Production and Rewrite Systems
Horatiu Cirstea, Claude Kirchner, Michael Mm Moossen, Pierre-Etienne Moreau

To cite this version:

HAL Id: inria-00280939
https://inria.hal.science/inria-00280939
Submitted on 20 May 2008

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

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

RETE et réécriture

Production and Rewrite Systems

<table>
<thead>
<tr>
<th>Description</th>
<th>This report studies the relationship between production systems and term rewrite systems.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auteur(s)</td>
<td>Horatiu CIRSTEA, Claude KIRCHNER, Michael MOOSSEN, Pierre-Etienne MOREAU</td>
</tr>
<tr>
<td>Référence</td>
<td>MANIFICO/Sous Projet 1/Fourniture 1.2/V0.8</td>
</tr>
<tr>
<td>Date</td>
<td>31 août 2004</td>
</tr>
<tr>
<td>Statut</td>
<td>à valider</td>
</tr>
<tr>
<td>Version</td>
<td>0.8</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 août 2004</td>
<td>V 0.8</td>
<td>création du document</td>
</tr>
<tr>
<td>15 septembre 2004</td>
<td>V1.0</td>
<td>version finale</td>
</tr>
</tbody>
</table>

Contents

1 Definition of production systems 3
   1.1 Informal presentation of production systems 3
   1.2 General Production systems 5

2 Production rules as objects 7

3 Production rules as AC-rewrite systems 7
   3.1 Snyder and Schmolze’s approach 7
   3.2 Using matching constraints 8

4 Production rules via Constraint handling 9

5 Criticism of these previous approaches 9

6 Appendix: Examples 10
   6.1 Abstract Notation 10
   6.2 OPS5 11
   6.3 CLIPS 12
   6.4 JJeOPS 13
   6.5 JRules, using TRL 15
   6.6 ELAN 16
   6.7 AC-Rewriting 18
      6.7.1 Snyder and Schmolze’s approach 18
      6.7.2 Using matching constraints 18

7 Benchmarking 19
   7.1 A more demanding benchmark 19
Introduction

There is a strong and somewhat renewed interest in production and business rule systems. For example it is significant that in April 2004 Gardner published a comparative study of the different business rule systems and vendors and estimated that the current market of 200 millions of dollars was promised to a sustained increased for several years.

Production systems, as we will call them from now on, are widely appreciated in industry for their agility, reactivity and flexibility. But putting them in action is also delicate since their semantics is unclear or even unknown and therefore make the development of large systems quite depend of test and experience.

Term rewriting is a different kind a rule based system that emerged from automated deduction and semantics of programming languages in the early seventies. It is now a pretty well understood concept and many implementations show their usefulness, robustness and efficiency.

This report intends to bridge these a priori different views of rule based systems.

Some landmarks on production systems are:
– the initial RETE (Latin for ‘net’) algorithm and its use in OPS5 [For82, For81],
– an alternative, the TREAT approach [Mir90, MBLG90],
– The work by François Fages and its collaborators at Thomson leading to initial version of ilog-RULE: [FL91, FL92]
– the first link between production rules and rewriting done by Snyder and Schmolze [SS96],
– production rules as strategic rewriting in ELAN: [Dub01, DK99a, DK99b, DK00b, DK00a]
– with concerns about the verification of rule based system, for example the verification of production rules applied to raw steel manufacturing process [KDK93],
– and emerging from the constraint (logic) programming community, Constraint Handling Rules, CHR, a language for defining constraint solvers, but at the same time it is one of the most powerful multiset rewriting languages [Frü98]. – and also real time approaches like in [FL91, LG89, Lop87].

This leaded to the emergence of many implementations, including: G2 used in chemical industry, BladeAdvisor, CLIPS, ILOG-Rule, RulesPower commercialized by Forgy, LibRT selling a business rule checker called Valens, X-Tra a French ops5 implementation written in lisp [X-T88], Jess a CLIPS implementation in Java, JeOPS a OPS5 implementation in Java.

1 Definition of production systems

This section recalls the main concepts and definitions as introduced in [CKMM04].

1.1 Informal presentation of production systems

A production system consists mainly of the following five components:

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

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

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

- Production Rules are conditional statements of the form

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

- The Production Memory (PM) is the set 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:

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

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

3. **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:

\(<\text{rule, list of facts that satisfy its LHS}>\).

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

A schematic view of the data work flow in a production system\(^2\) is shown in Figure 1.

---

\(^{1}\)this is a historical name, that does not reflect the common meaning of instantiation

\(^{2}\)With non-modifiable Production Memory
1.2 General Production systems

We consider a set $\mathcal{T}$ of function symbols, usually denoted $f, g, h, \ldots$, a set $\mathcal{P}$ of predicate symbols, and infinite sets $X$ 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 $\mathcal{T}(\mathcal{F}, X)$), ground terms (denoted $\mathcal{T}(\mathcal{F})$), atomic propositions, literals (i.e. atomic propositions 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 1.1 A fact $f$ is a ground term, $f \in \mathcal{T}(\mathcal{F})$.

Definition 1.2 We call working memory ($\text{WM}$) a set of facts, i.e. it is a subset of the Herbrand universe defined on the signature.

Definition 1.3 A production rule or simply rule or production, denoted

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

consists of the following components.

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

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, add and remove facts.

Definition 1.4 Given a set of facts $S$ and a set of positive patterns $p^+$, $p^+$ is said to match $S$ with respect to a theory $T$ and a substitution $\sigma$, written $p^+ \wpre{T} S$ if:

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

We say that a set of negative patterns $p^-$ dis-matches a set of facts $S$, written $p^- \npre{T} S$ iff:

$$\forall \neg p \in p^- \ \forall t \in S \ \forall \sigma \ | \ \sigma(p) \neq_T t$$

Definition 1.5 Given a substitution $\sigma$, a production rule $[l] \ p, c \Rightarrow r, a$ is ($\sigma, \text{WM}$)-fireable on a working memory $\text{WM}$ when

1. $p^+ \wpre{T} \text{WM}$
2. $p^- \npre{T} \text{WM}$
3. $T \models \sigma(c)$
for a minimal (with respect to the subset ordering) subset \( \mathcal{WM}' \) of \( \mathcal{WM} \). A fireable rule is also called an activation.

**Definition 1.6** 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 \( \mathcal{WM} \Rightarrow_{[l] \, p, c \Rightarrow r, a} \mathcal{WM}'' \) 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 1.7** For a given working memory \( \mathcal{WM} \) and a set of production rules \( \mathcal{R} \), the set

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

is called the \( \mathcal{R}@\mathcal{WM}\)-conflict set.

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 1.8** 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 1.9** A general production system is defined as

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

Where:

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

**Definition 1.10** The Inference Cycle is defined as follows:

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

2. **Firing**
   \( \mathcal{WM} \Rightarrow_{[l] \, p, c \Rightarrow r, a} \mathcal{WM}'' \)

3. **Selection**
   \[
   S(\mathcal{CS}, \mathcal{WM}_0 \Rightarrow \ldots \Rightarrow \mathcal{WM}_n) = (l, \sigma)
   \]

—

ILOG, INRIA Lorraine, INRIA Rocquencourt
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.

In order to get good performances when running production systems, suitable restrictions have been used for the so called rete algorithm.

2 Production rules as objects

Production systems can be implemented from an object oriented point of view. This is detailed in the thesis of Hubert Dubois [Dub01].

3 Production rules as AC-rewrite systems

3.1 Snyder and Schmolze’s approach

We summarize here the approach proposed by Snyder and Schmolze in [SS96].

The three basic components of a Production System:

- The working memory
- The collection of production rules
- The interpreter for applying rules repeatedly to the memory, subject to some conflict resolution strategy.

are represented, using the main notion of associative and commutative terms, in the following way.

**Working Memory** The memory is considered to be a set of positive ground atoms, completed with negatives according to the Reiter’s Closed World Assumption.  

\[ W = P \cup N \]

where:

- \( P \) is a finite set of ground atoms (the actual memory facts), and
- \( N = \{ \neg A | A \notin P \} \) (all other negative facts)

**Production Rules** Depending on the fact that a production rule contains a Negation-As-Failure (NAF) condition or not, it may be translated into two different types of rewrite rules:

- Preserving rewrite rules.
  
  For production rules without a NAF condition, has the form:

  \[ L_1, \ldots, L_n \rightarrow L'_1, \ldots, L'_n \]

  where on each side of the arrow we have sets of literals (possibly containing variables) and where:

  \[ \forall i \ L'_i = L_i \lor L'_i = \bar{L}_i \]

  In other words, the set of atoms is the same on each side, but the sign of a particular atom may have flipped from positive to negative (representing a deletion), from negative to positive (representing an addition), or has remained the same (representing a side condition). The application of such a rewrite rule preserves the property of being a working memory.

  All production rules that only test for membership, only add or delete facts, and do not have NAF conditions, can be formalized by one or more preserving rewrite rules.  

---

5The Snyder and Schmolze’s full paper also considers the use of a multiset
4A negative condition which contains a variable which appears nowhere else.
5More than one rule may be necessary, since a rule might add an atom without testing for its absence.
Constrained preserving rules.
This kind of rules have the form $S \rightarrow T[\varphi]$ where $S \rightarrow T$ is a preserving rewrite rule and $\varphi$ is a first-order formula where every free variable in $\varphi$ occurs in $S$ (and thus in $T$). However, we only need constraints of a particular form; roughly, we will transform each NAF condition $L_i$ involving NAF variables $x_{i,1}, \ldots, x_{i,m}$, into a subconstraint of the form $\varphi_i = \forall x_{i,1}, \ldots, x_{i,m}, L_i$, where none of the $x_{i,j}$ appear in $S \setminus \{L_i\}$. This results in a constraint of the form $[\varphi_1 \land \cdots \land \varphi_n]$ for a production rule with $n$ NAF conditions.

But some NAF variable $x_{i,j}$ may be constrained by some condition $C$ in $L_i$, if this is the case, the constraint $\varphi_i$ should be expanded to $\varphi_i \lor \neg C$.

Production Rule Interpreter We formalize the last main feature of a production rule system as a rewriting process subject to checking the constraints.

We say that a WM $W$ rewrites to $W'$ with respect to a set of constrained rules $R$, denoted $M \rightarrow_r M'$, if

- there exists a rule $P \rightarrow T[\varphi]$ in $R$,
- $M$ has the form $M[P\sigma]$ for some substitution $\sigma$ such that $M \models \varphi\sigma$, and
- $M'$ has the form $M[T\sigma]$.

The subset $P\sigma$ is called the redex.

Given this, we have a good approximation of a production rule system as an AC-Rewrite system, but we do not model the conflict resolution strategy.

Since we are acting on term rewriting (in this case modulo AC), the resolution of conflicts could be done in at least two ways.

The first one consists of having the production system encoded as a confluent AC-rewrite system. In this case, there is always a common reduct for any two terms issued from a common ancestor. If moreover the rewrite system is terminating, the result is uniquely defined, independently of the rewrite strategy in use. AC-term rewriting and its meta-properties have been extensively studied, from AC-unification [Sti75], AC-completion [PS81, Hul80] (generalizing Knuth and Bendix approach [KB70]), to AC-termination [RN93, Rub02]. Moreover, AC-matching [Hul79, Eke95] can be efficiently compiled [KM01]. Even if the theoretical background of confluent and terminating rewrite system is now well investigated, this approach has the drawback that, indeed, only few production systems are in essence confluent and terminating.

The second one consists in using explicitly rewrite strategies for explicitly guiding the term rewriting process. This is typical of the approach taken in the ELAN system where rewrite rule split in two categories. The first one called “un-labeled rules” are used for computation and are therefore assumed to form a confluent and terminating rewrite system. The second kind of rules are “labeled rules” and are assumed to be controlled by a strategy using the labels of the rule to fire the appropriate one. This is described in particular in [KKV94, BCD+04] and gave rise to the rewriting calculus [CK01], generalizing term rewriting as well as lambda-calculus and where strategy guided rewriting can be given a precise semantics [CKLW03]. Moreover (symbolic) constraint rewriting could be simply described in this setting, providing an explicit handling of substitutions and exceptions [FK02, CFK04] but also of structure with sharing and cycles [BBCK04].

For a complete example of such an encoding, see the appendix Section 6.7 page 18.

3.2 Using matching constraints
We model a production system using an approach simplifying the one by Schmolze and Snyder. The main originality is the introduction of a new predicate “don’t match” which is true if a condition does not match any fact in the working memory. It is therefore a dis-matching problem that shares many similarities with dis-unification one [Com91].

Definition 3.1 Let $s, t$ be terms and $T$ a theory. A matching constraint, denoted $s \triangleleft_T t$, has a solution $\sigma$ when $\sigma(s) =_T t$. When a matching constraint is unsatisfiable (i.e. has no solution), this
is denoted \( s \not\in_T t \): there exist no substitution \( \sigma \) such that \( \sigma(s) =_T t \). When \( T \) is the empty theory, it is just omitted.

A production system can be then modeled is the following way:
Propositions are considered as terms of type bool.
The working memory is a term, denoted \( \text{WM} \) and built using the constructors:

- nil
- an associative and commutative operator \( \cdot \) with neutral element nil.

A production rule \([l] \text{ if } p, c \text{ remove } r \text{ add } a \) is a conditional rewrite rule

\[
c, r \rightarrow a \text{ if } C
\]

with the standard semantics of AC-rewriting, that is:

\[
\text{WM} \rightarrow_{c \leftarrow_d} \text{if } C \Rightarrow \exists \sigma, \sigma(c) =_{AC} \text{WM} \text{ and } \text{WM}' = \sigma(a), \text{ provided } \sigma(C)
\]

See Section 6.7.2 on page 18 for a simple example of this encoding.

4 Production rules via Constraint handling

After discussing with HAK [ck: Mar 8 04]
Conditions are constraints
Actions are constraints of the form \( x' = f(x) \), where \( x' \) is the next value of \( x \).
After computing the value \( f(x) \), one gets the constraint: \( x = oldValue \land x' = newValue \).

Then one needs a meta operation that
1. remove the constraint \( (x = oldValue \land x' = newValue) \) from store
2. add the constraint \( x = newValue \) to store
3. propagate (may be not yet)

In this way \( x \) gets its new value
a bit (too?) magic i.e. too much adhoc??

Another try:
1. remove the constraint \( x = oldValue \) from store
2. add the constraint \( x = x' \) to store
3. propagate (to identify \( x \) and \( x' \) everywhere, but propagate will do more . . . )

5 Criticism of these previous approaches

All do not take care of the main pb of the rule firing semantics, i.e.

- when becomes a simple rule fire-able?
- could the user specify the order in which (s)he wants the fire-able rules to be fired?
- when unspecified what is the firing order?

Acknowledgments: Thanks to François Charpillet for sharing with us his X-tra experience.
6 Appendix: Examples

This appendix shows a same toy example, first in abstract notation and then implemented in several production system.

6.1 Abstract Notation

$$P := \{ \text{not, } <, =, \wedge \}$$

$$F := \{$$

- house_id: house -> number,
- color: house -> atom,
- price: house -> number,
- forrent: house -> atom \{true\|false\},
- houseaddress_id: houseaddress -> number,
- houseaddress_number: houseaddress -> number,
- houseaddress_street: houseaddress -> string,
- houseaddress_city: houseaddress -> string,
- myaddress_number: myaddress -> number,
- myaddress_street: myaddress -> string,
- myaddress_city: myaddress -> string,
- side1: war -> string,
- side2: war -> string

$$\}$$

$$W_0 := \{$$

- house(house_id()=1 \wedge color()=red \wedge price()=341 \wedge forrent()=true),
- houseaddress(houseaddress_id()=1 \wedge houseaddress_number()=251 \wedge houseaddress_street()="Rue Jeanne D'Arc" \wedge houseaddress_city()=Nancy),
- house(house_id()=2 \wedge color()=blue \wedge price()=390 \wedge forrent()=true),
- houseaddress(houseaddress_id()=2 \wedge houseaddress_number()=121 \wedge houseaddress_street()="Avenue de Brabois" \wedge houseaddress_city()=Villers-les-Nancy),
- house(house_id()=3 \wedge color()=red \wedge price()=415 \wedge forrent()=true),
- houseaddress(houseaddress_id()=3 \wedge houseaddress_number()=31 \wedge houseaddress_street()="Rue Carnot" \wedge houseaddress_city()=Vandoeuvre-les-Nancy),
- myaddress(myaddress_number()=2551 \wedge myaddress_street()=Gorbea \wedge myaddress_city()=Santiago),
- searching(),
- (war side1()=usa side2()=irak)

$$\}$$

$$R := \{$$

if (searching) \wedge
  (house_id(?h)=?id \wedge color(?h)=red \wedge price(?h)<400 \wedge forrent(?h)=true) \wedge
  (houseaddress_id(?ha)=?id \wedge houseaddress_number(?ha)=?num \wedge houseaddress_street(?ha)=?str \wedge
  houseaddress_city(?ha)=?cit) \wedge
  (myaddress) \wedge
  not (war(side1()=france)) \wedge
  not (war(side2()=france))

then
  remove (searching)
  forrent(?h)=false
  myaddress_number(myaddress)=?num \wedge
  myaddress_street(myaddress)=?str \wedge
  myaddress_city(myaddress)=?cit

end if

$$P := \{ \text{false/0, true/0, not/1, } <,/2, =/2, \wedge/2, \text{ remove/1, add/1} \}$$

$$F := \{ \text{house/4, houseaddress/4, myaddress/3, war/2, searching/0} \}$$

$$W_0 := \{$$

- 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),
- searching(),
- war(usa, irak)

$$\}$$
\[
\begin{align*}
R := \{ \\
& \text{if} \\
& \quad \text{searching()} \ \\
& \quad \text{house}(\text{id}, \text{red}, \text{?pr}, \text{true}) \ \\
& \quad \text{?pr < 400} \ \\
& \quad \text{houseaddress}(\text{id}, \text{?num}, \text{?str}, \text{?cit}) \ \\
& \quad \text{myaddress}(\text{?mn}, \text{?ms}, \text{?mc}) \ \\
& \quad \text{not war}(	ext{france}, \text{?s2}) \ \\
& \quad \text{not war}(\text{?s1}, \text{france}) \ \\
& \quad \text{then} \\
& \quad \text{remove}(\text{searching()}), \\
& \quad \text{remove}(\text{house}(\text{id}, \text{red}, \text{?pr}, \text{true})), \\
& \quad \text{add}(\text{house}(\text{id}, \text{red}, \text{?pr}, \text{false})), \\
& \quad \text{remove}(\text{myaddress}(\text{?mn}, \text{?ms}, \text{?mc})), \\
& \quad \text{add}(\text{myaddress}(\text{?num}, \text{?str}, \text{?cit}))) \ \\
& \quad \text{end if} \\
\}
\end{align*}
\]

### 6.2 OPS5

**OPS5 Core Syntax Explanations**

- `;`: line comment.
- `(\text{literate}<\text{factname}> <\text{slotname}'>*)` used for defining a new type of fact called `<factname>`, with 0, 1 or more slots (or properties) `<slotname>'*).
- `(p <\text{rulename}? <\text{pattern}'>* --> <\text{action}'>*)` used for defining a rule called `<rulename>`, with conditions or patterns `<pattern>'* and actions `<action>'*.
- `(\text{make}<\text{factname}> ["<\text{slotname}> <\text{value}>"]*)` used for creating a new fact, called `<factname>`, adding it to the WM.
- `(\text{remove}<\text{cid}>)` used for retracting the fact that satisfy the condition given by `<cid>`, which is just the number of the condition in the LHS.
- `(\text{modify}<\text{cid}> ["<\text{slotname}> <\text{newvalue}>"]*)` used for modifying the fact that satisfy the condition given by `<cid>`. Modifying just the values of the given slots by `<slotname>` replacing the value with `<newvalue>`. Internally implemented as a (remove `<cid>`) followed by a (make ...).

**OPS5 Example**

: new type(fact) definitions
:a fact called house(representing a house) with 4 "properties": id, color, price and forrent
(literate house id color price forrent)
a fact called houseaddress(representing the address of a house) with 4 "properties":
:id, number, street and city
(literate houseaddress id number street city)
a fact called myaddress(representing my address) with 3 "properties": number, street and city
(literate myaddress number street city)
a fact called searching with no properties, just for indicating that i am looking for a new house!
(literate searching)
a fact called war(representing a war between 2 sides) with 2 "properties": side1 and side2
(literate war side1 side2)

:a rule with the name search_for_house
(p search_for_house
;if the searching flag is set
(searching)
;search for a red house with a price less than 400
(house 'id <id> 'color red 'price < 400 'forrent true)
;get the address of that house
(houseaddress 'id <id> 'number <num> 'street <str> 'city <cit>)
;get my address
(myaddress)
if there is no war where france is involved
-(war `side1 france)
-(war `side2 france)
-->
;i found a nice house so i delete the searching fact
( remove 1)
: modify the state of the house
(modify 2 `forrent false)
: modify my address
(modify 4 `number `<num>`street `<str>`city `<cit>)
)

: working memory initialization
: first house
(make house `id 1 `color red `price 341 `forrent true)
(make houseaddress `id 1 `number 251 `street "Rue Jeanne D'Arc" `city Nancy)
;second house
(make house `id 2 `color blue `price 390 `forrent true)
(make houseaddress `id 2 `number 121 `street "Avenue de Brabois" `city Villers-les-Nancy)
;third house
(make house `id 3 `color red `price 415 `forrent true)
(make houseaddress `id 3 `number 31 `street "Rue Carnot" `city Vandoeuvre-les-Nancy)
;my address
(make myaddress `number 2551 `street Gorbea `city Santiago)
;searching flag
(make searching)
;war!
(make war `side1 usa `side2 irak)

6.3 CLIPS

CLIPS Core Syntax Explanations

- ;: line comment.

- (deftemplate <factname> (slot <slotname>)*): used for defining a new type of fact called <factname>, with 0, 1 or more slots (or properties) <slotname>*.

- (defrule <rulename>? ([<cid> <- ]<pattern>* => <action>*)): used for defining a rule called <rulename>, with conditions or patterns <pattern>* which can be associated to a given condition id <cid>, and actions <action>*.

- (assert <factname> (<slotname> <value>)*): used for creating a new fact, called <factname>, adding it to the WM.

- (deffacts <factgroupname> (<factname> (<slotname> <value>)*)*): used for creating several facts at once.

- (retract <cid>): used for retracting the fact that satisfy the condition given by <cid>.

- (modify <cid> (<slotname> <newvalue>)+): used for modifying the fact that satisfy the condition given by <cid>. Modifying just the values of the given slots by <slotname> replacing the value with <newvalue>. Internally implemented as a (retract <cid>) followed by a (make ...).

CLIPS Example

;new type(fact) definitions
;a fact called house( representing a house) with 4 "properties": id, color, price and forrent
(deftemplate house (slot id) (slot color) (slot price) (slot forrent))
;a fact called houseaddress( representing the address of a house) with 4 "properties":
;id, number, street and city
(deftemplate houseaddress (slot id) (slot number) (slot street) (slot city))
;a fact called myaddress( representing my address) with 3 "properties": number, street and city
(deftemplate myaddress (slot number) (slot street) (slot city))
;a fact called searching with no properties, just for indicating that i am looking for a new house!
(deftemplate searching)
; a fact called war (representing a war between 2 sides) with 2 "properties": side1 and side2
(deftemplate war (slot side1) (slot side2))

; a rule with the name search_for_house
(defrule search_for_house
 ; if the searching flag is set, and bind the matching fact to ?f1
  ?f1 <- (searching)
 ; search for a red house with a price less than 400, and bind the matching fact to ?f2
  ?f2 <- (house (id ?id) (color red) (price ?p) (forrent true))
      (test (< ?p 400))
 ; get the address of that house
  (houseaddress (id ?id) (number ?num) (street ?str) (city ?cit))
 ; get my address, and bind the matching fact to ?f3
  ?f3 <- (myaddress)
 ; if there is no war where france is involved
   (not (war (side1 france)))
   (not (war (side2 france)))
 =>
 ; i found a nice house so i delete the searching fact
   (retract ?f1)
 ; modify the state of the house
   (modify ?f2 (forrent false))
 ; modify my address
   (modify ?f3 (number ?num) (street ?str) (city ?cit))
)

; working memory initialization
(deffacts startup
 ; first house
  (house (id 1) (color red) (price 341) (forrent true))
  (houseaddress (id 1) (number 251) (street "Rue Jeanne D'Arc") (city Nancy))
 ; second house
  (house (id 2) (color blue) (price 390) (forrent true))
  (houseaddress (id 2) (number 121) (street "Avenue de Brabois") (city Villers-les-Nancy))
 ; third house
  (house (id 3) (color red) (price 415) (forrent true))
  (houseaddress (id 3) (number 31) (street "Rue Carnot") (city Vandoeuvre-les-Nancy))
 ; my address
  (myaddress (number 2551) (street Gorbea) (city Santiago))
 ; searching flag
 ; (searching)
 ; war!
  (war (side1 usa) (side2 irak))
)

6.4 JeOPS

JeOPS Core Syntax Explanations

• First you have to define one Java class for each type of Fact. They have not to implement or extend nothing special. And you can use any method for accessing they fields, ie: public attributes, getters and setters, or whatever.

• Then the rules definitions are described in a especial class which has to be compiled by jeops, with extension .rules. This class should define all the rules, and each rules should have three parts:
  - declarations: Here you have to specify the type of the involved facts, giving a variable name to each one. This step already does matching. If there are no facts in the WM of a required fact type, the rule is not executed.
  - conditions: In this section you can specify several Java conditions, using the variables defined in the previous section. There is also a section called preconditions which can be additionally used.
  - actions: in this section you can execute actions using the following methods: retract(FactObject), assert(FactObject) and modified(FactObject), the last one should be explicit called
after you modify any relevant information of the given FactObject, calling indirectly the rete algorithm.

- And at last, you have to define the main class which should instantiate an instance of the knowledge base generated by JeOPS, and initialize the WM by creating and asserting some Facts, for finally calling the run() method for execution.

**JeOPS Example**

```java
// definition of Fact classes
public class House {
    public int id; public String color;
    public double price; public boolean forRent;
}
public class HouseAddress {
    public int id, number;
    public String street, city;
}
public class MyAddress {
    public int number;
    public String street, city;
    public String toString() {
        return ""+number", "+street", "+city;
    }
}
public class Searching {}
public class War {
    public String side1, side2;
}

// the rules knowledge base
public ruleBase HousesBase {
    rule SearchForHouse {
        declarations
            Searching searching;
            House house;
            HouseAddress houseAddress;
            MyAddress myAddress;
            War war;
        conditions
            house.forRent;
            house.color.equals("red");
            house.price<400.0;
            house.id == houseAddress.id;
            !war.side1.equals("france");
            !war.side2.equals("france");
        actions
            retract(searching);
            house.forRent=false;
            modified(house);
            myAddress.number=houseAddress.number;
            myAddress.street=houseAddress.street;
            myAddress.city=houseAddress.city;
            modified(myAddress);
    }
}

// main program, initializing the WM and running
public class TestHouses {
    public static void main(String[] args) {
        HousesBase kb = new HousesBase();
        House h1 = new House();
        h1.id=1;h1.color="red";h1.price=341.00;h1.forRent=true;
        HouseAddress ha1 = new HouseAddress();
        ha1.id=1;ha1.number=251;ha1.street="Jeanne D’Arc";ha1.city="Wancy";
        kb.assert(h1);
        kb.assert(ha1);
    }
}
```
House h2 = new House();
h2.id=2;h2.color="blue";h2.price=390.00;h2.forRent=true;
HouseAddress ha2 = new HouseAddress();
ha2.id=2;ha2.number=121;ha2.street="Avenue de Brabois";ha2.city="Villers-les-Nancy";
kb.assert(h2);
kb.assert(ha2);

House h3 = new House();
h3.id=3;h3.color="red";h3.price=415.00;h3.forRent=true;
HouseAddress ha3 = new HouseAddress();
ha3.id=3;ha3.number=31;ha3.street="Rue Carnot";ha3.city="Vandoeuvre-les-Nancy";
kb.assert(h3);
kb.assert(ha3);

MyAddress ma = new MyAddress();
ma.number=2551;ma.street="Gorbea";ma.city="Santiago";
kb.assert(ma);

Searching s = new Searching();
kb.assert(s);

War w = new War();
w.side1="USA";w.side2="Irak";
kb.assert(w);

System.out.println("i lived here:
" + ma);
kb.run();
System.out.println("and moved to:
" + ma);
}

6.5 JRules, using TRL
Using the same fact classes defined in the JeOPS example.

Working Memory Initialization

assert [ ] [ ] House [ ]
so that id = 1
and color = "red"
and price = 341
and forRent = true
assert [ ] [ ] HouseAddress [ ]
so that id = 1
and number = 251
and street = "Rue Jeanne D Arc"
and city = "Nancy"
assert [ ] [ ] House [ ]
so that id = 2
and color = "blue"
and price = 390
and forRent = true
assert [ ] [ ] HouseAddress [ ]
so that id = 2
and number = 121
and street = "Avenue de Brabois"
and city = "Villers-les-Nancy"
assert [ ] [ ] House [ ]
so that id = 3
and color = "red"
and price = 415
and forRent = true
assert [ ] [ ] HouseAddress [ ]
so that id = 3
and number = 31
and street = "Rue Carnot"
and city = "Vandoeuvre-les-Nancy"
assert [ ] [ ] MyAddress [ ]
so that number = 2551
and street = "Gorbea"
and city = "Santiago"
assert [ ] [ ] Searching [ ]
[so that]
assert [ ] [ ] War [ ]
so that side1 = "USA"
and side2 = "Irak"

Knowledge Base

WHEN
there is a [ ] Searching [ ] [ called ?f1 ]
[where]
[such that]
there is a [ ] House [ ] [ called ?f2 ]
where id is called ?id
such that color = red
and price < 400
and forRent = true
there is a [ ] HouseAddress [ ] [ called ?f3 ]
[where]
such that id = ?id
there is a [ ] MyAddress [ ] [ called ?f4 ]
[where]
such that side1 = france
or side2 = france

THEN
retract ?f1
modify [ ] ?f2
so that forRent = false
modify [ ] ?f4
so that number = ?f3.number
and street = ?f3.street
and city = ?f3.city

ELSE

6.6 ELAN

module houses
import global bool builtinInt;
end

sort Object Space
House HouseAddress MyAddress Searching War;
end

operators global
@ U @ : (Space Space) Space (AC);
empty : Space;
@ : (Object) Space;

House[h_id=@, h_color=@, h_price=@, h_forRent=@]
: (builtinInt builtinInt builtinInt builtinInt) House;
@ : (House) Object;

HouseAddress[ha_id=@, ha_number=@, ha_street=@, ha_city=@]
: (builtinInt builtinInt builtinInt builtinInt) HouseAddress;
@ : (HouseAddress) Object;

MyAddress[ma_number=@, ma_street=@, ma_city=@]
: (builtinInt builtinInt builtinInt) MyAddress;
@ : (MyAddress) Object;

Searching[] : Searching;
Méta-compilation non intrusive du filtrage par contraintes

∅ : (Searching) Object;

War[w_side1=∅, w_side2=∅] : (builtinInt builtinInt) War;
∅ : (War) Object;

go : builtInt;
result(∅) : (Space) builtinInt;
occurWar1(∅, ∅) : (Space builtinInt) bool;
occurWar2(∅, ∅) : (Space builtinInt) bool;
end

stratop global
loop : <Space> bs;
end

rules for Space
S : Space;
id1, id2, pr : builtinInt;
um, str, cit : builtinInt;
um2, str2, cit2 : builtinInt;
global
[housesearch]
S U House[h_id=id1, h_color=1, h_price=pr, h_forrent=1]
S U HouseAddress[ha_id=id2, ha_number=num, ha_street=str, ha_city=cit]
S U MyAddress[ma_number=num2, ma_street=str2, ma_city=cit2]
S U Searching[] =>

S U House[h_id=id1, h_color=1, h_price=pr, h_forrent=0]
S U HouseAddress[ha_id=id2, ha_number=num, ha_street=str, ha_city=cit]
S U MyAddress[ma_number=num, ma_street=str, ma_city=cit]
if id1 == id2
if pr < 400
if not(occursWar1(S, 1))
if not(occursWar2(S, 1))
end
end

rules for builtinInt
S : Space;
um, str, cit : builtinInt;
global
[] go => result(S)
where S:=(loop) empty

U House[h_id=1, h_color=1, h_price=341, h_forrent=1]
U HouseAddress[ha_id=1, ha_number=251, ha_street=str, ha_city=cit]
U House[h_id=2, h_color=2, h_price=390, h_forrent=1]
U HouseAddress[ha_id=2, ha_number=121, ha_street=str, ha_city=cit]
U House[h_id=3, h_color=1, h_price=415, h_forrent=1]
U HouseAddress[ha_id=3, ha_number=31, ha_street=str, ha_city=cit]
U MyAddress[ma_number=num2, ma_street=str2, ma_city=cit2]
U Searching[]
U War[w_side1=2, w_side2=3]
end

[] result(S U MyAddress[ma_number=num, ma_street=str, ma_city=cit]) => num end

rules for bool
S : Space;
n, v : builtinInt;
global
[] occursWar1(S U War[w_side1=n, w_side2=v], n) => true end
[] occursWar1(S, n) => false end
[] occursWar2(S U War[w_side1=v, w_side2=n], n) => true end
[] occursWar2(S, n) => false end
end

strategies for Space
implicit
[] loop => repeat*(first one(housesearch)) end
6.7 AC-Rewriting

6.7.1 Snyder and Schmolze’s approach

\[ P := \{ \text{false}/0, \text{true}/0, \text{not}/1, < /2, = /2/2 \} \]
\[ F := \{ \text{house}/4, \text{houseaddress}/4, \text{myaddress}/3, \text{war}/2, \text{searching}/0 \} \]
\[ W_p := \{ \]
  \[ \text{house}(1, \text{red}, 341, \text{true}), \text{houseaddress}(1, 251, "\text{Rue Jeanne D'Arc"}, \text{Nancy}), \]
  \[ \text{house}(2, \text{blue}, 390, \text{true}), \text{houseaddress}(2, 121, "\text{Avenue de Brabois"}, \text{Villers-les-Nancy}), \]
  \[ \text{house}(3, \text{red}, 415, \text{true}), \text{houseaddress}(3, 31, "\text{Rue Carnot"}, \text{Vandoeuvre-les-Nancy}), \]
  \[ \text{myaddress}(2551, \text{Gorbea}, \text{Santiago}), \text{searching}(), \text{war}(\text{usa}, \text{irak}) \]
\[ \} \]
\[ W_0 := W_p \cup \{ A | \neg A \in W_p \} \]
\[ R := \{ \]
  \[ \text{searching}(), \]
  \[ \neg \text{house}(?, \text{red}, ?, \text{pr}, \text{true}), \neg \text{house}(?, \text{red}, ?, \text{pr}, \text{false}), \]
  \[ \neg \text{houseaddress}(?, \text{num}, ?, \text{str}, ?, \text{cit}), \neg \text{myaddress}(?, \text{num}, ?, \text{str}, ?, \text{cit}), \]
  \[ \neg \text{war}(\text{france}, ?, \text{s2}), \neg \text{war}(?, \text{s1}, \text{france}) \]
\[ \rightarrow \]
  \[ \neg \text{searching}(), \]
  \[ \neg \text{house}(?, \text{red}, ?, \text{pr}, \text{true}), \text{house}(?, \text{red}, ?, \text{pr}, \text{false}), \]
  \[ \text{houseaddress}(?, \text{num}, ?, \text{str}, ?, \text{cit}), \text{myaddress}(?, \text{num}, ?, \text{str}, ?, \text{cit}), \]
  \[ \text{war}(\text{france}, ?, \text{s2}), \text{war}(?, \text{s1}, \text{france}) \]
\[ \] \[ \neg \text{war}(\text{france}, ?, \text{s2}), \neg \text{war}(?, \text{s1}, \text{france}) \]
\[ \exists \text{pr} \quad \text{pr} < 400 \}
\[ \]

6.7.2 Using matching constraints

We follow here the methodology and encoding presented in section 3.2 on page 8.

\[ \mathcal{F} := \{ \text{false}/0, \text{true}/0, \text{not}/1, < /2, = /2/2 \} \cup \{ \text{house}/4, \text{houseaddress}/4, \text{myaddress}/3, \text{war}/2, \text{C}, \text{searching}/0 \} \]
\[ W_0 := \{ \]
  \[ \text{house}(1, \text{red}, 341, \text{true}), \text{houseaddress}(1, 251, "\text{Rue Jeanne D'Arc"}, \text{Nancy}), \]
  \[ \text{house}(2, \text{blue}, 390, \text{true}), \text{houseaddress}(2, 121, "\text{Avenue de Brabois"}, \text{Villers-les-Nancy}), \]
  \[ \text{house}(3, \text{red}, 415, \text{true}), \text{houseaddress}(3, 31, "\text{Rue Carnot"}, \text{Vandoeuvre-les-Nancy}), \]
  \[ \text{myaddress}(2551, \text{Gorbea}, \text{Santiago}), \text{searching}(), \text{war}(\text{usa}, \text{irak}) \]
\[ \} \]
\[ R := \{ \]
  \[ ?P \cup \text{searching}() \]
  \[ ?P \text{ house}(?, \text{red}, ?, \text{pr}, \text{true}) \]
  \[ ?P \text{ houseaddress}(?, \text{num}, ?, \text{str}, ?, \text{cit}) \]
  \[ ?P \text{ myaddress}(?, ?, ?, ?) \]
\[ \rightarrow \]
  \[ ?P \text{ house}(?, \text{red}, ?, \text{pr}, \text{false}) \]
  \[ ?P \text{ houseaddress}(?, ?, ?, ?) \]
  \[ ?P \text{ myaddress}(?, ?, ?, ?) \]
\[ if \quad ?\text{pr} < 400 \quad \land \quad \text{war}(?, \text{s1}, \text{france}) \neq \text{P} \]
\[ \]
7 Benchmarking

In this section we benchmark several languages for production rule systems. For this, we use the fibonacci numbers sequence, encoded as following (independent of the resolution strategy):

\[
P := \{ \text{false/0, true/0, not/1, } >/2, \text{ remove/1, add/1} \} \\
F := \{ \text{fib/2} \} \\
W_0 := \{ \text{fib(0, 1), fib(1, 1), fib(200, -1)} \}
\]

\[
R := \{ \\
\quad \text{if fib(?n, -1), not fib(?n-1, ?v) then} \\
\quad \quad \text{add(fib(?n-1, -1)) end if} \\
\quad \text{if fib(?n, -1), fib(?n-1, ?v1>0), fib(?n-2, ?v2>0) then} \\
\quad \quad \text{remove(fib(?n, -1)), add(fib(?n, ?v1+?v2)), remove(fib(?n-2, ?v2)) end if} \\
\quad \text{if fib(200, ?v>0), fib(199, ?v1) then} \\
\quad \quad \text{remove(fib(199, ?v1)) end if}
\}
\]

The benchmark has been executed on a Pentium IV 1.70Ghz 256Kb L2-cache and 384 MB RAM running Mandrake 9.2, see tables 1 to 4:

<table>
<thead>
<tr>
<th>Language</th>
<th>Time [ms]</th>
<th>Rules fired</th>
<th>Rules/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clips 6.1</td>
<td>40 ms</td>
<td>398 pr</td>
<td>9925 pr/sec</td>
</tr>
<tr>
<td>JeOPS 2.1</td>
<td>215 ms</td>
<td>398 pr</td>
<td>1847 pr/sec</td>
</tr>
<tr>
<td>JRules 6.0</td>
<td>686 ms</td>
<td>398 pr</td>
<td>580 pr/sec</td>
</tr>
<tr>
<td>ELAN 3.6g</td>
<td>810 ms</td>
<td>598 rwr</td>
<td>738 rwr/sec</td>
</tr>
<tr>
<td>JTom 2.0rc2</td>
<td>311 ms</td>
<td>398 pr</td>
<td>1276 pr/sec</td>
</tr>
</tbody>
</table>

Table 1: Fibonacci(200)

<table>
<thead>
<tr>
<th>Language</th>
<th>Time [ms]</th>
<th>Rules fired</th>
<th>Rules/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clips 6.1</td>
<td>160 ms</td>
<td>798 pr</td>
<td>4988 pr/sec</td>
</tr>
<tr>
<td>JeOPS 2.1</td>
<td>482 ms</td>
<td>798 pr</td>
<td>1654 pr/sec</td>
</tr>
<tr>
<td>JRules 6.0</td>
<td>1224 ms</td>
<td>798 pr</td>
<td>651 pr/sec</td>
</tr>
<tr>
<td>ELAN 3.6g</td>
<td>6590 ms</td>
<td>1197 rwr</td>
<td>181 rwr/sec</td>
</tr>
<tr>
<td>JTom 2.0rc2</td>
<td>1457 ms</td>
<td>798 pr</td>
<td>547 pr/sec</td>
</tr>
</tbody>
</table>

Table 2: Fibonacci(400)

7.1 A more demanding benchmark

The same fibonacci sequence, but without garbage collection.
Table 3: Fibonacci(1000)

<table>
<thead>
<tr>
<th>Language</th>
<th>Time [ms]</th>
<th>Rules fired</th>
<th>Rules/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clips 6.1</td>
<td>1010 ms</td>
<td>1997 pr</td>
<td>1977 pr/sec</td>
</tr>
<tr>
<td>JeOPS 2.1</td>
<td>903 ms</td>
<td>1997 pr</td>
<td>2211 pr/sec</td>
</tr>
<tr>
<td>JRules 6.0</td>
<td>4332 ms</td>
<td>1997 pr 197⁸ pr</td>
<td>460 pr/sec</td>
</tr>
<tr>
<td>JTom 2.0rc2</td>
<td>18557 ms</td>
<td>1997 pr</td>
<td>107 pr/sec</td>
</tr>
</tbody>
</table>

Table 4: Fibonacci(10000)

<table>
<thead>
<tr>
<th>Language</th>
<th>Time [ms]</th>
<th>Rules fired</th>
<th>Rules/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clips 6.1</td>
<td>163070 ms</td>
<td>19997 pr</td>
<td>123 pr/sec</td>
</tr>
<tr>
<td>JeOPS 2.1</td>
<td>59938 ms</td>
<td>19997 pr</td>
<td>334 pr/sec</td>
</tr>
<tr>
<td>JRules 6.0</td>
<td>259022 ms</td>
<td>19997 pr</td>
<td>77 pr/sec</td>
</tr>
<tr>
<td>ELAN 3.6g</td>
<td>83780 ms</td>
<td>287 rwr</td>
<td>3 rwr/sec</td>
</tr>
</tbody>
</table>

P := { false/0, true/0, not/1, >/2, remove/1, add/1 }
F := { fib/2 }
W₀ := { fib(0, 1), fib(1, 1), fib(200, -1) }

R := {
  if
    fib(?n, -1),
    not fib(?n-1, ?v)
  then
    add(fib(?n-1, -1))
  end if

  if
    fib(?n, -1),
    fib(?n-1, ?v1>0),
    fib(?n-2, ?v2>0)
  then
    remove(fib(?n, -1)),
    add(fib(?n, ?v1+?v2)),
  end if
}

The results are:

Table 5: Fibonacci(100)

<table>
<thead>
<tr>
<th>Language</th>
<th>Time [ms]</th>
<th>Rules fired</th>
<th>Rules/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clips 6.1</td>
<td>60 ms</td>
<td>397 pr</td>
<td>6617 pr/sec</td>
</tr>
<tr>
<td>JeOPS 2.1</td>
<td>370 ms</td>
<td>397 pr</td>
<td>1588 pr/sec</td>
</tr>
<tr>
<td>JRules 6.0</td>
<td>1171 ms</td>
<td>397¹¹ pr</td>
<td>339 pr/sec</td>
</tr>
<tr>
<td>ELAN 3.6g</td>
<td>2454680 ms</td>
<td>597 rwr</td>
<td>0.24 rwr/sec</td>
</tr>
</tbody>
</table>

Table 6: Fibonacci(200)
References


