MANIFICO Méta-compilation non intrusive du filtrage par contraintes
Non intrusive meta-compilation of matching by constraints

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 / 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

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. 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 ($\mathcal{WM}$) 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

  $\underline{\text{[Name] if Condition then Action}}$

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.

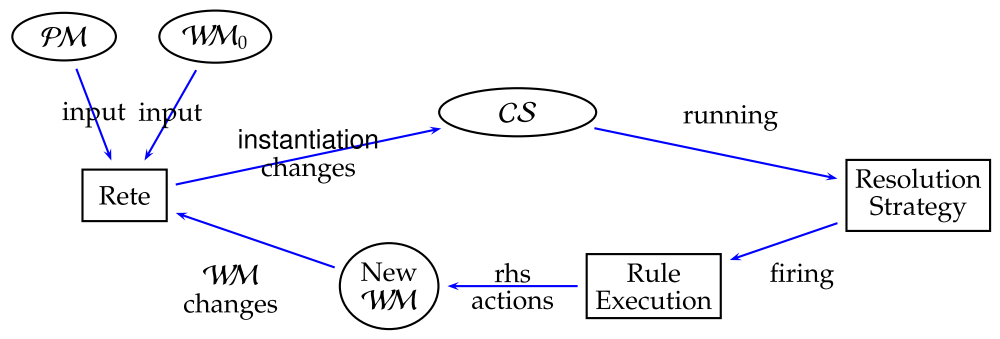


Figure 1: production system work flow

[1]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 $\mathcal{PM}$, 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:

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

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 General 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 whose set of free variables is a subset of the pattern variables: $Var(c) \subseteq Var(p)$.
- 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, $r = \{\ r_i\ \}_{i \in I_r}$, where $Var(r) \subseteq Var(p^+)$.
- 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, $a = \{\ a_i\ \}_{i \in I_a}$, where $Var(a) \subseteq Var(c)$.

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 $\mathcal{WM}$, p, r and a.

Definition 2.4 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 2.5 Given a substitution σ, so that $Dom(\sigma) = Var(c)$, 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. $\mathcal{T} \models \sigma(c)$
3. $\sigma p^- \not\ll_{\mathcal{T}} \mathcal{WM}$

for a minimal (with respect to the subset ordering) subset $\mathcal{WM}'$ of $\mathcal{WM}$. A fireable rule is also called an *activation*.

Definition 2.6 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 2.7 For a given working memory $\mathcal{WM}$ and a set of production rules $\mathcal{R}$, the set

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

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 2.8 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 2.9 A *general production system* is defined as

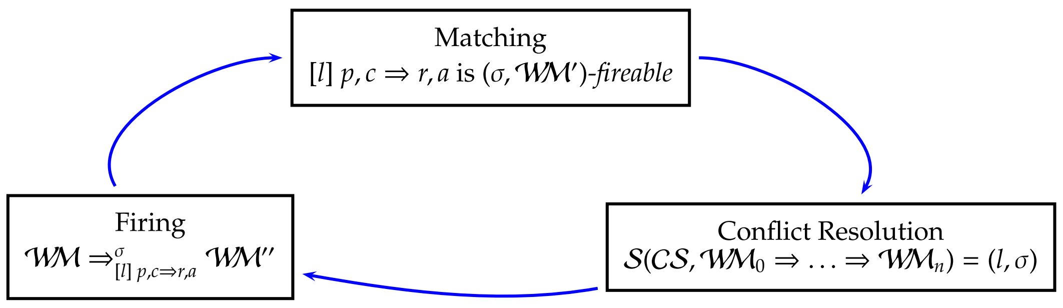
$$\mathcal{GPS} = (\ \mathcal{F},\ \mathcal{P},\ \mathcal{X},\ \mathcal{L},\ \mathcal{WM}_0,\ \mathcal{R},\ \mathcal{S},\ \mathcal{T}\ )$$

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,
- $\mathcal{WM}_0$ is the initial working memory,
- $\mathcal{R}$ is the set of production rules over $\mathcal{H} = (\ \mathcal{F},\ \mathcal{P},\ \mathcal{X}\ \mathcal{L}\ )$,
- $\mathcal{S}$ is the resolution strategy,
- $\mathcal{T}$ 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:


Matching
$[l]\ p, c \Rightarrow r, a$ is $(\sigma, \mathcal{WM}')$-*fireable*
Conflict Resolution
$\mathcal{S}(\mathcal{CS}, \mathcal{WM}_0 \Rightarrow \ldots \Rightarrow \mathcal{WM}_n) = (l, \sigma)$
Firing
$\mathcal{WM} \Rightarrow^{\sigma}_{[l]\ p,c \Rightarrow r,a} \mathcal{WM}''$

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

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

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 build dinamically.

$$\forall r_j \in r \ \exists p_i \in p^+ \ \mid \ \sigma r_j = \sigma p_i$$

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

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

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
- $\mathcal{S}$ 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 they capabilities in a short and simple way. We asume that numbers and arithmetics are builtin.

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

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:

$$
\begin{array}{rcl}
\mathcal{P} & := & \{ </2,\ =/2 \} \\
\mathcal{F} & := & \{ \\
 & & \quad house/4,\ houseaddress/4,\ myaddress/3,\ war/2,\ searching/0, \\
 & & \quad red/0,\ blue/0,\ usa/0,\ irak/0,\ france/0 \\
 & & \} \\
\mathcal{X} & := & \{ x \mid x \text{ begins with a question mark} \} \\
\mathcal{L} & := & \{ HouseSearch \} \\
\mathcal{WM}_0 & := & \{ \\
 & & \quad house(1,\ red,\ 341,\ true),\ houseaddress(1,\ 251,\ \text{"}rue\ jeanne\ d'arc\text{"},\ \text{"}nancy\text{"}), \\
 & & \quad house(2,\ blue,\ 390,\ true),\ houseaddress(2,\ 121,\ \text{"}avenue\ de\ brabois\text{"},\ \text{"}villers\ les\ nancy\text{"}), \\
 & & \quad house(3,\ red,\ 415,\ true),\ houseaddress(3,\ 31,\ \text{"}rue\ carnot\text{"},\ \text{"}vandoeuvre\ les\ nancy\text{"}), \\
 & & \quad myaddress(2551,\ \text{"}gorbea\text{"},\ \text{"}santiago\text{"}), \\
 & & \quad war(usa,\ irak), \\
 & & \quad searching() \\
 & & \}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{R} & := & \{ \\
& & \quad [HouseSearch] \\
& & \quad\quad searching() \ \wedge \\
& & \quad\quad house(?id,\ red,\ ?price,\ true) \ \wedge \\
& & \quad\quad houseadress(?id,\ ?number,\ ?street,\ ?city) \ \wedge \\
& & \quad\quad myaddress(?mn,\ ?ms,\ ?mc) \ \wedge \\
& & \quad\quad \neg war(?s1,\ france) \ \wedge \\
& & \quad\quad \neg war(france,\ ?s2)\ , \\
& & \quad\quad ?price < 400 \\
& & \quad \Longrightarrow \\
& & \quad\quad \{ \\
& & \quad\quad\quad searching(), \\
& & \quad\quad\quad house(?id,\ red,\ ?price,\ true), \\
& & \quad\quad\quad myaddress(?mn,\ ?ms,\ ?mc) \\
& & \quad\quad \}, \\
& & \quad\quad \{ \\
& & \quad\quad\quad house(?id,\ red,\ ?price,\ false), \\
& & \quad\quad\quad myaddress(?number,\ ?street,\ ?city) \\
& & \quad\quad \} \\
& & \} \\
\mathcal{S} & := & \text{a FIFO strategy}
\end{array}
$$

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:

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

When clear from the context a production system is just:

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

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

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

- 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:

$$\begin{tikzcd}
\text{Changes to the } \mathcal{WM} \arrow[d] \\
\text{Rete Algorithm} \arrow[d] \\
\text{Changes to the } \mathcal{CS}
\end{tikzcd}$$

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 *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, 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 *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 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 $\mathcal{CS}$. 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 3.

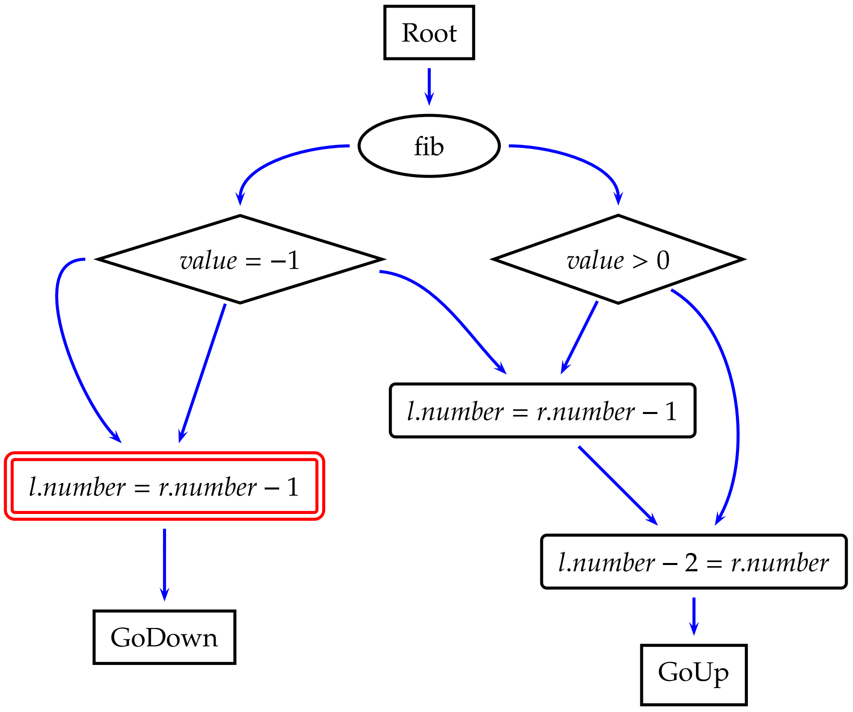


Figure 3: Rete network for Fibonacci example

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

$$\mathcal{WM}_0 \;:=\; \{\, fib(0,\ 1),\ fib(1,\ 1),\ fib(2,\ -1) \,\}$$

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* (it could be also numbered with 0 and 1; or 1 and 2).

Example 3.2 Rete Network Execution.
First, we initialize

$$\mathcal{WM} \;:=\; \emptyset \;\wedge\; \mathcal{CS} \;:=\; \emptyset$$

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

$$\mathcal{WM} \;:=\; \mathcal{WM} \cup \{\, fib(0,\ 1) \,\}$$

and we send the respective token to the rete network, which first checks for the type of the token, if it is equals to 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 $value = -1$, it is not the case, so the token on the left branch is cast away, on the right branch we check for $value > 0$, 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 $\mathcal{WM}_0$ to the $\mathcal{WM}$

$$\mathcal{WM} \;:=\; \mathcal{WM} \cup \{\, fib(1,\ 1) \,\}$$

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 $\mathcal{WM}_0$ to the $\mathcal{WM}$

$$\mathcal{WM} \;:=\; \mathcal{WM} \cup \{\, fib(2,\ -1) \,\}$$

in this case, the left branch is taken, and the check $value = -1$ 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 $fib(0,\ 1)$ and $fib(1,\ 1)$ waiting for action. And the condition holds for $fib(2,\ -1)$ and $fib(1,\ 1)$, so the just incoming token is given to the next two input token where $fib(0,\ 1)$ and $fib(1,\ 1)$ are waiting and now the condition holds with $fib(0,\ 1)$ and finally the *GoUp* rule is activated,

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

Now, our FIFO strategy selects the first instantiation from the $\mathcal{CS}$, updating it as

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

and executes the right hand side, first computing the substitution σ as

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

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

$$\mathcal{WM} \;:=\; \mathcal{WM} \setminus \{\, fib(2,\ -1) \,\}$$

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

$$\mathcal{WM} \ := \ \{\ fib(1,\ 1),\ fib(2,\ 2)\ \}$$

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* ω of a subterm in a term as:

$$\begin{aligned} \omega(t) \mid t &= 0 \\ \omega(s_i) \mid t(s_1, \ldots, s_n) &= i \\ \omega_1 = \omega(s) \mid t \ \wedge \ \omega_2 = \omega(t) \mid r \implies \omega(s) \mid r &= \omega_1.\omega_2 \end{aligned}$$

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

$$\begin{aligned}
comp(t) &:= comp(t, 0) \text{ for } t \in \mathcal{T}(\mathcal{X}, \mathcal{P}) \\
comp(t(s_1, \ldots, s_n)) &:= input.\omega = t \qquad \text{(1a)} \\
&\quad \bigwedge_{i \in I} comp(s_i, w.i) \\
&\quad \bigwedge_{\forall i<j \in I\ \exists \omega_1, \omega_2 \mid s_i|\omega_1 = s_j|\omega_2} input.\omega.\omega_1 = input.\omega.\omega_2 \qquad \text{(1b)} \\
comp(\neg t(s_1, \ldots, s_n)) &:= \nexists input \in \mathcal{WM} \ \mid \ comp(t(s_1, \ldots, s_n)) \\
comp(p, c) &:= \bigwedge_{i \in I} comp(p_i) \\
&\quad \bigwedge_{\forall i<j \in I(p^+) \vee I(P^-)\ \forall \omega_1, \omega_2 \mid p_i|\omega_1 = p_j|\omega_2 \in \mathcal{X}} input_i.\omega_1 = input_2.\omega_2 \qquad \text{(1c)} \\
&\quad \bigwedge_{\forall i \in I(p^+) \wedge j \in I(p^-)\ \forall \omega_1, \omega_2 \mid p_i|\omega_1 = p_j|\omega_2 \in \mathcal{X} \ \wedge \ \nexists f \in \mathcal{WM}} input_i.\omega_1 = f.\omega_2 \qquad \text{(1d)} \\
&\quad \bigwedge_{\forall i<j \in I\ \exists \omega \mid p_i|\omega = x \in Var(c)} input_i.\omega = x =: \sigma_i \ \wedge \ \sigma_i(c) \qquad \text{(1e)}
\end{aligned}$$

where *input* is a fact that may match the given pattern, as well $input_i$ for pattern p_i.
Remark: This is just an incremental definition of

$$p^+ \ll^\sigma \mathcal{WM} \ \wedge \ p^- \not\ll \mathcal{WM} \wedge \ \models \sigma(c)$$

as described in Definitions 2.4 and 2.5.

Now, we clasify all tests over $comp(p, c)$ for a given rule, identified by its label, l_j, in $\mathcal{PM}$. First, we call q_i^j 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^j the conjuction 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 π of a rule in *left and right memories*, *lm* and *rm*.

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

$$\begin{aligned} lm_i &= \{ (f_1, \dots, f_{k_i}) \in \mathcal{WM} \mid comp(\pi)(f_1, \dots, f_{k_i}) \} \\ rm_i &= \{ f \in \mathcal{WM} \mid q_{i+1}(f) \} \\ lm_1 &= rm_0 \end{aligned}$$

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 $l_j \in \mathcal{PM}$, we connect the *root* node with each $Anode(q^j_{i,0})$, and we connect the whole sequence of intra-element tests as follows:

$$\forall k \in K \mid Anode(q^j_{i,k}) \longrightarrow Anode(q^j_{i,k+1})$$

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

$$q^{j_1}_{i_1,k_1} = q^{j_2}_{i_2,k_2}$$

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

$$Anode(q^{j_1}_{i_1,k_1-1}) \longrightarrow A \longrightarrow Anode(q^{j_1}_{i_1,k_1+1}), \text{and}$$

$$Anode(q^{j_2}_{i_2,k_2-1}) \longrightarrow A \longrightarrow Anode(q^{j_2}_{i_2,k_2+1})$$

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 $Any^j_{i_1,i_2}$, for rule l_j and patterns i_1 and i_2, and the related to a subequation (1d) are called *negative Bnodes* and labeled $Not^j_{i_1,i_2}$. 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 ω names, as:

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

so, we have:

$$\begin{aligned} q^1_{1,0} = q^1_{2,0} = q^2_{1,0} = q^2_{2,0} = q^2_{3,0} &\equiv input.type = fib \\ q^1_{1,1} = q^2_{1,1} &\equiv input.value = -1 \\ q^2_{2,1} = q^2_{3,1} &\equiv input.value > 0 \\ Not^1_{1,2} &\equiv input_1.number - 1 = input_2.number \\ Any^2_{1,2} &\equiv input_1.number - 1 = input_2.number \\ Any^2_{1,3} &\equiv input_1.number - 2 = input_3.number \\ c^2_{2,3} &\equiv ?v = input_2.value + input_3.value \end{aligned}$$

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

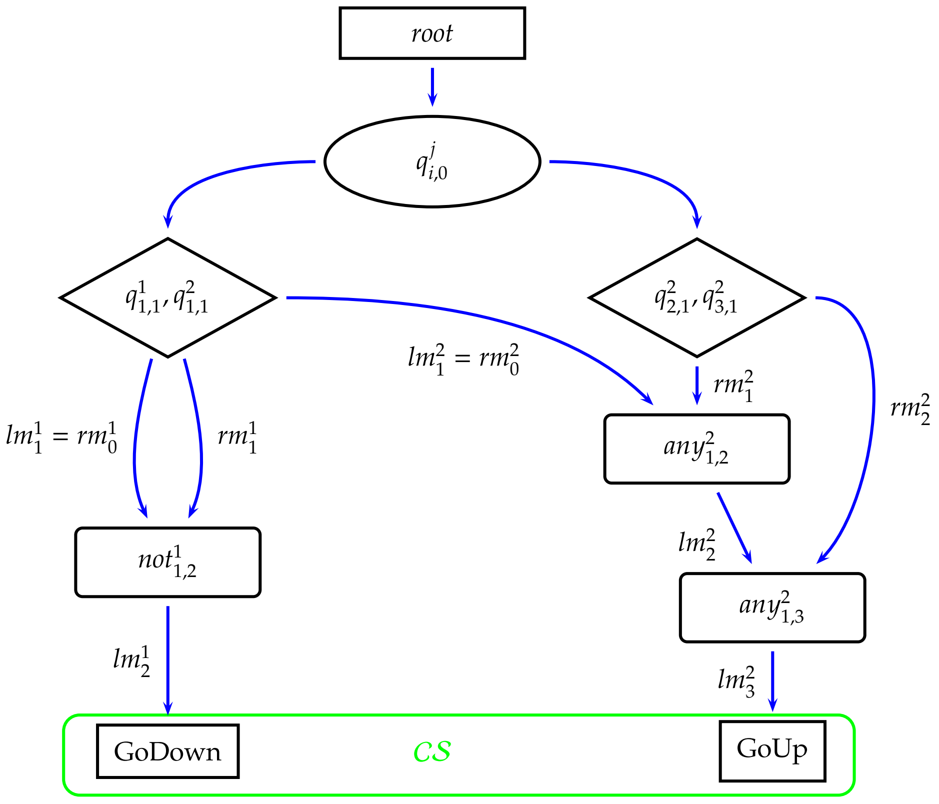


Figure 4: Rete network for Fibonacci example

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:

$$< + \mid -,\ f >? \longrightarrow \{< + \mid -,\ f >\}! \quad \text{if } q(f)$$

$$< + \mid -,\ f >? \longrightarrow \emptyset! \quad \text{if } \neg q(f)$$

And for *Bnodes*, each right memory behaves like:

$$< -,\ f >? \xrightarrow{rm \leftarrow rm - \{f\}} \{< -,\ f >\}!$$

$$< +,\ f >? \xrightarrow{rm \leftarrow rm \cup \{f\} \wedge \sigma \leftarrow \sigma \cup \sigma' \mid \exists R \in \mathcal{PM} \wedge \exists p \in p^+(R) \mid \sigma' p = f} \{< +,\ f >\}! \quad \text{if } q(f)$$

$$< +,\ f >? \longrightarrow \emptyset! \quad \text{if } \neg q(f)$$

Where each left memory behaves like:

$$< -,\ f >? \xrightarrow{lm \leftarrow lm - \{f\}} \{< -,\ f >\}!$$

$$< +,\ f >? \overset{lm \leftarrow lm \cup \{f\}}{\longrightarrow} \{< +,\ f >\}!$$

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:

$$< -,\ f >? \longrightarrow \{< -,\ f >\}!$$

$$< +,\ f >? \longrightarrow \{< +,\ (f,\ l) > \quad \forall l \in rm \mid r(l,\ f) >\}!$$

Right:

$$< -,\ f >? \longrightarrow \{< -,\ f >\}!$$

$$< +,\ f >? \longrightarrow \{< +,\ (l,\ f) > \quad \forall l \in rm \mid r(l,\ f) >\}!$$

And for negative *Bnodes*:

Left:

$$< -,\ f >? \longrightarrow \{< -,\ f >\}!$$

$$< +,\ f >? \longrightarrow \{< +,\ f >\}! \quad \text{if} \ \ \forall l \in rm \mid \neg r(l,\ f)$$

Right:

$$< -,\ f >? \longrightarrow \{< +,\ l > \quad \forall l \in lm \mid \forall f' \in rm \mid \neg r(l,\ f')\}!$$

$$< +,\ f >? \longrightarrow \{< -,\ l > \quad \forall l \in lm \mid r(l,\ f) >\}!$$

Finally for the $\mathcal{CS}$ we have:

$$< -,\ f >? \overset{lm \leftarrow lm - \{f\}}{\longrightarrow} \{< -,\ f >\}!$$

$$< +,\ f >? \overset{lm \leftarrow lm \cup \{f\}}{\longrightarrow} \{< +,\ f >\}!$$

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, $\mathcal{GPS}$s, and shallow, simple, syntactic and static production systems, $S^4\mathcal{PS}$s.

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 $\mathcal{WM}$ 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

[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.