Production Systems and Rete Algorithm Formalisation
Horatiu Cirstea, Claude Kirchner, Michael Mm Moossen, Pierre-Etienne Moreau

To cite this version:

HAL Id: inria-00280938
https://inria.hal.science/inria-00280938
Submitted on 20 May 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Sous Projet 1

RETE et réécriture

Production Systems and Rete Algorithm Formalisation

Description: The rete algorithm is a well-known algorithm for efficiently addressing the many patterns/many objects match problem, and it has been widely used and implemented in several applications, mainly production systems. But despite of the wide usage of production systems and the rete algorithm, to the best of our knowledge there has been just one proposition for a formal definition of the rete algorithm given by Fages and Lissajoux [FL92], but no attempt to give a formal description of production systems as a whole, giving rise to lots of ambiguities and incompatibilities between the different implementations. Therefore, the need for a formalisation is clear and we present in this report a first approach to it, refining Fages and Lissajoux’s approach to fit it in our general model of production systems.

Auteur(s): Horatiu Cristea,
Claude Kirchner,
Michael Moossen,
Pierre-Etienne Moreau

Référence: MANIFICO/Sous Projet 1/Fourniture 1.1/V0.9

Date: 31 août 2004

Statut: à valider

Version: 0.9

Réseau National des Technologies Logicielles
Projet subventionné par le Ministère de l’économie, des finances et de l’industrie
ILOG, INRIA Lorraine, INRIA Rocquencourt
Historique

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 août 2004</td>
<td>V 0.9</td>
<td>Création du document</td>
</tr>
<tr>
<td>15 septembre 2004</td>
<td>V1.0</td>
<td>Version finale</td>
</tr>
</tbody>
</table>

Contents

1 Introduction to production systems 3
   1.1 Informal presentation of production systems ........................................... 3
   1.2 Motivation for a formal description .......................................................... 5

2 Formal Description of Production Systems 5
   2.1 General Production systems ........................................................................... 5
   2.2 Restricted Production systems ....................................................................... 8
   2.3 Formal Abstract Language ............................................................................... 10

3 The Rete Algorithm 12
   3.1 Informal Description of the Rete Network .................................................... 12
   3.2 Rete Network Execution for our Fibonacci Example ....................................... 13
   3.3 Formal Definition of the Rete Network ......................................................... 15
   3.4 Execution of the Rete Algorithm .................................................................... 17

4 Conclusions 18
Introduction

The rete algorithm is a well-known algorithm for efficiently addressing the many patterns/many objects match problem. It has been first described in [For74], but it gains in popularity just after publication of [For82] where a more precise description is given.

Implementation issues related to this algorithm have been widely studied [TR89, Thé90, MBLG90, Alb89, Ish94a], and applications include, for instance, Petri Nets implementations based on the rete algorithm [BB01]. But the most important applications are production systems like OPS5 [X-T88, For81], Clips [CR03], JEOPS [dFFR00], Jess [FH03] and JRules [S.A04]. There are also many specialized production systems, like parallel, distributed, multi-agent and real-time production systems [LG89, Lop87, FL91, Ish94b]. But, despite of all this work around production systems and the rete algorithm, to the best of our knowledge there has been just one proposition for a formal definition of the rete algorithm given by Fages [FL92], but no one for production systems as a whole, giving rise to lots of ambiguities and incompatibilities between the different implementations, so the need for a formalisation is clear and we present in this work a first approach to it.

Outline of this report: For a better understanding of the context, we will begin with an intuitive introduction to production systems, and how they take advantage of the rete algorithm in Section 1.1. In Section 1.2, we present also our motivations to give a formal definition of production systems and an unambiguous formal abstract language for specifying a production system program as given in Section 2.

Then, in Section 3, we introduce intuitively the rete algorithm and in Section 3.3, we recall and refine the formal description proposed by [FL92] for getting it to fit in our general model and so obtaining a comprehensive and consistent view of production systems.

1 Introduction to production systems

An important class of rule based languages is based on the notion of production rule which is a statement of the form “if condition then action”. This class of programming languages has been emerging from the artificial intelligence community at the beginning of the seventies and are quite popular as they provide an attractive blend of declarative and imperative features. They are at the heart of expert systems and have been more recently used under the name of “business rule”. A first comprehensive comparison of production systems can be found in [Thé94].

Basically all of them provide the same semantics for programming, as shown in the following informal description of the behavior and main components of production systems.

1.1 Informal presentation of production systems

A production system consists mainly of the following five components:

- The Fact Types are user defined datatypes, like structs with fields or properties. There are intended for organizing the data that will be manipulated, for instance, we can have a fact type representing a house with properties like color, price, availability, and so on. But, notice that in most cases, we are restricted to basic types for the properties, so it could not be possible to have a property of type address in the house fact type defined before, if the address is a composed data type.

  We can then view a fact as a concrete assignment of values to the properties for a given fact type, for instance, an available red house that costs one thousand.

- The Working Memory (WM) is the current program state, it is a global structure of facts. We will see later that this structure could be implemented either by sets or multisets.

- Production Rules are conditional statements of the form

  [Name] if Condition then Action
A rule has a name and it acts by addition and retraction of facts on the ‘WM’ iff the condition is fulfilled. Here the condition is usually called the left hand side (LHS) of the production rule and the action its right hand side (RHS). The condition may or not be satisfied by the ‘WM’ as described in the next section together with more precise explanation for condition and action. When the LHS of a rule is satisfied, the rule is said to be activated.

- The Production Memory (PM) is a structure of production rules, also known as Knowledge Base. It is almost always unvarying, in spite of some production system implementations that provide facilities to manipulate the production memory as RHS actions.

- A Resolution Strategy that consists of an algorithm for selecting just one rule to execute, if the conditions of the LHS of more than one rule are satisfied at the same time.

The production system interpreter executes a production system by performing a sequence of operations called recognize-act cycle or inference cycle:

1. **Matching**: evaluate the LHS of each rule to determine which ones are activated given the current state of the ‘WM’. This is the most time consuming step in the execution of a production system, and here is were the rete algorithm is used.

2. **Conflict Resolution or Selection**: select one activated rule. If no rule is activated, halt the interpreter returning the current state of the ‘WM’.

3. **Firing or Act**: perform the actions specified in the RHS of the selected rule.

4. go to step 1.

When a rule is activated, an instantiation\(^1\) is generated as an ordered pair of the form:

<rule, list of facts that satisfy its LHS>.

Instantiations are maintained in the Conflict Set (CS). Then, the Resolution Strategy selects just one rule of this set, and its RHS is executed; it is said that the rule is fired.

A schematic view of the data work flow in a production system is shown in Figure 1.

---

\(^1\)this is a historical name, that does not reflect the common meaning of instantiation
1.2 Motivation for a formal description

Of course, each production system implementation has its own concrete syntax, facilities and behavior. For instance, facts usually could be non-ordered or ordered, and it could be possible to use objects as facts, as in the Java based implementations.

Most languages also consider the possibility to interact with the user by executing special commands in the RHS of the rules, or also the ability to modify the PM, or the possibility to choose between a set-based WM or a multiset-based WM (i.e. two identical facts can exist simultaneously in the WM).

In general, it is also possible to modify a fact, which is usually internally implemented as a retraction followed by an addition.

There are also languages that instead of a CS implement an Agenda, which is a list of (already) sorted instantiations. Thus, the resolution strategy consists in choosing the first element of the agenda.

In what follows we propose a formalisation that does not always handle these (implementation related) extensions but we discuss the way this can be implemented.

Example 1.1 A kind of identity rule could be described as:

\[
\text{[dummy]} \text{ if there is an instance of fact type A then Do Nothing}
\]

where the “Do Nothing” could be implemented in several ways, depending on the language facilities and the programmer interpretation, like, for instance:

- Really doing nothing, adding and retracting no fact.
- retracting a fact in the WM and adding it again. If, for instance, we do so with a fact of type A, we will obtain an infinite loop.
- Adding an arbitrary fact and retracting it. If the fact is not related to the current program, doing so is (almost) the same as really doing nothing.
- Using the language facility for modifying an arbitrary fact, but without really modifying any value of its properties. Usually, doing so generates a compilation time error, but there are production system that allows to do that.

This illustrates the fact that the handling of the conflict set could be unclear and that consequently a more precise description of the systems behavior should be provided. We will first identify the various formal components of a production system, getting the intended semantics from the behavior of existing implementations. In a second step not recorded in the report, we will address the specification of strategies as well as the description of their semantics.

2 Formal Description of Production Systems

Now that the need for a formal definition of production systems is clear, we will give a formal description of production systems.

2.1 General Production systems

We consider a set \( F \) of function symbols, usually denoted \( f, g, h, \ldots \), a set \( P \) of predicate symbols, and infinite sets \( X \) and \( L \) respectively called set of variables and of labels. Variables are denoted \( x, y, z, \ldots \). In most of the practical situations, finite set of labels are enough. These sets are assumed to be disjoint. Each function symbol and predicate symbol has a fixed arity. Nullary function symbols are called constants. We assume that there is at least one constant. The set of terms
(denoted $T(F, X)$), ground terms (denoted $T(F)$), atomic propositions, literals (i.e. atomic proposition or their negation), propositions, sentences (i.e. closed propositions) are defined as usual in term rewriting [KK99, BN98, “T02] and first-order logic [Gal86].

We will freely use the usual notion of substitution. Notice that since in general first order propositions are instantiated, the substitution mechanism works modulo alpha-conversion to take care of the variable bindings.

**Definition 2.1** A fact $f$ is a ground term, $f \in T(F)$.

**Definition 2.2** We call working memory (WM) a set of facts, i.e. it is a subset of the Herbrand universe defined on the signature.

**Definition 2.3** A production rule or simply rule or production, denoted

$$[l] \text{if } p, c \text{ remove } r \text{ add } a$$

consists of the following components.

- A name from the label set: $l \in L$.
- A set of positive or negative patterns $p = p^+ \cup p^-$ where a pattern $p_i \in T(F, X)$ and a negated pattern is denoted $\neg p_i$. $p^-$ is the set of all negated patterns and $p^+$ is the set of the remaining patterns.
- A proposition $c$ whose set of free variables is a subset of the pattern variables: $\text{Var}(c) \subseteq \text{Var}(p)$.
- A set $r$ of terms whose instances could be intuitively considered as intended to be removed from the working memory when the rule is fired, $r = \{ r_i \}_{i \in L}$, where $\text{Var}(r) \subseteq \text{Var}(p^+)$.
- A set $a$ of terms whose instances could be intuitively considered as intended to be added to the working memory when the rule is fired, $a = \{ a_i \}_{i \in L}$, where $\text{Var}(a) \subseteq \text{Var}(c)$.

Such a rule is also denoted $[l] p, c \Rightarrow r, a$.

**Remark:** Indeed in the previous definition, one can discuss the choice of set as the data structure to represent, the WM, $p$, $r$ and $a$.

**Definition 2.4** Given a set of facts $S$ and a set of positive patterns $p^+$, $p^+$ is said to match $S$ with respect to a theory $T$ and a substitution $\sigma$, written $p^+ \subseteq_T S$ if:

$$\forall p \in p^+ \exists t \in S \mid \sigma(p) =_T t$$

We say that a set of negative patterns $p^-$ dis-matches a set of facts $S$, written $p^- \not\subseteq_T S$ iff:

$$\forall \neg p \in p^- \forall t \in S \forall \sigma \mid \sigma(p) \not=_T t$$

**Definition 2.5** Given a substitution $\sigma$, so that $\text{Dom}(\sigma) = \text{Var}(c)$, a production rule $[l] p, c \Rightarrow r, a$ is $(\sigma, \text{WM}^\prime)$-fireable on a working memory WM when

1. $p^+ \subseteq_T \text{WM}^\prime$
2. $T \models \sigma(c)$
3. $\sigma p^- \not\subseteq_T \text{WM}$

for a minimal (with respect to the subset ordering) subset $\text{WM}^\prime$ of $\text{WM}$. A fireable rule is also called an activation.
**Definition 2.6** Given a production rule \([ l ] \ p, c \Rightarrow r, a\) which is \((\sigma, WM')\)-fireable on a working memory \(WM\), its *application* leads to the new working memory \(WM''\) defined as:

\[
WM'' = (WM - \sigma(r)) \cup \sigma(a)
\]

This is denoted \(WM \Rightarrow WM''\) or simply \(WM \Rightarrow WM''\). The couple \((\sigma, WM')\)-action of the production rule \([ l ] \ p, c \Rightarrow r, a\) on the working memory \(WM\).

**Definition 2.7** For a given working memory \(WM\) and a set of production rules \(R\), the set

\[
CS = \{ (l, \{ f_1, \ldots, f_k \}) | \exists ([l] \ p, c \Rightarrow a, r) \in R \text{ which is } (\sigma, WM')\text{-fireable on } WM
\]

is called the \(R@WM\)-conflict set, where \(WM'' = \{ f_1, \ldots, f_k \}\).

A conflict set could be either empty (no rule is fireable), unitary (only one rule can fire), finite (a finite number of rule is activated) or infinitary (an infinite number of matches could be found due to the theory modulo which we work [FH86]). Whether finite or infinite, one should decide which rule should be applied: this is one of the major topics of interest in production systems, addressed by *resolution strategies*.

**Definition 2.8** A *resolution strategy* is a computable function that given a set of production rules \(R\), and a production derivation

\[
WM_0 \Rightarrow WM_1 \Rightarrow \cdots \Rightarrow WM_n
\]

returns a unique element of the \(R@WM_n\)-conflict set.

We have now all the ingredients to provide a general definition of production systems:

**Definition 2.9** A *general production system* is defined as

\[
GPS = (\mathcal{F}, \mathcal{P}, \mathcal{X}, \mathcal{L}, WM_0, R, S, T)
\]

Where:

- \(\mathcal{F}\) is the set of function symbols,
- \(\mathcal{P}\) is the set of predicate symbols,
- \(\mathcal{X}\) is the set of variables,
- \(\mathcal{L}\) is the set of labels,
- \(WM_0\) is the initial working memory,
- \(R\) is the set of production rules over \(\mathcal{H} = (\mathcal{F}, \mathcal{P}, \mathcal{X} \mathcal{L})\),
- \(S\) is the resolution strategy,
- \(T\) is the matching theory.

**Remark:** This definition of a production system is quite general as the pattern can be deep and non-linear, the condition can be an arbitrary first-order proposition, the resolution strategy can take the full derivation history into account.

**Definition 2.10** The *Inference Cycle* is defined as follows:

1. **Matching**
   \([ l ] \ p, c \Rightarrow r, a\) is \((\sigma, WM')\)-fireable

2. **Firing**
   \(WM \Rightarrow WM''\)

3. **Conflict Resolution**
   \(S(CS, WM_0 \Rightarrow \cdots \Rightarrow WM_n) = (l, \sigma)\)
2.2 Restricted Production systems

We have a general formal description of a production system, but in order to get good performances when running production systems, suitable restrictions have been used for the so called rete algorithm. For presenting these restrictions we need the following definition

**Definition 2.11** The *height* of a term \( t \), is defined as

\[
h(t) = \begin{cases} 
0, & \text{if } t \text{ is a variable or a constant,} \\
1 + \max_{i \in \mathbb{N}} (h(s_i)), & \text{if } t = f(s_1, \ldots, s_n) \text{ with } n \geq 1.
\end{cases}
\]

We restrict general production system to be:

**Fact Shallow**: The facts are restricted to ground terms of height one,

\[ \forall i \in I_f \mid h(f_i) = 1 \]

**Pattern Shallow**: The patterns are restricted to non variable terms of height less or equal than one,

\[ \forall i \in I_p \mid h(p_i) \leq 1 \]

**Simple**: The condition \( c \) is restricted to a conjunction of literals (i.e. atomic formulas or their negation)

**Syntactic**: The matching theory is the empty theory.

**Static**: The facts to be removed can not be built dynamically.

\[ \forall r_j \in r \exists p_i \in p^+ \mid \sigma r_j = \alpha p_i \]

**Remark**: Some recent developed production systems, as they based on Java, may allow more general facts and patterns facts.

**Definition 2.12** A \( S^4 \) production system (Shallow, Simple, Syntactic, Static) is defined as

\[ \mathcal{PS} = ( \mathcal{F}, \mathcal{P}, \mathcal{X}, \mathcal{L}, \mathcal{WM}_0, \mathcal{R}, S ) \]

Where:

- \( \mathcal{F} \) is the set of function symbols,
- \( \mathcal{P} \) is the set of predicate symbols,
- \( \mathcal{X} \) is the set of variables,
- \( \mathcal{L} \) is the set of labels,
- \( \mathcal{WM}_0 \) is the initial working memory, restricted to terms of height 1,
- \( \mathcal{R} \) is the set of production rules over \( \mathcal{H} = ( \mathcal{F}, \mathcal{P}, \mathcal{X} ) \), where rules have patterns of height 1 and conditions restricted to a conjunction of literals
- \( S \) is the resolution strategy.

Using the above definition, we are able to specify a production system in an unambiguous, language independent and formal way.
Example 2.1 This example is a way for computing the Fibonacci number of 200, it is not the best example for the usage of production systems, but it shows most of their capabilities in a short and simple way. We assume that numbers and arithmetics are builtin.

\[ \mathcal{P} \ := \ \{ \ >/2, \ =/2 \} \]
\[ \mathcal{F} \ := \ \{ \ \text{fib}/2, \ =/-2 \} \]
\[ \mathcal{X} \ := \ \{ \ x \ | \ x \text{ begins with a question mark} \} \]
\[ \mathcal{L} \ := \ \{ \ \text{GoDown}, \text{GoUp} \} \]
\[ \mathcal{W}_{M0} \ := \ \{ \ \text{fib}(0, 1), \ \text{fib}(1, 1), \ \text{fib}(200, -1) \} \]
\[ \mathcal{R} \ := \ \{ \]
\[ \quad \left[ \text{GoDown} \right] \]
\[ \quad \text{fib}(?n, -1) \land \neg \text{fib}(?n1, ?v) , \]
\[ \quad ?n1 = ?n - 1 \]
\[ \quad \Rightarrow \]
\[ \quad \emptyset , \]
\[ \quad \left\{ \ \text{fib}(?n1, -1) \right\} \]
\[ \left[ \text{GoUp} \right] \]
\[ \quad \text{fib}(?n, -1) \land \text{fib}(?n1, ?v1) \land \text{fib}(?n2, ?v2) , \]
\[ \quad ?n1 = ?n - 1 \land ?v1 > 0 \land ?n2 = ?n - 2 \land ?v2 > 0 \land ?v = ?v1+?v2 \]
\[ \quad \Rightarrow \]
\[ \quad \left\{ \ \text{fib}(?n, -1), \ \text{fib}(?n2, ?v2) \right\} , \]
\[ \quad \left\{ \ \text{fib}(?n, ?v) \right\} \]
\[ \} \]
\[ \mathcal{S} \ := \ \text{a FIFO strategy} \]

Example 2.2 This is another example for searching for a house which is more near to the real usage of production systems, we assume numbers, arithmetics and strings are builtin:

\[ \mathcal{P} \ := \ \{ \ </2, \ =/2 \} \]
\[ \mathcal{F} \ := \ \{ \]
\[ \quad \text{house}/4, \ \text{houseaddress}/4, \ \text{myaddress}/3, \ \text{war}/2, \ \text{searching}/0, \]
\[ \quad \text{red}/0, \ \text{blue}/0, \ \text{usa}/0, \ \text{irak}/0, \ \text{france}/0 \]
\[ \} \]
\[ \mathcal{X} \ := \ \{ \ x \ | \ x \text{ begins with a question mark} \} \]
\[ \mathcal{L} \ := \ \{ \ \text{HouseSearch} \} \]
\[ \mathcal{W}_{M0} \ := \ \{ \]
\[ \quad \text{house}(1, \text{red}, 341, \text{true}), \ \text{houseaddress}(1, 251, \text{"rue jeanne d’arc"}, \text{"nancy"}), \]
\[ \quad \text{house}(2, \text{blue}, 390, \text{true}), \ \text{houseaddress}(2, 121, \text{"avenue de brabois"}, \text{"villers les nancy"}), \]
\[ \quad \text{house}(3, \text{red}, 415, \text{true}), \ \text{houseaddress}(3, 31, \text{"rue carnot"}, \text{"vandoeuvre les nancy"}), \]
\[ \quad \text{myaddress}(2551, \text{"gorbea"}, \text{"santiago"}), \]
\[ \quad \text{war}(\text{usa}, \text{irak}), \]
\[ \quad \text{searching}() \]
\[ \} \]
2.3 Formal Abstract Language

We will use in the rest of this report the same abstract syntax as in the example above. Therefore from a practical point of view, it is not needed to give everything in an explicit way for specifying a production system. For instance,

- if the resolution strategy is not explicitly given, a default resolution strategy, can be used, like a FIFO strategy, for instance.

- if the set of variables is missing, a default variables set can be used, like:

\[ X = \{ x \mid x \text{ begins with a question mark} \} \]

When clear from the context a production system is just:

\[ PS = ( F, WM_0, R ) \]

From the abstract syntax used in example 2.1, we can express common production systems programs in a formal way.

**Example 2.3** For instance, this JRules-TRL program [S.A04] is a concrete implementation of the Fibonacci example 2.1:

- Fact type declarations

\[
\begin{align*}
\text{public class Fib} & \{ \\
\text{public int number;} & \\
\text{public int value;} & \\
\} \\
\end{align*}
\]
• Working memory initialization

assert [ ] [ ] Fib [ ]
so that number = 0
and value = 1
assert [ ] [ ] Fib [ ]
so that number = 1
and value = 1
assert [ ] [ ] Fib [ ]
so that number = 200
and value = -1

• Rule declarations

[GoDown]
WHEN
there is a [ ] Fib [ ] [ called ?f ]
[where]
such that value = -1
there is no [ ] Fib [ ]
[where]
such that number = ?f.number - 1
THEN
assert [ ] [ ] Fib [ ]
so that number = ?f.number
and value = -1
ELSE

[GoUp]
WHEN
there is a [ ] Fib [ ] [ called ?f1 ]
[where]
such that value = -1
there is a [ ] Fib [ ] [ called ?f2 ]
[where]
such that number = ?f1.number - 1
and value > 0
there is a [ ] Fib [ ] [ called ?f3 ]
[where]
such that number = ?f2.number - 2
and value > 0
THEN
modify [ ] ?f1
so that value = ?f2.value + ?f3.value
retract ?f3
ELSE

Going one step forward, we are already able to develop a tool, for translating a program in this abstract notation into concrete programs for several specific production systems.

Remark: We already began to implement such a tool using TOM [MRV01].
3 The Rete Algorithm

The rete algorithm, as said before is an efficient algorithm for solving the many patterns/many objects match problem. Forgy’s paper [For82] represents the rete algorithm viewed as a black box as:

Changes to the $\mathcal{WM}$
\[ \downarrow \]
Rete Algorithm
\[ \downarrow \]
Changes to the $\mathcal{CS}$

Figure 2: Rete algorithm as a Black Box

But, more precisely it can just take one input token at once, and it may generate zero, one or more changes to the $\mathcal{CS}$.

The main idea of the rete algorithm, as stated by [FL92], is to compute the set of rule instantiations incrementally, based on two main approaches:

- **Memorisation:** in general, the set of rule instantiations does not change dramatically from one cycle to the next one. So, the idea is to compute just these changes. For this, partial rule instantiations are held and maintained.

- **Sharing:** if several rules are using some conditions in common, the rules are factorized over their common conditions.

We can also state that the rete algorithm is decomposable into two very different tasks; one is to build the *rete network* given the set of rules $\mathcal{PM}$, and the another is about the usage or execution of this network.

We will first give informal descriptions of these two tasks, and then recall and refine Fages and Lissajoux’ formalisation [FL92] to fit in our previous definition of production system.

3.1 Informal Description of the Rete Network

The rete algorithm works generating a workflow graph built based on the $\mathcal{PM}$. Where its input is the set of changes to the $\mathcal{WM}$, differentiating recently added facts, called *positive tokens*, and recently removed facts, called *negative tokens*, which are treated in a similar way. Its output are instantiations, positive and negative ones, to be added or removed from the $\mathcal{CS}$.

This directed graph may have several different kinds of nodes:

- **The Root Node:** this is the only entry point to the network, it receives the tokens and passes copies of them to all its successors. We represent it as a box node labeled **Root**.

- **One-Input Nodes:** these nodes, also called *Anodes*, perform the intra-elements tests, that are the conditions which depends on just a single pattern. If the test success it passes copies of the given token to all its successors. And they are of different types:
  - For type tests, for example, is the received token a fact of type *house*? Represented in oval shapes labeled with the fact type to be checked. These nodes are also called *Tnodes*.
  - For condition tests, for example, is the *color* of the received *house red*? Represented in diamond shapes and labeled with the condition to be checked.
  - For intra-relation tests, when a same variable appears more than once in a single pattern. Is the color of the *windows* equals to the color of the *doors* of the *house*??
• **Two-Input Nodes:** these nodes, also called Bnodes, are useful for testing the inter-element conditions, conditions which involve more than one single pattern. For instance, when two patterns are related due to a common variable. These nodes have two different local memory slots for storing the tokens arriving at each of its inputs. If a positive token arrives, it is stored in the local memory, if a negative token arrives, it is removed from the local memory. And, in both cases, if the test is fulfilled, the generated token passed to its successors will have the same sign than the last arrived token. Represented as rounded box shapes with two input arrows and labeled with the inter-element condition. If a token arrives at one of its input, the condition will be checked against all the tokens in the another input’s local memory. A common notation for referencing the different tokens in the condition is to use the prefixes $l$ for left and $r$ for right.

There are two different kinds of Bnodes:

- *Any* nodes, related to inter-element tests between only positive patterns or between only negative patterns.
- *Not* nodes, related to inter-element tests between a positive pattern and a negative pattern.

• **Terminal Nodes:** these nodes will just receive tokens which instantiate the LHS of a rule, so it will be the output of the network and the input for the CS. Represented as box shapes, and the label of the name of the activated or deactivated rule, depending of the sign of the arriving tokens.

**Example 3.1** The rete network for our Fibonacci example is shown in figure 3.

![rete network for Fibonacci example](figure3)

**Figure 3:** Rete network for Fibonacci example

### 3.2 Rete Network Execution for our Fibonacci Example

As an informal introduction to how the rete network is executed, we will take the rete network presented in the previous section, and execute the program given in example 2.1 but initializing
the working memory with

\[ \text{WM}_0 \ := \ \{ \text{fib}(0, 1), \text{fib}(1, 1), \text{fib}(2, -1) \} \]

Notice that the network is providing slots names to be more clear, so the first slot of the \text{fib} fact is called \textit{number}, and the second \textit{value} (it could be also numbered with 0 and 1; or 1 and 2).

**Example 3.2** Rete Network Execution.
First, we initialize
\[ \text{WM} := \emptyset \land \text{CS} := \emptyset \]
then, we add the first fact in the \text{WM}_0 to the \text{WM}
\[ \text{WM} := \text{WM} \cup \{ \text{fib}(0, 1) \} \]
and we send the respective token to the rete network, which first checks for the type of the token, if it is equals to \text{fib} as in this case, it passes the token to the next checks, if not the token is cast away.
At the second level it checks if \textit{value} = -1, it is not the case, so the token on the left branch is cast away, on the right branch we check for \textit{value} > 0, as it is true it gives the token to both successors, which are both two input nodes, with no token in theirs left memory, so the tokens remain waiting.
Then, we add the second fact in the \text{WM}_0 to the \text{WM}
\[ \text{WM} := \text{WM} \cup \{ \text{fib}(1, 1) \} \]
and the progress is the same as before, the token stays waiting at both two input nodes of the right branch in their right memory.
Finally, we add the last fact in the \text{WM}_0 to the \text{WM}
\[ \text{WM} := \text{WM} \cup \{ \text{fib}(2, -1) \} \]
in this case, the left branch is taken, and the check \textit{value} = -1 succeeds so the token propagates to all its successors. From left to right:
- the token is hold in the left memory of the negative two input node.
- the token is given to the right input of the negative node, and given that the condition is false, both tokens stay waiting for future tokens that could match.
- at least, the token is given to the two input node, where there are \text{fib}(0, 1) and \text{fib}(1, 1) waiting for action. And the condition holds for \text{fib}(2, -1) and \text{fib}(1, 1), so the just incoming token is given to the next two input token where \text{fib}(0, 1) and \text{fib}(1, 1) are waiting and now the condition holds with \text{fib}(0, 1) and finally the \textit{GoUp} rule is activated,
\[ \text{CS} := \text{CS} \cup < \text{GoUp}, \{ \text{fib}(0, 1), \text{fib}(1, 1), \text{fib}(2, -1) \} > \]
Now, our FIFO strategy selects the first instantiation from the \text{CS}, updating it as
\[ \text{CS} := \text{CS} \setminus < \text{GoUp}, \{ \text{fib}(0, 1), \text{fib}(1, 1), \text{fib}(2, -1) \} > \]
and executes the right hand side, first computing the substitution \( \sigma \) as
\[ \sigma := \{ \ ?n \rightarrow 2, \ ?n1 \rightarrow 1, \ ?n2 \rightarrow 0, \ ?v1 \rightarrow 1, \ ?v2 \rightarrow 1, \ ?v \rightarrow 2 \} \]
for then, removing \text{fib}(2, -1) from the working memory
\[ \text{WM} := \text{WM} \setminus \{ \text{fib}(2, -1) \} \]
So, a negative token is given to the rete network, which checks the type, then it checks the value and takes the left branch, arriving at the negative two input node, both previous stored positive tokens are cancelled with the new ones that are arriving. Where on the right it matches in the same way as the positive token, so the negative token arrives at the GoUp terminal node, removing the given instantiation if there would be one, but this is not the case, so it is casted away.

Follows to remove \( \text{fib}(0, 1) \), and to add \( \text{fib}(2, 2) \), so no more rules are activated and the final state of the \( \mathcal{WM} \) is

\[
\mathcal{WM} := \{ \text{fib}(1, 1), \text{fib}(2, 2) \}
\]

### 3.3 Formal Definition of the Rete Network

In this section we will describe how the rete network is build, for this we give a more general alternative of Fages and Lissajoux’ equations (5) and (6) of [FL92], using the same notations used for defining a production system in Section 2.

**Definition 3.1** We define the position \( \omega \) of a subterm in a term as:

\[
\begin{align*}
\omega(t) &= t = 0 \\
\omega(s_i) &\mid t(s_1, \ldots, s_n) = i \\
\omega_1 = \omega(s) \mid t &\land \omega_2 = \omega(t) \land r \implies \omega(s) \mid r = \omega_1.\omega_2
\end{align*}
\]

**Definition 3.2** We define the compilation \( \text{comp} \) of a given rule \( [l] p, c \Rightarrow r, a \), in an incremental way as:

\[
\begin{align*}
\text{comp}(t) &:= \text{comp}(t, 0) \text{ for } t \in T \ (X, \mathcal{P}) \\
\text{comp}(t(s_1, \ldots, s_n)) &:= \text{input}.\omega = t \\
&\bigwedge_{i \in I} \text{comp}(s_i, w.i) \\
&\bigwedge_{i, j \in I} \exists \omega_1, \omega_2 \mid s_j | \omega_1 = s_i | \omega_2 \\
\text{input}.\omega.\omega_1 = \text{input}.\omega.\omega_2 \\
\text{comp}(\neg t(s_1, \ldots, s_n)) &:= \emptyset \text{input} \in \mathcal{WM} \mid \text{comp}(t(s_1, \ldots, s_n)) \\
\text{comp}(p, c) &:= \bigwedge_{i \in I} \text{comp}(p_i) \\
&\bigwedge_{i, j \in I} \exists \omega_1, \omega_2 \mid p_i | \omega_1 = p_j | \omega_2 \in X \\
\text{input}_i.\omega_1 = \text{input}_2.\omega_2 \\
&\bigwedge_{i, j \in I} \exists \omega_1, \omega_2 \mid p_i | \omega_1 = p_j | \omega_2 \in X \land \mathcal{FM} \\
\text{input}_i.\omega_1 = f.\omega_2 \\
&\bigwedge_{i \in I} \exists \omega_1 \mid p_i | \omega_1 = x \in \text{Var}(c) \\
\text{input}_i.\omega = x =: \sigma_1 \land \sigma_1(c)
\end{align*}
\]

where \( \text{input} \) is a fact that may match the given pattern, as well \( \text{input}_i \) for pattern \( p_i \).

**Remark:** This is just an incremental definition of

\[
p^+ \ll^\omega \mathcal{WM} \land p^- \not\ll \mathcal{WM} \land \models \sigma(c)
\]

as described in Definitions 2.4 and 2.5.

Now, we classify all tests over \( \text{comp}(p, c) \) for a given rule, identified by its label, \( l_j \), in \( \mathcal{PM} \).

First, we call \( q_i \) the conjunction of all tests in the form of subequations (1a) and (1b) for rule \( l_j \) and pattern \( i \), these are the intra-elements tests or the test mono-schéma as defined in Section 3.1 of [FL92].

And, we call \( r_i \) the conjunction of all tests in the form of subequations (1c) and (1d) for rule \( l_j \), these are the inter-element tests or the test multi-schéma as defined in Section 3.1 of [FL92].
The incremental approach of the rete algorithm comes from the memorization of the tuples of facts that satisfy a partial left hand side $\pi$ of a rule in left and right memories, $lm$ and $rm$.

**Definition 3.3** For a partial left hand side of a rule $\pi$ we can define the left memory, $lm$, and the right memory, $rm$, as following:

$$
\begin{align*}
lm_i &= \{ (f_1, \ldots, f_k) \in \mathcal{WM} \mid \text{comp}(\pi)(f_1, \ldots, f_k) \} \\
rm_i &= \{ f \in \mathcal{WM} \mid q_{i+1}(f) \} \\
lm_1 &= rm_0
\end{align*}
$$

Finally, we can begin to build the network by a root node. Then, we will have two kinds of nodes Anodes for intra-element tests, and Bnodes for inter-element tests. For each rule $l_i \in \mathcal{PM}$, we connect the root node with each Anode($q_{i,0}^1$), and we connect the whole sequence of intra-element tests as follows:

$$
\forall k \in K \mid \text{Anode}(q_{i,k}^1) \rightarrow \text{Anode}(q_{i,k+1}^1)
$$

And, in the case there is the same intra-element condition in two different rules, we share it, i.e. if we have that

$$
q_{i_1,k_1}^h = q_{i_2,k_2}^h
$$

we build a single Anode $A$ for it, and we connect

$$
\text{Anode}(q_{i_1,k_1-1}^h) \rightarrow A \rightarrow \text{Anode}(q_{i_1,k_1+1}^h), \text{ and }
\text{Anode}(q_{i_2,k_2-1}^h) \rightarrow A \rightarrow \text{Anode}(q_{i_2,k_2+1}^h)
$$

Then, we connect the inter-element test nodes, or Bnodes, by associating each left and right memory, $lm$ and $rm$, to a Bnode. Bnodes related to a subequation (1c) are called positive Bnodes and labeled Any$^l_{i_1,i_2}$, for rule $l_i$ and patterns $i_1$ and $i_2$, and the related to a subequation (1d) are called negative Bnodes and labeled Not$^l_{i_1,i_2}$. Each Bnode is associated three memories: the left input memory $lm_{i-1}$, the right input memory $rm_{i-1}$ and the output memory $lm_i$.

**Example 3.3** For our Fibonacci example, given the positions $\omega$ names, as:

$$
\begin{align*}
\omega &= 0 \equiv \text{type} \\
\omega &= 1 \equiv \text{number} \\
\omega &= 2 \equiv \text{value}
\end{align*}
$$

so, we have:

$$
\begin{align*}
q_{1,0}^1 &= q_{2,0}^1 = q_{1,0}^2 = q_{2,0}^2 = q_{3,0}^2 \equiv \text{input.type} = \text{fib} \\
q_{1,1}^1 &= q_{1,1}^2 \equiv \text{input.value} = -1 \\
q_{2,1}^2 &= q_{3,1}^2 \equiv \text{input.value} > 0 \\
\text{Not}^l_{1,2} &= \equiv \text{input}_1.\text{number} - 1 = \text{input}_2.\text{number} \\
\text{Any}^l_{1,2} &= \equiv \text{input}_1.\text{number} - 1 = \text{input}_2.\text{number} \\
\text{Any}^l_{2,3} &= \equiv \text{input}_1.\text{number} - 2 = \text{input}_3.\text{number} \\
\text{Any}^l_{2,3} &= \equiv \text{input}_2.\text{value} + \text{input}_3.\text{value}
\end{align*}
$$

The figure 4 shows the formal version of the rete network for our Fibonacci example.
Figure 4: Rete network for Fibonacci example

3.4 Execution of the Rete Algorithm

Once the rete network is built, it can be executed receiving as input changes to the \( \mathcal{WM} \) as tokens in the form \(< +|-, f >\) where \( f \) is a fact and + means \( f \) has been added to the \( \mathcal{WM} \) and – means removed.

And we define the following general transition rule:

\[ m? \xrightarrow{a} ms! \]

where \( m? \) is the input token, \( ms! \) is a list of output token and \( a \) is a optional action to perform.

Now, we specify the transition rule for each type of node.

For \( Anodes \) we have the following behaviour:

\[< +|-, f >? \rightarrow \{< +|-, f >\}! \text{ if } q(f)\]

\[< +|-, f >? \rightarrow \emptyset! \text{ if } \neg q(f)\]

And for \( Bnodes \), each right memory behaves like:

\[< -, f >? \xrightarrow{rm\rightarrow rm-\{f\}} \{< -, f >\}!\]

\[< +, f >? \xrightarrow{rm\leftarrow rm \cup \{f\} \land \alpha\leftarrow \alpha \cup \alpha' | \exists \mathcal{PM} \land \exists \mathcal{PM}'(R) | \alpha' \models f} \{< +, f >\}! \text{ if } q(f)\]

\[< +, f >? \rightarrow \emptyset! \text{ if } \neg q(f)\]

Where each left memory behaves like:

\[< -, f >? \xrightarrow{lm\rightarrow lm-\{f\}} \{< -, f >\}!\]
Now a Bnode could receive different tokens depending on the memory that sent it. So, it will have different behaviors depending on the memory that sent the token. For positive Bnodes we have:

Left:
\(<-, f?> \rightarrow [<-, f>]!
\(+, f?> \rightarrow [<+, (f, l) > \forall l \in rm \mid r(l, f) >]!

Right:
\(<-, f?> \rightarrow [<-, f>]!
\(+, f?> \rightarrow [<+, (l, f) > \forall l \in rm \mid r(l, f) >]!

And for negative Bnodes:

Left:
\(<-, f?> \rightarrow [<-, f>]!
\(+, f?> \rightarrow [<+, f>]! \text{ if } \forall l \in rm \mid \neg r(l, f)

Right:
\(<-, f?> \rightarrow [<+, l > \forall l \in lm \mid \forall f' \in rm \mid \neg r(l, f') >]!
\(<+, f?> \rightarrow [<-+, l > \forall l \in lm \mid r(l, f) >]!

Finally for the CS we have:

\(<-, f?> \rightarrow [<-, f>]!
\(+, f?> \rightarrow [<+, f>]!

At last, we have to remark that the variables in subequation (1e) are bound at least before passing from an Anode to a Bnode.

4 Conclusions

The main contribution of this work has been the development of a whole formal description for production systems, including an ad-hoc formal definition of the rete algorithm.

During this work we could clearly identify some limitations of current implementations of production systems that leave us to make a distinction between general production systems, GPSs, and shallow, simple, syntactic and static production systems, S^4PSs.

This formal definition of production systems allows us to define a production system program in an unambiguous, language independent and formal way, being useful for generating code for concrete implementation.

With respect to the rete algorithm, we can say that we have rewritten Fages and Lissajoux’ previous formalisation, making clear that it can be used for handling deep facts and deep patterns.

So, if we compare the rete algorithm with a standard AC-rewriting algorithm, we can say that the only differences are:

- the rete algorithm manages a context, the WM which facilitates the implementation of negated patterns.
- the rete algorithm lacks on search capabilities, so it can not match subterms.

As a related work, we are working on an extensive comparison between the rewrite systems and production systems.

Acknowledgments: Thanks to François Charpillet for sharing with us his X-tra experience and to the Manifico project members for interactions and comments.
References


