PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

Implementing Prolog Extensions : a Parallel Inference Machine

Jean-Marc Alliot* (alliot@irit.fr)
Andreas Herzig† (herzig@irit.fr)
Mamede Lima–Marques‡ (mamede@irit.fr)

Institut de Recherche en Informatique de Toulouse
118 Route de Narbonne
31062 Toulouse cedex, France

Abstract

We present in this paper a general inference machine for building a large class of meta-interpreters. In particular, this machine is suitable for implementing extensions of Prolog with non-classical logics. We give the description of the abstract machine model and an implementation of this machine in a fast language (ADA), along with a discussion on why and how parallelism can easily increase speed, with numerical results of sequential and parallel implementation.

1 Introduction

In order to get closer to human reasoning, computer systems, and especially logic programming systems, have to deal with various concepts such as time, belief, knowledge, contexts, etc... Prolog is just what is needed to handle the Horn clause fragment of first order logic, but what about non-classical logics? Just suppose we want to represent in Prolog time, knowledge, hypotheses, or two of them at the same time; or to organize our program in modules, to have equational theories, to treat fuzzy predicates or clauses. All these cases need different ways of computing a new goal from an existing one.

Theoretical solutions have been found for each of the enumerated cases, and particular extensions of Prolog have been proposed in this sense in the literature. Examples are [BK82], [GL82], Tokio [FKTMO86], N-PROLOG [GR84], Context Extension [MP88], Templog [Bau89], Temporal Prolog [Sak89], and [Sak87].

For all these solutions it is possible to write specific meta-interpreters in Prolog that implement these non-classical systems ([SS86]). But there are disadvantages of a meta-interpreter: lower speed and compilation notoriously inefficient. If we want to go a step further, and to write proper extensions of Prolog, then the problem is that costs for that are relatively high (because for each case we will lead to write a new extension), and we are bound to specific domains: we can only do temporal reasoning, but not reasoning about knowledge (and what if we want to add modules?).

Our aim is to define a framework wherein a *superuser* can create easily "his" extension of Prolog. This framework should be as general as possible. Hence, we must provide a general methodology to implement non-classical logics.

There are four basic assumptions on which our frame is built:

1. to keep as a base the *fundamental logic programming mechanisms* that are backward chaining, depth first strategy, backtracking, and unification,
2. to *parametrize the inference step*: it is the superuser who specifies how to compute the new goal from a given one, and he specifies it in a logic form.
3. to be able to *rewrite goals*.
4. to *select clauses "by hand"*.

Points (2) and (3) postulate a more flexible way of computing goals than that of Prolog, where first a clause is selected from the program, then the Robinson unification algorithm is applied to the clause and the head of the goal, and finally a new goal is produced.

Point (4) introduces a further flexibility: the superuser may select clauses that do not unify exactly with the current goal, but just "resemble" it in some sense. Even more, if the current goal contains enough information to produce the next goal, or if we just want to simplify a goal or to reorder literals we don't need to select a fact clause at all.

The assumptions (1) and (2) were at base of the development of a meta-level inference system called MOLOG [FdC86], [ABFdC+86], [BFdCH88],

---

*Supported by the Centre d'Etudes de la Navigation Aérienne, France

†Supported by the Medlar Esprit Project

‡Supported by CAPES – Brasil

[Esp87b], [Esp87a]. The inference machine that is presented in this paper is a complete rewriting of MOLOG realizing assumption (4). It has been developped at IRIT ([Bri87] and [AG88]).

A formal specification of the inference mechanism called TIM : *Toulouse Inference Machine*, together with various examples, has been published in [BHLM91]. Here, in this paper, we present the TARSKI : *Toulouse Abstract Reasoning System for Knowledge Inference*, which is an abstract machine in which the inference mechanism can be implemented. In the preliminary version of this work nothing has been said about abstract machine and implementation, and the specifications are being defined more clearly now.

TARSKI was designed to implement parallelism (see sections 6 and 7). For example, for a given definite fact and goal clauses, more than one rule is possible. In this case it is possible to use a different processor for each rule. The parallel machine was developped and differents solutions was be done.

2 Horn clauses

The base of the language is that of Prolog. That language can (but need not) be enriched with *context operators* if one wants to mechanize non-classical logics.

Characteristically, non-classical logics possess symbols with a particular behaviour. These symbols are

- either classical connectors with modified semantics (e.g. intuitionist, minimal, relevant, paraconsistent logics)
- or new connectors called context operators (*necessary* and *possible* in modal, *knows* in epistemic, *always* in temporal, *if* in conditional logics).

Example In epistemic logics, the context operators are *knows* and *comp*, and

> *knows(a):P* means that agent *a* knows that *P*
>
> *comp(a):P* means that it is compatible with *a*'s knowledge that *P*

Hence inference engines for non-classical logics must reckon for the particular behaviour of some given symbols. These properties will be handled by built-in features of the inference engine.

The *conditio sine qua non* for logic programming languages is that they possess an implicational symbol to which a procedural sense can be given. To define a programming language it's less important if this is material implication or not, but it's rather the dynamic aspect of implication that makes the execution of a logic program possible. That is why the TIM language is built around some arrow-like symbol.

We suppose the usual definition of *terms* and *atomic formulas* of logic programming. Intuitively, *TIM Horn Clauses* are formulas built with the above connectors, such that dropping the context we may get a classical Horn clauses. Now for each logic programming language we suppose a particular set of context operators. This set depends on the logic programming language we want to implement, e.g. in epistemic logic it is {*knows, comp*} and in temporal logic it is {*always, sometimes*}. Formally we define by mutual recursion:

Definition 2. 1 - contexts

$\mathtt{m}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ is a context if $\mathtt{m}$ is a context operator $\mathtt{n} \geq 0$, and for $1 \leq \mathtt{i} \leq \mathtt{n}$ every $\mathtt{t}_\mathtt{i}$ is either a term or a definite clause.

Definition 2. 2 - goal clauses

$?P$ is a goal clause if P is an atomic formula

$?(G \wedge F)$ is a goal clause if $?G$, $?F$ are goal clauses

$?MOD : F$ is a goal clauses if $?F$ is a goal clause and MOD is a context

Definition 2. 3 - definite clauses

P is a definite clause if P is an atomic formula

$MOD : F$ is a definite clause if F is a definite clause and MOD is a context

$F \leftarrow G$ is a definite clause if F is a definite clause and G is a goal clause

Definition 2. 4 - TIM Horn clause

A TIM Horn clause (or Horn clause for short) is either a goal clause or a definite clause. Note that Horn clauses may contain several implication symbols.

We shall also use the term *Modal Horn clauses* if we are speaking of a modal logic. A set of definite clauses is called a *database*.

In the following sections we shall use the definition of the head of a Horn clause.

Definition 2. 5 - Head of a Horn clause

- H is a head of H.
- H is a head of $F \wedge G$ if H is a head of F.
- H is a head of $F \leftarrow G$ if H is a head of F.
- H is a head of $MOD : F$ if H is a head of F.

3 Writing meta-interpreters

3.1 General Mechanism

Just as in Prolog, to decide whether a given goal follows from the database essentially means to compute step by step new subgoals from given ones. In our

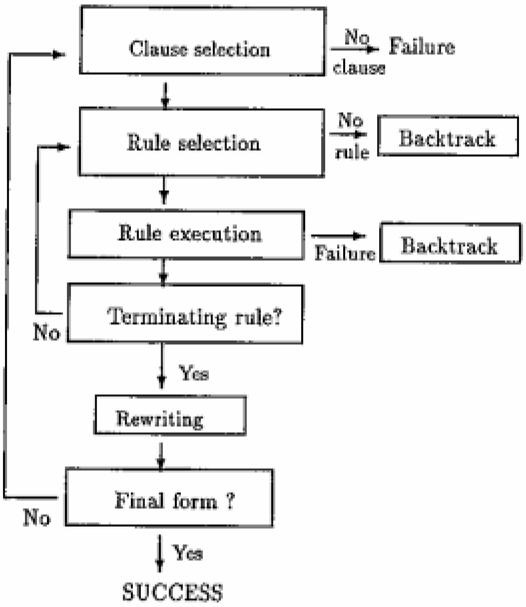

Figure 1: General mechanism of the TIM machine

case, the computation of the new subgoal is specified by the superuser. The general inference mechanism is described in figure 1. There are five steps:

Clause selection: We select a clause to solve the first sub-goal of the question.

Rule selection: We select a rule to be applied to the current clause and the current question.

Rule execution: The execution of the rule "modifies" the current clause and the current question and builds a *resolvent*.

Rewritting of the resolvent: When we reach a termination rule, we rewrite the resolvent into a new question.

End of resolution : A resolution is completed when we reach a *final form* : the goal clause *true*.

This system is doubly non determinist, because we have both a clause selection (as in standard Prolog) and a rule selection.

We are going in the next sections to explain how this mechanism can be implemented. In subsection 3.2, we will discuss rule selection and execution, in subsection 3.4 rewriting and in subsection 3.3 clause selection. In section 6, we will come back to rule selection to show how efficient mechanism can be used to improve resolution speed.

3.2 Selecting and Executing Inference Rules

An inference rule is of the form : $A, ?B \vdash ?C$ where A is a definite clause and B, C are goal clauses. It can be read: If the current goal clause unifies with B and the selected database clause unifies with A then a new goal can be inferred that is unified with C. In the style of Gentzen's sequent calculus, inference rules can be defined recursively as follows:

$$\frac{A, ?B \vdash ?C}{A', ?B' \vdash ?C'}$$

where A,A' are definite clauses and B, C, B', C' are goal clauses. As usual in metaprogramming, objects of the object language are represented by variables of the metalanguage[1].

Essentially, what can be tested here is any condition on the form of A,A', B, C, B', C', or on the existence of a database clause of a certain form. E.g. we can let an inference rule depend on the (non-)existence of some clause in some particular module of the database.

In the recursive definition the following conditions must be met[2]:

- $var(A') \subset var(A)$
- A' is a head of A or A is a head of A'
- C' is a variable
- C' is a head of C

A special category of inference rules are *reflexive rules*:

$$\frac{\text{true}, ?B \vdash ?C}{A', ?B' \vdash ?C'}$$

These rules use the special fact true. The conditions that these rules must meet are:

- A' is either:
  - a variable[3], or
  - any definite clause constructed from the variables in B and C and constants.
- C' is a variable
- C' is a head of C

Partial termination rules are written:

$$A, ?B \vdash ?C \text{ if Condition}$$

They end the recursivity in resolution.

These are some examples : the *Prolog rule for goal conjunctions*:

$$\frac{A, ?B \wedge C \vdash ?D \wedge C}{A, ?B \vdash ?D}$$

---

[1] To be correct, the real form of inference rule is a little different : a procedural condition expressed with elementary functions of the abstract machine (see section 5) can be added. This enables a more precise control over execution.

[2] It is these conditions on the form of the inference rules that warrant the efficiency of the implementation.

[3] This variable will be unified with a new fact taken in the clause base

the *Prolog rule for implications in database clauses*:

$$\frac{A \leftarrow B, ?C \vdash ?B \wedge D}{A, ?C \vdash ?D}$$

the *Prolog partial termination rule* is:

$$p, ?p \vdash ?true$$

Note that here we make use of unification. These three rules are exactly what is needed to implement Prolog.

To summarize, the execution of an inference rule modifies the current fact and the current question and constructs a resolvent. *The resolvent has the same structure than the question or any other fact.* Partial resolution is achieved when we reach a *partial termination* rule.

How rules are selected is defined by the user. We will see in the section 6 how this is exactly done. For the moment, we say that rules are taken in the order they appear in the rule base.

3.3 Rewriting the Resolvent into a New Question

As soon as we have reached a *partial termination* rule, we *rewrite* the resolvent to create the new question to solve. Rewriting is useful not only in order to simplify goals, but also in order to eliminate the *true* predicate from the new goal clause.

Rewrite rules are of the form:

$$G1 \rightsquigarrow G2$$

and allow to replace a term that is matched by G1 in the resolvent with some substitution σ by the term $(G2)\sigma$ in the new question.

For example, the *Prolog rewrite rule* is:

$$true \wedge A \rightsquigarrow A$$

In epistemic logic, the rule :

$$knows(a) : knows(a) : A \rightsquigarrow knows(a) : A$$

is a useful simplification.

3.4 Selecting Database Clauses

The user can define the way clauses are selected in the base. But this selection "by hand" must be chosen among a given set (that currently implements only two methods: classical Prolog selection and least used clause selection).

Using the abstract machine, it is possible to build another selection mechanism (for example indexing selection on the first operator) but it has not been implemented yet and it is not described in this paper.

4 Examples : Modules

In this section we are going to show how to specify modules with dynamic import. Here, any module name, such as m, $m1$, $m(2)$, etc... is considered to be a context.

Module logic
$\frac{C,?M:G \vdash ?M:NG}{C,?G \vdash ?NG}$
$\frac{M:C,?M:G \vdash ?M:NG}{C,?G \vdash ?NG}$
$true \wedge G \rightsquigarrow true$
$M:true \rightsquigarrow true$

Table 1: Rules for Module logics

The goal $m1 : m2 : G$ succeeds if G can be proved using clauses from the modules $m1$ and $m2$. The inference rules are that for Prolog, plus two supplementary rules to handle module operators (table 1).

The first rule represents the case where a module M is used to compute a new goal, and the second where another module name eventually occurring in G is used.

Others types of modules such as modules with static import or with context extension [MP88], can be specified by just adding as new inference rule. In [BHLM91], we have shown how temporal logics, hypothetical reasoning and logics of knowledge and belief can be specified elegantly in our framework.

5 The abstract machine

The goal of the TARSKI abstract machine is to bridge the gap between the description of inference rules in logical form as shown above, and the real implementation of the rule in an efficient programming language.

Compared to the WAM, the TARSKI abstract machine deals with different objects, and has a quite different goal, but on the whole, principles are identical; we will also define our machine in terms of data, stacks, registers and instructions set. We do not have enough room here to describe completely the machine. So, we shall not speak of the "classical" parts of resolution that are identical: i.e unification or backtracking. Let's say that the machine relies on classical structure sharing for unification, and on depth first search and backtracking.

Before going further, we must tell about the Great Lie. TARSKI does not use classical logic operators $\wedge$ or $\leftarrow$. For consistency and simplicity sake, all operators either modal, temporal, classical, are represented in our formalism in the same way *and are treated by the machine in the same way also.* Let's see that on an example: The logical clause written in Prolog:

$$A \leftarrow B \wedge C$$

will be written in TARSKI:

$$\wedge(C) : \wedge(B) : A$$

Here B is the *argument* of $\wedge$ and A is qualified by $\wedge(B)$. All operators have arguments, and qualify an object. For example, the S4 modal logic[4] clause:

$$\Box(X) : (\Box(a) : p \leftarrow \Diamond(a) : p)$$

will be written:

$$\Box(X) : \wedge(\Diamond(a) : p) : \Box(a) : p$$

and $\Diamond(a) : p$ is the argument of $\wedge$ that qualifies $\Box(a) : p$.

This could look like the polish reverse notation, but it is not exactly the same. In the polish reverse notation Kpq (that is $p \wedge q$) gives the same role to p and q. In $\wedge(p) : q$, p and q have really different parts to play: p is an operand of $\wedge$ and q is the object qualified by $\wedge(p)$. This destroys the symmetry of $\wedge$, but should be considered as an advantage here. In all classical Prolog, solving $p \wedge q$ is different from solving $q \wedge p$: the operator is not symmetric at all.

This formalism was not adopted lightly. The first versions did not use it, and gave a special place to the classical operators: we had a lot of problems to describe correctly the inference mechanism. Adopting this structure greatly enhanced the simplicity and the efficiency of the system.

5.1 Data structures

First of all, boolean objects (true, false) with classical operations associated (not, or, and) are implemented along with integer and floats, with their standard operations.

All data are organized in stacks. There are currently nine basic data types, and nine corresponding stacks.

The objects stack: holds all the objects on which the machine operates. An object can be either: an operator[5], a predicate[6], a variable, an integer, a float, a cons[7], *alfree*[8]. Elements of this stack will be called *ObjectElement*[9].

The operands stack: Objects do not hold their operands. Each object that has arguments holds the number of its operands and a pointer to an element of this stack that holds pointers to all the operands[10]. Elements of this stack are called *OperandElement*.

The clauses stack: Each element of this stack is composed of:

- a pointer in the object stack to the beginning of the clause
- a pointer to the head predicate[11]
- the number of free variables in the clause.

Elements of this stack are called *ClauseElement*.

The environments stack: Each element is a pair composed of a pointer to an object and a pointer in the environment stack in that the object has to be evaluated (classical structure sharing implementation). Elements of this stack are called *EnvironmentElement*.

The Trail stack: Pointers to the environment list for resetting to *free* some variables when backtracking (classical structure sharing implementation). Elements of this stack are called *TrailElement*.

The backtrack stack: Each element holds all information necessary to backtracking (values of top of stacks). Elements of this stack are called *BactrackElement*.

The question stack: Each element is a pair composed of a pointer of an object and a pointer to the environment where this object must be evaluated. The question stack holds goals to be solved. Elements of this stack are called *QuestionElement*.

The resolvent stack: stack for the resolvent elements. The resolvent is built with the current question and the current selected fact. When reaching a partial termination rule, the resolvent is re-written using rewriting rules on the top of the question and becomes the new question. Elements of this stacks are called *resolventElement*.

The predicates stack: Holds predicate structures.

There are also nine other types : pointers[12] to object in each stack, respectively *ObjectPointer, OperandPointer, ClausePointer, EnvironmentPointer, TrailPointer, BacktrackPointer, resolventPointer, QuestionPointer*.

At last, there is the *rules array*. This array describe how resolution rules behave in the system. We will come back to this later.

5.2 Registers

The registers described here are what we call *global registers* or *main registers* (see figure 2). There exists

---

[4] From now on, we will only use the S4 modal logic. A classical introduction is [HC72]. We will use the following notations : $\Box$ is *knows*, $\Diamond$ is *compatible*. Modal operators have arguments that must be constants. The new operator $\Diamond_I$ must be added to the original language as shown in ([CH88]).

[5] An operator is an object that has objects as arguments and qualify an other object.

[6] A predicate is an object that has arguments but do not qualify any other object.

[7] The classical LISP *cons*

[8] *alfree* is a special object quite similar in its behaviour to a variable that would always be free (*alfree* is the abbreviation of *always free*).

[9] Strings are currently not implemented.

[10] The operand stack is probably a technical mistake and will probably be suppressed in future versions of the machine

[11] Useful when using classical Prolog clauses selection to increase speed.

[12] We usually use the term *pointer* that is not exactly appropriate. Our *pointers* should be thought as abstract data types, that can be implemented as real pointers, or as indexes of an array, or anything similar.

Register	Description
Qcurr	Pointer to the current object in the question
FCurr	Pointer to the current object in the clause
FEnv	Pointer to the environment of the current clause
CClause	Pointer to the current clause
CRule	Index of the current rule used
TrTop	Pointer to the top of Trail Stack
ObTop	Pointer to the top of Object Stack
BTTop	Pointer to the top of Backtrack stack
Qtop	Pointer to the top of question stack
RTop	Pointer to the top of resolvent stack
EnvTop	Pointer to the top of environment stack

Figure 2: Abstract machine registers

Push(x : object) return pointer
Read(i : pointer) return object
Pull return object
Modify(x : object; i : pointer)
SetTop(i : pointer)
Position return pointer

Figure 3: Operations available on each stack

also general purpose registers that can be temporarily used for calculations. We will note them $R0, R1, \ldots$ in the following pages.

At time t, the machine is completely defined by the values of its stacks and its registers.

5.3 Instructions set

We describe here the instruction set of the abstract machine. We can not, because of lack of space, describe it extensively, but the next few lines give an intensive definitions of all instructions.

For each type of object, there are twice as many functions as there are components in the object, one for getting the value of the component and one for setting this value.

Moreover, for *each* of the nine stacks there are 6 basic operations implemented (see figure 3).

- **+(p:pointer; i:integer):pointer** Increments pointer p by i
- **-(p:pointer; i:integer):pointer** Decrements pointer p by i
- **-(p1,p2 : pointer):integer** Returns the number of elements between $p1$ and $p2$.

There are also some classical functions: **Assignment, Equality test, Conditional constructions.**

This ends the description of atomic functions. We will need in the following lines the classical macro-instruction **unify**, that unifies (Struct1, Env1) with (Struct2, Env2)[13].

Let's see on an example how the abstract machine code is used to implement rules[14] :

$$\frac{\Box(X) : A, ?\Box(X) : B \vdash ?\Box(X) : C}{\Box(X) : A, ?B \vdash ?C}$$

is translated into:

```
RO:=Read(Qcurr)
if not
 unify(Fcurr,Fenv,GetNumStruct(RO),GetNumEnv(RO)
 then return false
 else Pushresolvent(RO) endif
Qcurr := Qcurr+1
return true
```

6 Rule selection with parallelism

In section 3.4, we said that resolution rules were chosen in the rules base in order of appearance. We are going to show here that this mechanism can be greatly enhanced by indexing the rules base and using parallel execution of rules.

6.1 Indexation of rules

The rules necessary to implement S4 are shown on top of table 2.

Remember that due to the uniform notation of the abstract machine the clause $\wedge(A) : B$ of the second rule is in fact the implication $B \leftarrow A$. We can see that, for a given fact and a given question, we have to try a lot of different rules. This creates a second non-determinism that greatly slows down the implementation of the language.

But trying all rules is usually not useful, because for a given fact and a given question, only a few rules will match the shape of the fact and the shape of the question. For example, if the fact is $\Box(X) : A$ and the question $\Diamond_I(X, I) : B$ only rules 9 and 11 can be used.

So, for a given logic, we can develop extensively all possible cases. For S4, this gives table 2. This way, given a fact and a question, the array gives directly the rules that can be applied *and there is often only one rule that can be applied.* This transforms the double non-determinism in an almost simple non-determinism much closer to Prolog complexity. So, in a large number of cases, it is not necessary to backtrack on rule selection.

---

[13] **unify** is of course written with atomic instructions.

[14] Other examples can be found in [Alled]: full implementation of S4 logic, among others (Fuzzy logic, module logic).

Type	*Number*	*Form*
Rule	1	$p, ?p \vdash ?true$
Rule	2	$\frac{\wedge(A):B,?C\vdash?\wedge(A):D}{B,?C\vdash?D}$
Rule	3	$\frac{B,?\wedge(A):C\vdash?\wedge(A):D}{B,?C\vdash?D}$
Rule	4	$\frac{A,?\Diamond(X):B\vdash?C}{A,?B\vdash?C}$
Rule	5	$\frac{\Diamond_I(X,I):A,?\Diamond(X):B\vdash?\Diamond_I(X,I):C}{A,?\Diamond(X):B\vdash?C}$
Rule	6	$\frac{\Diamond_I(X,I):A,?\Diamond_I(X,I):B\vdash?\Diamond_I(X,I):C}{A,?B\vdash?C}$
Rule	7	$\frac{\Box(X):A,?\Box(X):B\vdash?\Box(X):C}{\Box(X):A,?B\vdash?C}$
Rule	8	$\frac{\Box(X):A,?\Diamond(X):B\vdash?\Diamond(X):C}{\Box(X):A,?B\vdash?C}$
Rule	9	$\frac{\Box(X):A,?\Diamond_I(X,I):B\vdash?\Diamond_I(X,I):C}{\Box(X):A,?B\vdash?C}$
Rule	10	$\frac{\Box(X):A,?\Diamond(X):B\vdash?\Diamond(X):C}{A,?\Diamond(X):B\vdash?C}$
Rule	11	$\frac{\Box(X):A,?B\vdash?C}{A,?B\vdash?C}$

Fact	*Question*	*Usable rules*
Pred	Pred	$p, ?p \vdash ?true$
Pred	$\wedge$	$\frac{A,?\wedge(X):B\vdash?\wedge(X):C}{A,?B\vdash?C}$
Pred	$\Diamond$	$\frac{A,?\Diamond(X):B\vdash?C}{A,?B\vdash?C}$
$\wedge$	Pred	$\frac{\wedge(X):A,?B\vdash?\wedge(X):C}{A,?B\vdash?C}$
$\wedge$	$\wedge$	$\frac{\wedge(X):A,?\wedge(Y):B\vdash?\wedge(Y):C}{\wedge(X):A,?B\vdash?C}$
$\wedge$	$\Diamond$	$\frac{\wedge(X):A,?\Diamond(Y):B\vdash?\wedge(X):C}{A,?\Diamond(Y):B\vdash?C}$
$\wedge$	$\Box$	$\frac{\wedge(X):A,?\Box(Y):B\vdash?\wedge(X):C}{A,?\Box(Y):B\vdash?C}$
$\wedge$	$\Diamond_I$	$\frac{\wedge(X):A,?\Diamond_I(Y,I):B\vdash?\wedge(X):C}{A,?\Diamond_I(Y,I):B\vdash?C}$
$\Box$	Pred	$\frac{\Box(X):A,?B\vdash?C}{A,?B\vdash?C}$
$\Box$	$\wedge$	$\frac{\Box(Y):A,?\wedge(X):B\vdash?\wedge(X):C}{\Box(Y):A,?B\vdash?C}$
$\Box$	$\Diamond$	$\frac{\Box(X):A,?\Diamond(Y):B\vdash?C}{A,?\Diamond(Y):B\vdash?C}$ $\frac{\Box(Y):A,?\Diamond(X):B\vdash?C}{\Box(Y):A,?B\vdash?C}$ $\frac{\Box(X):A,?\Diamond(X):B\vdash?\Diamond(X):C}{A,?\Diamond(X):B\vdash?C}$ $\frac{\Box(X):A,?\Diamond(X):B\vdash?\Diamond(X):C}{\Box(X):A,?B\vdash?C}$
$\Box$	$\Diamond_I$	$\frac{\Box(X):A,?\Diamond_I(Y,I):B\vdash?C}{A,?\Diamond_I(Y,I):B\vdash?C}$ $\frac{\Box(X):A,?\Diamond_I(X,I):B\vdash?\Diamond_I(X,I):C}{\Box(X):A,?B\vdash?C}$
$\Diamond_I$	$\wedge$	$\frac{\Diamond_I(Y):A,?\wedge(X):B\vdash?\wedge(X):C}{\Diamond_I(Y):A,?B\vdash?C}$
$\Diamond_I$	$\Diamond$	$\frac{\Diamond_I(Y):A,?\Diamond(X):B\vdash?C}{\Diamond_I(Y):A,?B\vdash?C}$ $\frac{\Diamond_I(X,I):A,?\Diamond(X):B\vdash?\Diamond_I(X,I):C}{A,?\Diamond(X):B\vdash?C}$
$\Diamond_I$	$\Diamond_I$	$\frac{\Diamond_I(X,I):A,?\Diamond_I(X,I):B\vdash?\Diamond_I(X,I):C}{A,?B\vdash?C}$

Table 2: S4 logic rules and their exhaustive development

	Fact	Question	Rules
R1	$\Box$	$\Diamond$	$\frac{\Box(X):A,?\Diamond(X):B\vdash?C}{A,?\Diamond(X):B\vdash?C}$
R2			$\frac{\Box(X):A,?\Diamond(X):B\vdash?C}{\Box(X):A,?B\vdash?C}$
R3			$\frac{\Box(X):A,?\Diamond(X):B\vdash?\Diamond(X):C}{A,?\Diamond(X):B\vdash?C}$
R4			$\frac{\Box(X):A,?\Diamond(X):B\vdash?\Diamond(X):C}{\Box(X):A,?B\vdash?C}$

Table 3: Rules $\Box$ against $\Diamond$

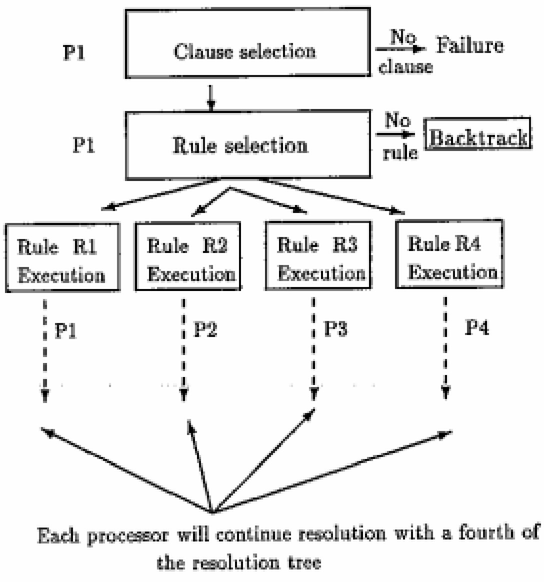

Figure 4: Parallel execution of S4 rules

6.2 Parallel rule execution

The abstract machine was designed to enable an easy implementation of parallelism. Sometimes, for a given definite fact and a given goal clause, more than one rule is possible : we can use a different processor for each rule. For example, in the S4 logic, if the fact is $\Box(X) : A$ and the question is $\Diamond(X) : B$, four rules can be used (table 3). With four processors, each one can continue the resolution with a different rule. Figure 4 shows how the inference system running originally on processor P1. With four processors P1, P2, P3, P4 available, it is possible to solve, in parallel, S4 rules described in table 3.

The information transferred from one processor (P1) to its children (P2, P3, P4) are *the abstract machine data stacks* and *the abstract machine registers*. Some stacks are never transferred (the backtrack stack, the trail stack) because the child does not need to backtrack over the current resolution point. This parallelism induces no side effects : as soon as one processor has received data, it will not have to communicate

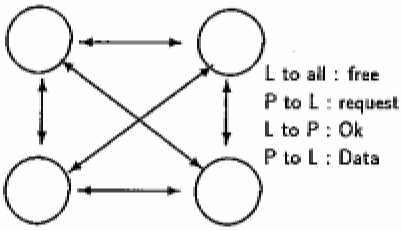

Figure 5: Fully interconnected network

with its parent any more until it has finished its own resolution. Moreover, there is no overhead in processing time because parallelism is explicit in the language itself : overhead comes only from communication between processes.

Four models (Master/slaves network, fully interconnected networks, ring networks, top-down networks) are under development; we just mention them and we will not discuss them in detail[15].

Fully interconnected network: Every processor can distribute work to any other processor that is free. A very simple protocol is used to prevent two processors to send at the same time data to the same processor (figure 5). This protocol will solve problems as represented in figure 4.

Master/slaves network: The master process distributes work to all other processes, which, in turn, can not distribute any work. This protocol will also solve problems as represented in figure 4.

Ring network: Here each processor can send work to the next one, and the last processor can send work to the first.

Top-Down network : In the Top-Down Network, each processor can only send information to the following one but the last processor can't send information to the first one. In ring networks and top-down networks, resolution is not exactly as represented in figure 4.

7 Implementing Parallelism

7.1 The "classical" machine

The new abstract machine specifications was the result that began with the first implementation of MOLOG, in C, in 1988.

Coding the new machine took less than two months. Of course, two years spent in coding other abstract machines (that proved to be unsatisfactory) helped a lot. From the beginning, the stress was on getting a program as close as possible to the specifications of the abstract machine. That was the reason why the ADA language has been chosen: the specifications of the abstract machine *are* exactly the specifications of the main package of the implementation. Moreover, compared to other implementations previously written in C, coding and debugging was a lot easier and faster. We wanted also to be able to easily implement parallelism. So, for example, stacks are implemented with arrays and there is not a single real pointer in the system, only indexes. It has an interesting well known side effect: we never run out of stack space, because if a stack becomes full, we just have to copy it to a new larger stack. All indexes are still valid. The mechanism is invisible to the programmer and the user and very useful with some very recursive non-classical problems.

This was done at the loss of performance. Accessing any object in a stack requires two function calls and three tests plus the classical indirection. The TARSKI machine runs about fifteen times slower than C-Prolog[16] on PROLOG problems. This could easily be enhanced by recoding the machine with efficiency in mind.

Coding a logic is very easy as soon as it follows the general framework given in section 3.2. The S4 logic was implemented in *one* day. and tested with the classical "wise men" puzzle. The puzzle is solved in three minutes on a HP-720 workstation *with the full amount of knowledge* (more than twenty clauses). With only the five clauses necessary to solve the problem, the solution is found in less than a second, hundred times faster than the MOLOG interpreter.

7.2 The parallel machine

The parallel machine was developped with an ETHERNET network as medium for data transfer. The parallel system is made of many TARSKI machines running on different workstations, linked by INTERNET sockets[17]. The only configuration tested was a top-down network. Results are shown in table 4. It would be too long to discuss them here in detail. Full explanations can be found in [Alled].

We can briefly say that, over three processors, the network is clearly too slow and becomes the bottleneck of the system. A large part of time is lost in communicating with other processors. There are different solutions that could be used to enhance performances:

- We can use parallelism only for branches that are

---

[15] On all practical implementations issues, details can be found in [Alled].

[16] It is however faster than some classical PROLOG written in compiled Common Lisp

[17] It was quite easy to do, because all necessary packages for communication and parallelism had been developped previously for other projects. Reusability of software is a major advantage of ADA.

# of Procs	P1	P2	P3	P4
1	319+1			
2	166+10	145+6		
3	129+24	142+50	77+17	
4	129+26	140+46	46+31	22+9

Table 4: CPU+system time used

close to the root of the tree. This will decrease the number of sent packets.

- We can try a master/slave network. The master processor will be almost devoted to sending packets but slaves would not spare time on this.
- We can improve the amount of sent data; some stacks can only grow, and are never modified under a certain depth. We could only send new data, and not the whole stack.
- We could try to use a different medium. An ethernet network is a very slow device for parallelism, and, moreover, our network is usually crowded with packets coming from other stations or other X-terminals. It would be very interesting to implement the machine on a multi-processor computer with shared memory segments, or on a transputers network. We were not able to do it yet because we lack access to such a machine. We are very eager to try such an approach. If we are able to find a machine with many processors, the inference machine could be almost as fast as a standard PROLOG even when solving non-classical logic problems, because the double non-determinism would be almost reduced to classical PROLOG non-determinism.

8 Conclusion

We think the implementation of any logic given by inference rules of the form defined in the earlier sections can be done in a very short amount of time (one or two days at most). The development of an automatic translator from the logical shape of the rules to the abstract machine specifications suggests itself and is a subject of current work.

Now, it is hoped that fast, general and efficient implementations of such logics could bring a new area of development for expert systems. In particular, in the C.E.N.A.[18] a large expert system (3,000 rules) using fuzzy and temporal logics has been developped in Prolog ([AL91]). This expert systems could be an excellent test for TARSKI.

9 Acknowledgements

We wish to thank Luis Fariñas Del Cerro for valuable discussions.

References

[ABFdC$^+$86] R. Arthaud, P. Bieber, L. Fariñas del Cerro, J. Henry, and A. Herzig. Automated modal reasoning. In *Proc. of the Int. Conf. on Information Processing and Management of Uncertainty in Knowledge-Based Systems*, Paris, july 1986.

[AG88] J. M. Alliot and J. Garmendia. *Une Implementation en "C" de MOLOG.* Rapport D.E.A, Université Paul Sabatier, Toulouse, France, 1988.

[AL91] Jean-Marc Alliot and Marcel Leroux. En route air traffic organizer, un système expert pour le contrôle du trafic aérien. In *Proceedings of the International Conference on Expert systems and their applications*, Avignon, May 1991.

[Alled] Jean-Marc Alliot. Tarski: une machine parallèle pour l'implémentation d'extensions de prolog. Master's thesis, Université Paul Sabatier, To be published.

[Bau89] Marianne Baudinet. *Logic Programming Semantics: Techniques and Applications.* PhD thesis, Stanford University, feb 1989.

[BFdCH88] P. Bieber, L. Fariñas del Cerro, and A. Herzig. MOLOG – a modal PROLOG. In E. Lusk and R. Overbeek, editors, *Proc. of the 9th Int. Conf. on Automated Deduction*, LNCS 310, pages 487–499, Argonne – USA, may 1988. Springer Verlag.

[BHLM91] P. Balbiani, A. Herzig, and M. Lima-Marques. TIM: The Toulouse Inference Machine for non-classical logic programming. In M.M. Richter and H. Boley, editors, *Processing Declarative Knowledge*, number 567 in Lecture Notes in Artificial Intelligence, pages 365–382. Springe-Verlag, 1991.

[BK82] K. A. Bowen and R. A. Kowalski. Amalgamating language and metalanguage in logic programming. In K. Clark and S. Tarnlund, editors, *Logic Programming*, pages 153–172. Academic Press, 1982.

[Bri87] M. Bricard. Une machine abstraite pour compiler MOLOG. Rapport D.E.A., Université Paul Sabatier – LSI, 1987.

[CH88] Luis Fariñas Del Cerro and Andreas Herzig. Linear modal deductions. In E. Lusk and R. Overbeek, editors, *Proc. of the 9th Int. Conf. on Automated Deduction Computer Systems.* Springer-Verlag, 1988.

[Esp87a] Esprit Project p973 "ALPES". *MOLOG Technical Report*, may 1987. Esprit Technical Report.

[Esp87b] Esprit Project p973 "ALPES". *MOLOG User Manual*, may 1987. Esprit Technical Report.

[FdC86] L. Fariñas del Cerro. MOLOG: A system that extends PROLOG with modal logic. *New Generation Computing*, 4:35–50, 1986.

---

[18] The CENA is an institution responsible for studies of new systems for Air Traffic Control in France

[FKTMO86] M. Fujita, S. Kono, H. Tanaka, and T. Moto-Oka. Tokio: Logic programming language based on temporal logic and its compilation to prolog. In *Third Int. Conf. on Logic Programming*, pages 695–709, jul 1986.

[GL82] M. Gallaire and C. Lasserre. Meta-level control for logic programs. In K. Clark and S. Tarnlund, editors, *Logic Programming*, pages 173–188. Academic Press, 1982.

[GR84] D. Gabbay and U. Reyle. N-prolog: An extension of prolog with hypothetical implications. *Jounal of Logic Programming*, 1:319–355, 1984.

[HC72] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logics.* Methuen & Co. Ltd, USA, 2 edition, 1972.

[MP88] Luis Monteiro and Antonio Porto. Modules for logic programming based on context extension. In *Int. Conf. on Logic Programming*, 1988.

[Sak87] Y Sakakibara. Programming in modal logic: An extension of PROLOG based on modal logic. In *Int. Conf. on Logic Programming*, 1987.

[Sak89] Takashi Sakuragawa. Temporal PROLOG. In *RIMS Conf. on software science and engineering*, 1989.

[SS86] L Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, USA, 1986.