Production Systems and Rete Algorithm Formalisation

Horatiu Cirstea, Claude Kirchner, Michael Moossen, Pierre-Etienne Moreau

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 Cirstea,
Claude Kirchner,
Michael Moossen,
Pierre-Etienne Moreau

Référence : Manifico/Sous Projet 1/Fourniture 1.1/V1.0

Date : 15 septembre 2004

Statut : validé

Version : 1.0

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

31 août 2004	V 0.9	création du document
15 septembre 2004	V1.0	version finale

Contents

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 and in Section 1.2, we present also our motivations to give a formal definition of production systems. Section 2.1 will present a formal description of production systems, and an unambiguous formal abstract language for specifying a production system program as given in Section 2.2.

Then, in Sections 3.1 and 3.2, we introduce intuitively the rete algorithm and in Sections 3.3 and 3.4, 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 ($\mathcal{WM}$) 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 $\mathcal{WM}$ 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 $\mathcal{WM}$ 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 ($\mathcal{PM}$) 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 $\mathcal{WM}$. 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 $\mathcal{WM}$.
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* ($\mathcal{CS}$). 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.

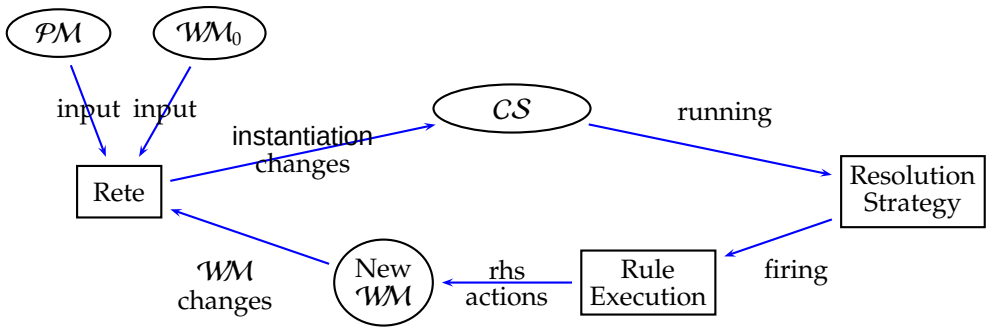

Figure 1: production system work flow

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.

[1] this is a historical name, that does not reflect the common meaning of instantiation

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 $\mathcal{PM}$, or the possibility to choose between a set-based $\mathcal{WM}$ or a multiset-based $\mathcal{WM}$ (i.e. two identical facts can exist simultaneously in the $\mathcal{WM}$).

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 $\mathcal{CS}$ 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:

[dummy] 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 $\mathcal{WM}$ 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 illustrate 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 Production systems

We consider a set $\mathcal{F}$ of function symbols, usually denoted $f, g, h, \ldots$, a set $\mathcal{P}$ of predicate symbols, and infinite sets $\mathcal{X}$ and $\mathcal{L}$ 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 $\mathcal{T}(\mathcal{F}, \mathcal{X})$), *ground terms* (denoted $\mathcal{T}(\mathcal{F})$), *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, $f \in \mathcal{T}(\mathcal{F})$.

Definition 2.2 We call *working memory* ($\mathcal{WM}$) 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

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

consists of the following components.

- A name from the label set: $l \in \mathcal{L}$.
- A set of positive or negative *patterns* $p = p^+ \cup p^-$ where a *pattern* is a term $p_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ 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
- 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, $r = \{ r_i \}_{i \in I_r}$, where $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(p) \cup \mathcal{V}ar(c)$.
- 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, $a = \{ a_i \}_{i \in I_a}$, where $\mathcal{V}ar(a) \subseteq \mathcal{V}ar(p) \cup \mathcal{V}ar(c)$.

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

Definition 2.4 We call *production memory* ($\mathcal{PM}$) a set of production rules.

Remark 2.1 Indeed in the previous definitions, one can discuss the choice of set as the data structure to represent the $\mathcal{WM}$, p, r, a and $\mathcal{PM}$.

Definition 2.5 Given a set of facts $\mathcal{S}$ and a set of positive patterns p^+, p^+ is said to *match* $\mathcal{S}$ with respect to a theory $\mathcal{T}$ and a substitution σ, written $p^+ \ll^{\sigma}_{\mathcal{T}} \mathcal{S}$ if:

$$\forall p \in p^+ \quad \exists t \in \mathcal{S} \mid \sigma(p) =_{\mathcal{T}} t$$

We say that a set of negative patterns p^- dis-matches a set of facts $\mathcal{S}$, written $p^- \not\ll_{\mathcal{T}} \mathcal{S}$ iff:

$$\forall \neg p \in p^- \quad \forall t \in \mathcal{S} \quad \forall \sigma \mid \sigma(p) \neq_{\mathcal{T}} t$$

Definition 2.6 Given a substitution σ, so that $Dom(\sigma) = \mathcal{V}ar(p) \cup \mathcal{V}ar(c)$, a production rule $[l]\ p, c \Rightarrow r, a$ is $(\sigma, \mathcal{WM}')$-*fireable* on a working memory $\mathcal{WM}$ when

1. $p^+ \ll^{\sigma}_{\mathcal{T}} \mathcal{WM}'$
2. $\hat{\mathcal{T}} \vDash \sigma(c)$
3. $\sigma p^- \not\ll_{\mathcal{T}} \mathcal{WM}$

for a minimal (with respect to the subset ordering) subset $\mathcal{WM}'$ of $\mathcal{WM}$ and a matching theory, $\mathcal{T}$, and an additional theory $\hat{\mathcal{T}}$ so that $\mathcal{T} \subseteq \hat{\mathcal{T}}$, for the condition c. A fireable rule is also called an *activation*.

Definition 2.7 Given a production rule $[l]\ p, c \Rightarrow r, a$ which is $(\sigma, \mathcal{WM}')$-fireable on a working memory $\mathcal{WM}$, its *application* leads to the new working memory $\mathcal{WM}''$ defined as:

$$\mathcal{WM}'' = (\mathcal{WM} - \sigma(r)) \cup \sigma(a)$$

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

Definition 2.8 For a given working memory $\mathcal{WM}$ and a set of production rules $\mathcal{R}$, the set

$$\mathcal{CS} = \{ (\, l, \{ f_1, \ldots, f_k \}) \mid \exists ([l]\, p, c \Rightarrow a, r) \in \mathcal{R} \wedge \exists \sigma \text{ so that } l \text{ is } (\sigma, \mathcal{WM}')\text{-fireable on } \mathcal{WM} \}$$

is called the $\mathcal{R}@\mathcal{WM}$-*conflict set*, where $\mathcal{WM}' = \{ f_1, \ldots, f_k \}$

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 2.9 A *resolution strategy* is a computable function that given a set of production rules $\mathcal{R}$, and a production derivation

$$\mathcal{WM}_0 \Rightarrow \mathcal{WM}_1 \Rightarrow \ldots \Rightarrow \mathcal{WM}_n$$

returns a unique element of the $\mathcal{R}@\mathcal{WM}_n$-conflict set.

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

Definition 2.10 A *production system* is defined as

$$\mathcal{GPS} = \left(\mathcal{P}, \mathcal{F}, \mathcal{X}, \mathcal{L}, \mathcal{WM}_0, \mathcal{PM}, \mathcal{S}, \mathcal{T}, \hat{\mathcal{T}} \right)$$

Where:

- $\mathcal{P}$ is the set of predicate symbols,
- $\mathcal{F}$ is the set of function symbols,
- $\mathcal{X}$ is the set of variables,
- $\mathcal{L}$ is the set of labels,
- $\mathcal{WM}_0$ is the initial working memory,
- $\mathcal{PM}$ is the production memory over $\mathcal{H} = (\mathcal{F}, \mathcal{P}, \mathcal{X}, \mathcal{L})$,
- $\mathcal{S}$ is the resolution strategy,
- $\mathcal{T}$ is the matching theory.
- $\hat{\mathcal{T}}$ is the constraint theory, and $\mathcal{T} \subseteq \hat{\mathcal{T}}$.

Remark 2.2 This definition of a production system is quite general as facts and patterns can be deep and non-linear, conditions can be any arbitrary first-order propositions, and resolution strategies can take the full derivation history into account.

Definition 2.11 The *Inference Cycle* is described in figure 2.

2.2 Formal Abstract Language

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 they capabilities in a short and simple way. We asume that numbers and arithmetics are builtin.

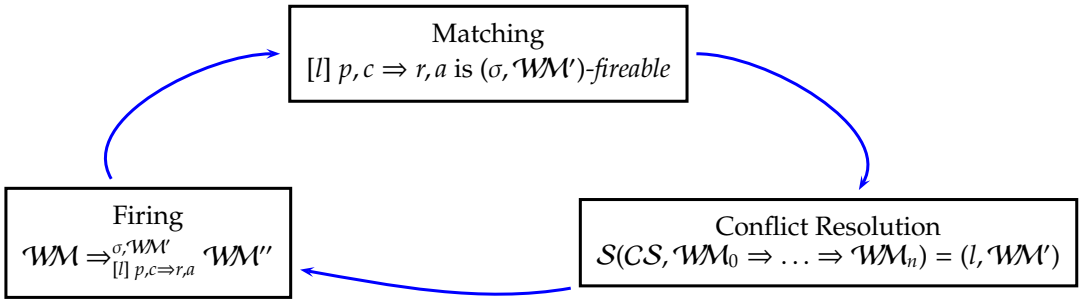

Figure 2: Inference Cycle

$$
\begin{array}{rcl}
\mathcal{P} & := & \emptyset \\
\mathcal{F} & := & \{\ fib/2,\ -/2\ \} \\
\mathcal{X} & := & \{\ x \mid x \text{ begins with a question mark}\ \} \\
\mathcal{L} & := & \{\ GoDown, GoUp\ \} \\
\mathcal{WM}_0 & := & \{\ fib(0,\ 1),\ fib(1,\ 1),\ fib(200,\ -1)\ \} \\
\mathcal{PM} & := & \{ \\
 & & \quad [GoDown] \\
 & & \qquad fib(?n,\ -1)\ \wedge\ \neg fib(?n1,\ ?v)\ , \\
 & & \qquad ?n1 = ?n - 1 \\
 & & \quad \Longrightarrow \\
 & & \qquad \emptyset, \\
 & & \qquad \{\ fib(?n1,\ -1)\ \} \\
 & & , \\
 & & \quad [GoUp] \\
 & & \qquad fib(?n,\ -1)\ \wedge\ fib(?n1,\ ?v1)\ \wedge\ fib(?n2,\ ?v2), \\
 & & \qquad ?n1 = ?n - 1 \wedge ?v1 > 0 \wedge ?n2 = ?n - 2 \wedge ?v2 > 0 \wedge ?v = ?v1 + ?v2 \\
 & & \quad \Longrightarrow \\
 & & \qquad \{\ fib(?n,\ -1),\ fib(?n2,\ ?v2)\ \}, \\
 & & \qquad \{\ fib(?n,\ ?v)\ \} \\
 & & \} \\
\mathcal{S} & := & \text{a FIFO strategy} \\
\mathcal{T} & := & \emptyset \\
\hat{\mathcal{T}} & := & \emptyset
\end{array}
$$

Example 2.2 This is another example for searching for a house which is more near to the real

usage of production systems, we asume numbers, arithmetics and strings are builtin:

$\mathcal{P}$:= $\emptyset$
$\mathcal{F}$:= {
house/4, *houseaddress*/4, *myaddress*/3, *war*/2, *searching*/0,
red/0, *blue*/0, *usa*/0, *irak*/0, *france*/0
}
$\mathcal{X}$:= { x | x begins with a question mark }
$\mathcal{L}$:= { *HouseSearch* }
$\mathcal{WM}_0$:= {
house(1, *red*, 341, *true*), *houseaddress*(1, 251, "*rue jeanne d'arc*", "*nancy*"),
house(2, *blue*, 390, *true*), *houseaddress*(2, 121, "*avenue de brabois*", "*villers les nancy*"),
house(3, *red*, 415, *true*), *houseaddress*(3, 31, "*rue carnot*", "*vandoeuvre les nancy*"),
myaddress(2551, "*gorbea*", "*santiago*"),
war(*usa*, *irak*),
searching()
}
$\mathcal{PM}$:= {
[*HouseSearch*]
searching() ∧
house(?*id*, *red*, ?*price*, *true*) ∧
houseadress(?*id*, ?*number*, ?*street*, ?*city*) ∧
myaddress(?*mn*, ?*ms*, ?*mc*) ∧
¬*war*(?*s*1, *france*) ∧
¬*war*(*france*, ?*s*2) ,
?*price* < 400
⟹
{
searching(),
house(?*id*, *red*, ?*price*, *true*),
myaddress(?*mn*, ?*ms*, ?*mc*)
},
{
house(?*id*, *red*, ?*price*, *false*),
myaddress(?*number*, ?*street*, ?*city*)
}
}
$\mathcal{S}$:= a FIFO strategy
$\mathcal{T}$:= $\emptyset$
$\hat{\mathcal{T}}$:= $\emptyset$

We will use in the rest of this report the same abstract syntax as in the examples 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:

$$\mathcal{X} = \{\ x\ \mid\ x \text{ begins with a question mark}\ \}$$

So, when clear from the context a production system is just:

$$\mathcal{PS} = (\ \mathcal{F},\ \mathcal{WM}_0,\ \mathcal{PM}\ )$$

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 ($\mathcal{F}$)

```
public class Fib {
  public int number;
  public int value;
}
```

- Working memory initialization ($\mathcal{WM}_0$)

```
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 ($\mathcal{PM}$)

```
[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 2.3 We already have implemented a prototype of such a tool using Java and 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 described in figure 3:

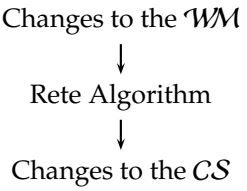

Figure 3: Rete algorithm as a Black Box

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, starting from the $\mathcal{WM}_0$.

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 there are 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 *window*s equals to the color of the *door*s 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, the test result will determine, if a new token is generated and the sign of it. Represented as rounded box shapes with two input arrows and labeled with the inter-element conditions, maybe more than one, or no condition at all. 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 *Bnode*s:
  - *Any* nodes, related to inter-element tests between only positive patterns or between only negative patterns.
  - *Not* nodes, related to inter-element tests between both, positive and negative, patterns.
- **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 $\mathcal{CS}$. 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 4.

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

$$\mathcal{WM}_0 \coloneqq \{\ fib(0,\ 1),\ fib(1,\ 1),\ fib(3,\ -1)\ \}$$

Notice that the network is providing slots names to be more clear, so the first slot of the fib fact is called *number*, and the second *value* (they could be also numbered with 1 and 2, for instance).

Example 3.2 Rete Network Execution.
First, we initialize

$$\mathcal{WM} \coloneqq \emptyset \wedge \mathcal{CS} \coloneqq \emptyset$$

then, we add the first fact in $\mathcal{WM}_0$ to $\mathcal{WM}$

$$\mathcal{WM} \coloneqq \mathcal{WM} \cup \{\ fib(0,\ 1)\ \}$$

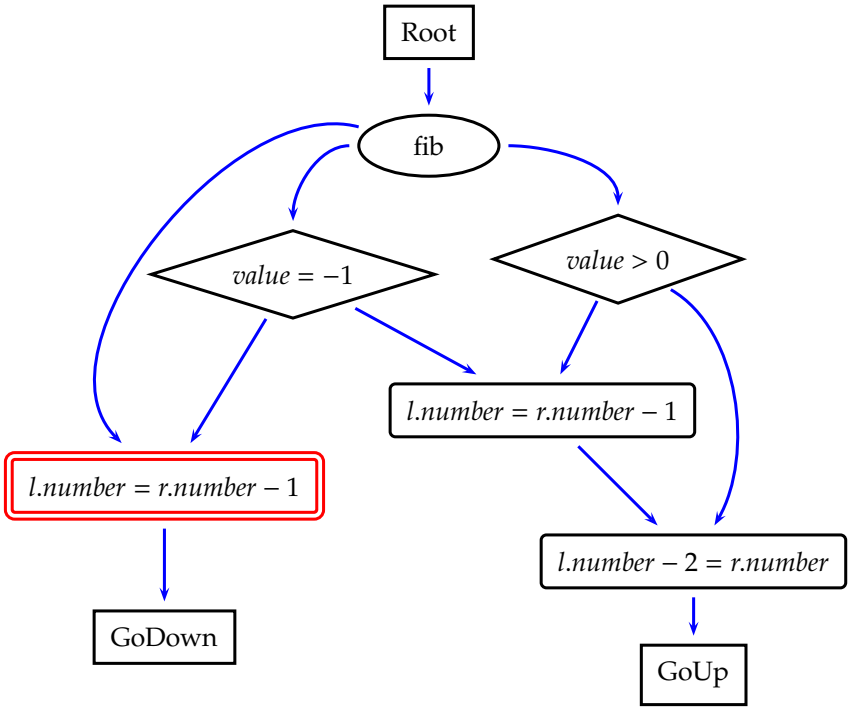

Figure 4: Rete network for Fibonacci example

and we send the respective token, $< +,\ fib(0,\ 1) >$, to the rete network, which first checks for the type of the token, if it is equals to fib as in this case, it passes the token to the next connected nodes, also called successors, if not the token is casted away.

At the second level, first, the token travels to the left input of the *Not* node and the fact is stored in the left memory of the node, waiting for some input at the right side. On the other side, the token also travels to the node for checking if $value = -1$, it is not the case, so the token is casted away. Finally, on the right branch, the token arrives to the node for checking the condition $value > 0$, as it is true it gives the token to both successors, which are both two input nodes, since the sign of the token is positive we add the given fact $fib(0,\ 1)$ to the right memory, and because the nodes have both empty left memories no condition can be checked so no input is generated.

Then, we add the second fact in $\mathcal{WM}_0$ to $\mathcal{WM}$

$$\mathcal{WM} \coloneqq \mathcal{WM} \cup \{\ fib(1,\ 1)\ \}$$

and the process is the same as before, the fact stays waiting at both *Any* nodes of the right branch in their right memory, and also in the left memory of the *Not* node of the left branch.

Finally, we add the last fact in $\mathcal{WM}_0$ to $\mathcal{WM}$

$$\mathcal{WM} \coloneqq \mathcal{WM} \cup \{\ fib(3,\ -1)\ \}$$

in this case, the left branch is taken, so first the fact is stored in left memory of the *Not* node, and then the check $value = -1$ succeeds so the token propagates to all its successors, first to the right hand side of the *Not* node, where the condition is checked against all the fact in the left memory, and given that the condition can not be satisfied by any combination an output token is generated, $< +,\ fib(3,\ -1) >$, activating the rule *GoDown*, so we have:

$$\mathcal{CS} \coloneqq \mathcal{CS} \cup\ < GoDown,\ \{\ fib(3,\ -1)\ \} >$$

On the other hand, the fact $fib(3,\ -1)$ is also stored in the left memory of the first *Any* node, where no combination satifies the condition so no output is generated.

Now, the rete network is waiting for new input, so we have to fire a new activation, and there is no choice, and we fire the given activation of the *GoDown* rule, actualizing the $\mathcal{CS}$ and

computing the substitution σ:

$$\mathcal{CS} \coloneqq \mathcal{CS} \setminus < GoDown, \{ fib(3, -1) \} >$$

$$\sigma \coloneqq \{ ?n \to 3, ?n1 \to 2 \}$$

And we perfom the actions of the RHS of the given rule, which just adds a new fact, $fib(2, -1)$, to the $\mathcal{WM}$,

$$\mathcal{WM} \coloneqq \mathcal{WM} \cup \{ fib(2, -1) \}$$

So, we generate the respective token, $< +, fib(2, -1) >$, which behaves similar to the previous one, $< +, fib(3, -1) >$, except that now, when the token arrives at the left input of the *Not* node, the condition can be satisfied in conjunction with the fact $fib(3, -1)$ of the right memory, so, a new token is generated $< -, fib(3, -1) >$ deactivating the rule *GoDown*, if it would be activated. On the other hand, the new fact $fib(2, -1)$ also riches the right input of the *Not* node, but since there exists a combination with a fact in left memory, $fib(1, 1)$, validating the condition again a new token is generated for deactivating the *GoDown* rule. And finally, now, when the token arrives the left input of the first *Any* node, the condition can be satisfied, using fact $fib(1, 1)$ of the right memory, so a new token is generated, $< +, \{ fib(1, 1), fib(2, -1) \} >$, and given to the left input of the second *Any* node, storing the facts in the left memory, and checking the condition, which can be satisfied by the fact $fib(0, 1)$ of the right memory, generating a new token for activating the *GoUp* rule, and:

$$\mathcal{CS} \coloneqq \mathcal{CS} \cup < GoUp, \{ fib(0, 1), fib(1, 1), fib(2, -1) \} >$$

Again the rete network is stucked, waiting for more input, so we have to select an activation from the $\mathcal{CS}$, we select the only one activation in the $\mathcal{CS}$, of rule *GoUp*, and we actualize the $\mathcal{CS}$

$$\mathcal{CS} \coloneqq \mathcal{CS} \setminus < GoUp, \{ fib(0, 1), fib(1, 1), fib(2, -1) \} >$$

and we execute the right hand side, first computing the substitution σ as

$$\sigma \coloneqq \{ ?n \to 2, ?n1 \to 1, ?n2 \to 0, ?v1 \to 1, ?v2 \to 1, ?v \to 2 \}$$

for then, removing $fib(2, -1)$ from the working memory

$$\mathcal{WM} \coloneqq \mathcal{WM} \setminus \{ fib(2, -1) \}$$

So, a negative token, $< -, fib(2, -1) >$, is given to the rete network, which checks the type, then the token first arrives at the left input of the *Not* node and the given fact is removed from the left memory, and now the condition can not be satisfied by the $fib(3, -1)$ fact of the right memory, so a new token $< +, fib(3, -1) >$ is generated and the *GoDown* rule is activated:

$$\mathcal{CS} \coloneqq \mathcal{CS} \cup < GoDown, \{ fib(3, -1) \} >$$

Then, the token pass also through the $value = -1$ check, so the token arrives at the right input of the *Not* node and after actualising the right memory by $rm = rm \setminus \{ fib(2, -1) \}$ there is still a combination, with fact $fib(1, 1)$ that satisfies the condition so a negative token is generated for deactivating the *GoDown* rule. Then, on the other branch, the negative token arrives at the left input of the first *Any* node, removing the given fact from the left memory, and given that the condition can be still satisfied by the fact $fib(1, 1)$ from the right memory a new negative token, $< -, \{ fib(1, 1), fib(2, -1) \} >$ is generated, arriving at the left input of the second *Any*, which in the same way removes the item from the left memory and generates a new negative token, deactivating the given instantiationof the *GoUp* rule.

Follows to remove $fib(0, 1)$, and to add $fib(2, 2)$, then a new activation of rule *GoUIP* is fired, and facts $fib(3, -1)$ and $fib(1, 1)$ are removed, and fact $fib(3, 3)$ added, so no more rules are activated and the final state of the $\mathcal{WM}$ is

$$\mathcal{WM} \coloneqq \{ fib(2, 2), fib(3, 3) \}$$

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' formalisation of [FL92], using the same notations used for defining a production system in Section 2.1.

First we will describe how to compute a normal form for a $\mathcal{PM}$, and then how to build the rete network from this normal form.

3.3.1 Compilation of a $\mathcal{PM}$

In this section we describe how to compile a $\mathcal{PM}$, that is a set of rules, and more specifically, a set of LHS, patterns and conditions, for obtaining a normal form, allowing us to easily build the rete network as described in the next section.

Definition 3.1 A term t can be viewed as a mapping from a subset of the monoid $\mathbb{N}^*$ called its *domain*, denoted $\mathcal{D}om(t)$, to $\mathcal{F}$.

Definition 3.2 The subterm s in term t at *occurrence* $\omega \in \mathcal{D}om(t)$, denoted as $s = t|_\omega$, is defined as:

$$t|_\epsilon = t$$
$$f(t_1, \ldots, t_n)|_{i.\omega} = t_i|_\omega$$

Definition 3.3 The *top symbol* of term t is written as $top(t)$.

Definition 3.4 We define $\mathcal{MV}$ as an enumerable set of metavariables, in general, denoted in_i.

Definition 3.5 We define the *compilation* of a given pattern $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and a metavariable $in \in \mathcal{MV}$, denoted $p \ll^\epsilon in$, or simply $p \ll in$, via the following set of rewrite rules:

$$f(t_1, \ldots, t_n) \ll^\omega in \quad \rightarrow \quad top(in|_\omega) = f \wedge \bigwedge_{i=1}^{n} t_i \ll^{\omega.i} in \tag{1a}$$

$$x \ll^{\omega_1} in \wedge x \ll^{\omega_2} in \quad \rightarrow \quad x \ll^{\omega_1} in \wedge in|_{\omega_1} = in|_{\omega_2} \tag{1b}$$

Where:

- $x \in \mathcal{X}$, $f \in \mathcal{F}$, $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,
- p is a positive or a negative pattern, we compile them in the same way,
- in is a placeholder for a fact that may match the pattern, and
- ω is an occurrence in in.

Example 3.3 The pattern $f(x, g(a, x))$ will compile into:

$$\begin{aligned} in|_\epsilon &= f \\ in|_1 &= in|_{2.2} \\ in|_2 &= g \\ in|_{2.1} &= a \\ x &\ll^1 in \end{aligned}$$

Definition 3.6 We also define incrementally the *compilation* of a given rule $[l]\ \{p_1, \ldots, p_m\}, c \Rightarrow r, a$, by compiling each pattern, starting from

$$p_1 \ll in_1 \wedge \ldots \wedge p_m \ll in_m \wedge c$$

and applying the following set of rewrite rules:

$$f(t_1, \ldots, t_n) \ll^{\omega} in_i \quad \rightarrow \quad top(in_i\,|_{\omega}) = f \wedge \bigwedge_{i=j}^{n} t_j \ll^{\omega} in_i \tag{2a}$$

$$x \ll^{\omega_i} in_i \ \wedge\ x \ll^{\omega_j} in_j \quad \rightarrow \quad x \ll^{\omega_i} in_i \wedge in_i\,|_{\omega_i} = in_j\,|_{\omega_j} \tag{2b}$$

$$c\,[x] \ \wedge\ x \ll^{\omega} in_i \quad \rightarrow \quad c\,[in_i\,|_{\omega}] \ \wedge\ x \ll^{\omega} in_i \tag{2c}$$

Remark 3.1 This is just an incremental definition of a rule's left hand side satisfaction as described in Definitions 2.5 and 2.6.

Remark 3.2 Notice that equation (2a) is just the same as equation (1a) and that equation (2b) is just a generalisation of equation (1b).

Example 3.4 The following left hand side of a rule

$$f(x, y) \ \wedge\ f(a, x)\,,$$
$$y > 5 \ \wedge\ z = x + y$$

will compile into:

$$\begin{aligned} in_1\,|_{\epsilon} &= f \\ in_2\,|_{\epsilon} &= f \\ in_1\,|_1 &= in_2\,|_2 \\ in\,|_2 &> 5 \\ z &= in_1\,|_1 + in_1\,|_2 \\ x &\ll^1 in_1 \\ y &\ll^2 in_1 \end{aligned}$$

Definition 3.7 Now, we define an *optimisation* for such compilation, which is strong related to the sharing concept in the Rete Algorithm, as described in section 3.3.2. The main idea is that for testing conditions depending on just one single pattern, we can forget the related pattern and treat two conditions that only differ in the related pattern as the same. So we can have this additional rewrite rule for optimization:

$$R\,(in_i\,|_{\omega_1}, \ldots, in_i\,|_{\omega_n}) \ \wedge\ R\left(in_j\,|_{\omega_1}, \ldots, in_j\,|_{\omega_n}\right) \longrightarrow R\,(in_i\,|_{\omega_1}, \ldots, in_i\,|_{\omega_n})$$

So, in the example 3.4, we should remove, for instance, the relation

$$in_2\,|_{\epsilon} = f$$

Definition 3.8 This takes us to define the compilation of a whole production memory $\mathcal{PM}$ as the compilation of each rule of the $\mathcal{PM}$, obtaining a *Normal Form* like:

$$\bigwedge_{i \in \mathcal{I}} top(in\,|_{\omega}) = f_i \tag{3a}$$

$$\bigwedge_{j \in \mathcal{J}} in\,|_{\omega_{1_j}} = in\,|_{\omega_{2_j}} \tag{3b}$$

$$\bigwedge_{(l, r) \in (\mathcal{L}, \mathcal{R})} in_l\,|_{\omega_l} = in_r\,|_{\omega_r} \tag{3c}$$

$$\bigwedge_{k \in \mathcal{K}_1} c_k \ \mid\ \mathcal{V}ar\,(c_k) = \emptyset \tag{3d}$$

$$\bigwedge_{k \in \mathcal{K}_2} c_k \ \mid\ \mathcal{V}ar\,(c_k) \neq \emptyset \tag{3e}$$

where

- Subequations (3a) and (3b) come from subequations (1a) and (1b),
- Subequation (3c) comes from subequation (2b) for inter-pattern relations, and
- Subequations (3d) and (3e) comes from subequation (2c) for general conditions, and the conditions in subequation (3d) have no free variables, so that they can be used for building the Rete Network, while the conditions of subequation (3e) still have free variables, so they have to be handled at runtime by an additional Constraint Solver. This is, for instance, the case of the condition $?v = ?v1+?v2$ in our Fibonacci example.

Definition 3.9 To put everything together, we have to sort all the relations appearing in the Normal Form, except the relations of subequation (3e), for this we first define the *Dependency Set* of a relation $R(t_1, \ldots, t_n)$ as the set of involved patterns:

$$\mathcal{DS}(R) \coloneqq \{ p \mid \mathcal{V}ar(p) \cap \mathcal{V}ar(R) \neq \emptyset \}$$

Definition 3.10 We define $\mathcal{R}^k$ as the set of all relations depending on exactly k patterns:

$$\mathcal{R}^k \coloneqq \{ R(t_1, \ldots, t_n) \mid |\mathcal{DS}(R)| = k \}$$

Definition 3.11 And finally, we define $R^k_{p_1, \ldots, p_k}$ as the set of all relations depending on exactly k patterns, patterns $p_1, \ldots, p_k$ as:

$$\mathcal{R}^k_{p_1, \ldots, p_k} \coloneqq \{ R(t_1, \ldots, t_n) \mid \mathcal{DS}(R) = \{ p_1, \ldots, p_k \} \}$$

Remark 3.3 The set of relations $\mathcal{R}^1$ is the set of intra-elements tests or the *test mono-schéma* as defined in Section 3.1 of [FL92].

And $\mathcal{R} \setminus \mathcal{R}^1$ is the set of the inter-element tests or the *test multi-schéma* as defined in Section 3.1 of [FL92].

3.3.2 Rete Network

So, given a normal form for a $\mathcal{PM}$, we can begin to build the network by creating a *root* node, then for each pattern i we take any intra-element relation $r \in \mathcal{R}^1_i$ and we build a new *Anode* or one-input node for each of them and we connect them in a serial way and an arbitrary absolute order to the *root* node, removing first any duplicated relation in $\mathcal{R}^1_i$.

And, in the case there is the same intra-element test, in two different rules or patterns, we share it, i.e. if we have that two relations R and R', so that:

$$R = R' \in \mathcal{R}^1$$

Instead of two different nodes, one for R and one for R', we build a single *Anode*, A, and if $Anode(R_p)$ and $Anode(R'_p)$ are the predecessors and $Anode(R_s)$ and $Anode(R'_s)$ are the successors of R and R' respectively, we connect them as follows:

$$Anode(R_p) \longrightarrow A \longrightarrow Anode(R_s), \text{and}$$

$$Anode(R'_p) \longrightarrow A \longrightarrow Anode(R'_s)$$

In other words, you can see it graphically as described in the example of figure 5, where we begin by an empty graph, then we add a *root* node, and then following the same absolute order we connect the intra-element relations of a single pattern to the root node for each pattern in every rule, obtaining one independent path for each one. Then we remove duplicate instances of relations in a single pattern/path.

Once we have shared the intra-element relations for each pattern, we have still one path for each pattern, but now we will share also relations between different patterns, and may be also different rules, as the example of figure 6 shows.

We will call leaf-nodes every node which is still unconnected at its output, so after you connect every intra-element relation, you will have one leaf-node for each pattern, non considering sharing. In our example, y, z and v are leaf nodes, which is clear in figure 5, but tricky in figure 6.

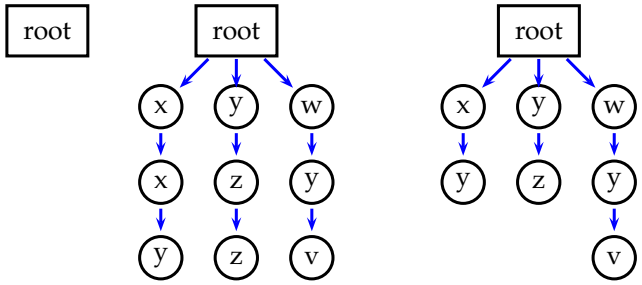

Figure 5: An example how to share intra-element tests for single patterns

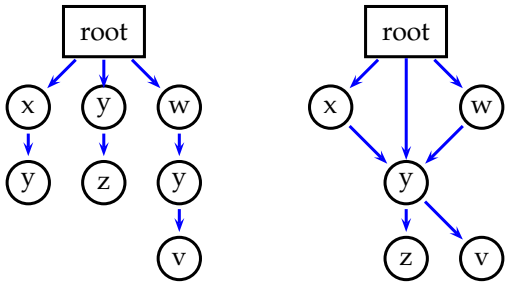

Figure 6: An example how to share intra-element tests

Now, for building the *Bnode*s or two-input nodes we have to consider that each *Bnode* is asociated a list L of the relations to be checked by the node, so this node will depend on the union of each dependency set

$$\mathcal{DS}(n) := \bigcup_{R \in L} \mathcal{DS}(R)$$

for n a *Bnode*. This set will be a superset of the union of the patterns related to the facts arriving at both inputs.

Remark 3.4 A *Bnode*, depending on the relation to a negative pattern, is called *Any* node, if there are only negative or only positive patterns involved, or *Not* node, if there are negative as also positive patterns involved.

Then, for building and connecting the *Bnode*s, for inter-element relations, we select one of the remaining relations $R \in \mathcal{R}^k$ and dependency set $P = \{ p_1, \ldots, p_k \}$, Here, we will have to decide how to select this relation $R \in \mathcal{R}$. A good criterion seems to be to select the relation with smallest dependency set.

Then, if there is a *Bnode*, n, does not matter if it is a leaf-node or not, so that $\mathcal{DS}(R) \subseteq \mathcal{DS}(n)$, we have just to add the relation R to this node, and if there are more than one possibilities, we add it to the node n, so that $|\mathcal{DS}(n)|$ is minimal.

If there is no node n, so that $\mathcal{DS}(R) \subseteq \mathcal{DS}(n)$, create a new node n for this relation R, and locate the minimal amount of leaf-nodes $n_1, \ldots, n_l$, so that, $\mathcal{DS}(R) \subseteq \mathcal{DS}(n_1) \cup \ldots \cup \mathcal{DS}(n_l)$, and connect the output of the nodes $n_1, \ldots, n_l$ to the inputs of node n, keeping in mind that a *Bnode* have exactly two inputs, and one output. So, we proceed as follows: If

l=2 , there is just two possibilities to connect the outputs of n_1 and n_2 to the inputs of n, one structural shape, with 2! posibilities. Here is the picture:

l=3 , there are several, exactly 3! possibilities to connect them, the point is that we have to connect all them two nodes by two at each time, creating empty nodes, nodes that only collects their inputs but do not check any relation, at least yet, may be later you will have to put some relations in these nodes. The best way to connect them will depend on each production system, but some metrics for taking a good decision includes the height of the final rete network and the number of empty nodes. Here a picture of the different possibilities for

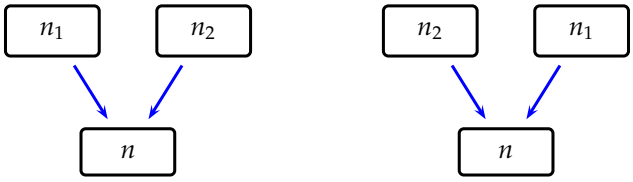

Figure 7: How to create a new *Bnode n* from 2 sources, n_1 and n_2

connecting three nodes, structuraly there is only one choise, but there are three different configurations swapping the nodes:

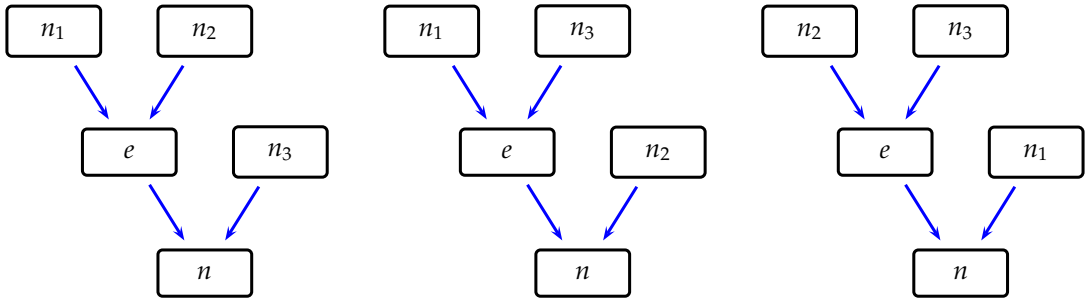

Figure 8: How to create a new *Bnode n* from 3 sources, n_1, n_2 and n_3, by inserting an extra empty node e.

l=4 , as before, there are $4! \times 2 = 48$ possibilities, two different structures with $4! = 24$ different configurations each:

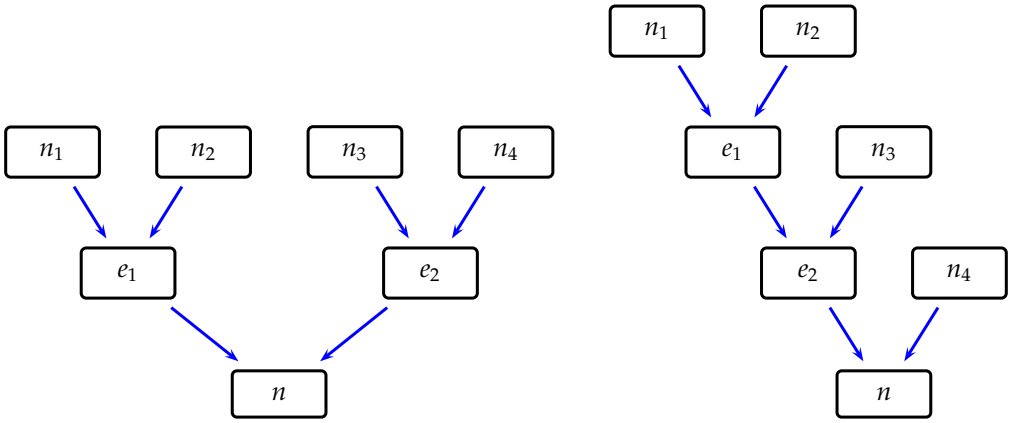

Figure 9: How to create a new *Bnode n* from 4 sources, n_1, n_2, n_3 and n_4, by inserting two extra empty nodes e_1 and e_2, and showing just the two different structural approaches, each one with three possible combinations.

l=? , in general, there will be exactly $l! \times s$ possibilities for connecting the nodes, where s is the number of different structural posibilities, using a similar strategy as showed in the examples above.

Remark 3.5 Until now, we made no difference for building nodes for relations that involve positive and negative patterns. And, in general, for building the Rete Network there is no special threatment for negative patterns, just that, if you have a relation which involves both positive and negative patterns, you have to build your network in such a way that you collect all the

negative patterns and all the positive patterns, so that the new node will have one input for positive patterns and one for negative patterns.

Once all the relations have been included in the Rete Network, we create for each rule a terminal node, and we just take all the outputs of leaf-nodes related to a single rule and we connect them with the corresponding terminal node, and we do not worry about the number of inputs of terminal nodes, since they do not perform any condition check, they can have as many inputs as needed.

Remark 3.6 The so built Rete Network will have exactly one entry point, the *root* node, one terminal node for each rule and, unless there are two or more identical patterns in the same rule, one path from the root node to a terminal node will represent exactly one pattern.

Example 3.5 For our Fibonacci example, we have:

$$q^1_{1,0} = q^1_{2,0} = q^2_{1,0} = q^2_{2,0} = q^2_{3,0} \equiv input.\epsilon = fib$$
$$q^1_{1,1} = q^2_{1,1} \equiv input.2 = -1$$
$$q^2_{2,1} = q^2_{3,1} \equiv input.2 > 0$$
$$Not^1_{1,2} \equiv input_1.1 - 1 = input_2.1$$
$$Any^2_{1,2} \equiv input_1.1 - 1 = input_2.1$$
$$Any^2_{1,3} \equiv input_1.1 - 2 = input_3.1$$
$$c^2_{2,3} \equiv ?v = input_2.2 + input_3.2$$

So, just considering intra-element relations, we will have the network showed in figure 10.

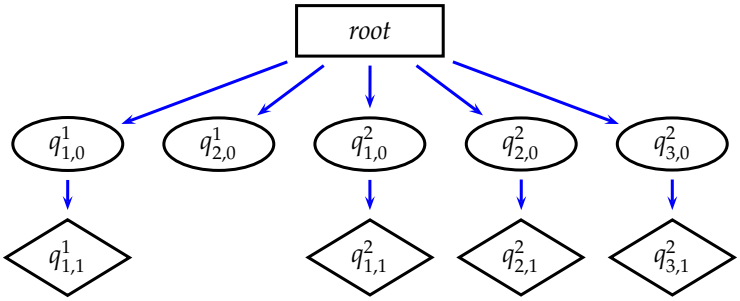

Figure 10: Intra-element nodes for Fibonacci without sharing

Then, figure 11, shows the network after sharing.

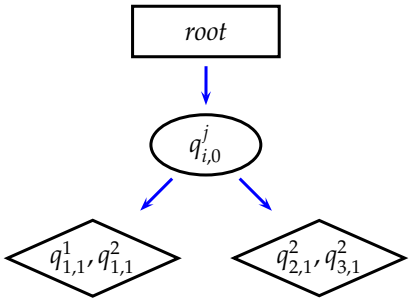

Figure 11: Intra-element nodes for Fibonacci with sharing

Now, we have to build the *Bnode*s for the inter-element tests. Notice that our leaf-nodes are: $q^2_{2,0}, q^1_{1,1}, q^2_{1,1}, q^2_{2,1}$ and $q^2_{3,1}$. We first take relation $Not^1_{1,2}$, this relation depends on pattern p_1 and p_2

for the *GoDown* rule, so, since there is no node n so that $\{ p_1, p2 \} \subseteq \mathcal{DS}(n)$, we have to create a new node and we connect them to the corresponding leaf-nodes, checking $q^1_{1,1}$ for the positive pattern p_1 and $q^1_{2,0}$ for the negative pattern p_2. Then, we take the next relation $Any^2_{1,2}$, related to patterns p_1 and p_2 of the *GoUP* rule, and after looking for a node n so that $\{ p_1, p_2 \} \subset \mathcal{DS}(n)$ with no success, we create a new node connecting the corresponding leaf-nodes, checking $q^2_{1,1}$ for p_1 and $q^2_{2,1}$ for p_2. Now, for the last relation $Any^2_{1,3}$ we have also to create a new node, but now the leaf-node involving p_1 is not $q^2_{1,1}$ instead it is the node $Any^2_{1,2}$, so we connect the new node to it and also to the node containing $q^3_{3,1}$. So, we get the graph as presented in figure 12

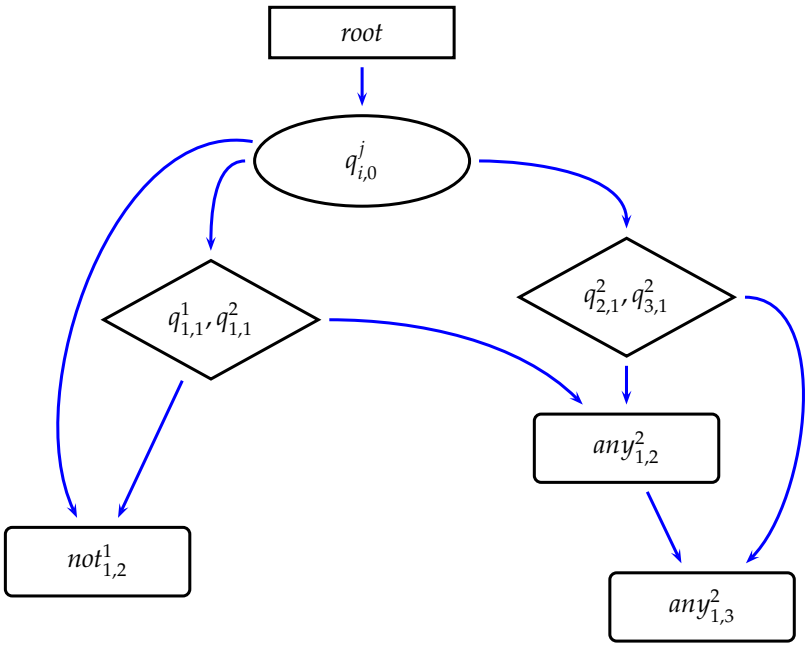

Figure 12: Rete network for Fibonacci example after adding the *Bnode*s

And finally, after adding the terminal nodes, figure 13 shows the formal version of the rete network for our 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.

Definition 3.12 We define a *token*, t, as:

$$t =< s, \{ f_1, \ldots, f_n \} >$$

where

- $f_1, \ldots, f_n$ are facts in $\mathcal{WM}$, and
- s is a sign, + or −, for indicating that a fact has been added or removed from $\mathcal{WM}$.

So, if a fact, f, is added to $\mathcal{WM}$, we generate a token $< +, f >$, and we pass it to the entry point of the rete network, the *root* node. If f was removed, we generate a negative token $< -, f >$.

Now, we will define how the diferent node types behaves on the arrival of tokens. First, when the *root* node gets a new token, it forward it without question to all its successors in the network, this means possibly, a clonation of the token, so that every successor, get his own copy.

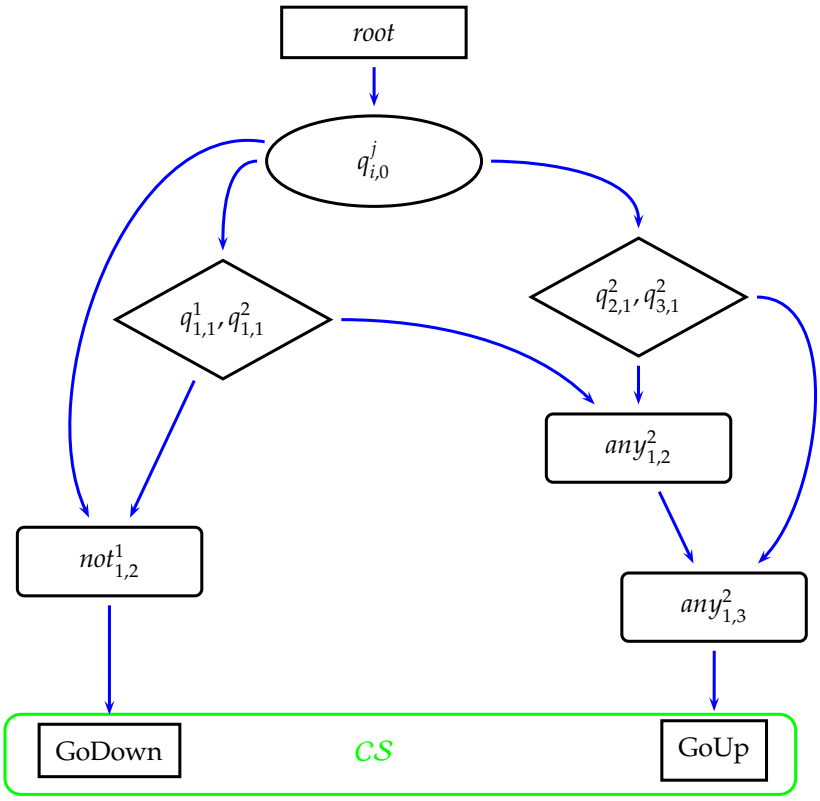

Figure 13: Rete network for Fibonacci example

Definition 3.13 Then, we define an *execution rule* for a given *Anode*(R), n, and a given input token, t, as:

$$\mathcal{E}xe\,(n,\ t) = \begin{cases} t & \text{if } R(f) \\ \emptyset & \text{if } \neg R(f) \end{cases}$$

That means, that when a *Anode* receives an input token and the respective relation can be satisfied, it generates the same input token as output, for all its successors. And if not, nothing happens.

The incremental approach of the rete algorithm comes from the memorization of the tuples of facts that satisfy a partial left hand side of a rule in *left and right memories*, *lm* and *rm* for each *Bnode*. Each *Bnode*, is associated to three memories, the left input memory *lm*, the right input memory *rm* and the output memory, or input memory for the next node.

Definition 3.14 A *partial left hand side* of a rule j, $[l]\ p, c \Rightarrow r, a$, asociated to a *Bnode*, n, is $\pi^j_n = (p', c')$, where

$$p' = \mathcal{DS}\,(n)$$

$$c' = \{\ c_i\ \mid\ \forall x \in \mathcal{V}ar\,(c_i) \quad x \in \mathcal{V}ar\,(\mathcal{DS}\,(n)) \wedge x \notin \mathcal{V}ar\,(p \setminus p')\ \}$$

We also define a partial left hand side of a rule, for a single pattern p_i asosiated to the last *Anode* n in a branch, where

$$p' = p_i$$

$$c' = \{\ c_i\ \mid\ \forall x \in \mathcal{V}ar\,(c_i) \quad x \in \mathcal{V}ar\,(p_i) \wedge x \notin \mathcal{V}ar\,(p \setminus p_i)\ \}$$

Definition 3.15 So, we can define the *left memory*, lm^j_n, and the *right memory*, rm^j_n of a *Bnode* n in a rule j, if it is connected to the nodes n_1 and n_2, as following:

$$lm^j_n = \left\{\ F \subseteq \mathcal{WM}\ \mid\ F \vDash \pi^j_{n_1}\ \right\}$$

$$rm_n^j = \left\{ F \subseteq \mathcal{WM} \mid F \vDash \pi_{n_2}^j \right\}$$

Then, when a token $< s,\ F >$ arrives to one input of a *Bnode*, either *Any* or *Not* node, the respective memory(lm or rm) is updated as follows:

$$mem \leftarrow \begin{cases} mem \cup \{ F \} & \text{if } s = + \\ mem \setminus \{ F \} & \text{if } s = - \end{cases}$$

For a *Any* node, we define the execution rule as:

$$\mathcal{E}xe\left(Any_i(R),\ < s,\ F >\right) = \begin{cases} < s,\ F \cup F' > & \text{if } \exists F' \in mem \mid F \cup F' \vDash R \\ \emptyset & \text{if } \forall F' \in mem \mid F \cup F' \nvDash R \end{cases}$$

Where mem is the memory from the opposite side where the token has arrived.

And for a *Not* node, we define the execution rule as follows, if the token arrives at the side related to positive patterns:

$$\mathcal{E}xe\left(Not_i(R),\ < s,\ F >\right) = \begin{cases} < s,\ F > & \text{if } \forall F' \in mem \mid F \cup F' \nvDash R \\ < -s,\ F > & \text{if } \exists F' \in mem \mid F \cup F' \vDash R \end{cases}$$

and if the token arrives at the side related to negative patterns:

$$\mathcal{E}xe\left(Not_i(R),\ < s,\ F >\right) = \begin{cases} < -s,\ F' > & \text{if } \exists F' \in mem \mid F \cup F' \vDash R \\ \emptyset & \text{if } mem = \emptyset \ \vee\ \forall F' \in mem \mid F \cup F' \nvDash R \end{cases}$$

Where mem is the memory from the opposite side where the token has arrived.

At last, the terminal nodes, that can have several inputs, in general, have to wait for tokens at every input, memorising the facts at every input, these memories work in the same way as a lm or rm, if a positive token, $< s,\ F_j >$ arrives at input j, F_j is stored in the memory, if a negative token arrives, F_j is removed from the memory. So, when a new positive token arrives, the other inputs are checked for the different combinations for activating the given rule l,

$$\mathcal{CS} = \mathcal{CS} \cup\ < l,\ \bigcup_{i \in \mathcal{I}} F_i >$$

Where F_i is an element at input i, but F_j is fixed. So, all inputs have to have at least one element, for generating the activation. But, if a negative token arrives, and there is a combination, then the activation is retracted:

$$\mathcal{CS} = \mathcal{CS} \setminus\ < l,\ \bigcup_{i \in \mathcal{I}} F_i >$$

The only exception is, if one of the inputs is comming directly from an *Anode* related to a negative pattern. In that case, the activation is generated just if all inputs have elements, except the input comming from the mentioned *Anode*, which has to be empty.

Remark 3.7 Notice that the activation do not include any information about the binding of the variables, this is because it is trivial to obtain the mapping given the facts that match the patterns. But there is also the possibility to track the binding of the variables inside of the tokens, and include that information in the activation of a rule.

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.

With respect to production systems, we give a formal definition of common used concepts, like: facts, working memory, positive and negative patterns, production rules, production memory, activations, instantiations, conflict set, conflict resolution strategy, and so on.

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 we have rewritten Fages and Lissajoux' previous formalisation, making clear some important points, like:

- The fact that it can be used for handling deep facts and deep patterns.
- The need for correlation between different relations involving common patterns.
- The need, in some cases, for empty nodes in the rete network, just for collecting the facts pairwise.

We also presented a normal form for a production memory for being able to build the rete network.

On the other side, 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 $\mathcal{WM}$ which facilitates the implementation of negative 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

[Alb89] Luc Albert. Average case complexity analysis of rete pattern-match algorithm and average size of join in databases. Rapport de recherche no. 1010, INRIA-ROCQUENCOURT, 1989.

[BB01] Dumitru Dan Burdescu and Marius Brezovan. Algorithms for high level petri nets simulation and rule-based systems. *Acta Universitaris Cibiniensis*, XLIII:33 – 39, 2001.

[BBCK04] Clara Bertolissi, Paolo Baldan, Horatiu Cirstea, and Claude Kirchner. A rewriting calculus for cyclic higher-order term graphs. In Maribel Fernandez, editor, *Proceedings of the 2nd International Workshop on Term Graph Rewriting*, Roma (Italy), September 2004. to appear.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.

[CKMM04] Horatiu Cirstea, Claude Kirchner, Michael Moossen, and Pierre-Etienne Moreau. Production systems and rete algorithm formalisation. Manifico deliverable, LORIA, Nancy, September 2004.

[CR03] Chris Culbert and Gary Riley. *Basic Programming Guide*, June 2003.

[dFFR00] Carlos Santos da Figueira Filho and Geber Lisboa Ramalho. Jeops - the java embedded object production system. *Lectures Notes in Artificial Intelligence*, 1952, 2000.

[DK99a] Hubert Dubois and Hélène Kirchner. Modelling planning problems with rules and strategies. Technical Report 99-R-029, LORIA, Nancy, France, March 1999.

[DK99b] Hubert Dubois and Hélène Kirchner. Rule based programming with constraints and strategies. Technical Report 99-R-084, LORIA, Nancy, France, November 1999. ERCIM workshop on Constraints, Paphos (Cyprus).

[DK00a] Hubert Dubois and Hélène Kirchner. Objects, rules and strategies in `ELAN`. In *Proceedings of the second AMAST workshop on Algebraic Methods in Language Processing, Iowa City, Iowa, USA*, May 2000.

[DK00b] Hubert Dubois and Hélène Kirchner. Rule Based Programming with Constraints and Strategies. In K.R. Apt, A.C. A. C. Kakas, E. Monfroy, and F. Rossi, editors, *New Trends in Constraints, Papers from the Joint ERCIM/Compulog-Net Workshop, Cyprus, October 25-27, 1999*, volume 1865 of *Lecture Notes in Artificial Intelligence*, pages 274–297. Springer-Verlag, 2000.

[Dub01] Hubert Dubois. *Systèmes de règles de production et calcul de réécriture*. PhD thesis, Université Henri Poincaré - Nancy 1, September 2001.

[Duf84] Pierre Dufresne. *Contribution algorithmique a l'inference par regles de production*. PhD thesis, Université Paul Sabatier de Toulousse, Juny 1984.

[FH86] F. Fages and G. Huet. Complete sets of unifiers and matchers in equational theories. *Theoretical Computer Science*, 43(1):189–200, 1986.

[FH03] Ernest J. Friedman-Hill. *JESS in Action*. Manning Publications Co., 2003.

[FL91] Francois Fages and Rémi Lissajoux. Systèmes experts temps-réel: une introduction au langage xrete. *Revue Technique Thomson-CSF*, 23 - 3:633–699, 1991.

[FL92] Francois Fages and Rémi Lissajoux. Sémantique opérationnelle et compilation des systèmes de production. *Revue d'intelligence artificielle*, 6 - 4:431–456, 1992.

[For74] Charles Forgy. A network fast routine for production systems. Working paper, Carnegie-Mellon University, 1974.

[For81] Charles Forgy. Ops5 user's manual. Technical Report CMU-CS-81-135, Carnegie-Mellon University, Pittsburgh, USA, July 1981. 62 pages.

[For82] Charles Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[Frü98] Thom Frühwirth. Theory and practice of constraint handling rules. *Special Issue on Constraint Logic Programming*, 37(1-3):98–135, October 1998.

[Gal86] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*, volume 5 of *Computer Science and Technology Series*. Harper & Row, New York, 1986.

[Ish94a] Toru Ishida. An optimization algorithm for production systems. *IEEE Transactions on Knownledge and Data Engineering*, 6 - 4:549–558, 1994.

[Ish94b] Toru Ishida. Parallel, distributed and multiagent production systems. *Lecture Notes in Artificial Intelligence*, 878, 1994.

[KDK93] Francis Klay, Eric Domenjoud, and Claude Kirchner. Vérification sémantique de spécifications métallurgiques. Rapport de fin de contrat, Inria Lorraine & Crin, 1993.

[KK99] Claude Kirchner and Hélène Kirchner. Rewriting, solving, proving. A preliminary version of a book available at `www.loria.fr/~ckirchne/rsp.ps.gz`, 1999.

[LG89] Thomas Laffey and Anoop Gupta. Real-time knownledge-based systems, 1989.

[Lop87] Frank Lopez. *The Parallel Production System*. PhD thesis, University of Illinois, Urbana-Champaign, Illinois, USA, 1987.

[MBLG90] Daniel Miranker, David Brant, Bernie Lofaso, and David Gadbois. On the performance of lazy matching in production systems. *Knowledge Presentation*, pages 685–692, 1990.

[Mir90] Daniel Miranker. *Treat: a new and efficient match algorithme for AI production systems*. Morgan Kaufmann, 1990.

[MRV01] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler. In D. Parigot and M. G. J. van den Brand, editors, *1st International Workshop on Language Descriptions, Tools and Applications*, 2001.

[S.A04] ILog S.A. Ilog jrules 4.6 technical white paper, 2004.

[SS96] Wayne Snyder and James Schmolze. Rewrite semantics for production rule systems: Theory and applications. In Michael McRobbie and John Slaney, editors, *Proceedings 13th International Conference on Automated Deduction, New Brunswick NY (USA)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 508–522. Springer-Verlag, July 1996.

[“T02] “Terese” (M. Bezem, J. W. Klop and R. de Vrijer, eds.). *Term Rewriting Systems*. Cambridge University Press, 2002.

[Thé90] Philippe Théret. De l’efficacité des systèms de règles de production. Technical Report 90.09 / 002 / P, IBSI électronique, September 1990.

[Thé94] Philippe Théret. *De l’efficacité des interpréteurs de systèms de règles de production dans les systèmes à base de connaissances*. PhD thesis, Université Paris XIII, February 1994.

[TR89] Milind Tambe and Paul Rosenbloom. Eliminating expensive chunks by restricting expressiveness. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 431–456, 1989.

[X-T88] X-tra 1.0 - manuel de reference, 1988.