The RETE Algorithm in OPS5

A naive implementation of the recognize-act cycle in which for each loop iteration all database elements are compared with the conditions of all rules, would lead to bad runtime performance, especially for large rule sets.

Two fundamental ideas to improve efficiency:

1) Limitation of database elements that have to be checked:
   During the evaluation of a rule, only a small part of the working memory is changed. A pattern matching with unchanged elements doesn’t change the conflict set.
   The results of a comparison can be saved. In the next cycle, only the changes in the database are checked whether they lead to instantiations of new rules or whether they remove existing instantiations.

2) Limitation of conditions that have to be checked:
   In general, the condition parts of rules are not disjunct. So, the costs of the recognition cycle can be reduced by executing identical tests only once.

The RETE Algorithm in OPS5

The RETE algorithm allows an efficient implementation of pattern matching in production systems.

The algorithm receives all database changes as input and uses this information to compute the necessary changes of the conflict set.

Rules are transformed into a network of code snippets representing condition parts (RETE = Latin for „network“).

The network consists of:
- 1-input nodes: Representation of conditions concerning only one database element.
- 2-input nodes: Relations between the single conditions.

A single condition is transformed into a sequence of 1-input nodes:

```
(Pyramid ^Color << Yellow White >> ^Weight <= heavy)
```

Class name = Pyramid ?
^Color << yellow white >> ^Weight <= heavy ?

The RETE Algorithm in OPS5

Each node is created by a procedure which checks the restrictions of an attribute.

There are no nodes for variable bindings as long as the variable appears only once. Otherwise, a node is created:

```
(Box ^Material Wood ^Height <X> ^Diameter >= <X>)
```

```
Class name = Box ?
^Material = Wood ?
^Diameter >= Value of Height ?
```

The input of such a sequence of 1-input nodes are tokens. These tokens represent the value and timestamp of database elements together with a label:

```
+: newly created element
- : element to be deleted
```

The RETE Algorithm in OPS5

Each node only transfers tokens to the child node, which fulfills its condition.

Multiple single conditions on the left hand side of a rule can be connected by a 2-input node.

```
(Cuboid ^Name ^stands_on <Cuboid_1> ^stands_on Bottom)
```

```
(Cube ^Weight heavy)

Class name = Cuboid ?
^Name = Cube ?
^Weight = heavy ?
```

Beta Memory

Alpha Memory
The RETE Algorithm in OPS5

A 2-input node has two inputs. If the condition part of a rule has more than two clauses, additional 2-input nodes are created. Thereby, one input is connected with the output of a previous 2-input node.

2-input nodes have two lists in which tokens can be stored.

The outputs of the two previous nodes are stored in the **alpha and beta memory**.

If one of the two previous nodes is a 2-input node, the output of that node is stored in the **beta memory**.

If both inputs of a 2-input node have no common variables, the result is the **cross product** of its alpha and beta memory.

If there are common variables, the 2-input node checks what elements fit together to get a **consistent variable configuration**.

The output of the last 2-input node that belongs to a rule, represents the **changes of the conflict set**.

---

The RETE Algorithm in OPS5

If a new element is added to the database, it passes the network as a plus token, whereby the alpha and beta memory have to be updated.

If an element has to be removed from the database, it passes the network as a minus token, whereby the entries in the alpha and beta memory that belong to this element will also be removed.

**Example for the use of alpha and beta memory:**

```
(P Search_Pyramid
 (Quader
  ^Name <Cuboid_1>
  ^stands_on Bottom)
 (Cube
  ^Weight heavy )
 (Pyramid
  ^Color << Yellow White >>
  ^stands_on <Cuboid_1>
  ^Weight <> heavy )
 --> (action part)
)
```

---

The RETE Algorithm in OPS5

The tokens in the memories are represented by their time stamps.

The entries in bold shows the changes after the insertion of the following element into the database: (15, Cube, W2, Blue, Heavy, Nil)

<table>
<thead>
<tr>
<th>Time stamp</th>
<th>Class</th>
<th>Name</th>
<th>Color</th>
<th>Weight</th>
<th>Stands_on</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Blue</td>
<td>Q1</td>
<td>Blue</td>
<td>Heavy</td>
<td>Bottom</td>
</tr>
<tr>
<td>4</td>
<td>Cuboid</td>
<td>Q2</td>
<td>Red</td>
<td>Light</td>
<td>Bottom</td>
</tr>
<tr>
<td>6</td>
<td>Pyramid</td>
<td>P1</td>
<td>Yellow</td>
<td>Light</td>
<td>Bottom</td>
</tr>
<tr>
<td>9</td>
<td>Pyramid</td>
<td>P2</td>
<td>White</td>
<td>NIL</td>
<td>Q1</td>
</tr>
<tr>
<td>12</td>
<td>Cube</td>
<td>W1</td>
<td>Blue</td>
<td>Heavy</td>
<td>Bottom</td>
</tr>
</tbody>
</table>

The RETE algorithm merges identical subparts of all networks of the rule base.
The RETE Algorithm in OPS5

During the transformation the conditions are not rearranged. Although the content of the following single conditions is identical, only the test for the class name is merged.

```
(* Box Color blue Material wood Size <= 100 *)
(* Box Material wood Color blue Size <= 100 *)
```

In the beginning of their first condition, the following two rules can use a common part of the network.

```
(* Rainstorm warning *)
(Weather-forecast Status current Source radio-news Weather rain)
(Observation Type dark-clouds)
--> (WRITE [There will be rain.] [CRLF]
[I recommend to take an umbrella with you.])
```

```
(* Trip recommendation *)
(Weather-forecast Status current Source country-saying Weather sunshine)
(Date Weekday friday Month june jul aug sep)
--> (WRITE [The weather will be fantastic.] [CRLF]
[I recommend this weekend to go to the sea.])
```

The RETE Algorithm in OPS5

A disadvantage of the RETE algorithm is that rules, that are newly created during runtime, only react on database elements which enter the database after the rule creation.

The RETE algorithm allows a more efficient computation with production systems.

One should try to minimize the content of the various alpha memories. That can be done by formulating the rule conditions as specific possible.

Example: Rule that determines the location of the workplace for each person in the database.

```
(* Location_of_workplace *)
(Control-element request Status active)
(Person Name <name> employed_at <employer>)
(Company Name <employer>)
(Location <place_searched>)
--> (WRITE <name> works in <place_searched>)
```

The RETE Algorithm in OPS5

It may happen that 2-input nodes have more than one output, e.g. if the condition part of a rule includes the condition part of another rule.

```
(* Rule_1 *)
(Condition_1) (Condition_2) --> (action part)
(* Rule_2 *)
(Condition_1) (Condition_2) (Condition_3) --> (action part)
```

For changing an element in the database, a plus token and a minus token is sent through the network. In doing so, changing an element is twice as expensive as creating or removing an element.

A disadvantage of the RETE algorithm is that rules, that are newly created during runtime, only react on database elements which enter the database after the rule creation.

The RETE algorithm allows a more efficient computation with production systems.

One should try to minimize the content of the various alpha memories. That can be done by formulating the rule conditions as specific possible.

Example: Rule that determines the location of the workplace for each person in the database.
If we have the additional information that persons under 16 and persons over 65 as well as pupils and students don’t have an employer, the alpha memory can be reduced.

(P Location_of_workplace
  (Control-element
    ^Context request
    ^Status active)
  (Person
    ^Name <name>
    ^Age { >= 16 <= 65 }
    ^Profession { <> pupil <> student }
    ^employed_at <employer>)
  (Company
    ^Name <employer>
    ^Location <place_searched>)

--> (WRITE <name> | works in | <place_searched>)
)

The RETE Algorithm in OPS5

The content of the beta memory can be reduced by taking over only a subset of the preceding 2-input node cross products.

Example:

(Condition_A <var_1>)
(Condition_B <var_2>)
(Condition_C <var_3>)
(Condition_D <var_1> <var_2> <var_3>)

The first three conditions are independent of each other. But there is a dependency between the fourth and the first three conditions.

In our example, we assume that for each of the four conditions 10 matching elements exist in the database. The dependency described in the fourth condition results in 10 different instantiations.

In node Z3, the cross product consisting of 10,000 elements has to be evaluated. But only 10 instantiations remain as an output.

Example of an OPS5 production rule in the factory scheduling system FERPLAN

{p w18
  (goal ^run <u>
  ^status active
  ^type place
  ^object workpiece)
  (Counter ^run <u>
  ^value1 <n>
  ^value2 <m>
  ^value3 <l>)
  (Tool ^run <u>
  ^type <p>
  ^pnr <l>)
  (Tool ^run <u>
  ^type <s>
  ^pnr <n>
  ^protgrid yes
  ^empty yes)
  (Protgrid ^state open)

--> (write | Insert workpiece in | <s>)
  (remove 1)
  (modify 4 ^empty no)
  (make goal ^run <u>
    ^status active
    ^type close
    ^object protgrid)
  )

If there is an active goal to place a workpiece into a tool 's', 'Counter' is set to a value of '1'. A following tool named 's' with position number 'n' for which an opened protective grid is needed, then:

remove the reached goal, change the attribute value and create a goal for closing the protective grid.
Expansion of Rule Set by Actions of Rules

In OPS, new rules can be created at runtime with the action BUILD, e.g. to enlarge the knowledge base or to improve runtime efficiency.

Parameters for BUILD: a unique rule name, the condition part, the atom ‘^Value’ and the action part of the new rule.

Example:

(P Rule builder
  (Final-value ^Value <x>)
  -->(BUILD \ (GENATOM)
     (Result ^Value \ \ <x> )
  -->(WRITE [Final value reached] )
  (STOP))

If the database includes the element (Final-value ^Value 2009), the following new rule is created:

(P G:372
  (Result ^Value 2009 )
  -->(WRITE [Final value reached] )
  (STOP))

Default Reasoning - Motivation

The most what we know about the world is only „almost always” true.

We never have total knowledge about the world.

→ Typically, a bird can fly. Unless it is a penguin, ostrich, kiwi etc.

Possible formalization in PL1 (first order predicate logic):

∀x: Bird(x) ∧ ¬Penguin(x) ∧ ¬Ostrich(x) ...
⇒ Flies(x)

→ Problems:

1) Can we ever specify „...” completely?
2) We cannot conclude for any bird that it can fly, if we do not know that it is not a penguin, not a ostrich, not a kiwi and not a ...

Approach

→ We would like to have something like: „typically, birds can fly”.

• Possibility: Additional (non-logical) inference rules:

\[
\text{Bird}(x): \text{Flies}(x) \\
\text{Flies}(x)
\]

Intended meaning:
If x is a bird and there is no evidence that Flies(x) might be false, then assume Flies(x).

• Exceptions are represented by simple implications:

\[
\forall x: \text{Penguin}(x) \Rightarrow \neg \text{Flies}(x) \\
\forall x: \text{Ostrich}(x) \Rightarrow \neg \text{Flies}(x) \\
\forall x: \text{Kiwi}(x) \Rightarrow \neg \text{Flies}(x)
\]

Monotonicity of PL1: if X is a consequence of S, then it is also a consequence of any set containing S. One cannot preempt conclusions by adding new premises.

Formal Framework for Non-Monotonic Reasoning

• PL1 with “¬” traditional provability and Cn consequence operation. We consider only sets of closed formulas.

• Default rules

\[
\alpha(x), \beta(x), \ldots, \beta_n(x) \implies \gamma(x)
\]

in which \(\alpha(x), \beta(x), \ldots, \beta_n(x), \gamma(x)\) are formulas with free variables \(x = x_1, \ldots, x_n\).

o \(\alpha(x)\) : Prerequisite \(\rightarrow\) must be provable
o \(\beta(x)\) : Justification \(\rightarrow\) consistent assumptions
o \(\gamma(x)\) : Consequent

A default rule is called closed, if it does not have any free variables.

• Default theory: A tuple \((D, W)\) in which \(D\) is a countable set of default rules and \(W\) is a countable set of PL1 formulas.

• Closed default theory: All default rules are closed.

→ From now on, we consider closed default theories.
Properties of Default Reasoning

**Non-Monotonic Reasoning**
Assume there is exactly one default \[ \frac{A}{B} \] and nothing more is known. Then we can conclude B.
If the fact \( \neg A \) is added, B cannot be concluded!

**Non-Determinism**
\[
\begin{align*}
\text{Spouse}(x, y) \land \text{hometown}(y) = z & : \text{hometown}(x) = z \\
\text{hometown}(x) = z & : \text{hometown}(x) = z \\
\text{Employer}(x, y) \land \text{location}(y) = z & : \text{hometown}(x) = z \\
\text{hometown}(x) = z & : \text{hometown}(x) = z
\end{align*}
\]
Furthermore:
\[
\text{Spouse}(\text{mary}, \text{tom}), \text{hometown}(\text{tom}) = \text{toronto}, \text{Employer}(\text{mary}, \text{univ}), \text{location}(\text{univ}) = \text{vancouver}
\]

→ Default rules are used to complement incomplete theories. Thereby, several complements can be possible: „multiple extensions“ (credulous strategy).

→ Alternatively, the intersection over all possible extensions can be used. (skeptical strategy).

Additional Examples

- Default assumptions in frames and semantic networks.
Example: By default, the hometown of a person is Palo Alto.

  ```
  [Person UNIT Basic
   
   <hometown ((a city) Palo Alto; DEFAULT)>
   
   Person(x) : hometown(x) = y
   hometown(x) = y
  ]
  ```

- **Closed World Assumption.** For each base relation \( R \) in a DB:

  \[
  \frac{\neg R(x, \ldots, x_n)}{\neg R(x, \ldots, x_n)}
  \]

- **Frame Default (as an alternative to frame axioms).** For each relation \( R \) with a situation argument \( s \) and transition function \( f \):

  \[
  \frac{R(x, s) : R(x, f(x,s))}{R(x, f(x,s))}
  \]

Problems in Default Logic:
Case Analysis in Default Logic

Example:

\[
\frac{\text{Emu}(x) ; \neg \text{Fly}(x)}{\neg \text{Fly}(x)}, \quad \text{Ostrich}(x) ; \neg \text{Fly}(x)
\]

We cannot infer from \( (\text{Emu}(\text{Tweety}) \lor \text{Ostrich}(\text{Tweety})) : \neg \text{Fly}(\text{Tweety}) \).

Two solutions:

1) Combine the default rules to a single default:

\[
\frac{\text{Big-Bird}(x) ; \neg \text{Fly}(x)}{\neg \text{Fly}(x)}
\]

and add \( \text{Emu}(x) \lor \text{Ostrich}(x) \Rightarrow \text{Big-Bird}(x) \) to the facts.

2) Reformulate the defaults (no prerequisite)

\[
\begin{align*}
\text{Emu}(x) & \Rightarrow \neg \text{Fly}(x) \\
\text{Ostrich}(x) & \Rightarrow \neg \text{Fly}(x)
\end{align*}
\]

Extensions of Default Theories

The default rules \( D \) extend the PL1 theory that is given by \( W \): Extensions of \( D, W \).

Desirable properties of an extension \( E \) of \( D, W \):

- includes the facts: \( W \subseteq E \)
- is deductively closed: \( E = \text{Cr}(E) \)
- all applicable defaults „fired“:

  If

  1) \( \frac{\alpha ; \beta_1, \ldots, \beta_n}{\gamma} \in D \),
  2) \( \alpha \in E \),
  3) \( \neg \beta_1, \ldots, \neg \beta_n \in E \)
  then \( \gamma \in E \).

- \( E \) only consists of those facts that must be in \( E \) according to these prerequisites.
Formal Definition of Extensions of Default Theories

Let $\Delta = (D, W)$ be a closed default theory. For each set of closed formulas $S$, $\Gamma(S)$ is the smallest set that meets the following requirements:

D1: $W \subseteq \Gamma(S)$,
D2: $Cn(\Gamma(S)) = \Gamma(S)$, (Fixpoint)
D3: If

1) $\left( \beta_1, \ldots, \beta_n \right) \in D,$
2) $\alpha \in \Gamma(S)$ and
3) $\neg \beta_1, \ldots, \neg \beta_n \notin S$,  
then $\gamma \in \Gamma(S)$.

A set of closed formulas $E$ is an extension of $\Delta$, if $E = \Gamma(E)$.

Examples

1) $D = \left\{ A, B, C \right\}, \quad W = \{ B \Rightarrow \neg A \land \neg C \},$ 
$E_1 = Cn(W \cup \{ A, C \}), \quad E_2 = Cn(W \cup \{ B \})$

2) $D = \left\{ C, D \right\}, \quad W = \{ \}, \quad E_1 = Cn(\neg D, \neg F)$

3) $D = \left\{ C, D \right\}, \quad W = \{ \}, \quad E_1 = Cn(\neg C), \quad E_2 = Cn(\neg D)$

4) $D = \left\{ A : \exists x P(x), \quad \neg A \right\}, \quad W = \{ \},$  
$E_1 = Cn(\neg A)), \quad E_2 = Cn(A, \exists x P(x))$

Semi-Normal Defaults

Besides normal defaults, the semi-normal defaults are useful in AI systems:

$$\frac{\alpha \land \gamma}{\neg \beta}$$

Important for interacting normal defaults:

$$\begin{align*}
\text{Adult(x)} : & \text{Employed(x)} \\
\text{Dropout(x)} : & \text{Adult(x)} \\
\text{Dropout(x)} - & \text{Employed(x)}
\end{align*}$$

For so called „ordered default theories”, a proof theory exists [Etherington 88].
But, this kind of conflict resolution is not simple. Especially for more complex theories.

Temporal Projection in Situation Calculus

Given: the description of a situation and sequence of events.  
Which properties are valid after these events?

Situations: $S_i, s, \ldots$ (temporal intervals without any changes).
Properties: awake, dead, $f, \ldots$ (situations, for which the property is valid)

Situational statements: Holds(awake, $S_i$), ...

Events: wakeup, shoot, $e, \ldots$

Results of events are situations: Result(wakeup, $S_i$), ...

Laws of causality: $\forall s$: Holds(awake, Result(wakeup, $s$)), ...
The Frame Problem

Given:

\[ \forall s \text{:: } \text{Holds(awake, Result(wakeup, s))} \]
\[ S_1 = \text{Result(wakeup, } S_0) \]
\[ S_2 = \text{Result(eat-breakfast, } S_1) \]

Is: Holds(awake, S_2) valid?

Adding frame axioms. But

1) requires huge number of axioms \(|\text{Properties}| \times |\text{Events}|\);

2) the frame axioms must not always be true.

→ default: Holds(f, s) ⇒ Holds(f, Result(e, s)).

McCarthy’s formulation: All „normal“ facts are valid after „normal“ events:

\[ \forall f, e, s: \text{Holds}(f, s) \land \neg \text{ab}(f, e, s) \Rightarrow \text{Holds}(f, \text{Result}(e, s)) \]

„Minimization“ of the ab-predicate (→ circumscription with abnormal predicate).

Same result with DL:

\[ \text{\underline{\neg ab}(f, e, s)} \]