
- W3 : s
- W4 : se.
- Alpha memory of CE5 : s

I2 is considered next, it does not satisfy any of the three conditions listed in step 3 of the *'select'* phase of the algorithm due to conflicting entries in the RqJist of W4. Now I3 is considered. It satisfies condition b and c of step 3 of the *'select'* phase of the algorithm. So it is entered into *SL*. Working memory elements W5, W6, W7 and alpha memory of CE5 of P2 are now tagged. The tags would now be

- Wl : s
- W2 : s
- W3 : s
- W4 : se.

- W5 : s
- W6 : s
- W7 : se
- Alpha memory of CE5 : s

Then instantiation I4 is considered and not entered into the select list *SL* because it's request for working memory element W7 is conflicting with the tag of W7. So, the select list SL now has instantiations I1 and 13. The two instantiations are fired in the firing cycle. If instantiation I2 was considered first then I1 will be discarded and any one of the instantiations I3 or I4 would be selected. This example shows that the initial instantiation selected for inclusion in SL has a bearing on the subsequent instantiations selected. In order to select a maximal subset of the conflict set as the select list, the initial instantiation should be selected such that it results in the maximal subset being chosen. However this is an intractable problem [81] and an arbitrary selection is to be made.

6.3.4 Proof of Correctness

It is proved in [37] that the concurrent firing of a set of rule instantiations is serializable if there is no cycle of interference in the set. For example the equivalent serial execution for the dependency chain shown in Figure 6.2, is $P_n, P_{n-1}, ..., P_1$. The proof that the selection criteria in the *'select'* phase ensure that there is no cycle of interference in the select list *SL* is presented below.

Theorem 1: The select list *SL* does not contain any cycle of interference.

Proof:

Assume that there is only one rule instantiation r_1 in *SL*. *SL* does not possess a cycle of interference now. Let us assume that another rule instantiation r_2 is added to *SL* by the *'select'* phase and a cycle of interference has resulted due to the

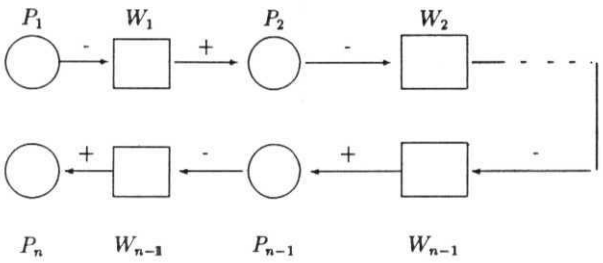


Figure 6.2: Chain of dependencies in productions

addition. There are four possible ways in which interference can result between the two rules r_1 and r_2. They are

1. a working memory element w_1 is '-' changed by r_1 is '+' referenced by r_2 and another working memory element w_2 is '-' changed by r_2 and '+' referenced by r_1.
2. a working memory element w_1 is '-' changed by r_1 is '+' referenced by r_2 and another working memory element w_2 is '+' changed by r_2 and '-' referenced by r,.
3. a working memory element w_1 is '+' changed by r_1 is '-' referenced by r_2 and another working memory element w_2 is '-' changed by r_2 and '+' referenced by r_1.
4. a working memory element w_1 is '+' changed by r_1 is '-' referenced by r_2 and another working memory element w_2 is '+' changed by r_2 and '-' referenced by r,.

These possibilities are represented in Figure 6.3.

In case (1), r_1, r_2 make the following entries in the RqJists of w_1 and w_2.

Rq_list(w_1): (n, e), (r_2, s)
Rq_list(w_2): (r_1, s), (r_2, e)

If r_1 is entered earlier into *SL* as assumed first, then the working memory element w_1 would be tagged as exclusive and the working memory element w_2 would be tagged as shared. Next, if r_2 is considered, it would not satisfy conditions a, b or c in step 4 of the *'select'* phase of the algorithm. So, r_2 would not be entered in *SL*.

In case (2), r_1, r_2 make the following entries in the RqJists of w_1 and w_2.

Rq_list(w_1): (r_1,e), (r_2, s)
Rq_list(α-mem of w_2): (r_1, s), (r_2, e)

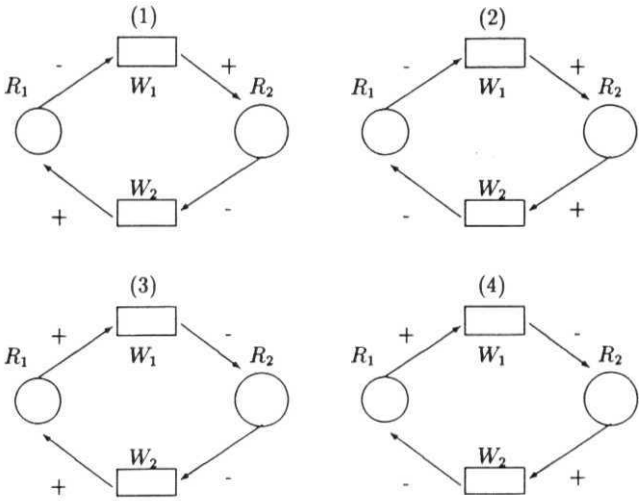


Figure 6.3: Four possible modes of interference between two productions

Since, the working memory element w_2 is being '-' referenced by r_1, i.e. by a negative condition element nCE of r_1, the shared request of r_1 has to be made on the alpha memory of w_2 and weaker alpha memories. Similarly since w_2 is being '+' changed (added to working memory) by a positive action element pAE of r_2, the exclusive request of r_2 has to be made on the alpha memory of w_2 and weaker alpha memories. For simplicity we have assumed that there are no weaker alpha memories (this assumption will not affect the proof). Hence, in this case the above requests are made.

If r_1 is entered first into *SL* as assumed first, then the working memory element w_1 would be tagged as exclusive and the alpha memory corresponding to w_2 would be tagged as shared. Next, if r_2 is considered, it would not satisfy conditions a, b or c in step 4 of the *'select'* phase of the algorithm. So, r_2 would not be entered in *SL*.

In case (3), r_1, r_2 make the following entries in the RqJists of w_1 and w_2.

Rq_list(α-mem of w_1): (r_1, e), (r_2, s)
Rq_list(w_2): (r_1,s), (r_2, e)

If r_1 is entered first into *SL* as assumed first, then the alpha memory corresponding to w_1 would be tagged as exclusive and the working memory element w_2 would be tagged as shared. Next, if r_2 is considered, it would not satisfy conditions a, b or c in step 4 of the *'select'* phase of the algorithm. So, r_2 would not be entered in *SL*.

In case (4), r_1, r_2 make the following entries in the RqJists of w_1 and w_2.

Rq_list(α-mem of w_1): (r_1, e), (r_2, s)
Rq_list(α-mem of w_2): (r_1, s), (r_2, e)
If r_1 is entered first into *SL* as assumed first, then the alpha memory corresponding to w_1 would be tagged as exclusive and the alpha memory corresponding to w_2 would be tagged as shared. Next, if r_2 is considered, it would not satisfy conditions a, b or c in step 4 of the *'select'* phase of the algorithm. So, r_2 would not be entered in *SL*.

Hence, in all the four possible cases of cyclic interference, r_2 would not be entered into *SL,* by the *'select'* phase of the algorithm if r_1 is already entered. So, if r_2 is entered in *SL,* it implies that there is no cyclic interference between r, and r_2.

By induction, let the rules $\{r_1, r_2, ..., r_k\}$ be selected by *SL.* Assume that there is no cycle of interference. Also assume that the addition of r_{k+1} by the algorithm causes a cycle of interference. This implies r_{k+1} has an exclusive request on a working memory element (alpha memory) which is already tagged as shared by some rule instantiation present in *SL,* and that r_{k+1} has a shared request on a working memory element (alpha memory) tagged as exclusive by some rule instantiation in *SL.* So, r_{k+1} will not satisfy either conditions a,b or c of step 4 of the *'select'* phase. Hence r_{k+1} would not entered into *SL.* So, if r_{k+1} is entered into *SL,* it implies that no cycle of interference is formed. Hence the theorem.

Theorem 2: The rule instantiations in the select list *SL* are serializable.

Proof: It follows from theorem 1 that rule instantiations in *SL* do not form a cycle of interference. Hence from [37] instantiations in *SL* are serializable.

Theorem 3: The *'select'* phase of the algorithm takes 0(n) time for 'n' instantiations in the conflict set.

Proof. The *'select'* phase consists of two operations viz. the verification of tags in step 4 and tagging entities in step 5 for every rule instantiation. For a rule instantiation, the number of working memory elements, matching each positive condition element and negative action element, is one. For each negated condition element and positive action element, the number of weak alpha memories in a production system program is constant. Hence, the number of tag verifications for any instantiation of a given production is constant. So, the verification process would be a constant time operation for each instantiation. Hence for 'n' instantiations the time required would be 0(n). The tagging of working memory elements and alpha memories by the selected rule instantiation takes again constant time. Hence the *'select'* phase takes 0(n) time.

The proposed access control scheme is deadlock free. Every rule instantiations

requesting access is queued and need not wait to be queued in the Rq_list. A rule instantiation is blocked by the algorithm from firing only when it interferes with another rule instantiation that is already selected for firing. Hence, the access control scheme in the algorithm is free from deadlocks.

6.4 Interference Analysis in REX

OPS5 production systems are designed to suit the efficient implementation of RETE match algorithm [53]. These semantics make interference a complex task. It has been argued in [53] that OPS5 is not the only production system and other production systems with semantics to meet differing needs may have to be designed. REX has been designed to meet the specific requirements of real time expert systems dealing with continuous streams of input data. Hence, the semantics of data and rule objects in REX are different from those of OPS5. The interference analysis algorithm presented in the earlier section has to be modified to meet the REX semantics.

In, REX there is no deletion of object instances already created. Every rule action will perform updates to attribute values in the Attribute Table. This results in new object instances in the Object Instance space of Work Area. In REX, all rules are matched against the current contents of the Attribute Table. Premises of Spanning rules though match against Object Instance space, but perform updates only to the Attribute Table. So, it would be sufficient if the integrity of the **Attribute** Table is maintained during multiple rule firings. The operations performed by the rules on the Attribute Table are *'read'* during premises match and *'update'* during rule firing.

The dependency graph of rules in a REX rule base can be constructed from the following primitives.

- A *Rule node (R-node)* representing a matched rule. It is shown as a circle in the graph.
- A *Attribute node (A-node)* representing an object attribute defined in the

object taxonomy. This node is shown as squares in the graph.

- A *directed edge from a R-node to a A-node* representing the fact that the R-node's rule updates the value of the attribute corresponding to the A-node. The edge is marked V, to indicate update operation.

- *Adirected edge from a A-node to a R-node* representing the fact that the attribute represented by the A- node is referenced by the premises of the rule corresponding to the R-node. The edge is marked V, to indicate an read operation.

The *All Rules Condition* described in the earlier section has to be restated for REX semantics. The restatement of the *All Rules Condition* is :
Let $R_1, R_2, ..., lR_{i+1}, .., R_n R_{n+1}$, *where* $R_{n+1} = R_n$ *be acyclic sequence of matched rules. Cyclic interference is said to exist, if for all i, there exists an attribute thai is 'updated' by* R_i *and is 'read' by* R_{i+1}. *Interference is said to occur between two rules* R_A *and RB, if there exists cyclic interference that includes* R_A *and* R_B.

6.4.1 The Algorithm for REX

Unlike, OPS5 there are no alpha memories in REX. There is only an Attribute Table. Further, it is not necessary to maintain separate RqJists for each attribute. The *shared request* count and *exclusive request* count entries in the Attribute Table will be sufficient. In this algorithm, the Attribute Table and Rule Table are used. The structures of these two tables are repeated in Tables 6.1 and 6.2 for ready reference. The input to the algorithm is the Matched Rule Set (MRS) and the output is Eligible Rule Set(ERS). MRS is similar to the conflict set CS of OPS5 and ERS is similar to the *Select List (SL)* defined in the earlier section. The tags of attributes are maintained as the shared request count and exclusive request counts in the Attribute Table. For each rule in REX, a list of attributes that are read in the premises and another list of attributes that are updated by the actions are maintained. These are used instead of maintaining RqJists for each attribute.

In the algorithm for REX, there would be only two phases. They are the *'select'*

Att. No.	Att. name	Type	Attribute® time	value	shared request count	exclusive request count	update flag

Table 6.1: Structure of the Attribute Table

Rule id.	Priority	Read List	Write List	Premise id.s	Action id.	Event id.	Hold id.

Table 6.2: Structure of Rule Table

and *'post firing'* phases.

begin algorithm
Select phase

1. While there exists a unmarked rule in *MRS* do steps 2 to 5.
2. Select an unmarked rule r_i in the Matched Rule Set *MRS.*
3. For r_i do steps 4 to 8.
4. Collect the attributes referenced by the premises of r_i in the set READ-SET (these are obtained from the Read List column of the Rule Table).
5. Collect the attributes updated by the actions of r_i in the set WRITE-SET (these are obtained from the Write List column of the Rule Table).
6. Add r_i to the Eligible Rule Set *ERS,* if r_i satisfies at least one of the following conditions.
   (a) for each attribute in READ-SET, the corresponding exclusive request count in the Attribute Table is zero.
   (b) for each attribute in WRITE-SET, the corresponding shared request count in the Attribute Table is zero.
7. If r_i is entered into the Eligible Rule Set *ERS,* do
   (a) for each attribute in the READ-SET, increment the corresponding shared request count in the Attribute Table

(b) for each attribute in the WRITE-SET, increment the corresponding exclusive request count in the Attribute Table

8. Mark r_i as 'considered' and go to step 1.

Post firing phase

1. For every rule r_i fired, decrement the shared request count for all attributes in the Read List of the rule. Similarly, decrement the exclusive request count for all attributes in the Write List of the rule.

end algorithm

The proof of correctness of the algorithm in the earlier section (for OPS5) can be extended to this algorithm.

6.4.2 An Example

The functioning of the algorithm has been explained with the following two interfering rules. The dependency graph is in Figure 6.4.

R1:
Priority: 1
Premise : $A.a_1 < 30$; $B.b_2 > 40$
Action : $D.d_1 = 100$

R2:
Priority: 2
Premise : $D.d_1 < 50$
Action : $A.a_1 = 100$

The Attribute Table and the Rule Table for the two rules are in Tables 6.3 and 6.4. The Attribute Table represents the initial state.

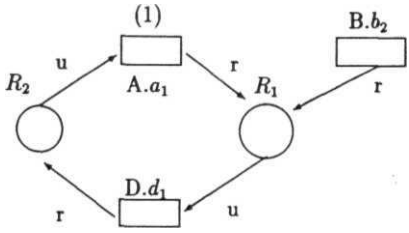


Figure 6.4: Dependency graph for rules Rl and R2

Att. No.	Att. name	Type	Attribute@ time	value	shared request count	exclusive request count	update flag
1	A.a_1	int		25	0	0	
2	B.b_1	int		5	0	0	
3	B.b_2	int		46	0	0	
4	D.d_1	int		12	0	0	

Table 6.3: Structure of the Attribute Table

Rule id.	Priority	Read List	Write List	Premise id.'s	Action id.'s	Event id.'s	Hold id.
1	1	1,3	4	1,2	1		
2	2	4	1	3	2		

Table 6.4: Structure of Rule Table

Att. No.	Att. name	Type	Attribute® time	value	shared request count	exclusive request count	update
1	$A.a_1$	int		25	1	0	
2	$B.b_1$	int		5	0	0	
3	$B.b_2$	int		46	1	0	
4	$D.d_1$	int		12	0	1	

Table 6.5: Structure of the Attribute Table after R1 is entered in ERS

With, the initial state of the Attribute Table , both the rules' premises have been satisfied and hence they together form the Matched Rule Set (MRS).

MRS = {1,2}

Now the Interference Analysis Task(IAT) will have to run the algorithm presented above to detect interference among rules in the MRS. The algorithm has to select one of the rules in MRS to start with. Since, REX is a real time expert system tool, it has been decided to consider the highest priority rule first. Hence, in this example, the rule with Rule id. 1 has to be considered first. Since the shared request counts of attributes $A.a_1$ and $B.b_2$ are zero, the rule satisfies condition 'a' of step 6 in the *'select'* phase. So, it is entered in the Eligible Rule Set(ERS) and the shared and exclusive request counts of the appropriate parameters are incremented. The Attribute Table would now be as in Table 6.5.

ERS = {1}

Next, as rule 2 is the only rule left in MRS, it is considered next. For rule 2 to be included in the ERS, either the exclusive request count of attribute $D.d_1$ should be zero, or the shared request count of attribute $A.a_1$ should be zero. Neither of these counts is zero. Hence, rule 2 would not satisfy the conditions a and b, in step 6 of the *'select'* phase. So, rule would not be entered into ERS. So, ERS will consist of only one rule i.e. rule 1 and it would be scheduled for firing by the *Rule Firing Scheduler*.

6.5 Asynchronous Rule Firing

The rule firings in REX update the Attribute Table. The rules in the *Eligible Rule Set (ERS)* have the property of maintaining Attribute Table integrity without any synchronisation measures, when they are fired concurrently. Hence, the rules in ERS can be scheduled for asynchronous firing. In, REX the actions of every rule are compiled into executable code and this code would be executed by the rule firing tasks. The priorities of the different rule firing tasks would be based on the rule priorities. The rule firing task of the rule with the highest priority among rules in ERS would have the highest priority among the rule firing tasks.

During the match and interference analysis phases, the Attribute Table is locked and external data updates by *Attribute Update Task* would not be allowed. This has been done because the MRS and ERS have to be consistent with reference to the Attribute Table contents. Rule firing tasks perform updates to Attribute Table. The set of attributes updated by the *Rule Firing Tasks* and the set updated by the External Data Interface *(Attribute Update Task)* are disjoint (sensor data cannot be updated by rule firings). Hence asynchronous updates by the *Rule Firing Tasks* and *External Data Interface* could be allowed. So the Attribute Table is unlocked in the rule firing phase and asynchronous updates by the *Rule Firing Tasks* and *Attribute Update Task* would be allowed.

6.6 Summary

In this chapter, a new interference analysis technique has been presented. It is based on access control of working memory. The algorithm detects cyclic interference between rules in the conflict set. If any cyclic interference is detected, it inhibits interfering rules from firing concurrently. Access requests are queued in request lists of respective entities, and these are used to detect interfering rules and rules for firing. The interference analysis process is deadlock free and does not block any rule in the interference analysis phase.The proof of correctness of the algorithm has been presented. An example of the functioning of the algorithm has been given. This

algorithm was adapted to suit REX semantics. An example of the REX interference algorithm has also been presented. Finally, the asynchronous rule firing scheme in REX is discussed. In the next chapter, some of the significant implementation aspects of REX are presented.

Chapter 7

IMPLEMENTATION ASPECTS

7.1 Introduction

In the preceding chapters, the design aspects of REX, which include the object oriented data and knowledge representation scheme, the match algorithm, the interference analysis technique and the asynchronous rule firing policy have been discussed. The implementation decisions play a major role in shaping up the designed system. In this chapter, some of the important aspects of REX implementation are presented. In the next section, the main memory data structures, secondary storage structures and techniques used by the *Object Manager* in implementing the object oriented data and knowledge management functions are presented. In Section 7.3, the Work Area data structures and the function defined for implementing the match algorithm are presented. This is followed by interference analysis. The user interface has a major part in the effective presentation of a system. REX is provided with a X - window based interface, developed using X Toolkit and Athena Widgets. The interface is presented in Section 7.5.

The object oriented data and knowledge representation scheme forms the backbone of REX. To, facilitate its implementation, an object oriented language is necessary. Further, as REX has to be integrated into a target environment, it is necessary for the language to provide the needed interface. So, C++ language is chosen for the implementation. The total system is on a computing platform with Lynx OS. Lynx OS is a UNIX like real time OS. It provides the POSIX thread facilities, which are used in implementing asynchronous tasks.

7.2 Object Manager

An unified object oriented paradigm has been used to store data and knowledge in REX. So, the *Object Manager* has to perform both data and knowledge management tasks.

7.2.1 Data Management

The Object Manager in REX provides facilities to

- create and maintain the object schema
- store and retrieve objects from secondary store
- create and maintain object instances in main memory

The evolution of a schema in REX is through the definition of classes and the hierarchical relationship among the classes. Each class consists of attributes and methods to manipulate those attributes. The attributes of the class can be of the types integer, float, string or another class. The user can also define facets *timeout, default value and legal range of values* for the class attributes. Multiple inheritance is supported. In case, of an ambiguity regarding the inheritance of an attribute, there is a provision to specify from which class the attribute has to be inherited.

The following are the functions available to create the object schema.

Creating a class:

Addclass(classname);
Addclass(classname,superclass string);

Creating an attribute and defining the default value and legal values:

Addattr(classname,attrname,type,timeout);
Defaultvalue(classname,attrname,defaultvalue string);
Legalvalues(classname,attrname,legalvalues string);

Creating a method:

Add_method(classname,method);

Resolve conflict:

Inherit_from(classname,attrname,from-classname);

Deleting a class:

Dropclass(classname);

Deleting an attribute:

Dropattr(classname,attrname);

Deleting a method:

Del_method(classname,method);

The schema so defined using the above functions is stored in a file called *schema definition file.* If there are any modifications to the schema, the file is updated to reflect the changes. The *Schema Definition File* consists of 2 parts the *the index block* and *the data block.*

Each index record pertains to a class and provides access to the class related data like attributes,methods etc., stored in the data blocks. The *index block* is of size 2K and consists of a

- header and
- index records

The *index block header* contains information about the number of classes in taxonomy, last object identifer allotted and the base classes. The index block contains essential and minimal information about a class and the location of the class record in the file. The record structure and sizes of the fields in the *index block* are shown

in the Figure 7.1. The data block has detailed information regarding the class attributes, their properties, default values and legal values. The structure of the *data block* is shown in Figure 7.2.

From, the schema definition, a C++ class declaration file is generated for the object taxonomy. These declarations are used by the *Object Manager* for managing the object instances in the main memory. A partial set of class declarations generated for the schema depicted in Figure 7.3 are,

```
class OBJECT
{
public :
    long clock;
};


class WSS : public OBJECT
{
public :
    float speed;
    String direction;
    WSS() {speed = 0; direction = NULL;}
    WSS(long clockl, float speedl, String dir)
        {
            clock = clockl;
            speed = speedl;
            direction = dir;
        }
};
```

The object instances for each class are stored in a separate *doubly linked list* in the main memory. The latest object is inserted at the head of the list. Each node in the linked list consists of an Object Identifier (OID), a pointer to the corresponding

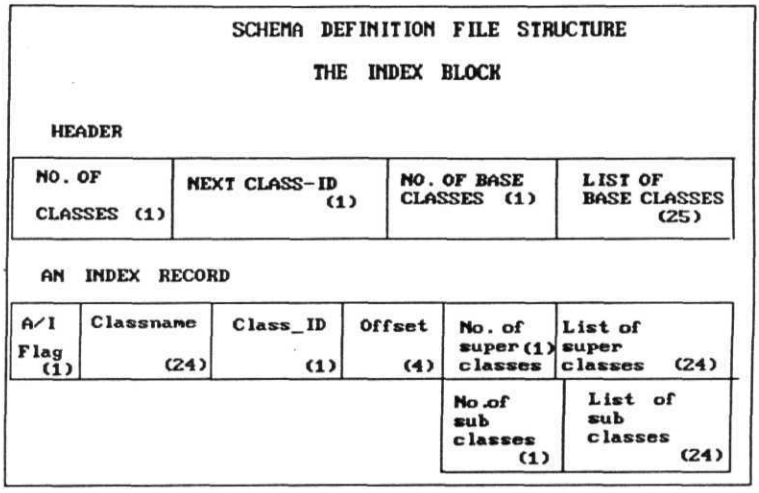


Figure 7.1: Index Block of the Object Schema File

SCHEMA DEFINITION FILE STRUCTURE

THE DATA BLOCK

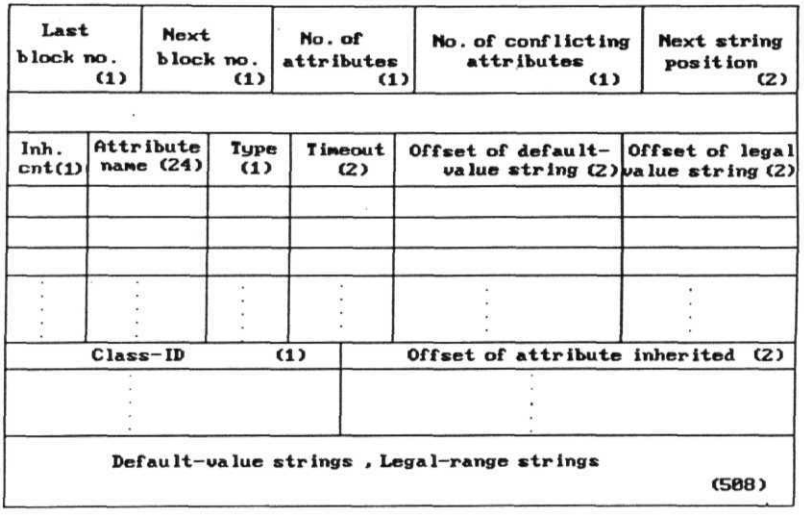


Figure 7.2: Data Block of the Object Schema File

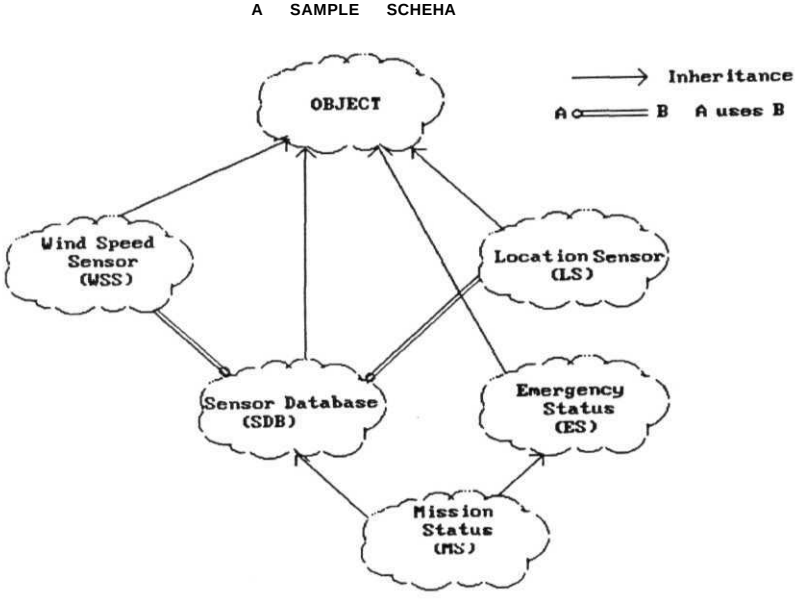


Figure 7.3: Example Schema

Ptr. to previous instance	OID	Object Ptr.	Ptr. to next instance

Table 7.1: Doubly Linked List for storing Object Instances

Record Type (1)	No. of keys in record (2)	Child Ptr. (4)	Key (2)	Child Ptr.	

Table 7.2: Structure of B+ index record

object instance and pointers to the previous and next object instances. If the number of instances exceed a fixed number, the objects at the tail of the list are dumped to the secondary storage. This linked list structure will be very useful for evaluating spanning premises. The structure of the doubly linked list is given in Table 7.1.

The Physical Storage Model used to store the instances in main memory is similar to PSM-1 [94] i.e., each instance occurs only once in the most specific class to which it belongs. A B+tree indexed file structure using the OID as the key is implemented for this purpose. The sequence set consisting of the keys (OID's), and their offsets (in the data file) is implemented as a doubly linked list.The reverse links provide access to the previous instances of the object. This will facilitate retrieval of history data needed during evaluation of spanning premises. The structure of the index and leaf records in the B+ tree are given in Tables 7.2 and 7.3.

Instances of each class defined in the taxonomy are stored in a separate file. The structure and the length of the file record depends on the class structure. Thus every class will have one data file to store instances and an B+ tree index file.

Record Type	No. of keys in record	Ptr. to left leaf	Ptr. to right leaf	Key	Data ptr.	

Table 7.3: Structure of B+ leaf record

Priority	No. of premises	Premise records	Action records	Action	Hold

Table 7.4: Storage record structure for Autonomous Rules

Autonomous rule structure	Event id.	Time limit	Time operator

Table 7.5: Storage record structure for Clock Synchronised Rules

7.2.2 Rule Management

The *Object Manager* provides the following facilities for rule management.

- create the rules
- check for syntactic correctness and completeness
- store in a rule base file
- compile the premises, spanning premises, events and actions into C++ methods
- convert the rules from the rule base into a form suitable for table creation during run time to facilitate the reasoning process.

An X-window based user interface has been provided to facilitate rule creation. Templates are provided for the four different types of rules. The rule entered by the user is parsed to detect errors. This parser is implemented using the *'lex'* and *'yacc'* utilities. The rules are stored in the secondary storage according to the rule taxonomy defined in Chapter 4 (Figure 4.1). The record structures used for storing the different kinds of rules are given in Tables 7.4 to Tables 7.7.

Autonomous rule structure	Event id.	Time limit

Table 7.6: Storage record structure for Event Spanning Rules

Autonomous rule structure	Cycle time	Time limit

Table 7.7: Storage record structure for Time Spanning Rules

This rule base storage file and the object schema file are used in creating the Work Area table structures and their initialisation.

The match phase has to evaluate the *events, premises* and *spanning premises* of the different rules to check if the rule conditions are met. This evaluation can be made faster if the premises and events are compiled rather than being interpreted. Similarly the rule actions are also be compiled for faster rule firing. Hence, after a rule is found to be syntactically correct, the events, premises, spanning premises and actions are converted to corresponding C++ functions. The methods that are generated by the *rule compiler* have the following form,

```
int event_id()
{
    code for evaluating the condition;
}

int premise_id()
{
    code for evaluating the condition;
}
```

The *id* in the generated function name is the unique identifier of the premise or event whose code is to be generated.

All the methods generated are compiled and stored in a file. This compiled code is made available for run time processing.

7.3 Match Implementation

The match algorithm used in REX is presented in Chapter 5. This algorithm uses the Attribute Table, Premise Table, Event Table, Rule Table and AEP Index Table extensively. Each table record is defined as a class. For example each entry in the Rule Table are instances of a class called RULE.REC. A constructor and I/O operators are defined for each such class. The Rule Table is itself implemented as a class whose members are instances of RULE_REC.

These data structures are defined to implement the match algorithm. The locking of the Attribute Table is done by defining a mutex made available by the operating system. The Lynx OS has thread facility. In the match process, the 'SPremise Evaluator' is scheduled as a separate thread. This thread waits on a 'condition variable'. Whenever spanning premises have to be evaluated, this condition variable is set and the thread is triggered. A list of modified attributes (alist) is provided as input to the match task. The output of the match is the mrule_list which is the MRS.

7.4 Interference Analysis

The mrule_list is the input to the interference analysis algorithm called make_request( ). This function executes the algorithm discussed in Section 6.4.1 to determine the ERS. The output is of the function is erule_list. This list is used by the rule scheduler. The rule scheduler function create separate threads to perform each rule firing. The OS facilities to create and schedule threads are used by the scheduler function.

7.5 User Interface

The user interface module is implemented using the Athena Widget set and the X Window System. The interface of REX is graphical in that the primary means of interaction between the user and the system is by manipulation of visual items

through a mouse, pull down menus and push buttons. The range of functionality of the interface includes schema design, data entry and presentation and modification.

The design of the user interface has been influenced by the following factors.

1. the user is given sufficient information to accomplish the tasks,
2. the user can control the pace of interaction with the system,
3. the user can easily correct the errors made,
4. the software performance conforms to the user's expectations

The overall user interface provided is managed by an Interface Manager. When the REX shell is initiated, the interface manager presents the user with the **Main Menu** as shown in Figure 7.4. Under the **SYSTEMS** option in the Main Menu, options are provided for configuring an expert system. The user can specify the name of the expert system to be created and configure it by specifying the names of the corresponding object schema, rule base and external input files. The other options provided in the Main Menu are the **Objects, Rules, Agents, Inference, Ext Jnput** and *Quit.*

Under the **Objects** option in the **Main Menu,** options to create a **Schema** and **Objects.** The **Schema** option provides facilities to create, browse and modify through the object taxonomy. The schema browse menu is shown in Figure 7.5. The schema modification menu is shown in Figure 7.6. The dialog boxes to create classes is shown in Figure 7.7. The **Object** option provides options to create and browse object instances . The dialog box to create object instances is shown in Figure 7.8. The object browsing screen is shown in Figure 7.9.

Under the **Rules** option in the *Main Menu,* options for creating, modifying, deleting, displaying, printing and verifying a rule are given. The templates to create an autonomous rule, a clock synchronised rule, and a time spanning rule are shown in Figures 7.10, 7.11 and 7.12 respectively.

Under **Agents** options we can define multiple expert systems as a single unit. An *agent* is an individual expert system complete with its knowledge base, object base

and inference process. The eventual aim is to create a system of cooperating agents. Further work in defining the cooperative system paradigm and its implementation is under progress.

The **Ext_Input** option is used to specify the source of external input. Using the **Inference** option , the type of inference process can be specified. Presently, only the forward chaining process is possible in REX. Help is provided through the use of **'Alt-H'** keys. The user can exit the shell by choosing the **Quit** option.

7.6 Summary

In this chapter, the discussion is concentrated on the implementation issues of the REX system. First, the implementation aspects of the *Object Manager* are discussed. This is followed by a discussion about the match and interference analysis algorithms. Finally, the user interface of REX is presented. Thus this chapter, presents a general outline of the implementation aspects of REX.

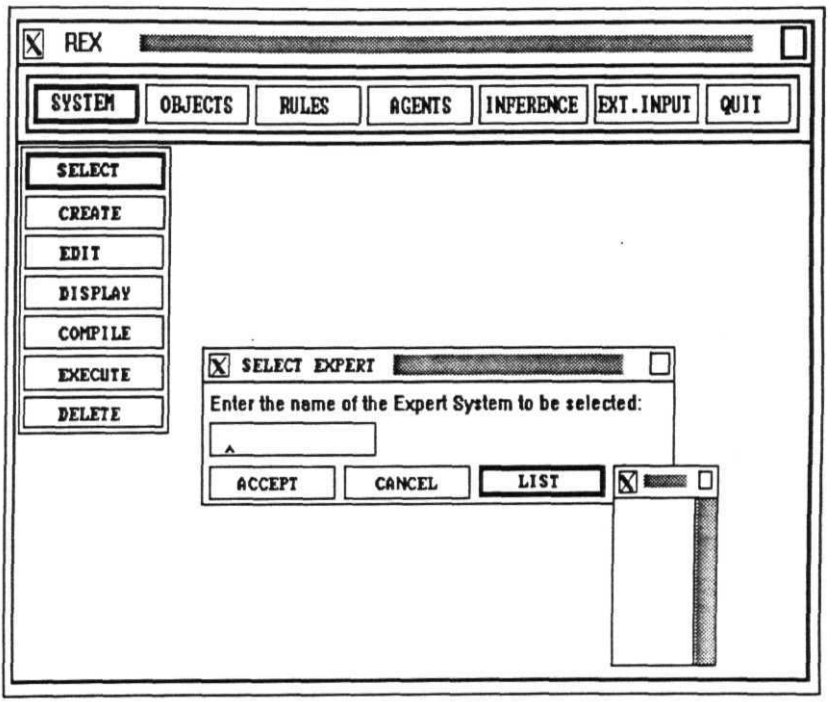


Figure 7.4: Main Menu of REX User Interface

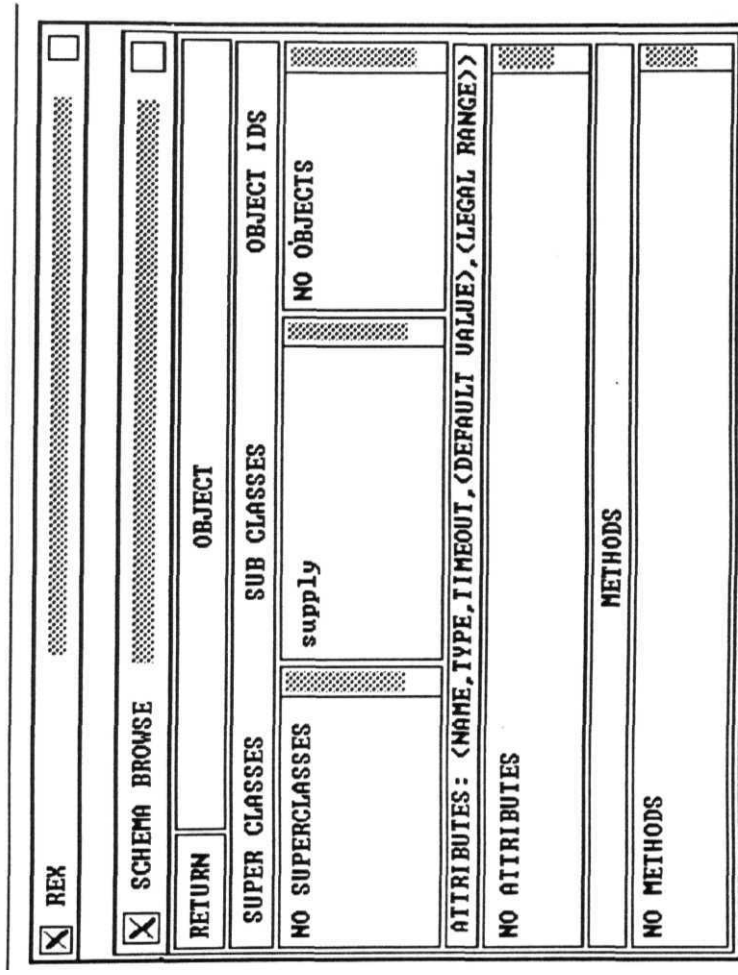


Figure 7.5: Schema Browsing Menu of REX User Interface

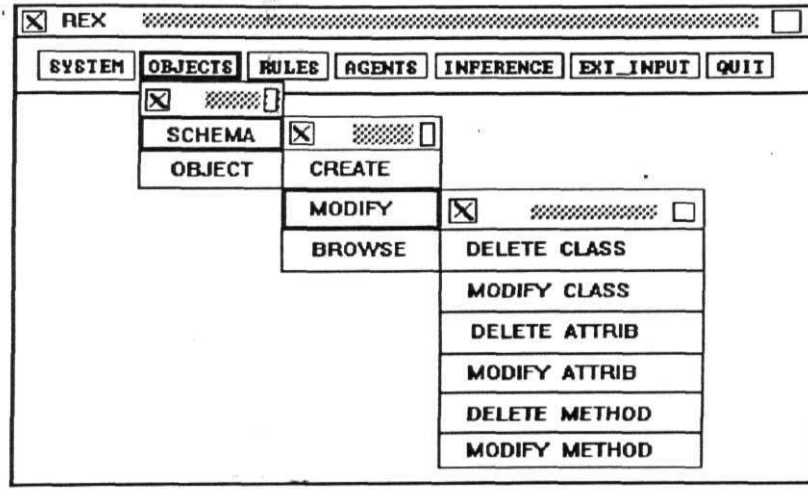


Figure 7.6: Schema Modification Menu of REX User Interface

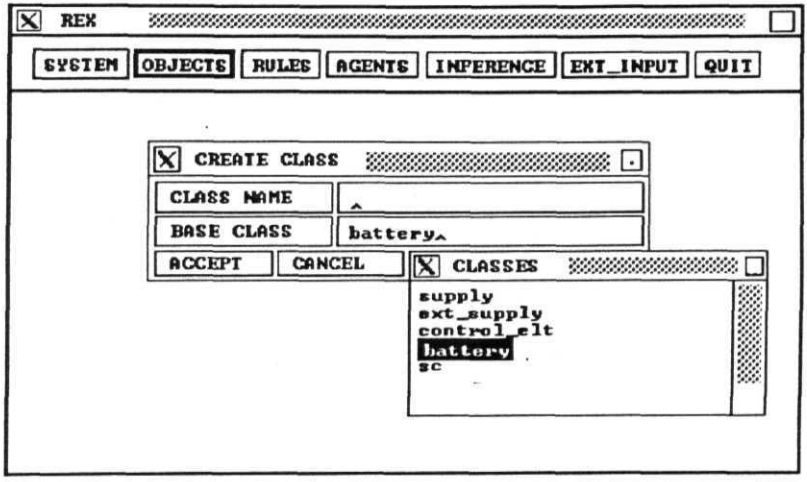


Figure 7.7: Dialog Box to create a class definition using REX User Interface

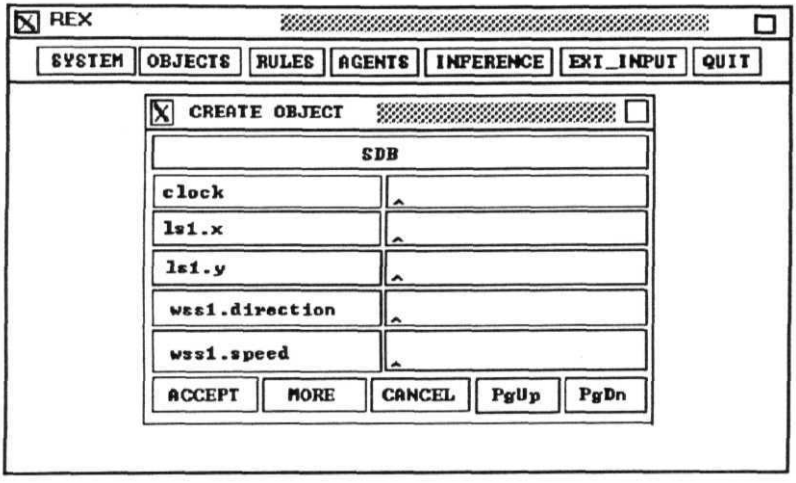


Figure 7.8: Dialog Box to create an class instance using REX User Interface

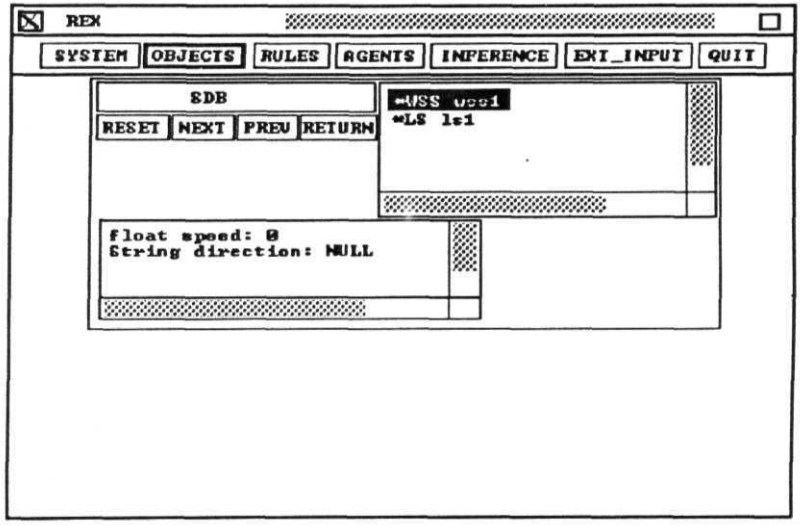


Figure 7.9: Object Browse Screen in REX User Interface

TEMPLATE ***FOR*** AUTONOMOUS RULE

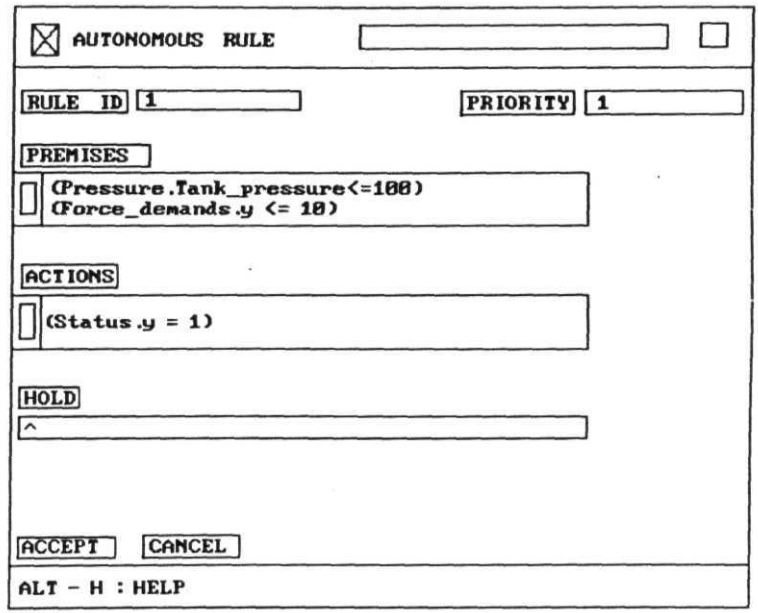


Figure 7.10: Template for creating an Autonomous Rule

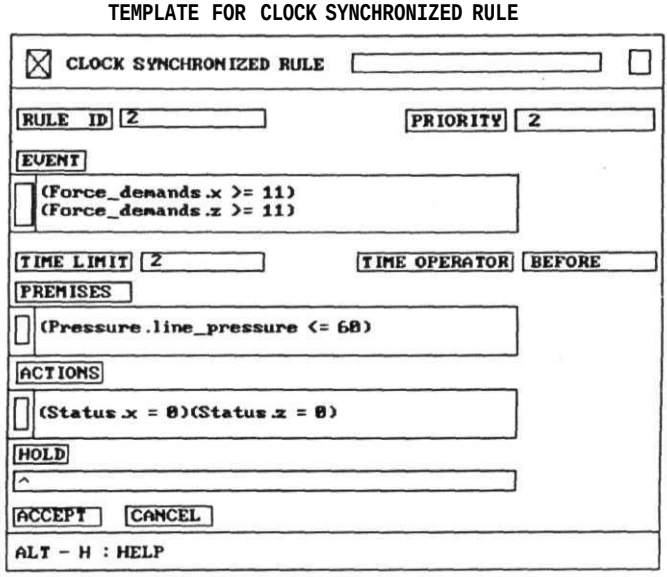


Figure 7.11: Template for creating a Clock Synchronised Rule

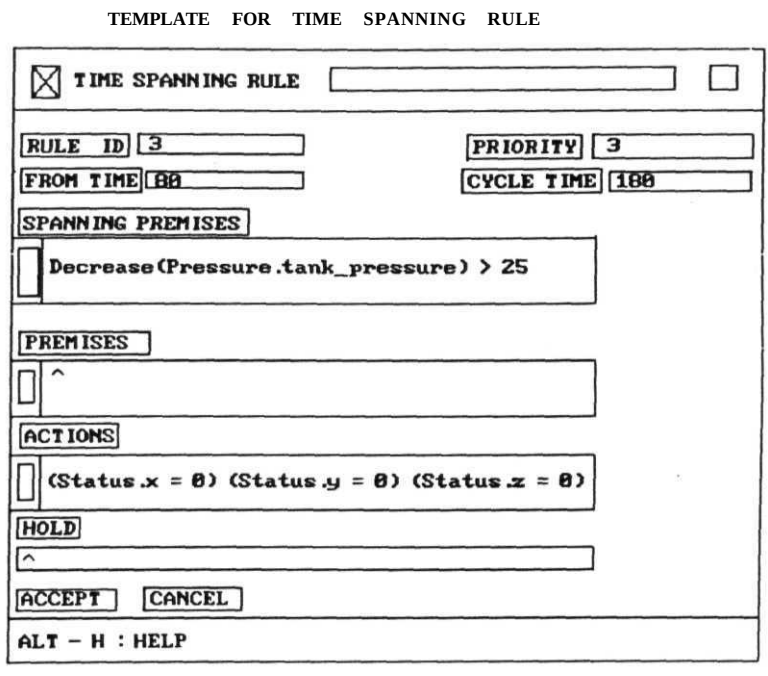


Figure 7.12: Template for creating a Time Spanning Rule

Chapter 8

CONCLUSIONS

8.1 Introduction

The need for real time expert systems is being felt increasingly and their design is receiving much attention during recent past. The significance attached to real time expert systems can be gauged by the vast and expanding literature on this subject.

In the preceding chapters, we considered an architecture and its implementation for developing real time expert system. The purpose of this chapter is to summarize the significant aspects of the previous chapters and suggest further work in this area.

8.2 A Summary of the Work

The need to design any system arises out of certain requirements not being met by similar systems already available. Real time expert systems have some specific requirements like continuous reasoning, temporal data and knowledge representation, interruptability and reactive response behaviour. These requirements could not be met by traditional expert system architectures. So, a new architecture has to be designed for real time expert system.

We started out with identifying the specific requirements of real time expert systems. This was followed by the identification of necessary features to be incorporated in the real time expert system architecture. The key requirements identified are

- Temporal Representation of Data

- Representing and Reasoning about Temporal Relationships
- Modelling Delayed Feedbacks
- Continuous Reasoning
- Simultaneous Handling of Multiple Events
- Interruptible Reasoning
- Embedded Operation
- Reactive Response
- Focusing Attention
- Handling Asynchronous Events.

Recently, many real time expert systems are being reported in literature. New knowledge representation techniques and inference strategies to address specific application requirements have been reported. A study of such techniques, tools and systems from the perspective of the above requirements was carried out. The results of this study are contained in Chapter 2.

Based on the specific requirements identified, a formal model to represent various aspects of a real time expert system has been defined. This model is an Extended Petri Net (EPN). The EPN model has the ability to abstract various aspects of real time expert systems like representation of temporal data and knowledge, knowledge base verification and reasoning. On the basis of the EPN model, an object oriented real time asynchronous production system architecture called REX has been developed. The key features of this architecture are

- an unified object oriented data and knowledge representation model.
- asynchronous multiple rule firing reasoning process.
- integration of external inputs into the reasoning process.

The proposed object oriented data and knowledge representation model is a distinguishing feature of REX. The data model has been designed to facilitate representation and storage of temporal data. Since, real time data have life spans, a 'timeout' facet has been provided to represent data life span. Unlike other expert system architectures, in which rules are defined above objects and are treated differently [9], rules in REX are treated as objects. A rule schema and taxonomy have been defined in REX. Three different rule types viz. Autonomous rules, Clock synchronised rules and Spanning rules have been defined. Autonomous rules deal with current data values. Clock synchronised rules represent knowledge about temporal relationships and deal with data to be obtained in future. Spanning rules pertain to trends in past (historical) data. These different rule types can effectively capture knowledge in real time domains.

The reasoning process in real time expert systems should meet the reactive response requirements and suit the semantics of the data and knowledge representation scheme. The reasoning algorithms in most production systems fail to meet real time response requirements. This is because of their combinatorial match phase. In order to achieve a reactive response behaviour it is necessary to reduce the amount of time spent in the match phase. This can be achieved by designing a suitable match algorithm. The design of a match algorithm should take into account the rule semantics. Most match algorithms model the match process as a sequence of Select and Join operations on the working memory. In a similar vein, REX models premise evaluations as query evaluations with Select and Join operations. The evaluation of a Spanning premise is different from that of other premises. The evaluation of a Spanning premise can be modelled as an aggregation query on a set of object instances. With these models of evaluations, anew match algorithm to suit REX rule semantics was designed. It has been shown that the number of evaluations made per attribute update is a linear function of the number of premises and events.

Real time expert systems have to deal with simultaneously occurring multiple events in the external world. If the concept of single rule firing in an inference cycle is adopted, then it will not be possible to handle multiple events occurring simultaneously. So, the concept of firing multiple rules concurrently in a single inference

cycle has been adopted in REX. If a multiple rule firing model is adopted, it is necessary to ensure the integrity of the working memory. This process in production systems is termed as interference analysis. Interference analysis techniques reported in literature are compute intense. Some techniques are prone to deadlocks. We have proposed a deadlock free interference analysis algorithm and proved its correctness. The algorithm is 0(n) complex, where 'n' is the number of matched rule instantiations. This algorithm is not specific to REX alone, but can be used to production systems like OPS5.

The current prototype is being used to implement an aerospace application.

8.3 Scope for further work

The size of the knowledge base in real time expert systems increases drastically with the increase in application complexity. In such cases, it is necessary to adopted a distributed problem solving paradigm. Two specific issues have to be addressed if such a paradigm is adopted. They are

- knowledge base partitioning and
- communication paradigm

These two factors are interwoven and can have a significant effect on the real time performance of the system. Though work on partitioning for real time expert systems is reported [7], it is necessary to study the effect of partitioning strategies on the communication. Similarly, the effect of communication paradigm on rule base partitioning have to be studied. Further work in defining the cooperative system paradigm and its implementation is under progress.

Another possible direction is in predicting the upper bound on the match time. Recently, a method that predicts a run time upper bound on the match time for RETE is reported [89]. The worst case upper bound predicted by this technique is five time more than the actual run time. Further work on developing such prediction

techniques for the REX system can be undertaken. It is also necessary to develop techniques whose predictions are more accurate.

Interference analysis is performed using serializability as the correctness criterion. It is observed that serializability is a strict criterion in real time systems. Correctness criterion based on application semantics are being proposed for real time databases [41]. The applicability of such criteria to interference analysis in real time expert systems can be studied.

Bibliography

[1] F. Barachini. The evolution of PAMELA. *Expert Systems*, 8(2), May 1991.

[2] L. S. Baum, R. T. Dodhiawala, and V. Jagannathan. The Erasmus System. In V. Jagannathan, R. Dodhiawala, and L. S. Baum, editors, *Blackboard Architectures and Applications*, pages 347-370. Academic Press, 1989.

[3] R. Bisiani, F. Alleva, A. Forin, R. Lerner, and M. Bauer. The Architecture of the AGORA environment. In M. N. Huhns, editor, *Distributed Artificial Intelligence*, pages 99-118. Morgan Kaufmann, 1987.

[4] G. Booch. *Object Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[5] B. Chandrasekharan, R. Bhatnagar, and D. D. Sharma. Real Time Disturbance Control. *Communications of the ACM*, 34(8):32-47, Aug. 1991.

[6] T. Chehire and E. Onaindia. Applying Time Map Manager in a Real Time Expert System for Alarm Filtering. In *Proceedings of IEEE Conference on Tools with Artificial Intelligence*, pages 14-21. IEEE CS Press, 1992.

[7] I.-R. Chen and B. L. Poole. Performance Evaluation of Rule Grouping on a Real Time Expert System Architecture. *To appear in IEEE Transactions on Knowledge and Data Engineering*, 1993.

[8] S. M. Chen, J. S. Ke, and J. F. Chang. Knowledge representation using Fuzzy Petri nets. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):311-319, Sept. 1990.

[9] B. Czejelo, C. F. Eich, and M. Taylor. Integrating Sets, Rules, and Data in an Object Oriented Environment. *IEEE Expert*, 8(l):59-66, Feb. 1993.

[10] H. Dai. *Knowledge based systems for real time applications - A parallel processing approach.* PhD thesis, University of Ulster, May 1991.

[11] R. Davis, H. Shrobe, and P. Szolovits. What is a Knowledge Representation. *AI magazine*, 14(l):17-33, 1993.

[12] U. Dayal, B. Blaustien, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The HiPAC Project: Combining Active Databases and Timing Constraints. *SIGMOD RECORD*, 17(l):51-70, Mar. 1988.

[13] D. Deeter and R. Stack. G2: Two thumbs up .. if you've got the cash. *IEEE Expert,* 8(2):92–94, Apr. 1993.

[14] A. Delis, L. Raschid, and T. Sellis. Experiments on the concurrent rule execution in database systems. In *Proceedings of IEEE Conference on Tools with Artificial Intelligence,* pages 405-416. IEEE CS Press, 1992.

[15] Y. Deng and S. K. Chang. A G-net model for knowledge representation and reasoning. *IEEE Transactions vn Knowledge and Data Engineering,* 2(3):295-310, Sept. 1990.

[16] R. T. Dodhiawala, N. S. Sridharan, and C. Pickering. A Real Time Blackboard Architecture. In V. Jagannathan, R. Dodhiawala, and L. S. Baum, editors, *Blackboard Architectures and Applications,* pages 219-238. Academic Press, 1989.

[17] D. Drovak and B. Kupiers. Process Monitoring and Diagnosis: A model based approach. *IEEE Expert,* 6(3):67-74, June 1991.

[18] R. O. Duda. Knowledge Based Expert Systems Come of Age. *Byte,* 6(9):238-281, Sept. 1981.

[19] M. R. Fehling, A. M. Altman, and B. M. Wilber. The Heuristic Control Virtual Machine: An Implementation of the Schemer Computational Model of Reflective Real Time Problem Solving. In V. Jagannathan, R. Dodhiawala, and L. S. Baum, editors, *Blackboard Architectures and Applications,* pages 191-218. Academic Press, 1989.

[20] E. A. Feigenbaum. Knowledge Engineering in the 1980's. Technical report, Stanford University, 1982.

[21] R. Fikes and T. Kchler. The Role of Frame Based Representations in Reasoning. *Communications of the ACM,* 28(9):904-920, Sept. 1985.

[22] C. L. Forgy. RETE: A fast match algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence,* 19(l):17-37, 1982.

[23] L. Gasser, C. Bragazan, and N. Herman. MACE: A flexible testbed for Distributed AI research. In M. N. Huhns, editor, *Distributed Artificial Intelligence,* pages 119-152. Morgan Kaufmann, 1987.

[24] W. B. Gevartor. The Basic Principles of Expert Systems. In P. G. Raeth, editor, *Expert Systems: A Software Methodology for Modern Applications,* pages 17-32. IEEE CS Press, 1990.

[25] T. V. S. Gireendranath. Knowledge base verification using petri nets. Master's thesis, Department of Computer Science, University of Hyderabad, Mar. 1989.

[26] P. E. Green. AF - A Framework for Real Time Distributed Cooperative Problem **Solving. In M. N. Huhns, editor, *Distributed Artificial Intelligence*, pages 153–176.** Morgan Kaufmann, 1987.

[27] A. Gupta. ***Parallelism** in production systems.* Morgan Kaufmann, 1988.

[28] A. Gupta and B. E. Prasad. Knowledge Representation. In A. Gupta and B. E. Prasad, editors, *Principles of **Expert** Systems*, pages 75-77. IEEE Press, 1988.

[29] W. Harvey, D. Kalp, M. Tambe, D. McKweon, and A. Newell. The effectiveness of task level parallelism for Production systems. *Journal of Parallel and **Distributed Computing,** 13:395-411, Jan. 1991.*

[30] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6):34-44, Dec. 1992.

[31] T. Ishida. Parallel Rule Firing in Production systems. *IEEE Transactions on **Knowledge and Data Engineering,** 3(1):11–17, Mar. 1991.*

[32] T. Ishida. Organising Self Development and Distributed Production Systems. ***IEEE Transactions on Knowledge and Data Engineering,** 4(2):123–134, Apr.* 1992.

[33] T. Ishida and S. J. Stolfo. Towards the parallel execution of rules in production **system programs. In *Proceedings of International Conference on Parallel** Processing*, pages 568-575, 1985.

[34] F. Jakob and P. Suslenschi. Situation Assessment for Process Control. *IEEE Expert*, 5(2):49-59, Apr. 1990.

[35] K. Kaiser, D. Blevins, B. Miller, L. S. Baum, and V. Jagannathan. Adapting the Blackboard Model for Cockpit Information Management. In V. Jagannathan, R. Dodhiawala, and L. S. Baum, editors, *Blackboard Architectures and Applications*, pages 481-500. Academic Press, 1989.

[36] S. Kao, T. J. Laffey, J. Schmidt, J. Read, and L. Dunham. Real time Analysis of **Telemetry Data. In *Proceedings of Third annual Expert systems in Government** conference*, pages 137-143. IEEE CS Press, 1987.

[37] C. Kuo, D. P. Miranker, and J. C. Browne. On the performance of the CREL **system. *Journal of Parallel and Distributed Computing,* 13:424-441, Jan. 1991.**

[38] S. Kuo and D. Moldovan. Implementation of Multiple Rule Firing Production **Systems on Hypercube. *Journal of Parallel and Distributed Computing,* 13:383–** 394, Jan. 1991.

[39] S. Kuo and D. Moldovan. The State of the Art in Parallel Production Systems. ***Journal of Parallel and Distributed Computing,* 15:1-26, 1992.**

[40] S. Kuo, D. Moldovan, and S. Cha. Control in production system with multiple rule firing. In *Proceedings of International Conference on Parallel Processing*, pages 243-246, 1990.

[41] T.-W. Kuo and A. K. Mok. Application Semantics and Concurrency Control of Real Time Data Intensive Applications. In *Proceedings of IEEE Symposium on Real Time Systems*, pages 35–45. IEEE CS Press, 1992.

[42] T. J. Laffey, P. A. Cox, J. L. Schmidt, S. Kai, and J. Y. Read. Real time knowledge based systems. *AI magazine*, 9(1):27-45, 1988.

[43] K. S. Leung and M. H. Wang. An Expert System Shell using structural knowledge : An object oriented approach. *IEEE Computer*, 23(3):38-47, Mar. 1990.

[44] N. G. Leveson and J. L. Stolzy. Safety Analysis Using Petri Nets. *IEEE Transactions on Software Engineering*, 13(3):386-397, Mar. 1987.

[45] N. K. Liu and T. Dillon. Detection of consistency and completeness using numerical petri nets. In J. Gero and R.Stentor, editors, *Artificial Intelligence developments and applications*, pages 119-134. Elsivier, 1988.

[46] C. G. Looney. Fuzzy Petri nets for rule based decision making. *IEEE Transactions on Systems Man and Cybernetics*, 18(1):178–183, Jan/Feb 1988.

[47] R. Maiocchi and B. Pernici. Temporal Data Systems: A comparative view. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):504-524, Aug. 1991.

[48] W. S. Mark and R. L. S. Jr. Knowledge Based Systems. *IEEE Expert*, 6(3):12-17, June 1991.

[49] K. Matsuzawa. A parallel execution method of production systems with multiple worlds. In *Proceedings of IEEE Conf. on Tools for AI*, pages 339-345. IEEE CS Press, 1989.

[50] D. Michie. Knowledge Based Systems. Technical Report 80-1001, University of Illinois Urbana-Champaign, Jan. 1980.

[51] D. Miranker. TREAT: A better match algorithm for AI production systems. In *Proceedings of Sixth National Conference on Artificial Intelligence AAAI-87*, pages 42-47, 1987.

[52] D. Miranker and B. Lofaso. The organisation and performance of a TREAT based production system compiler. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):3-11, 1991.

[53] D. P. Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production Systems*. Morgan Kaufmann, 1989.

[54] D. P. Miranker, D. A. Brant, B. Lofaso, and D. Gadbois. On the performance of lazy matching in production systems. In *Proceedings of National Conference on Artificial Intelligence AAAI-90*, pages 685-692, 1990.

[55] W. K. Nelson. REACTOR: An Expert sytwm for Diagnosis and Treatment of Nuclear Reactor Accidents. In *Proceedings of AAAI-82*, pages 296-301, 1982.

[56] D. Niemann. Parallel OPS5 User's manual and technical report. Technical Report COINS Technical report 92-28, University of Massachusetts, Apr. 1992.

[57] D. Niemann. Issues in the Design and Control of Parallel Rule Firing Production Systems. Mar. 1993.

[58] D. P. Niemann. Control issues in parallel rule firing production systems. In *Proceedings of Ninth National Conference on AI*, pages 310-316, 1991.

[59] H. P. Nii. Introduction. In V. Jagannathan, R. Dodhiawala, and L. S. Baurn. editors, *Blackboard Architectures and Applications.* Academic Press. 1989.

[60] H. P. Nii, N. Aiello, and J. Rice. Framework for Concurrent Problem Solving: A report on CAGE and POLIGON. In R. Engelmore and T. Morgan, editors. *Blackboard Systems*, pages 475-502. Addison Wesley, 1988.

[61] S. Padalkar, G. Kansai, C. Biegi, and J. Szatipanovits. Real Time Fault Diagnosis. *IEEE Expert*, 6(3):75-85, June 1991.

[62] A. Paterson and P. Sachs. Causal Reasoning in Real time expert systems. In S. G. Tzafesta, editor, *Knowledge based Systems : Diagnosis, Supervision and Control*, pages 217-229. Plenum Press, 1989.

[63] C. J. Paul, A. Acharya, B. Black, and J. K. Strosnider. Reducing Problem Solving Variance to improve Predictability. *Communications of the ACM*, 34(8):80-93, Aug. 1991.

[64] W. A. Perkins and A. Austin. Adding Temporal reasoning to expert system building environments. *IEEE Expert*, 5(l):23-30, Feb. 1990.

[65] M. Perlin. Panel discussion: Is production system match interesting. In *Proceedings of the 1992 IEEE Int. Conf. on Tools with AI*, pages 2-3. IEEE CS Press, 1992.

[66] M. Perlin and J. Debaurd. MatchBox: Fine Grained Parallelism in Match. In *Proceedings of IEEE Conference on Tools with Artificial Intelligence*, pages 428-434. IEEE CS Press, 1989.

[67] T. S. Perraju and B. E. Prasad. On the Design of a Real Time Expert System Kernel. In *Proceedings of All India Conference on Expert Systems in Engineering.* Institution of Engineers(India), 1992.

[68] T. S. Perraju and B. E. Prasad. An Algorithm for Maintaining Consistency in Multiple Rule Firing Systems. *Under Consideration in IEEE Transactions on Knowledge and Data Engineering,* 1993.

[69] T. S. Perraju and B. E. Prasad. Interference Analysis in Multiple Rule Firing Systems. *Under Consideration in IEEE Transactions on Knowledge and Data Engineering,* 1993.

[70] T. S. Perraju, B. E. Prasad, G. Uma, and P. Umarani. REX: An Object Oriented Asynchronous Real Time Expert System Shell for Aerospace Applications. *Submitted to IEEE Expert,* 1993.

[71] T. S. Perraju, G. Uma, and B. E. Prasad. A scheme for knowledge representation, verification and reasoning in real time asynchronous production systems. Technical report, University of Hyderabad, 1993.

[72] J. L. Peterson. *Petri net theory and modelling of systems.* Prentice Hall Inc., 1981.

[73] L. Portinale. Verification of causal models using petri nets. *International Journal of Intelligent Systems,* 7:715-742, Jan. 1992.

[74] L. Raschid, T. Sellis, and C.-C. Lin. The concurrent execution of production systems in database implementations. Technical Report UMIACS-TR-91-25, University of Maryland, Feb. 1991.

[75] D. Reynolds. MUSE: A toolkit for embedded real time AI. In R. Engelmore and T. Morgan, editors, *Blackboard Systems,* pages 519-532. Addison Wesley, 1988.

[76] J. P. Rosenking and S. P. Roth. REACT: Cooperating Expert Systems via A Blackboard Architecture. In *Proceedings of SP1E Application of Artificial Intelligence VI,* pages 143-150, 1988.

[77] F. H. Roth. Rule Based Systems. *Communications of the ACM,* 28(9):21-32, Sept. 1985.

[78] A. Sabharwal, S. S. Iyengar, G. de saussure, and C. R. Weisben. Parallelism in Rule based Production Systems. In *Proceedings of SPIE Application of Artificial Intelligence VI,* pages 360-372, 1988.

[79] A. Sabharwal, S. S. Iyengar, C. R. Weisbin, and F. G. Pin. Asynchronous Production Systems. *Journal of Knowledge Based Systems,* 2(2):117–127, June 1989.

[80] K. Sanou, S. G. Romaniuk, and L. O. Hall. A Hybrid/Symbolic Connectionist Production System. In *Proceedings of IEEE Conference on Tools with Artificial Intelligence,* pages 44-53. IEEE CS Press, 1992.

[81] J. G. Schmolze. Guaranteeing serializable results in synchronous parallel production systems. *Journal of Parallel and Distributed Computing,* 13:348 365. Jan. 1991.

[82] K. D. Schnelle and R. S. H. Mah. A Real Time Expert system for Quality Control. *IEEE Expert,* 7(5):36-42, Oct. 1992.

[83] A. S. Sohn and J. L. Gaudiot. A survey of parallel distributed production systems. *International Journal of Artificial Intelligence Tools,* 1(2), 1992.

[84] S. J. Stolfo, O. Wolfan, P. K. Chan, H. M. Dewan, L. Wodbury, J. S. Glazier, and D. Oshie. PARULEL: Parallel rule processing using Metarules for Redaction. *Journal of Parallel and Distributed Computing,* 13:336-382, Jan. 1991.

[85] M. Suwa, A. L. Scott, and E. H. Shortliffe. Completeness and consistency in a rule based system. In B. G. Buchanan and E. H. Shortliffe, editors, *Rule based expert systems,* pages 159-170. Addison Wesley, 1985.

[86] A. Tailor. MXA - A Blackboard Expert System Shell. In R. Engelmore and T. Morgan, editors, *Blackboard Systems,* pages 315-334. Addison Wesley, 1988

[87] M. Tambe, D. Kalp, and P. S. Rosenbloom. An efficient algorithm for production systems with linear-time match. In *Proceedings of the 1992 IEEE Int. Conf. on Tools with AI,* pages 36-43. IEEE CS Press, 1992.

[88] G. Uma. *A framework for modelling and analysis of distributed intelligent systems.* PhD thesis, Department of Computer Science, University of Hyderabad. May 1991.

[89] G. VERTENEUL. Further Results for Estimating Match-Time for RETE. In *Proceedings in Workshop on Advances in Real Time Expert System Technologic? - 10th European Conference on Artificial Intelligence,* 1992.

[90] C.-K. Wang, A. K. Mok, and A. M. K. Chang. MRL: A real time rule based production system. In *Proceedings of 11th IEEE Real Time Systems Symposium.* pages 267-276, 1990.

[91] C.-K. Wang, D.-C. Tsou, R.-H. Wang, J. C. Browne, and A. K. Mok. Automated Analysis of Bounded Response Time for Two NASA Expert systems. In *Proceedings of ACM SIGSOFT '91 Conference on Software for Critical Systems,* pages 147-161, 1991.

[92] D. A. Waterman and F. Hyes-Roth. An Overview of Pattern Directed Inference Systems. In D. A. Waterman and F. Hayes-Roth, editors, *Pattern Directed Inference Systems,* pages 3-22. Academic Press, 1978.

[93] M. A. William. Hierarchial Multi Expert Signal Understanding. In R. Engelmore and T. Morgan, editors, *Blackboard Systems,* pages 387-415. Addison Wesley, 1988.

[94] M. J. Willshire and H.-J. Kim. Properties of Physical Storage Models for Object Oriented Databases. In *Proceedings of IEEE Data Engineering Conference*, 1990.

[95] Y. T. J. Wu and U. Dayal. A Uniform model for Temporal Object Oriented Databases. In *Proceedings of Data Engineering '92*, 1992.

[96] R. Zarconato. An Object Oriented Blackboard system framework for reasoning in time. In R. Engelmore and T. Morgan, editors, *Blackboard Systtms*, pages 335-345. Addison Wesley, 1988.