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, R1 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, R1 will not match and cannot be fired. However, if a multiple rule firing policy is adopted and both R1 and R2 are fired concurrently, then there is no equivalent serial execution. Hence, interference is said to occur between R1 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.
Chapter 6.   INTERFERENCE ANALYSIS

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 interfering 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 animal frog action 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
Chapter 6. INTERFERENCE ANALYSIS

Figure 6.1: Dependency Graph for Productions P and Q

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)
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.
Chapter 6. INTERFERENCE ANALYSIS

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.
Chapter 6. INTERFERENCE ANALYSIS

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 RqJists 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 RqJists 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 RqJists of all working memory elements to be deleted. Exclusive access requests are queued in the RqJists 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 an 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 $\text{'pCE'}$ of $r_i$, add a shared request on behalf of $r_i$ in the RqJist of $w_j$.

3. For each negated CE $\text{'nCE'}$ of $r_i$, 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 $\text{'nAE'}$ of $r_i$, add an exclusive request on behalf of $r_i$ to the RqJist of the WME to be deleted.

5. For each positive action element $\text{'pAE'}$ 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 $pAE$. 
Chapter 6. INTERFERENCE ANALYSIS

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 \( r_i \).

   (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
Chapter 6. INTERFERENCE ANALYSIS

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 CE1 to CE\( n \) 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 -)

- (labeling-candidate junction-ID <y> line-2 -)

-->
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 J1 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-ID 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(W1) : (I1,s), (I2,s), (I3,s), (I4,s)
- RqJist(W2) : (I1,s), (I2,s)
- RqJist(W3) : (I1,s), (I2,s)
Chapter 6. INTERFERENCE ANALYSIS

- RqJist(W4) : (I1,se), (I2,se)
- RqJist(W5) : (I3,s), (I4,s)
- RqJist(W6) : (I3,s), (I4,s)
- RqJist(W7) : (I3,se), (I4,se)
- RqJist(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 W1, W2, W3, W4 and the alpha memory of CE5 of P1. The tags would be

- W1 : 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

- W1 : s
- W2 : s
- W3 : s
- W4 : se.
• W5 : s
• W6 : s
• W7 : se
• Alpha memory of CE5 : s

Then instantiation 14 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 11 and 13. The two instantiations are fired in the firing cycle. If instantiation 12 was considered first then 11 will be discarded and any one of the instantiations 13 or 14 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 \( P_n, P_{n-1}, \ldots, P_1 \). 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 \( r_1 \).

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 \( r_1 \).

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 \( w_2 \).

\[
\begin{align*}
\text{RqJist}(w_1) & : (n, e), (r_2, s) \\
\text{RqJist}(w_2) & : (r_1, s), (r_2, e)
\end{align*}
\]

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 \( w_2 \).

\[
\begin{align*}
\text{RqJist}(w_1) & : (r_1, e), (r_2, s) \\
\text{RqJist}(\alpha\text{-mem of } w_2) & : (r_1, s), (r_2, e)
\end{align*}
\]
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 $w_2$.

$$Rq\_list(\alpha\text{-mem of } w_1): (r_1, e), (r_2, s)$$
$$Rq\_list(w_2): (r_1, s), (r_2, e)$$

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 $w_2$.

$$Rq\_list(\alpha\text{-mem of } w_1): (r_1, e), (r_2, s)$$
$$Rq\_list(\alpha\text{-mem of } w_2): (r_1, s), (r_2, e)$$

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 $O(n)$ 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 $O(n)$. The tagging of working memory elements and alpha memories by the selected rule instantiation takes again constant time. Hence the 'select' phase takes $O(n)$ 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.

- A directed 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 a 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 \( R_1, R_2, ..., R_{i+1}, ..., R_n R_{n+1} \), where \( R_{n+1} = R_n \) be an acyclic sequence of matched rules. Cyclic interference is said to exist if for all \( i \), there exists an attribute that is 'updated' by \( R_i \) and is 'read' by \( R_{i+1} \). Interference is said to occur between two rules \( R_A \) and \( R_B \), 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'
Chapter 6. INTERFERENCE ANALYSIS

<table>
<thead>
<tr>
<th>Att. No.</th>
<th>Att. name</th>
<th>Type Attribute</th>
<th>Attribute® value</th>
<th>shared request count</th>
<th>exclusive request count</th>
<th>update flag</th>
</tr>
</thead>
</table>

Table 6.1: Structure of the Attribute Table

<table>
<thead>
<tr>
<th>Rule id.</th>
<th>Priority</th>
<th>Read List</th>
<th>Write List</th>
<th>Premise id.s</th>
<th>Action id.</th>
<th>Event id.</th>
<th>Hold id.</th>
</tr>
</thead>
</table>

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.

\[ \text{R1:} \]

- Priority: 1
- Premise: \( A.a_1 < 30; B.b_2 > 40 \)
- Action: \( D.d_1 = 100 \)

\[ \text{R2:} \]

- 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 R1 and R2

<table>
<thead>
<tr>
<th>Att. No.</th>
<th>Att. name</th>
<th>Type</th>
<th>Attribute@ time</th>
<th>value</th>
<th>shared request count</th>
<th>exclusive request count</th>
<th>update flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A.a₁</td>
<td>int</td>
<td>time</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B.b₁</td>
<td>int</td>
<td></td>
<td>5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>B.b₂</td>
<td>int</td>
<td></td>
<td>46</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>D.d₁</td>
<td>int</td>
<td></td>
<td>12</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.3: Structure of the Attribute Table

<table>
<thead>
<tr>
<th>Rule id.</th>
<th>Priority</th>
<th>Read List</th>
<th>Write List</th>
<th>Premise id.’s</th>
<th>Action id.’s</th>
<th>Event id.’s</th>
<th>Hold id.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3, 4</td>
<td>1, 2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1, 3</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4: Structure of Rule Table
Chapter 6.  *INTERFERENCE ANALYSIS*

<table>
<thead>
<tr>
<th>Att. No.</th>
<th>Att. name</th>
<th>Type</th>
<th>Attribute® time</th>
<th>value</th>
<th>shared request count</th>
<th>exclusive request count</th>
<th>update</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A.(a_1)</td>
<td>int</td>
<td></td>
<td>25</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B.(b_1)</td>
<td>int</td>
<td></td>
<td>5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>B.(b_2)</td>
<td>int</td>
<td></td>
<td>46</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>D.(d_1)</td>
<td>int</td>
<td></td>
<td>12</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

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).

\[
\text{MRS} = \{1, 2\}
\]

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.

\[
\text{ERS} = \{1\}
\]

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*.
Chapter 6. INTERFERENCE ANALYSIS

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.