October <Technical> Rules Fest

ORF - 2008: Friday, October 24th

Rete Pattern-Matching Algorithm:

(and other subjects?)

James Owen

Senior KE

KnowledgeBased Systems Corporation

http://www.kbsc.com

October <Technical> Rules Fest

This talk will deal with two subjects, maybe three if we have the time.

- **Original 1979 Rete Algorithm (using the original code from the thesis)**
- **Chaining**
  - **Forward**
  - **Goal-Oriented Forward Chaining**
  - **Rule-Generated Backward Chaining Using Forward Chaining Rulebase**
  - **Full Opportunistic Backward Chaining**
- **Introduction to Parallel Rulebased Engine History, etc.**

Rete Algorithm

Rete Pattern-Matching Algorithm:

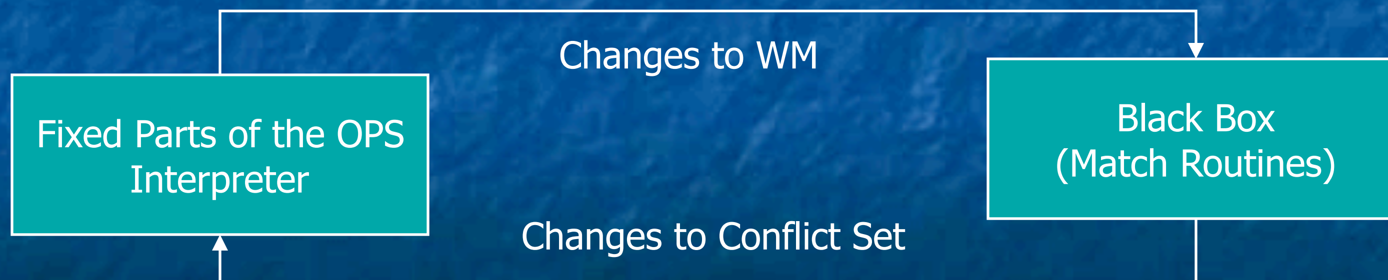
If you read the 1979 dissertation and compare that with numerous commercial texts dealing with the Rete Algorithm you will discover that very few actually have any idea of how the Rete Algorithm works. Sad but true. This talk will cover parts of the 1979 dissertation. (This may or may not help you learn to write rules.) However, the next few slides will follow the dissertation SO that you can read it later and make some sense of it. ☺ Taking a rule from an examples in CLIPS [MB11]

```
( defrule move-object-to-place ""
  ( goal-is-to  ( action  move ) ( arguments  ?obj  ?place ) )
  ( monkey  ( location  ~?place ) ( holding  ?obj ) )
  ( not ( goal-is-to  ( action  walk-to ) ( arguments  ?place ) ) )
  =>
  ( assert  ( goal-is-to  ( action  walk-to ) ( arguments  ?place ) ) ) )
```

Rete Algorithm

Rete Pattern-Matching Algorithm:

First, in order for the Rete Algorithm to function, it must first process all of the LHS elements before beginning execution. This is (today) a very short set-up time to discover the similarities. This is the interpreter that compiles a program to perform a match for that one production system, set of rules. The most unusual feature of the match routine is that it never examines working memory. Instead, it monitors the *changes* made to the working memory and maintains internal information that is *equivalent* to working memory.



Rete Algorithm

Rete Pattern-Matching Algorithm:

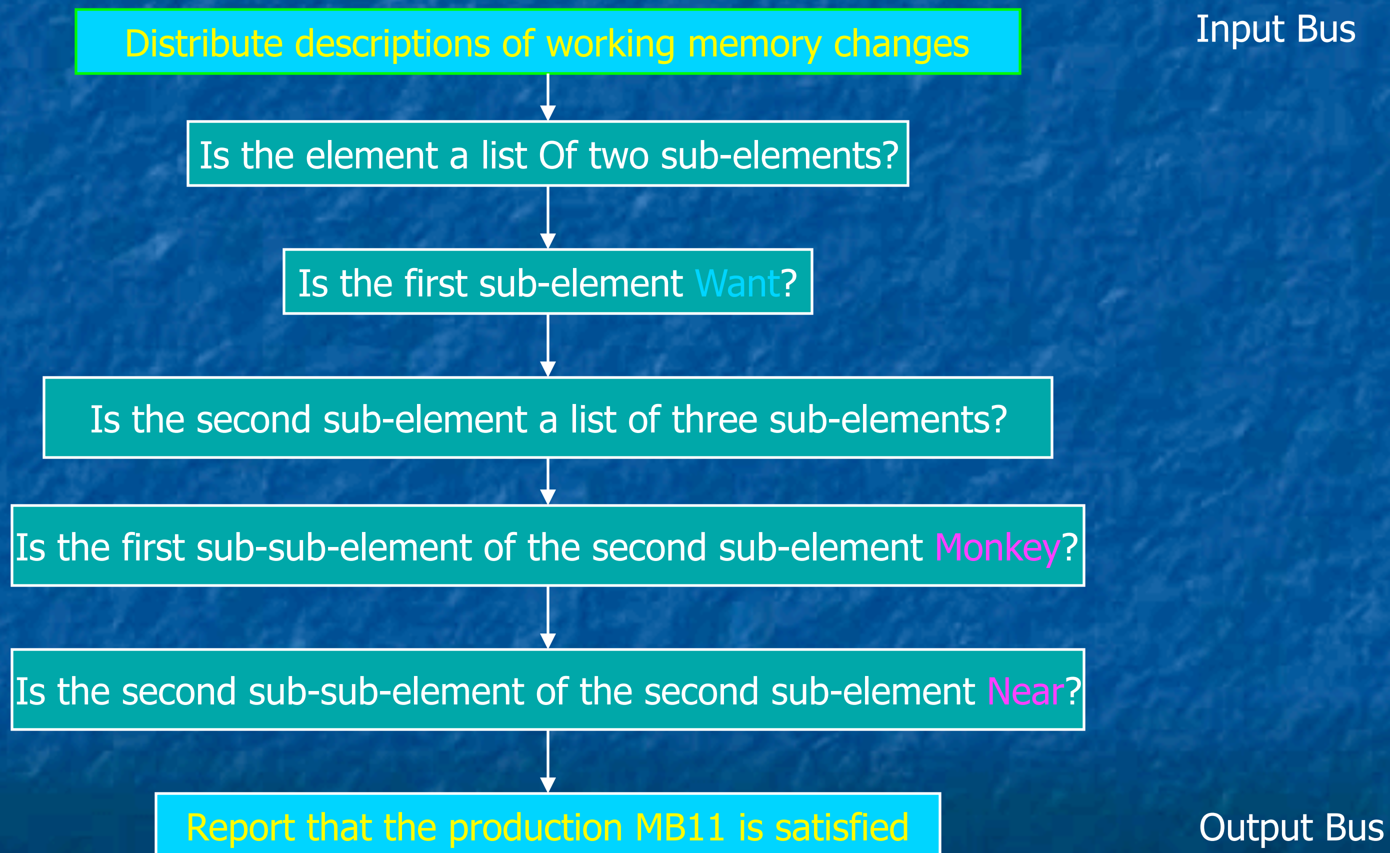
If we look at MB11 in the 1979 dissertation, p. 26, MB11 and MB12 are stated

```
MB11    ( ( Want  ( Monkey  Near  =P ) )
        -->
        ( Want  ( Monkey  On  Floor ) )  )
MB12    ( ( Want  ( Monkey  Near  =P ) )  ( Monkey  On Floor )
        ( Monkey  Near  =C  &  #P )
        -->
        (  < WRITE >  "The monkey walks from " =C  " to "  =P )
        (  <DELETE >  ( Want  ( Monkey  Near  =P )  )  )
        (  <DELETE >  ( Monkey  Near  =C )  )
        ( Monkey  Near  =P  )
```

Note that we have two CE's that are parsed as follows: Two Sub-elements (SE) where the first SE is composed of only one SE called Want and the second SE is composed of three Sub-SE's where the first two are Monkey and Near.

Rete Algorithm

Rete Pattern-Matching Algorithm:



Rete Algorithm

Rete Pattern-Matching Algorithm:

The packets sent between the nodes are called tokens. Each token has two components: the tag part and the data part. The tag part is either <VALID> or <INVALID> while the data part holds either a WME or list of WME's. <VALID> indicates elements that have been added to WM and <INVALID> indicates elements that have been deleted from WM. For example, assume that we insert into WM the two elements and delete one element. These would be:

<VALID, (Want (Monkey Holds Bananas)) >

<VALID, (Want (Monkey On Floor)) >

<INVALID, (Want (Monkey On Floor)) >

Note that the tag is VALID or INVALID while the text following the comma is the data part of the token.

Rete Algorithm

Rete Pattern-Matching Algorithm:

Now, combine the thought process of the rules with the inserted elements:

<VALID, (Want (Monkey Holds Bananas)) >

1st : Is the element a list of two sub-elements. YES

2nd : Is the first sub-element Want? YES

3rd : Is the second sub-element a list of three sub-sub-elements? YES

4th : Is the first sub-sub-element of the second sub-element Monkey? YES

5th : Is the second sub-sub-element of the second sub-element Near? NO

Because the second sub-sub-element of the second sub-element is Holds the token fails so nothing is sent to any following nodes.

Rete Algorithm

Rete Pattern-Matching Algorithm:

Continuing that thought process of the rules with the inserted elements and MB11 rule :

<VALID, (Want (Monkey Near (2 2))) >

1st : Is the element a list of two sub-elements. YES

2nd : Is the first sub-element Want? YES

3rd : Is the second sub-element a list of three sub-sub-elements? YES

4th : Is the first sub-sub-element of the second sub-element Monkey? YES

5th : Is the second sub-sub-element of the second sub-element Near? YES

All five nodes are YES so the token is passed as VALID. Actually, MB12 would also pass the token since the first CE for both MB11 and MB12 are the same. But that leads to the network itself – how does Rete combine these for better efficiency?

Rete Algorithm

Rete Pattern-Matching Algorithm:

Consider MB18 from the dissertation, about page 32.

```
MB 18   ( ( Want  ( EmptyHanded  Monkey ) )  ( Monkey  Holds  =X )  )
    -->
    (  <WRITE>  " The Monkey drops the "  =X )
    ( <DELETE>  ( Want  ( EmptyHanded  Monkey )  )  )
    ( <DELETE>  ( Monkey  Holds  =X )  )  )
```

Consider the first CE is instantiated by an data element that has the following:

1. **Has two sub-elements**
2. **Has the constant Want as its first sub-element**
3. **Has a list of two sub-sub-elements as the second sub-element**
4. **Has the constant EmptyHanded as the first sub-sub-element of its second sub-element**
5. **Hs the constant Monkey as the second sub-sub-element of its second sub-element**

Rete Algorithm

Rete Pattern-Matching Algorithm:

Consider the second CE is instantiated by an data element that has the following:

1. **Has three sub-elements**
2. **Has the constant Monkey as its first sub-sub-element.**
3. **Has the constant Holds as it second sub-sub-element.**

Now we have a rule that incorporates TWO condition elements and we have to evaluate both before we can join the tokens, one from the left branch and one from the right branch. Since elements do not normally arrive at the same time, it must save state from one activation to the next.

This is discussed in great detail (well, some detail anyway) in section 2.2.4 of the 1979 dissertation.

Rete Algorithm

Rete Pattern-Matching Algorithm:

Once more considering MB18, Consider that we have previously inserted four elements into working memory.

(Monkey Holds Ladder)
(Monkey Holds Orange)
(Monkey Holds Glass)
(Monkey Holds Box)

And, later, (Want (EmptyHanded Monkey)) is entered into working memory. Its token would reach the left input of the of the two-input node and cause processing that would result in for tokens being sent out:

< VALID, (Want (EmptyHanded Monkey)) (Monkey Holds Ladder) >
< VALID, (Want (EmptyHanded Monkey)) (Monkey Holds Orange) >
< VALID, (Want (EmptyHanded Monkey)) (Monkey Holds Glass) >
< VALID, (Want (EmptyHanded Monkey)) (Monkey Holds Box) >

Rete Algorithm

Rete Pattern-Matching Algorithm:

Now, we are ready to consider the example given in all text books on the subject of the Rete Algorithm:

```
MB15   ( ( Want ( Monkey  On  =0 ) ) ( =0  Near  =X )
        ;  A       B1      B2   B3       C1   C2   C3
      -->
      ( Want  ( Monkey  Near  =X ) ) )
       ;  W       X1       X2    X3
MB16   ( ( Want ( Monkey  On  =0 ) ) ( =0  Near  =X ) ( Monkey  Near  =X )
       ;   D       E1       E2   E3       F1   F2    F3       G1        G2   G3
      -->
      ( Want  ( EmptyHanded  Monkey ) ) )
          ;  Y         Z1            Z2
```

When these two rules are entered into the network, most of the nodes in the network are shared as shown in the next slide:

Rete Algorithm

Rete Pattern-Matching Algorithm:

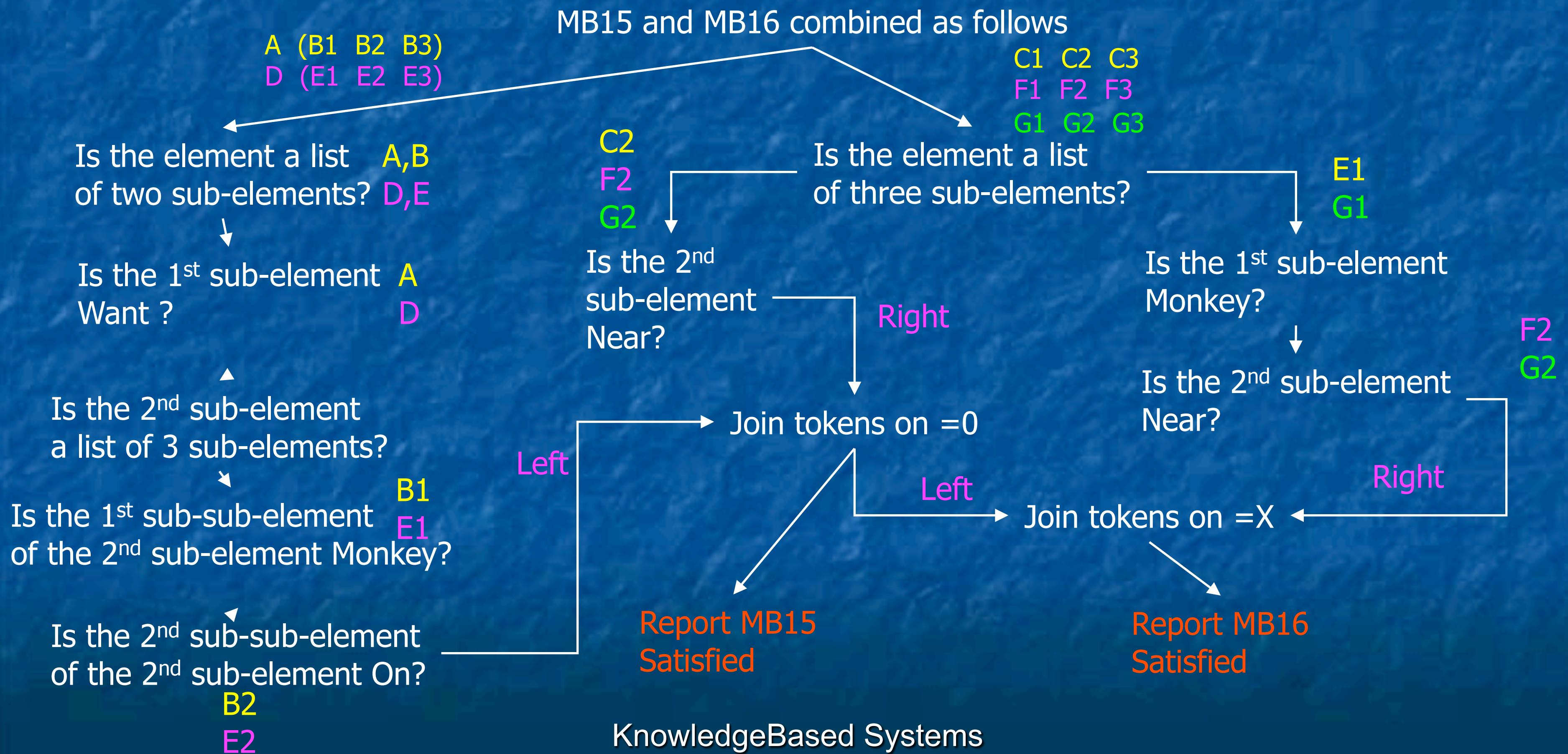
Now, we are ready to consider the example given in all text books on the subject of the Rete Algorithm:

```
MB15   ( ( Want ( Monkey  On  =0 ) ) ( =0  Near  =X )
         ;  A      B1     B2   B3      C1   C2   C3
       -->
       ( Want  ( Monkey  Near  =X ) ) )     // inserts into WM
        ;  W      X1      X2   X3
MB16   ( ( Want ( Monkey  On  =0 ) ) ( =0  Near  =X ) ( Monkey  Near  =X )
        ;   D      E1     E2   E3      F1   F2   F3     G1      G2    G3
       -->
       ( Want  ( EmptyHanded  Monkey ) ) )   // inserts into WM
          ;  Y        Z1           Z2
```

Since the two rules share several nodes (shown with connecting arrows above) the the network can be created as shown in the next slide:

Rete Algorithm

Rete Pattern-Matching Algorithm:



Rete Algorithm

Rete Pattern-Matching Algorithm:

Here is what we are NOT going to consider at this point in time:

Sharing among non-similar rules
Parallelism - Last part of this presentation
One-input Nodes
Two-input Nodes
The NOT Node problem
A Node that changes the conflict set
The Bus Node
Memory Nodes
Synchronizing a Divided Two-Input Node
OLD-VALID tags

Dr. Forgy has said that if you absolutely MUST study the 1979 dissertation, then concentrate on Chapters Two, Three and Four. So far, we have done only chapter Two and not much of it. The rest would be an Advanced Class.

Chaining

Four Main Groups:

- **Forward Chaining (Normally uses some variant of the Rete Algorithm)**
- **Goal-Oriented Forward Chaining (sometimes erroneously called backward chaining) that is used by a Forward Chaining engine.**
- **Backward Chaining using Forward Chaining rules to generate goals to find missing attributes.**
- **Full Opportunistic Backward Chaining (FOBC) - the main point of this talk is to show the more traditional way to do backward chaining.**

Forward Chaining

- **Forward Chaining** (Normally uses some variant of the Rete Algorithm) is normally used for forecasting, monitoring and control. These rule programs are normally non-monotonic in nature - as opposed to Java or C/C++ which are monotonic and procedural in nature.
- This is discussed in-depth in Girratano & Riley, Chapter 3.14. A simple definition is that we move from assumption to conclusion and is called by most at "data-driven" reasoning.
- A more complex point of view might be that we have chaining from a given set of datum to a conclusion or, internally, that we have chaining between rules.

Rule1
If CE1 AND
CE2
Then AE1

Goal-Oriented Forward Chaining

Goal-Oriented Forward Chaining (sometimes called backward chaining) that is used by a Forward Chaining engine.

However, most practioners follow the method used by vendors such as ILOG JRules, Fair Isaac Blaze Advisor or PST OPSJ which is Goal-Oriented Chaining using a Forward Chaining rule engine.

To understand GOFC we must first understand Conflict Resolution Strategy (CRS) and how it plays a major part.

Goal-Oriented Forward Chaining

Conflict Resolution Strategy (note that this talk now has a sub-goal)

Rules that have a potential to be "fired", meaning that the LHS is true, are all placed on an "Agenda Table" where they are evaluated. For any given set of rules, they are resolved in the following order:

1. Refraction
2. Priority
3. **Recency**
4. Specificity
5. Arbitrary

Goal-Oriented Forward Chaining

Conflict Resolution Strategy

Step 3 (Recency) on the previous slide can be implemented using either LEX or MEA. This is discussed in Cooper and Wogrin Ch. 3.3.3 as well as others. Also, some RBS - such as OPSJ - use an "enhanced" CRS such as

1. Refraction
2. Rule Class
3. **Recency**
4. Priority
5. Specificity
6. Arbitrary

Goal-Oriented Forward Chaining

The purpose of discussing CRS first was to show that Recency (3rd element on both CRS) can be used to "drive" a rulebased system where the focus is moved to any rule or group of rules that have a certain goal as the first CE of those rules. For example (pseudo code can be used by any rulebased system)

IF there is a Goal (value == GOAL1)

AND …

AND …

THEN … (insert new Goal (GOAL2)

The initial Goal in the first CE is NOT retracted but due to Refraction when there are no more rules where the first CE is true and there are no more rules to fire with that goal CE then the goal is no longer in focus and won't fire - but control could return to that set of rules at any time.

However, sometimes (as shown here) a new Goal can be inserted into Working Memory and then that set of rules with GOAL2 as the first CE will become the focus of the rulebase.

Goal-Oriented Forward Chaining

Per conversation in Dallas on 23 October 2008 with Dr. Charles Forgy:

"Conflict Resolution cannot be a locus of intelligence."

"Using rule order alone for conflict resolution is unacceptable."

QED

Back Chaining Using Rules

Backward Chaining using a forward chaining engine is discussed in some detail in Girratano and Riley, Chapter 12.4. While this is not terribly complex to use there has not been a lot done with this in applications. (Most will use Prolog for something like this.) Here is the part of the example from G&R, 4th Ed., p724+

```
procedure  Solve_Goal (goal)
    goal: the current goal to be solved
    if value of the goal attribute is known
        Return the value of the goal attribute
    end if
    for each rule whose consequence is the goal attribute do
        call Attempt_Rule with the rule
        if Attempt_Rule succeeds
            then Assign the goal attribute the value indicated and Return that value
        endif
    end do
end procedure
```

Full Opportunistic BackwardChaining

Full Opportunistic Backward Chaining (FOBC) - is a slightly different type of rulebase that is backward chaining as default and the rulebase is constructed as being backward chaining. Neuron Data (predecessor to Fair Isaac) had such a system called ND Nexpert, or later just ND Expert.

The main point of this talk is simple: that a rulebase user could just suggest a hypothesis (usually focused on the action part of an IF-THEN rule) as being either true or false (or that the value is some number or condition) and then examine all of the data behind any conclusion of that rule to support that conclusion as being true or false.

If we think of each rule as being an AND gate, OR gate, NAND gate or NOR gate then this makes the analysis much simpler. The next slide will demonstrate this idea.

Full Opportunistic BackwardChaining

However, for an FOBC there are three states of each value or output:

- **Known**: The value of the input or output is known
- **Unknown**: The value of the input or output is unknown and unknowable because we have tried to find out the value and could not determine it.
- **Not-Known**: The value of the input or output is not known and we have not tried to find its value.

When dealing with an any of the gates, if the

- **Output** is **known** then we can readily examine it and see whether it is true or false.
- **Output** is **unknown** then we stop evaluation and report that the value can not be determined.
- **Output** is **Not-Known** then we will have to find the value by examining the inputs by back-chaining through the attributes / slots.

Full Opportunistic BackwardChaining

For an FOBC then, for an AND or NAND gate, if any input is

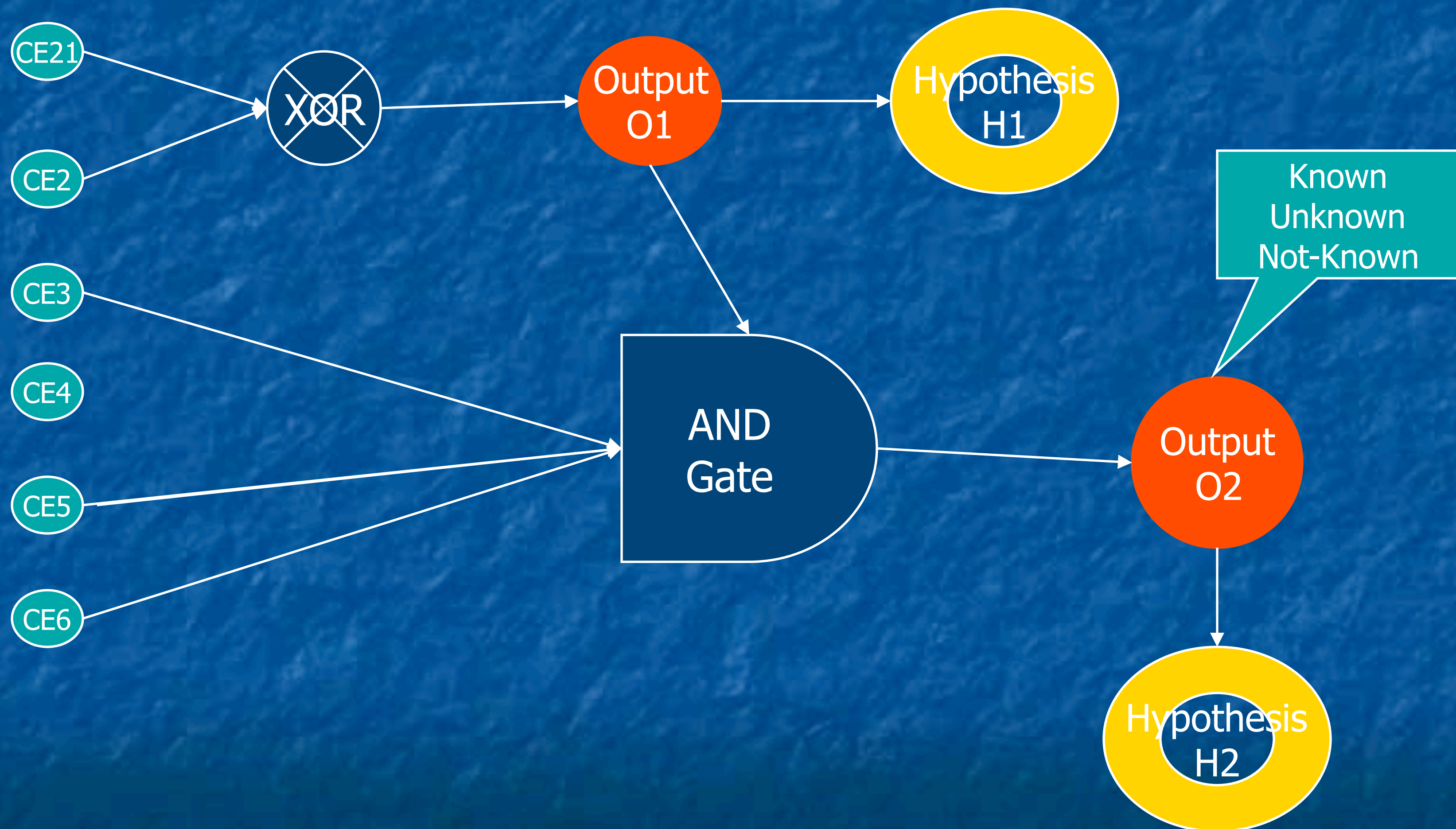
- **Known: If the value is true, examine the next inputs. If false, report the AND gate as false.**
- **Unknown: the value of the could not be determined and, therefore, the value of the AND gate could not be determined.**
- **Not-Known: If there is a "question handler" for that slot then we begin to trace backward to see if the value can be determined.**

For an FOBC, for an OR or NOR gate, if the input is

- **Known then we can readily examine it and see whether it is true or false. If true, the gate is true. If false, we examine the other inputs.**
- **Unknown AND there are no more inputs to examine AND the other inputes are false or unknown, then we stop evaluation and report that the value can not be determined ELSE we continue to examine inputs.**
- **Not-Known If there is a "question handler" for that slot then we begin to trace backward to see if the value can be determined.**

Full Opportunistic BackwardChaining

An AND gate with an XOR gate might appear like this



October <Technical> Rules Fest

Rete Algorithm and Rule Chaining

References:

1979 Ph.D. Thesis by Dr. Charles Forgy

1995 – Robert Doorenbos – Production matching for Large Learning Systems

On file at http://www.kbsc.com/WhitePapers.html

Expert Systems – Principles and Practices, Joseph Girratano and Gary Riley, 4th Ed., 2005, ISBN 0-534-38477-1

Rule-Based Programming With OPS5, Thomas Cooper and Nancy Wogrin, 1988, ISBN 0-934613-51-6 (out of print)

Rule-Based Expert Systems – The MYCIN Experiments of the Stanford Programming Project, Bruce Guchanan and Edward Shortliffe, 1984/1985, ISBN 0-201-10172-6 (out of print)

Introduction: Parallel Rulebase Programming

Early Work by Gupta, Forgy, Newell in 1986:

Of the three steps (Match, Conflict Resolution, Act) the most time -consuming is the Match phase.

Consider that if we have 1000 rules, 1000 WME elements and each rule has three CEs, if you match all tuples of size three from WM you will have over 1T ($1000 * 1000^3$) comparisons (called cross matches) that is the initial cross match product. Fortunately, the Rete algorithm handles all subsequent matches so that we don't have to do the naïve matching on each cycle.

This part of the presentation deals with Parallelizing Rete.

Introduction: Parallel Rulebase Programming

Parallelizing Rete:

By parallelizing Rete at a relatively fine grain it is possible to exploit it is possible to evaluate multiple activations of the same node in parallel and to process multiple changes of WM in parallel.

Because the Match step takes about 90% of the time this should be the first to consider for parallelization. In the 86-GFN paper,

"The obvious way to use parallelism in a Rete matcher is to allow more than one token to be flowing through the graph at any one time. In fact, this is how the the Rete Algorithm was intended to be used…**"**

Introduction: Parallel Rulebase Programming

Parallelizing Rete:

Again, in the 86-GFN paper, they suggest the following procedure:

- Each node is permitted to process only one input token at a given time
- The node has only a single thread of control internally
- The processor(s) should be shared-memory CPUs
- You should have between 32 - 64 CPUs minimum for any discernable effect
- Processors should be connected to shared memory via shared bus
- Hardware should have hardware task scheduler

Introduction: Parallel Rulebase Programming

Parallelizing Rete:

Again, in the 86-GFN paper, they found the following results:

- An increase speed of “only” ten fold because
  - The number of rules relevant to any change is small
  - There is a large variation in the processing requirements of relevant rules
- Of the 32 processors only 16 were kept busy

Another paper written by Frank Lopez in 1987 sheds very little light on the problems in Parallel Rete.

Introduction: Parallel Rulebase Programming

Parallelizing Rete:

A working version of Para-OPS-5 was developed by Gupta et al in 1987-1988 and a manual was developed for it that is still available. Some of the insights from that system were:

- Avoid small cycles of the Recognize-Act cycle with less than 50 tokens.
- Avoid long chains of dependent node activations
- Avoid large cross-products (much like any other rulebased system)

These bottlenecks are discussed in some detail in the manual itself.

October <Technical> Rules Fest

Parallelizing the Rete Algorithm

References:

1979 Ph.D. Thesis by Dr. Charles Forgy

1986 - Anoop Gupta, Charles Forgy and Alan Newell - Parallel Algorithms and Architectures for Rule-Based Systems.

1988 - Gupta et al, 1988 - Dirk Kalp, Milind Tambe, Anoop Gupta, Charles Forgy, Allen Newell, Aurag Acharya, Brian Milnes, Kathy Swedlow: Parallel OPS5 User's Manual

On file at http://www.kbsc.com/WhitePapers.html