The University of Bologna Laboratory for Computer Science Research Technical Reports are available via anonymous FTP from the area ftp.cs.unibo.it:/pub/TR/UBLCS in compressed PostScript format. Abstracts are available from the same host in the directory /pub/TR/UBLCS/ABSTRACTS in plain text format. All local authors can be reached via e-mail at the address last-name@cs.unibo.it.

UBLCS Technical Report Series

93-7 Distributed Programming with Logic Tuple Spaces, by P. Ciancarini, April 1993.
93-8 Coordinating Rule-Based Software Processes with ESP, by P. Ciancarini, April 1993.
93-14 Guard Compilation in Logic Shared Dataspace Languages, by M. Garpari, June 1993.
Extending Prolog with Data Driven Rules

Mauro Gaspari

Technical Report UBLCS-94-3

February 1994

Abstract
We describe an extension to Prolog supporting data driven rules. The implementation is based on a compiler exploiting a non-state saving technique coupled with an indexing mechanism to achieve efficiency. The runtime system supports interoperability between the forward chaining language and the underlying Prolog engine.

---

1 Mauro Gaspari, Dipartimento di Matematica, Università di Bologna, Piazza Porta S. Donato, 5, Bologna, Italy, email: gaspari@cs.unibo.it, Tel: +39 (547) 642813, Fax: +39 (547) 610100.
1 Introduction

It has been recognized that "there is no single knowledge representation that is best for all problems, nor is there likely to be one" [Nec91]. For this reason, knowledge engineers require AI toolkits which can offer a wider range of representation and reasoning paradigms [Dom93] than currently supported by available knowledge engineering shells and toolkits. These more powerful toolkits are also needed to support the ‘pick and mix’ approach to KBS design [Rei86], [Sch88]. This consists of producing guidelines to help the knowledge engineer to select the right knowledge representation mechanism for a given problem, or part of it. Such guidelines will be of little practical use if suitable, hybrid knowledge engineering toolkits, providing a wide choice of representation alternatives, are not devised.

Prolog is one of the most successful AI languages since it supplies built-in mechanisms such as unification, backward chaining search, backtracking and dynamic database handling. Several AI tools have been built using Prolog such as Epsilon [Cos88], Blackshell [Bro88], and Flex [Qui91]. A common limitation of these tools is caused by the fact that Prolog does not support data driven computations; thus the performance of the provided forward chaining engines is not satisfying. Moreover is difficult to implement efficiently in Prolog standard rule matching algorithms such as rete match [For82] and to support interoperability with the underlying Prolog engine.

The term interoperability, in this context, refers to the possibility of sharing data and pass them back and forth between the two engines without the overhead of transforming data representations [Att90]. If working memory elements are stored into network nodes, it is difficult to access to them from Prolog and using them as facts in deduction; thus assertions shared between the two engines must be duplicated. A consequence of these limitations is that the user is forced to translate knowledge, which could be otherwise easily expressed using data driven rules, into a different language.

In this paper we present an extension to Prolog supporting data driven rules designed to overcome these limitations. The extension is based on an indexing technique on the content of the working memory. We exploit interoperability between forward chaining and backward chaining (Prolog) production rules, since terms in the working memory are represented as Prolog facts.

The paper is organized as follows in Section 2 we introduce forward chaining production systems and we describe our extension to Prolog. In Section 3 we present the compiler and the indexing technique, and finally in Section 4 we discuss related works and benchmark results.

2 Data Driven Rules

Forward chaining production systems such as OPS5 [For82] are composed of a set of production (if-then) rules and a database of ground assertions. Rules have the following general structure rule-name LHS --> RHS. The execution mechanism is based on the recognize-act cycle which consists of three phases:

- In the match phase the LHS of all rules is matched with the working memory (exploiting pattern matching). The instantiations of all the satisfied productions are collected into the conflict-set.

---

2 LHS and RHS means respectively Left/Right Hand Side.
In the conflict resolution phase one of the productions in the conflict set is chosen for execution.

In the act phase the RHS of the production selected in the previous phase is executed. This may change the working memory.

The execution terminates when no productions are satisfied in the match phase.

The language we propose is a simple extension to Prolog supporting production rules. Rules have the following format: \( \text{LHS then RHS} \). LHS is a sequence of tests on the working memory including Prolog goals and a set of Prolog built-in predicates. Prolog goals which appear in the LHS can not contain working memory update operations. RHS is a sequence of assert and retract operations on the working memory and of Prolog goals which may include also update operations. Retract operations appearing in the RHS always succeed: if a fact unifying the argument does not exist the working memory remain the same; this is different from the semantics of retract in Prolog.

The name and the arity of all the facts appearing in the working memory must be declared, defining the predicate \( \text{wm} \). The initial working memory is specified by Prolog facts. Facts on the working memory are not necessarily ground and to evaluate the LHS of rules we exploit unification instead of pattern matching.

The engine is activated issuing a goal \( \text{run} \) which can be included in a Prolog predicate. This starts a recognize-act cycle which terminates successfully when no rules are satisfied; \( \text{run} \) is a deterministic predicate. Another instance of the activation predicate is \( \text{run(fact)} \) which terminates successfully when \( \text{fact} \) is asserted into the WM otherwise fails\(^3\).

The default conflict resolution strategy takes the first recognized rule as the next rule to be fired. Other conflict resolution strategies are supported based on the following criteria: how recently an element has been added to the working memory, how significant is the element (a declaration \( \text{most\_significant(Name/Arity)} \) must be inserted in the program), and on the complexity of the LHS, i.e. the number of conditions which are present. Combining together these criteria it is possible to obtain more complex strategies such as the standard LEX and MEA [For81].

An example taken from [Bra86] is presented in figure 1 to illustrate the language.

---

\(^3\) In the current version if a failure occurs the WM is not restored. A correct use of this predicate can be achieved exploiting the module system of Prolog.
% a small knowledge base for locating faults in an electronic network

wm  on/1, connected/2, working/1, device/1, proved/1, samefuse/2.

device(heater). on(light1). connected(light1,fuse1).
devive(light1). on(light2). connected(light2,fuse1).
devive(light2). on(heater). connected(heater,fuse2).

on(Device),devive(Device),
not working(Device),connected(Device,Fuse),
proved(intact(Fuse))
then
assert(proved(broken(Device))).

connected(Device,Fuse),
working(Device)
then
assert(proved(intact(Fuse))).

connected(Device1,Fuse),on(Device1),
not working(Device1),samefuse(Device2,Device1),
on(Device2),not working(Device2)
then
assert(proved(failed(Fuse))).

connected(device1,Fuse),connected(Device2,Fuse),
different(Device1,Device2)
then
assert(samefuse(Device1,Device2)).

different(X,Y):-\+ (X = Y).

Figure 1: An example.

different/2 is a Prolog predicate in the LHS of rules since it does not appear in the wm declaration. (See [Bra86] for a detailed comments on this example).

3 Compilation

We exploit a non-state saving technique based on a compilative approach. Rules are translated into Prolog clauses and the evaluation of the LHS with respect to the working memory is performed evaluating the body of the clause. Since compilative techniques have been recently extensively studied for Prolog dynamic code [Clo85], [Dem89], [Lin87] and almost all the available Prolog implementations provide mechanisms to manage the dynamic data areas (garbage collection), we choose to implement working memory elements as Prolog dynamic code. As a consequence, updating operations are implemented by Prolog assert and retract built-in predicates supporting interoperability with the underlying Prolog engine i.e. facts asserted during forward chaining are also available for backward deductions and vice versa.

The rete match algorithm is based on a state-saving technique since both the conflict set and intermediate results are stored into the network.
Since rule preconditions are evaluated left to right, it may happen that a failure occurs because one of the last elements of s LHS is not present in the working memory. This is really a drawback if the number of rules and the number of conditions in the LHS of rules is large. To overcome this problem we devise an indexing mechanism which allow us to select a subset of the rules which potentially might match the working memory (the candidate rules).

Indexing is based on a bitmap representation of the working memory and of the rule preconditions. Bitwise operations are used to select candidate rules. To do this we consider an abstract representation of rules and of the working memory (WM). The abstract WM and the abstract rules are represented by sets of abstract terms. Given a term, the corresponding abstract one is represented by its name, and its arity, for instance \( a(X,Y) \) is represented as \( a/2 \). As an example the abstract WM of the working memory \( \{a(1), a(2), b(X)\} \) is the set: \( \{b/1, a/1\} \), and the abstract precondition of the LHS \( a(X), b(1), \text{not}(f(X)) \) is the set: \( \{a/1, b/1\} \). In the current version we do not exploit indexing on negated conditions.

Let \( R \) a set of rules, \( WM \) the initial working memory, and \( LHS_{i} \in R \) a rule precondition. To implement indexing we add to the Prolog representation of the working memory and rules a set of bitmaps, one (\( WMB \)) stores the presence of facts in the WM and the other \( LHSB_{i} \in R \) store the presence of conditions in rules.

Let \( AWM \) the set representing the abstract WM, and \( ALHS_{i} \in R \) the family of sets representing abstract rules. Let \( \Sigma = AWM \cup ALHS_{i} \in R \) the union of all the abstract sets\(^5\). We define a mapping \( \phi \) from \( N_{|\Sigma|} \) to \( \Sigma \) which associates an element of \( \Sigma \) to each natural number in the interval \([0,|\Sigma|]\).

The bitmaps \( WMB \) and \( LHSB_{i} \in R \) are defined in figure 2.

![Figure 2: Working Memory and Rules bitmaps](image)

We say that a rule \( i \in R \) is a candidate rule if its precondition bitmap \( LHSB_{i} \) satisfies the test: \( LHSB_{i} \land WMB = LHSB_{i} \).

As an example in figure 3 there is the Prolog code which is the result of the compilation of the example in figure 1.

---

\(^5\) Note that in order to apply this technique it must be possible to statically calculate the set \( \Sigma \).
dynamic on/1, connected/2, working/1, device/1, proved/1, samefuse/2.

device(heater).
device(light1).
device(light2).
connected(light1,fuse1).
connected(light2,fuse1).
connected(heater,fuse2).
on(light1).
on(light2).
on(heater).
p(1):-on(Device),device(Device),
    not working(Device),connected(Device,Fuse),proved(intact(Fuse)),!,
    assert(proved(broken(Device))),upd(4).
p(2):-connected(Device,Fuse),working(Device),!,
    assert(proved(intact(Fuse))),upd(4).
p(3):-connected(Device1,Fuse),on(Device1),
    not working(Device1),samefuse(Device2,Device1),
on(Device2),not working(Device2),!,
    assert(proved(failed(Fuse))),upd(4).
p(4):-connected(device1,Fuse),connected(Device2,Fuse),
different(Device1,Device2),!,
    assert(samefuse(Device1,Device2)),upd(5).

different(X,Y):-\+ (X = Y).

Figure 3:  Rules compilation.

The \textit{wm} declaration has two effects: first it define the mapping $\varphi$ which follows the order of the predicates in the WM declaration, secondly it is translated in a \textit{dynamic} declaration, to state Prolog \textit{dynamic} predicates. Rules are translated into clauses for a predicate $p/1$. We associate an index to each rule (a natural number) in order to exploit also Prolog indexing. The bitmaps generated in the translation of the previous example are presented in figure 4.

<table>
<thead>
<tr>
<th>WM:</th>
<th>110100</th>
</tr>
</thead>
<tbody>
<tr>
<td>RULE1:</td>
<td>111110</td>
</tr>
<tr>
<td>RULE2:</td>
<td>011000</td>
</tr>
<tr>
<td>RULE3:</td>
<td>110001</td>
</tr>
<tr>
<td>RULE3:</td>
<td>010000</td>
</tr>
</tbody>
</table>

Figure 4:  Bitmaps.

Indexing is performed by a C routine which applies the test to find candidate rules and execute them exploiting Prolog indexing on the first argument. Conflict resolution strategies are obtained by preprocessing rules and exploiting different instance of the \textit{assert} predicate such as \textit{asserta} or \textit{assertz}. The tool has been implemented in Sicstus$^{\text{TM}}$ Prolog using the foreign language interface to C and bitwise C operations. The size of the bitmap used depends on the cardinality of $\Sigma$; large bitmap are also supported. Assert and retract
operations on the working memory are extended with bitmap operations and may appear also in Prolog code. These operations are part of the runtime system.

4 Related Work and Performance

Shintani [Shi88] proposed a compilation technique for forward rules in Prolog which exploits indexing based on the working memory contents. The resulting knowledge engineering tool is called KORE/IE. The main difference between the Shintani and our approach is that we perform indexing on all the conditions of a rule LHS using only the functors and the arities of the conditions, while Shintani uses one working memory fact for each rule to select candidate rules. The two approaches are complementary since there are programs which may run fast in our system with respect to KORE/IE and vice versa. Our approach is more adequate for production system with large and heterogeneous LHSs while examples such as the monkey and banana [Brw85] in which the structure of the LHSs is uniform (there is almost always the same set of conditions) run fast with KORE/IE.

Furukawa and Fujita [Fur89] proposed an approach based on partial evaluation of meta-programs. They generate Prolog code for a production system exploiting the same indexing technique of Shintani.

The performance achieved by forward chaining production systems exploiting network algorithm is 400 wme (Working Memory Element) changes per second on a one Mips machine [Gup89]. This is obtained compiling the network directly into machine code. We do not expect to improve these results since our mechanism is more complex because it supports full unification.

In order to evaluate the impact of the proposed techniques we tested three different production system implementations in Prolog: a naive interpreter for production rules without any optimization, a compiler which compile productions rules into Prolog clauses and does not exploit indexing, and our compiler exploiting indexing. The benchmarks are executed on a Sun sparcstation 2. The results are summarized in the tables in figure 4.

<table>
<thead>
<tr>
<th>11 Rules</th>
<th>Interpreter</th>
<th>Compiler</th>
<th>Indexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec</td>
<td>118.2</td>
<td>31.4</td>
<td>22.4</td>
</tr>
<tr>
<td>wme change/sec</td>
<td>338.4</td>
<td>1273.8</td>
<td>1785.7</td>
</tr>
<tr>
<td>Rule fire/sec</td>
<td>84.6</td>
<td>318.4</td>
<td>446.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>22 Rules</th>
<th>Interpreter</th>
<th>Compiler</th>
<th>Indexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec</td>
<td>420.7</td>
<td>119.3</td>
<td>53.8</td>
</tr>
<tr>
<td>wme change/sec</td>
<td>190.1</td>
<td>670.4</td>
<td>1487</td>
</tr>
<tr>
<td>Rule fire/sec</td>
<td>47.5</td>
<td>167.6</td>
<td>371.7</td>
</tr>
</tbody>
</table>

Figure 4: Benchmark Results
The programs are composed respectively of 11 rules and 22 rules with 10 terms in the precondition, 4 wme updates for each rule. The execution of the programs involves respectively 10000 and 20000 rule firing. This example is similar to the example presented in [Gas93] where the guards of a rule-based parallel language are compiled exploiting the same technique; the performance obtained here are different since guards in the parallel language include also update operations. The speed-up achieved with the indexing technique is very good considering that the larger program is composed of just 22 rules. We obtained a performance improvement of 7.8 times with respect to the naive interpreter for the larger program. The program we reported in this paper correspond to the test program presented in [Shi88] since the number of candidate rules is one at every execution step. Shintani reported an execution time of 4.7 sec for one hundred rule firing on a SUN3/52m obtaining 21.27 Rule fire/sec.

Furukawa and Fujita [Fur89] reported a performance improvement of 4.7 with respect to the naive interpreter solving the Rubik’s cube example which consist of 61 rules.

Sicstus compiles in bytecode abstract instructions. The compilation in native machine instructions does not provide a great benefit: 2% for the compiler, and no benefit for the compiler + indexing. This is due to the extensive use of dynamic code and to conversion problems in the Sicstus interface with C.

5 Conclusion and Further Work

We propose an indexing technique based on the working memory to support the embedding of data driven rules into Prolog.

A number of issues are still open. First comparative benchmark with different OPS5 implementations are needed (some of them will be included in the final version of the paper). Second, it is interesting to study the impact of the presented indexing technique in the WAM [War83] code of the LHS of rules, and the possible optimizations. Finally a preprocessing phase could be introduced to allow an efficient execution of programs such as the monkey and banana which have an uniform structure of the LHS.

---

6 A careful study of these problems could provide further optimizations. This kind of applications, which exploit a lot of interaction between Prolog and C, could benefit of environments which support full interoperability between the two languages [Att91].
Bibliography


