MASTER IN COMPUTER SCIENCE

A Blackboard Approach to Concurrent Constraint Programming

Pires da Costa, João Manuel

Award date:
1996

Awarding institution:
University of Namur

Erasmus Stage Thesis  in Logic Programming

presented at

Facultés Universitaires Notre-Dame de la Paix
Namur - Belgium

A Blackboard Approach to Concurrent Constraint Programming

---

by

João Manuel Pires da Costa

in the academical year

95 / 96

supervised by

Jean-Marie Jacquet

Table of Contents

Table of Figures

Tables

Table of Examples

Part I

Preface

It's always hard to put on paper ones experience in a foreign country doing something new and exciting as logic programming. My previous interest in this particular field was somewhat limited and through this *ERASMUS* stage, I have gained the sensibility to the problems and possibilities offered by this dynamic research area.

Despite the fact that almost all the written code was in the C language, the study of the WAM allows me to better understand the process of logic programming, which I believe is an add-on for the future.

Abstract

This thesis is about a client server application using the concepts of logic, constraint and concurrency. Its main goal is to built a framework to exchange data using blackboards in concurrent constraint programming.

Organization of the Thesis

The thesis is devised in four logical parts:

- ✧ Introduction, states the author forewords about the thesis;
- ✧ Background, where an insight is given into the foundations of Logic Programming, Constraint Logic Programming and Concurrent Programming;
- ✧ Work Report, states the work done during the stage;
- ✧ Summary, where the conclusion, bibliography, index and other information of the sort is putted.

The thesis is also organized by chapters:

- Chapter One, introduces the thesis and it's objectives;
- Chapter Two, gives a background on Logic Programming;
- Chapter Three, Constraint Logic Programming is explained;
- Chapter Four, is about Concurrent Programming;
- Chapter Five, states the work done by the author;
- Chapter Six, draws conclusions.

Acknowledgments

The author would like to thank those people who, directly or indirectly, made this work possible.

I am particularly grateful to Jean-Marie Jacquet, my thesis supervisor, who patiently answered so many endless questions.

I want to thank Koenraad De Bosschere who give me some valuable information about the WAM.

I also would like to acknowledge the F.U.N.D.P.[1] where this work was developed.

[1] Facultés Universitaires Notre-Dame de la Paix

Chapter 1 -- Introduction

1.1 Introduction

One of the interesting features of logic programming is probably its appeal to first-order logic. Hence in contrast with imperative languages such as Pascal, C or Fortran, the programmer is not faced with assignments and loops but with logic formulae. This declarative appeal turns the programming task in specifying the problem to be solved as opposed to expliciting how to solve it. This has lead to an easy yet efficient way of solving many complex problems.

A generalization of unification to constraint solving has pushed this idea further, with the result that nowadays constraint programming is employed in the industry to solve such problems as scheduling for factories and for computer instructions sets; options and portfolio analysis, modeling water usage, DNA and electrical circuits analysis.

The development of parallel architectures for computers and networks of computers have evidenced, if need still be, the interest of parallel computations. One natural next step is thus to parallelize constraint logic programming. Our thesis takes place in this context; we shall indeed propose a new model based on blackboards.

Understanding the differents concepts used in the thesis is important to better understand the work performed during the stage. The background knowledge is given in part II and is divided in chapters in which logic programming, constraint programming and concurrent programming are explained.

The chapter dedicated to logic programming presents its three components: syntax, operational semantics and declarative semantics. Syntax introduces the reader to the construction of well formed formulas and to the heart of logic programming - the unification mechanism. Operational semantics presents how formulas are processed, the model of execution of programs and the possible states. Declarative semantics deals with true values, with models and logical consequences in regard to formulas. The implementation section was written to explain the breakthrough that occurred in LP (Prolog) - the definition of the Warren's Abstract Machine.

The constraint logic programming chapter offers a view of this very active field. The main idea of CLP is to replace the unification mechanism of LP with a solver of constraint in a particular domain. The solver can be seen as a "black box" responsible to test the satisfiability of the constraints and to possibly reduce them to a normal form. A different approach is presented there called "glass box", which uses a constraint $X \; in \; r$ over finite domains.

Parallel systems requires concurrent programming which is the subject discussed in chapter four. The chapter presents a new interesting framework (μLog) using logic programming to achieve concurrency through blackboards. Using blackboards avoids problems like for instance the synchronization and mutual exclusion. Two categories of objects are described as well as the operations that can be performed on them. Processes are also differentiated.

1.2 Contributions

The differents concepts explained in part II are putted into practice in part III through the development of a client server application using blackboards. In chapter five, the application development steps are stated.

The inclusion in a program of the client functions created, makes it a client program capable of communicate with the server. To exemplify the use of clients functions on different programming languages, the Wamcc(fd) is changed. Moreover functions interfacing the code generated by Wamcc(fd) for a given program and the functions on the client side, are added.

To allow a better debugging of any errors occurring during the execution of the daemons server, the system logger is used.

An attempt was made to make the code as much as possible POSIX 1 compliant, which gives us some guaranties that it will run on most machines. Making the code more portable has its backsides, namely when a FIFO [1] was chosen instead of a more efficient solution (for instance shared memory which would speed up the process of reading / writing the blackboard).

Finally, for ending the continuous reading in the several fifos either by the server and client, a signal solution was found so that when a client has send a request (or the answer by the server) a signal is sent to make the corresponding sleepy process active.

1.3 Conventions

The following symbols are used throughout the thesis to denote a particular class of textual expression (i. e. definition, proposition, question, enumeration) :

❶,❷,...,①,②,...,	The first symbols denotes an enumeration of items, if there is a sub-numeration than the seconds are used;
✧,▪	When there is no need for an enumeration of items, this symbol is used.

[1] Also called *named pipes*.

- • Denotes a question;
- ✓ Denotes a definition or a proposition;
- □ Used in the beginning of definitions, propositions and questions;
- 🕮 Introduces a note, a remark or a comment.
- ☒ Gives a warning, when a danger of any sort occurs;
- T Capital T - Indicates a theorem

The use of an expression in italic has the purpose of making clear its use.

1.4 Credits

The author acknowledge the sources for the chapters in the thesis, as the following:

- ✧ Chapter two - the authors and books referenced in [4], [5], [6] ;
- ✧ Chapter three - [4], [10], [11], [12] ;
- ✧ Chapter four - [7] ;
- ✧ Chapter five - [2], [3], [10].

All the numbered references belong to the bibliography.

Part II

Chapter 2 -- Logic Programming

This chapter is dedicated to the introduction of logic programming to provide background knowledge to understand in what context this work was achieved. Logic programming has three components: syntax, operational semantics and declarative semantics.

2.1 Syntax

Syntax specifies the well formed formulae.

Logic programming uses first-order logic, but from a programmer point of view this means first-order formulae. These are built from variables and from atomic formulae. Variables must be quantified either existentially or universally (using the quantifier $\exists$ or $\forall$, respectively). An atomic formulae is composed of an atom.

Example 1

$$\forall X\ \forall Y\ (\text{grand\_son}(X,Y) \Leftrightarrow (\ \exists Z\ (\text{son}(X,Z) \wedge (\text{son}(Z,Y))))$$

X,Y,Z are variables;
grand_son(X,Y), son(_,_) are atoms;
grand_son, son are predicates.

Atoms may be linked together by logical connectives,

¬ (negation), ∧ (conjunction), ∨ (disjunction), ⇒ and ⇐ (implication) and ⇔ (equivalence).

For grouping formulas we can use the quantifiers (∀,∃) and brackets, but to avoid their intricate use, an association rule is used, with the usual left to right and top to bottom precedence.

¬ , ∀ , ∃
∨
∧
⇐ , ⇒ , ⇔

Table 1 : Precedence Operators

After introducing the logical connectives and quantifiers the definition of the well formed formulae is the next step.

✓ Well formed formulae **1**

❶ Well formed formulae or wff, for short, are inductively defined as follow:

① atomic formulae are wff's;

② if F and G are wff's, then ¬F, (F), F∧G, F∨G, F⇐G, F⇒G, F⇔G are wff 's;

③ if X is a variable and F is a wff, then ∀X(F) and ∃X(F) are wff's. The scope of ∀X and ∃X in the above formulae is the wff F. An occurrence of X is said to be bound if it is under the scope of a quantifier; otherwise it's free.

❷ A closed wff is a wff with no free occurrences of any variable. For notation convenience, we shall use ∀(F) and ∃(F) to denote the universal closure of F and the existential closure of F, respectively, that is the wff obtained by respectively adding a universal or an existential quantifier for every variable having a free occurrence in F.

Example 2

Well formed formulae

$$\forall X(p(X,Z)) \Leftrightarrow (q(Y) \wedge a)$$

Using the Edinburgh Syntax several concepts are now explained.

✓ **Variables** **2**

- What is the composition of variables ?
  - Letters, digits and the special symbol "_".
- How to recognize a variable ?
  - Variables start with an upper case letter or with the special symbol "_".

✓ **Predicates and functions** **3**

- What is the composition of functions and predicate symbols ?
  - Letters, digits and the special symbol "_".
- How to recognize a predicate or a function ?
  - They do not begin neither with an upper case letter nor with the symbol "_".

✓ **Constants** **4**

- Constants are functions of arity 0.

✓ **Terms** **5**

- Variables and constants are terms;
- If the arity of a function f[1] is greater than zero and $t_1,...,t_n$ are terms then $f(t_1,...,t_n)$ is a term.

✓ **Atoms** **6**

- An atomic formula or atom, is an expression of the form $p^1(t_1,..., t_n)$ where $t_1,...,t_n$ are terms and p is a predicate symbol.

Example 3

Here are some examples of the concepts just defined.

Examples	*Type*
Belgium, _abc, Bx15	Variables
namur, pc486, 1111	Constants
f(h(h(Y))), [H|T][2]	Term
greater(X,Y),q	Atom

Table 2 : Examples

[1] The symbols f,p are called the functor of the term and atom, respectively.

[2] The reason why this is a term is that, $[t_1,t_2]$ is in fact $\bullet(t_1,t_2)$, being $\bullet$ the list constructor (function) with arity 2.

The atoms and their negation are important in logic programming as we shall see below.

✓ **Literal** **7**

- A literal is an atom or the negation of an atom. The first is called positive literal and the last negative literal.

Full first-order logic is limited to wff composed of definitive and normal clauses for efficiency purposes. The following definitions are first required.

✓ **Clauses** **8**

- A clause is a wff of the form $\forall X_1 \ldots X_n (L_1 \vee \ldots \vee L_m)$, where L_i is a literal and $X_1,\ldots,X_n$ are all variables.

Example 4

$$\forall X \forall Y \, (\text{odd}(X) \vee \text{even}(Y))$$

✓ **Horn Clauses** **9**

- A Horn clause is a clause with at most one positive literal.

Example 5

$$\forall X \forall Y (\neg \text{even}(X) \vee \neg \text{odd}(Y))^1$$

✓ **Definite Clauses** **10**

- A definite clause is a clause containing exactly one positive literal.
  They are normally denoted as follows in the view of the equivalence between the formula $\neg A \vee B$ and $A \Rightarrow B$:

$$H \leftarrow B_1,\ldots,B_n$$

  H is called the head of the clause and $B_1,\ldots,B_n$ the body of the clause.
  Note that the atom H is the positive literal of the clause and the atoms $B_1,\ldots,B_n$ are the conjunction of the positive literals associated with the negatives literals of the clause.

Example 6

$$\text{odd}(3) \leftarrow \text{even}(2), \text{even}(0)$$

[1] In this particular case there is no positive literal.

Definite clauses having only the atom H are also called facts.

✓ **Normal Clauses** **11**

- Normal clauses are generalizations of definite Horn clauses where a negative literal occur in the body of the clause. They are denoted as above but now with $B_1,...,B_n$ being arbitrary literals, and any negative literal $\neg L$ being written as not(L).

Example 7

odd(3) ←even(2), not(even(3))

Procedures and programs are formed from definite and normal clauses.

✓ **Definite (normal) procedure** **12**

- Finite set of definite (resp. normal) clauses with same head functor and same head arity.

✓ **Definite (normal) program** **13**

- Finite set of definite (resp. normal) procedures.

✓ **Logic Programs** **14**

- Programs that are either definite or normal are called logic programs. If there is no ambiguity in their nature they are simply called programs.

List of literals considered as conjunctions are represented by $\leftarrow L_1,...,L_m$. The empty conjunction is denoted by □.
Conjunctions of both kinds are called goals.

Any computation in logic programming consists essentially of proving that the variables of a given goal can be replaced by terms such, that the goal becomes a logical consequence of a given logic program and, of actually delivering only the most general terms to variables. To that end, the computation attempts to progressively transform the given goal to empty goal □. Each step of the transformation consists of substituting the variables of the current goal and of a program clause by terms such that the head of the clause identifies with an atom of the goal. In this case a atom is replaced by the body of the clause.

Example 8

Goal: ← a,b,c.

Clause: b←d,f,g.

Result 1: ← a,d,f,g,c.
...

The substitutions are the classical way of reporting values for variables.
The process of identifying an atom and the head of a clause is called unification.

Substitutions

Substitutions may be presented in three ways:

❶ Set of bindings

❷ Functions

❸ Solutions of Equations

The first approach is the usual one in the logic programming community, so it is also adopted here.

✓ **Substitution** **15**

▫ A substitution is a (possibly empty) finite set of the form $\{X_1/t_1,...,X_m/t_m\}$ where $X_1,...,X_m$ are distinct variables and $t_1,...,t_m$ are terms respectively distinct from their corresponding variables. Each element X_i/t_i is called a *binding* or an *instantiation* for X_i. The empty set of bindings is called the *identity* substitution[1].

[1] Represented by the letter ε. Substitutions are typically denoted by Greek letters.

The set of variables appearing in a term, a literal, a wff or more generally in any construct C is represented by var(C). The set of variables of a substitution $\theta = \{X_1/t_1,...,X_m/t_m\}$ is referred by dom(θ); the set of terms of the substitution θ, $\{t_1,...,t_m\}$ is named cod(θ). The set of variables appearing in $t_1,...,t_m$ is designated by varcod(θ).

In the computation process, generally, auxiliary variables are calculated but whose values are irrelevant to the final result; so a restriction of substitutions to the given set of variables must be defined.

✓ **Restriction of Substitutions** **16**

- The restriction of substitution $\theta = \{ X_1/t_1,...,X_m/t_m\}$ to the set of variables S, denoted by $\theta_{|S}$ is the substitution obtained from θ by deleting all the bindings X_i/t_i for which $X_i \notin S$.

Substitutions take their essential meaning through their application to expressions.

✓ **Expression** **17**

- An expression is a term, a literal, a disjunction of literals or a conjunction of literals.

✓ **Instance** **18**

- Let $\theta = \{ X_1/t_1,...,X_m/t_m\}$ be a substitution and E an expression.
  The construct Eθ denotes the expression obtained from E by simultaneously replacing in E each occurrence of the variable X_i by the corresponding term t_i ($1 \le i \le n$). It is called an *instance* of E.
  This situation is also referred to as θ being applied to E.
  After such an operation, X_i is said to be bound or instantiated to t_i.

Let σ and τ be two substitutions. The following propositions are equivalent:
① the substitutions σ and τ are identical;
② for any expression E, the expression Eσ and Eτ are identical;
③ for any variable X of dom(σ)∪dom(τ), the terms Xσ and Xτ are identical.

Substitutions can be composed.

✓ **Composition of Substitutions** **19**

- The composition of the substitutions $\sigma = \{X_1/t_1,...,X_m/t_m\}$ and $\tau = \{Y_1/u_1,..., Y_n/u_n\}$, denoted by $\sigma\circ\tau$, or $\sigma\,\tau$ for short, is the substitution obtained from the set $\{X_1/t_1\tau,...,X_m/t_m\tau, Y_1/u_1,...,Y_n/t_n\}$ by deleting any binding $X_i/t_i\tau$ for which $X_i = t_i\tau$ $(1 \le i \le m)$ as well as any binding Y_j/u_j for which $Y_j \in X_1,...,X_m$.

Three elementary properties of the composition of substitutions:
① $\theta\varepsilon = \theta = \varepsilon\theta$;
② for every expression E, $(E\sigma)\tau = E(\sigma\,\tau)$;
③ $(\theta\sigma)\tau = \theta(\sigma\,\tau)$.

Closely related to equations, the idempotent substitutions, which form a particular class of substitutions has some interesting properties.

✓ **Idempotent Substitution** **20**

- A substitution θ is idempotent if, and only if, $\theta = \theta \circ \theta$.

✓ **Properties of the Idempotent Substitution** **21**

❶ A substitution θ is idempotent if, and only if, $dom(\theta) \cap varcod(\theta) = \varnothing$;

❷ Let σ and τ be substitutions. Suppose τ is idempotent. Then $\sigma \ge \tau$ if, and only if, $\sigma = \tau \circ \sigma$;

❸ Let σ and τ be substitutions. If $\sigma \subseteq \tau$ then $\sigma \le \tau$.

Unification

Since the structures of atoms and terms are similar, in the unification process they have a common behavior. That is the reason of the next definitions.

✓ **E-term** **22**

- An e-term is either a term or an atom.

✓ **Most General Unifier** **23**

- A substitution θ unifies 2 e-terms E and F if, and only if, the instances of Eθ and Fθ are syntactically identical. If so, E and F are said to be unifiable and θ is called a unifier of E and F. If θ is more general than all unifiers of F and E it is called a most general unifier[1].

The unification mechanism is the core of first-order logic. Several algorithms have been proposed[2], the one presented here is due to Herbrand ([Her67]).

Trying to unify E and F is trying to solve the equation E=F (or solving systems of equations).

Let S be a system of equations. Repeat the following actions as long as possible. Choose an equation Eq non-deterministically from S in one of the following forms and perform the associated action.

EQUATION	ACTION
Eq is $f(t_1,...,t_m) = g(u_1,...,u_n)$ with $m,n \geq 0$, and f syntactically different from g	Halt with failure
Eq is $f(t_1,...,t_m) = g(u_1,...,u_n)$ with $m,n \geq 0$, and $m \neq n$	Halt with failure
Eq is $f(t_1,...,t_m) = g(u_1,...,u_m)$ with $m \geq 0$	Replace Eq by the equations $t_1=u_1,...,t_m=u_m$
Eq is X = X with X a variable	Remove the equation
Eq is t = X with X a variable and t a non-variable term	Replace Eq by the equation X = t
Eq is X = t with X a variable appearing in another equation and t a term containing no occurrence of X	Replace X by t in every other equation than Eq
Eq is X = t with X a variable appearing in another equation and t a term different from X but containing an occurrence of X	Halt with failure

[1] Or mgu, for short.
[2] See for instance [R 1, R 4, R 6, R 10].

- How to recognize a solved system of equations ?
  - A system of equations is in solved form if it has the form

$$\begin{cases} X_1 = t_1 \\ \cdots \\ X_m = t_m \end{cases}$$

where the X_i's are distinct variables that do not appear in any of the t_j's.

- How to unify an atom and a normal clause ?
  - An atom *A* and a normal clause *C* unify with most general unifier θ if, and only if, θ is a most general unifier of *A* and of the head of *C*. It is assumed that, if necessary, variables of C have been renamed so that no variable of *A* appears in *C*.

2.2 Declarative Semantic

Declarative Semantics defines the meaning of formulas in terms of true values, of models and of logical consequences.

The following notions of interpretation and model are essentially addressed to definite programs.

✓ **Interpretation** **24**

- An interpretation consists of four parts:
  1. a nom empty set *D*, called the domain of the interpretation;
  2. for each constant, the assignment of an element in *D*;
  3. for each n-ary function, the assignment of a mapping from D^n to *D*;
  4. for each n-ary predicate, the assignment of a mapping from D^n to {true, false}.

✓ **Variable assignment** **25**

▫ Let I be an interpretation. A variable assignment with respect to I is an assignment to each variable of an element in the domain of I.

F	G	¬F	F∧G	F∨G	F⇒G	F⇐G	F⇔G
true	true	false	true	true	true	true	true
true	false	false	false	true	false	true	false
false	true	true	false	true	true	false	false
false	false	true	false	false	true	true	true

Table 3 : Truth tables for logical connectives

✓ **Term assignment** **26**

▫ Let I be an interpretation and V be a variable assignment. The term assignment with respect to I and V of the terms is defined as follows:

❶ each variable is given its assignment according to V;

❷ each constant is given its assignment according to I;

❸ if $\tilde{t}_1, ..., \tilde{t}_n$ are the term assignments of $t_1,...,t_n$ with respect to I and V and $\tilde{f}$ is the assignment of f with respect to I, then $\tilde{f}(\tilde{t}_1, ..., \tilde{t}_n)$ is the term assignment of $f(t_1,...,t_n)$ with respect to I and V.

✓ **True value of a wff** **27**

▫ Let I be an interpretation with domain D and let V be a variable assignment. Then a wff is given a truth value, true or false, with respect to I and V as follows.

❶ If the wff is an atom $p(t_1,...,t_n)$ then the truth value is obtained by calculating the value $\tilde{p}(\tilde{t}_1, ..., \tilde{t}_n)$ where $\tilde{p}$ is the mapping assigned to p by I and $\tilde{t}_1, ..., \tilde{t}_n$ are the term assignment of $t_1,...,t_n$ with respect to I and V.

❷ If the wff has the form ¬ F, F∧G, F∨G, F⇐G, F⇒G, F⇔G, then the truth value of the wff is given by the usual truth values for the logical connectives, listed in table 1.

❸ If the wff has the form ∃X(F), then the truth value of the wff is true if there is some d in D such that F has as truth value with respect to I and $V[X/d]$, where $V[X/d]$ is V except that X is assigned to d; otherwise, its truth value is false.

❹ If the wff has the form ∀X(F), then the truth value of the wff is true if, for all d in D, the truth value of F with respect to I and $V[X/d]$ is true; otherwise, the truth value is false.

As we can deduct from the last two items from definition 26, the truth value of a closed wff does not depend on a variable assignment.

✓ **Model** **28**

▫ Let S be a set of closed wff 's. An interpretation I is a model for S if the truth value of any formula of S with respect to I is true.

With all this concepts we can define the declarative semantics of logic programming.

✓ **Logical Consequence** **29**

▫ Let S be the set of closed wff 's. A closed wff is a logical consequence of S if, and only if, every model for S is a model for F too. This situation is denoted by $S \models F$, for any unclosed wff F, to really mean $S \models \forall(F)$.

✓ **Declarative Semantics** **30**

▫ The function $\mathcal{D}$: Sdprog→Sgoal→Ssubst, for any program $P \in$ Sdprog, any goal $G \in$ Sgoal, defines the following declarative semantics:

$$\mathcal{D}(P)(G) = \{\theta \in \text{Ssubst} : P \models G\theta \}$$

The basic problem from the declarative point of view is to determine whether $P \models G$ holds for some given program P and goal G. The following propositions allows to reduce this problem to the problem of checking that $P \cup \{\neg G\}$ has no model.

Let S be a set of wff 's and F be a closed wff.
Then F is a logical consequence of S, if and only if, $S \cup \{\neg F\}$ has no model.

Using Herbrand interpretations, a particular interpretation, simplify the checking of models. Before defining Herbrand interpretation a few preliminary notions are necessary.

✓ **Ground Term** **31**

▫ A ground term is a term containing no variables.

✓ **Ground Atom** **32**

- A ground atom is an atom containing no variables.

✓ **Ground Instance** **33**

- A ground instance of a clause C is a clause obtained from C by removing the universal quantifiers and by replacing each variable of C by a ground term. The set of ground instances of the clauses of the program *P* is subsequently denoted by ground(*P*).

✓ **Herbrand Universe** **34**

- Set of all ground terms.

✓ **Herbrand Base** **35**

- Set of all ground atoms.

✓ **Herbrand Interpretation** **36**

- An interpretation is an Herbrand interpretation if the following conditions are satisfied:

  ❶ the domain of the interpretation is the Herbrand universe U_H;

  ❷ constants are assigned to themselves in U_H;

  ❸ if f is an n-ary function, then f is assigned to the mapping $\tilde{f}$ from $(U_H)^n$ into U_H defined by $\tilde{f}(t_1,\ldots,t_n) = f(t_1,\ldots,t_n)$.

It is possible to identify an Herbrand interpretation by a set of ground atoms since the assignments of constants and functions is fixed. These atoms are considered to be true with respect to the interpretation

✓ **Herbrand Model** **37**

- Let S be a set of closed wff 's. An Herbrand model for S is an Herbrand interpretation which is a model for S.

✓ **Proposition** **38**

- A set of clauses S has no model, if and only if, it has no Herbrand model.

The least Herbrand model is a particular model that is used to reduce the checking of Herbrand models.

✓ **Least Herbrand Model** **39**

▫ Let *P* be a definite program and $\{M_i\}_{i \in I}$ be the set of Herbrand models. Then $\cap_{i \in I} \{M_i\}$ is an Herbrand model, called the least Herbrand model. It is denoted by Mp.

T The least Herbrand model Mp of a definite program *P* satisfies the equality $Mp = \{A \in B_H : P \models A\}$.

It is possible to further characterize the least Herbrand model with the help of the immediate consequence operator, denoted by Tp.

✓ **Definition** **40**

▫ Let *P* be a Horn clause program. The mapping Tp: $\mathcal{P}(B_H) \rightarrow \mathcal{P}(B_H)$ is defined as follows, for any Herbrand interpretation I.

$$Tp = \{A \in B_H : (A \leftarrow A_1,...,A_m) \in \text{ground}(P), \{A_1,...,A_m\} \subseteq I\}$$

The following auxiliary notions are needed to characterize the operator Tp.

Notation

Let *A* be a set and let $T:\mathcal{P}(A) \rightarrow \mathcal{P}(A)$ be a function from $\mathcal{P}(A)$ to $\mathcal{P}(A)$.
We define:

$$T\uparrow 0 = \varnothing; \quad T\uparrow n = T(T\uparrow(n-1)); \quad T\uparrow\omega = \bigcup_{n=0}^{\infty} T\uparrow n.$$

For any set of interpretations $S \subseteq \mathcal{P}(B_H)$, the least upper bound is the union and the greater lower bound is the intersection of the interpretations of S. The least Herbrand model can be linked to the immediate consequence operator as follows.

T Let *P* be a Horn clause program. Then,
$Mp = \text{lfp}(Tp) = Tp\uparrow\omega$
where lfp(Tp) is the least fixed point of Tp.

2.3 Operational Semantic

Operational Semantics refers to a model of execution and states, in a abstract way, how formulae are processed.

There are many methods of computing logic programs, each of them defining a logic programming language. The refutation proof procedure from Robinson[1] and Kowalski procedural interpretation, which is a specialized version (of the first) adapted to the context of Horn clauses is the source for most methods.

We will show how it applies to definite and normal programs.

Definite Programs

A computation in logic programming consists in trying to solve, in a sense to be made precise in a moment, a given conjunction of atoms $\leftarrow A_1,...,A_m$ with respect to a given definite program. More specifically, it consists of trying to find instantiations for the variables appearing in $A_1,...,A_m$ that conjointly solve these atoms. This can lead to two opposite results: success or failure. In the first case, the values assigned to the variables constitute the output of the computation. In the later case, no output is generated. Note that several successful computations may exist, each resulting in a different output.

The computation steps are non-deterministic.

Reduction steps **41**

Given $\leftarrow G_1,...,G_i,...,G_n \ (n \geq 1)$ and a clause $H \leftarrow B_1,...,B_k$.

❶ Select an arbitrary Gi;

❷ Let θ be the most general unifier of G_i and H.
Try to unify G_i with the clause $H \leftarrow B_1,...,B_k$;
If needed, the variables of the clause are previously renamed to avoid their appearance in the computation and specially in G_i.

❸ Transform the conjunction:

$\leftarrow G_1,...,G_i,...,G_n$ into $\leftarrow G_1,...,B_1,...,B_k,G_{i+1},...,G_n$	**Reduction / Deduction step**

❹ End the reduction when the current conjunction is empty or when no further reduction can be made.

[1] In [R 5].

☒ Be aware that infinite reduction processes may exist. They correspond to imperative infinite loops and do not produce any result.

Any sequence of the above transformation steps is called a SLD[1] - derivation. A complete SLD-derivation is successful[2] when it ends with the empty goal. It is qualified as a failure, otherwise.

The computation terminates successfully if the conjunction $\leftarrow A_1,...,A_m$ can be reduced to the empty conjunction. It fails if whatever atom and clause are selected at each step, the reduction process ends with a non-empty conjunction of goals.

The operational semantics may be defined using the following concepts:

✓ **Computed Answer Substitution** **42**

- Let $\theta_1,...,\theta_n$ be the successive mgu's involved in a (successful) SLD - - refutation for the query $\leftarrow A_1,...,A_m$ and let S be the set of variables of this query. The restriction of the substitution $\theta_1 \circ \cdots \circ \theta_n$ is called a computed answer substitution for $\leftarrow A_1,...,A_m$.

✓ **Derivation relation** **43**

- Let P be a definite program, G a goal and θ a substitution. The expression $P \vdash G[\theta]$ means that there is a SLD-refutation of G with respect to P giving as result the computed answer substitution θ. In other words, G is successful completed after n steps using the definite clauses of program P, yielding θ as a set of instantiated variables.

The derivation relation itself is defined by rules of the form:

$$\frac{\text{Assumptions}}{\text{Conclusion}} \quad \text{if Conditions}$$

that states that a Conclusion is reached if the Assumptions and the Conditions hold.

The derivation relation can be express by:

❶ $\frac{}{P \vdash \square\,[\varepsilon]}$ (no Assumptions, no Conditions)

which states that the empty conjunction $\square$ is derivable from any program with the empty substitution ε as a computed answer substitution.

[1] for Selection Linear Derivation.
[2] Also called a SLD-refutation.

❷ $$\frac{P \vdash (A_1,\ldots,A_{i-1},B_1,\ldots,B_k,A_{i+1},\ldots Am)\theta\ [\sigma]}{P \vdash A_1,\ldots,Am\ [(\theta\sigma)_{|\ vars\ (\{A_1,\ldots,A_m\})}]}$$

If ($H \leftarrow B_1,\ldots,B_n$ is a fresh renaming of the clause P) AND (A_i and H unify with mgu θ)

which express the reduction step explained in 37.

Example 9

Program

p(3)←q(Y).
q(4).

Goal: p(X)

Resolution

Unification of p(X) and p(3) giving $\theta = \{X/3\}$.
Unification of q(Y) and q(4) giving $\sigma = \{Y/4\}$.
Restriction to the composition $(\theta\sigma) = [\{X/3\}\{Y/4\}]_{|\{X\}}$ giving $\{X/3\}$.

✓ **Operational Semantics** **44**

- The operational semantics can be define as the following function $\mathcal{O}$: $Sdprog^1 \rightarrow Sgoal^2 \rightarrow \mathcal{P}(Ssubst^3)$: for any program $P \in$ Sdprog, any goal $G \in$ Sgoal,

$$\mathcal{O}(P)(G) = \{\theta \in Ssubst : P \vdash G\ [\theta]\}$$

Normal Programs

Many proposal have been made to treat normal clauses[4]. An early proposal ([R 8]) which is simple and easy to implement consists in the reduction of atoms (as describe above) and of reducing negative literals by reducing the associated positive literal and by inverting the computation results, thus reporting success if failure and vice-versa. In the reduction of atoms the empty conjunction is reported as the computed answer substitution.

[1] Set of definite programs.
[2] Set of goals.
[3] Set of substitutions.
[4] See for instance [R 2, R 3, R 7, R 9, R 11, R 13, R 14].

The type of negation described here is know as negation as failure.

$$not(p(a)) \Rightarrow p(a) \Rightarrow not(false/true)$$
$$\Downarrow$$
$$True/False$$

Negation as failure is not, in general, the real logical negation except when no variables appear in the negative literal, so it is usual programming practice to make sure that reduction progresses in such a way that, when they are selected for reduction, negative literals do not contain any variables. Alternatively, some logic programming languages delay the reduction of negative literals until they do not embody variables.

Search Tree

Prolog uses what is call a search tree. A search tree is defined by the following concepts.

✓ **Search state** **45**

- A search state is defined as:

  $< n, \theta, c >$ where
  - *n* is the number of the clause;
  - θ is a substitution;
  - *c* is the continuation atoms $B_1,...,B_p$.

✓ **Search Tree** **46**

- A standard search tree having $Q_1,...,Q_m$ as initial goals, is a finite or infinite tree, where each node is an instance of a search state.
  The root is an instance of $<0, \{\}, Q_1,...,Q_m>$.
  A instance node of $< n, \theta, c >$ where *c* is empty is called a success leaf.
  For every other instance node *v* of $< n, \theta, c >$ and for every *v'* (son of *v*), instance of $< n', \theta', c' >$ then:
  - The mapping $v' \rightarrow n'$ is a one-to-one mapping from the set of nodes sons of *v* to the set of numbers of clause *n'* such that, B_1 is unifiable with the head of the clause numbered *n'* (Cn');
  - *v* is a failure leaf if there is no *n'*;
  - $\forall$ v' $\exists$ $A_0' \leftarrow A_1',...,A_n'$ a variant of *Cn'* (clause renaming) without common variables with the goals of nodes states prior to *v'*;
  - θ' is the mgu of B_1 and A_0', *c'* is the result of the replacement in

θ'(c) of B_1 by $\theta'(A_1', \ldots, A_n')$;

- The order in the sons of v is the order in θ'.

In a search tree several success leaves may exist. The first to be found depends on the strategy used. Prolog strategy consists of a depth-first left-to-right traversal of the search tree.

2.4 Relation between the semantics

An equivalence between the declarative an operational semantics allows, in logic programming, programs to be made by declaring an information set of relations in the information, instead of a set of tasks to execute. The relation itself can be studied in the context of definite clauses through two properties: soundness and completeness.

✓ **Soundness** **47**

- Soundness expresses that any computed answer substitution for a goal and with respect to a given definite program instantiates the goal to a logical consequence of the program.

✓ **Completeness** **48**

- Completeness states the converse result that any substitution that instantiates the given goal to a logical consequence of the given definite program is less general than a computed answer substitution for the goal with respect to the program.

In other words, everything that is calculated at the operational level is true at the declarative level (soundness) and vice-versa. Together these properties allows to relate the declarative and operational semantics.

T

Let P be a definite program and G be a goal.
Soundness: For any θ, if $P \vdash G[\theta]$ holds, then so does $P \models G\theta$.
Completeness: For any θ, such that $P \models G\theta$ holds, there is a substitution μ such that $P \vdash G[\mu]$ and $G\mu \leq G\theta$.

2.5 WAM Implementation

The WAM is an abstract machine consisting in a memory architecture and instruction set tailored to Prolog[1].

This words from D. Warren are certainly the best description of what is know as the Warren's Abstract Machine, or for short, the WAM. To understand why the WAM is so important we must realize that Prolog programming language was built using the concepts of logic programming set by Kowalski, and before Warren's report[2] in 83, writing compilers for Prolog was, at least, a hard task.

The WAM is certainly a good starting point for studying Prolog implementation technology[1].

Alain Colmerauer and colleagues conceived Prolog in the beginning of the seventies at the University of Marseille.

One of the features that distinguishes the WAM is its memory architecture map, which is presented in figure 1.

There is, as we can see in the memory map, a Code Area, a Heap, a Stack, a Trail and a Push Down List. The purpose of the code area is, like the name suggest, the storage area for code; the Heap is used as a global stack to store any permanent information[3]; the Stack is where variable data are putted; the Trail supports Prolog's mechanism of backtracking and finally the Push Down List is a global dynamic area used as unification stack.

The registers arguments A1,...,An are used (in a query or body goal) as auxiliary and temporary storage areas for passing information between the caller and the called, like arguments to a function. The same registers are used to store the contents of temporary variables in which case a different notation $X_1,...,X_n$ is used to differentiate the use. These registers also exist in the choice point frame as a backup to allow a full restoration of an early state of the computation.

Opposed to the temporary and auxiliary registers there are others which are local and permanent (Y1,...,Yn), in the environment frame. Facts uses the temporary registers and rules uses the permanents.

To understand the utility of the choice point and environment frames we must first explain how Prolog search is done, with an example.

[1] Quoted from David H. D. Harren, forewords in [5].

[2] In which the principles of the WAM are set.

[3] Namely, variables and structures.

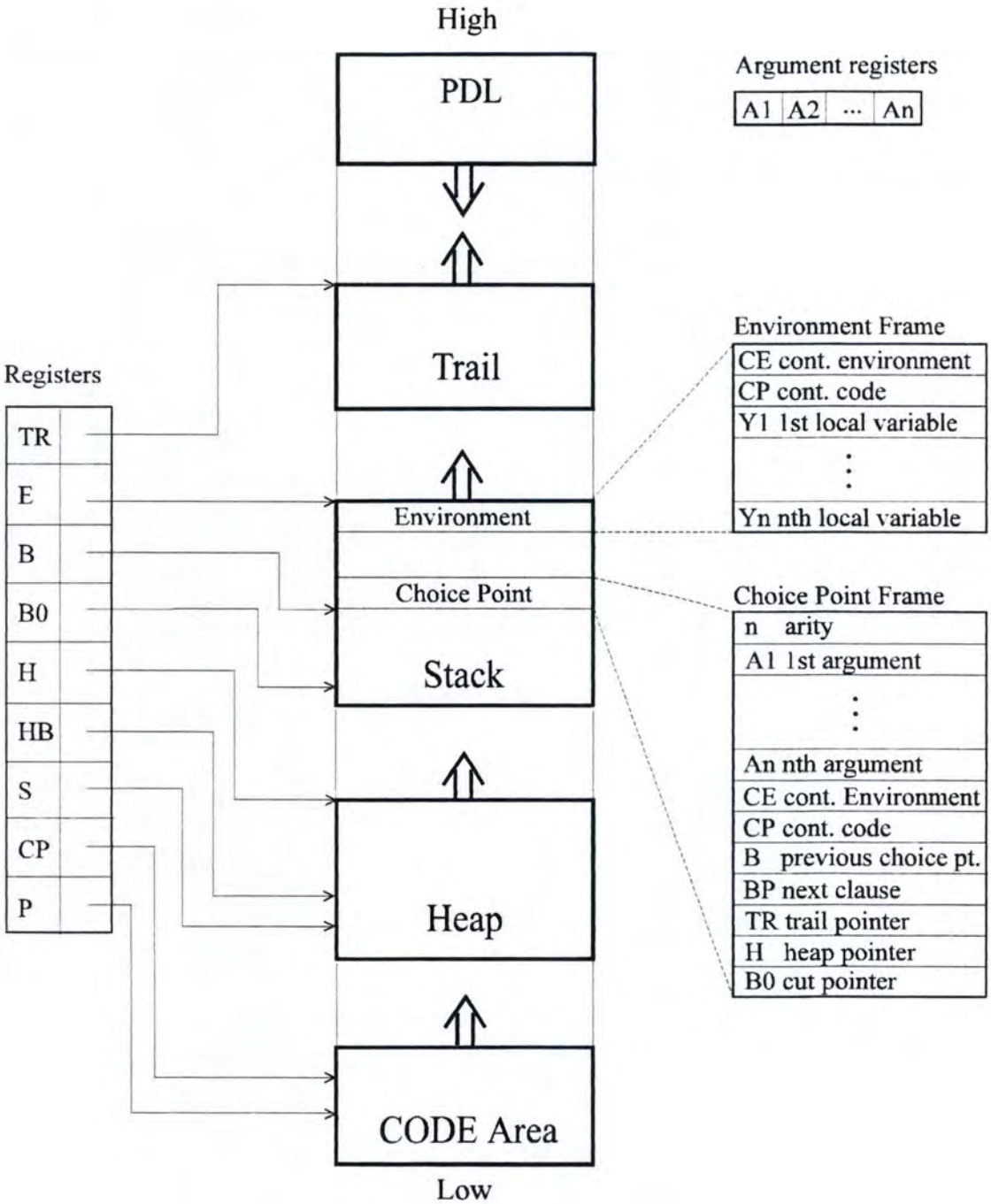
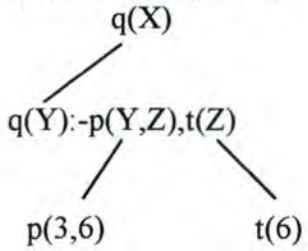


Figure 1 : WAM memory layout and registers

Example 10

Let :-q(X) be a query[1], q(Y):- p(Y,Z),t(Z) be a rule and p(3,6), t(6) be facts.

Prolog will try to find an answer for the query in a tree like search.



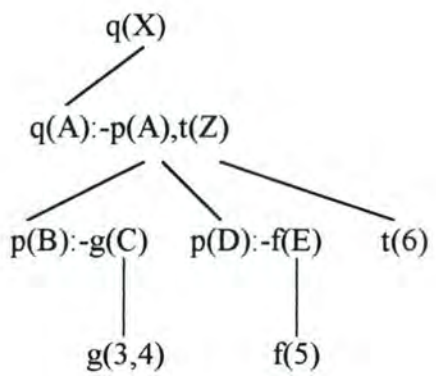
In the example above we must make a choice in the rule, what goal do we try first p(Y,Z) or t(Z) ? Prolog always tries the left one first, continuing as deep as possible and proceed with the others (in the example t(Z)). Since the variable Z appears in more than one goal it must be made permanent, because it must outlive the goal in which it was first created (i. e. p(Y,Z)). The storage is done in the environment frame, for each variable appearing in more than one body goal, the head atom is considered as part of the first body goal[2] for this purpose.

An environment frame is created by the first body goal and removed by the last goal[3].

Environment frames are also called AND-stack since each environment is created for a body goal and as we know they are all linked by the logical connective AND.

The previous example will be changed to explain choice point frames.

Example 11



[1] A query is a rule with no head.
[2] That's why Y, in the example, is not a permanent variable.
[3] In reality, for efficiency purposes, the environment is destroyed before the call to the last goal.

Now there are two rules for the definition of clause p. Since Prolog tries the left one first and goes as deep as possible to find a match[1], the sequence q(X), q(A):-p(A), p(B):-g(C), g(3,4) is first tried, but since the unification fails at this point (in the goal g(C) and g(3,4)), Prolog must clime in the search tree so that the clause p(D):-f(E) can be explored downwards. The problem here is that if the computation state at q(A):-p(A),t(Z) is not saved, it's impossible to go back when there is a failure in the unification process. For this purpose, when more than one definition of a clause exist it is necessary to create a choice point to save the current state and restore it upon failure using backtracking (return to a previous state where a choice, of what clause definition to try, was made).

Choice point frames have also another name: OR-stack, given by the fact that the different alternatives are logically or-alternatives, since if one fails the next is tried and so on until one succeeds or the last one fails, in this way succeeding the goal or failing otherwise.

A choice point frame is created by the first alternative, updated by intermediate[2] alternatives (as far as which alternatives to try next) and finally discarded by the last alternative.

A closer look to the choice point frame and environment frame, in fig. 1, will call the attention to the cells named B and CE, pointer to last choice point and pointer to last environment respectively, forming in this manner a linked list of frames of the same type; furthermore both kinds of frames are mixed.

An explanation of the others components of the WAM follows.

Trail

The need for choice points is to restore a previous state with everything that defines that state: contents of variables, registers, integers, and so on. Our concern is with unbound variables that are instantiated with values or memory references in one alternative and upon backtracking they need to be set to unbound again, to allow a full reposition of a previous state; as in the last example , where the variable *A* is unified with *B* in the first alternative and when backtracking is performed, due to unification failure of g(C) with g(3,4), p(D):-f(E) is tried, but a problem occurs since *A* can not be tied to *D* since it is already tied with *B*. What to do ?
The best way to handle this situation is to store the location of the variable *A* in a heap called Trail.

When a variable previous to the last choice point must be tied, its address is putted in the trail. To make this arrangement work we must put, in the choice point frame, the location of the top of the trail. When backtracking it's only necessary to set

[1] A successful unification between a query or body goal and a atom (i.e. fact or head atom of a rule).
[2] But not ultimate alternatives.

to unbound all memory cells referenced in the trail, between the previous top (stored in the last choice point frame) and the current top pointed by the register TR.

When two variables are tied, as in the last example, *A* and *B*, the variable that will contain the reference to the other must be the one that belongs to the current clause, so B←A, avoiding in this way the storage of the location of variable *A* in the trail.

Heap

As seen before, variables are either stored in temporary registers or in environment frames, but where to store the terms specially those whose arity is greater than one ? They must be stored in the Heap, whose top is given by the base register H[1]. Upon backtracking, all the terms created after the last choice point are removed like a garbage collector. To allow this, each point frame has a cell named H that points to a previous top of the heap, everything between the current top and the previous, is disposed.

We now know where terms are stored, but how are they built ?

Term representation

There are several term types to store so the need arises to represent them with the help of a tag to identify each term type.

TERM	*TAG*	*VALUE*
Variable	REF	Pointer to an address in the Heap
Integer	INT	Integer
Constant	CST	Pointer to a table of constants
Empty list	----	Represented by the constant '[]'
Non empty list	LST	Pointer to an Heap cell containing the CAR[2]. Next cell will always be the CDR[3].
Structure	STR	Pointer to an Heap cell containing the functor[4] and arity. Next to them comes the n sub-term elements.

Table 4 : Definition of Terms

[1] Pointer to next free cell.
[2] The Head of the list.
[3] The remainder of the list.
[4] More precisely, a pointer to hash-code table of constants.

The representation of terms is as follows:

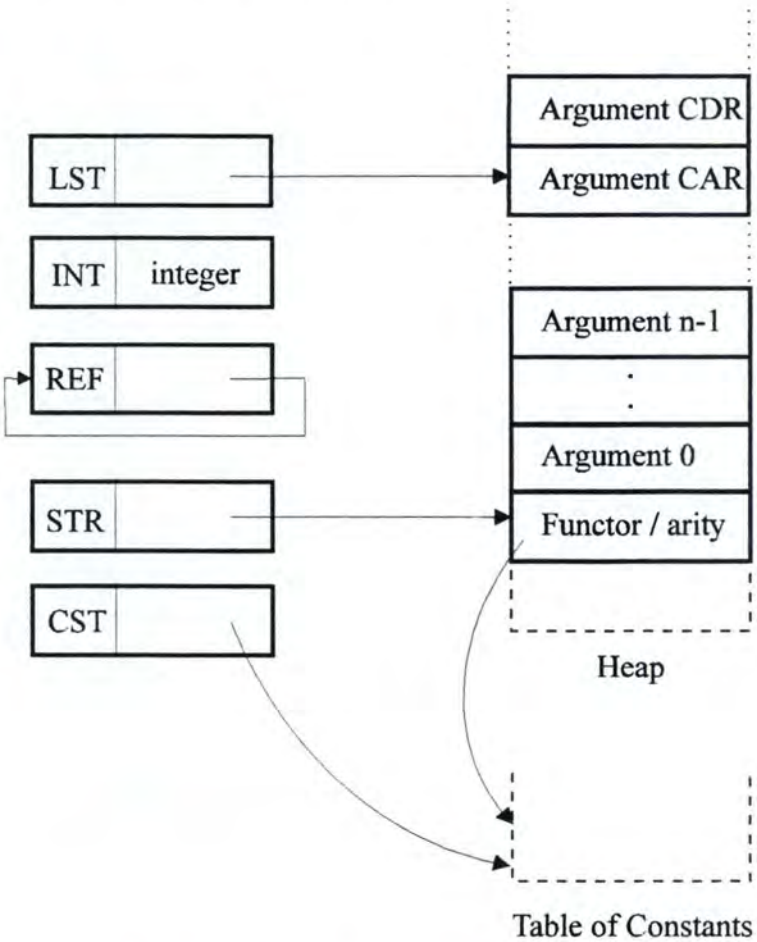


Figure 2 : Term representation in the WAM

An unbound variable points, by convention, to itself. The operation of following linked variables until one unbound variable or a term different from a variable is found is called *dereferentiation*.

Unification of structured terms

Suppose that we have a program *p*, a query *q* that has build a term on the heap and a register X1 to contain the term's address. Thus unifying *p* to *q* can proceed by following the term structure already present in X1 as long as it matches functor for functor the structure of *p*. The only complication is that when an unbound REF cell is encountered in the query term in the heap, then it is to be bound to a new term that is built on the heap, as an example of the corresponding sub-term in *p*.

As we can see, two different modes are used in the unification process:

- ✧ Read mode, in which data on the heap is matched against. The base register S is set to contain at all times the heap address of the next term to be matched.
- ✧ Write mode, in which a term is copied to a new location in the heap.

To avoid the use of another register to identify the mode, the register S is used since in write mode S is set to Null (i.e. if S>0 then mode is read, otherwise it's write).

Indexing

Each time choice points are created variables must be put in the Trail; to save memory it would be useful to reduce their number. The principle of indexing is to divide the set of clauses of the same predicate using the possible values of certain arguments as keys and to generate the instructions for the management of choice points, for each sub-set separately. Since there is only a concern with one part of the set, the probability of creating choice points is reduced.

In the WAM, the key for indexing is the main functor of the first argument of the head, which can be:

- ✧ a Variable,
- ✧ a List (not empty),
- ✧ an Integer,
- ✧ a Constant (including the constant for the empty list),
- ✧ a Structure.

In the last three items, another division is made using their value.

Instruction Set

A predicate may be compiled independently of any context as a result of using the register arguments for exchanging data and there is no need for a predicate clause to be aware of the others; only the instructions who manage choice points needs to know all clauses. Studying predicates at compile time allows the WAM to produce code more efficient, faster and that saves memory. As an example, choice point management is done by specialized instructions as is the unification that is decomposed according to the arguments of the head of the clause to avoid using the general unification algorithm.

The WAM instruction set may be grouped in four categories:

- ✧ Registers recovery instructions,
- ✧ Registers setting instructions,
- ✧ Control instructions,
- ✧ Indexing[1] instructions.

Registers recovery instructions

These instructions are produced by the compilation of the head of clauses. For reasons of performance they are decomposed; their type, comments and respective instruction follows.

Let *V* be a temporary or permanent variable and *A* a register argument .

First occurrence of a variable get_variable V,A	Makes a copy from V to A.

Other occurrence of a variable get_value V,A	Unify V with A.

constant C get_constant C,A	Checks that *A* is tied to *C* or to an unbound variable, in which case this variable is tied to C.

The instruction for the integers, *get_integer N,A* is like *get_constant*.

empty list get_nil A	Abbreviation of get_constant '[]',A.

The recovery of a structured term uses the unify instructions. The unification of a composed term behaves in two different ways depending upon the type of argument being unified with the term:

- If the term has the same functor and the same arity then a real unification takes place regarding the sub-terms (Read mode).
- If it is an unbound variable then the term is created in the Heap and the variable is tied to it.

[1] Choice point management.

list not empty	These instructions dereference *A*; if the dereference word is an unbound variable, it is tied to a list created in the Heap; if the word is a list, the instructions *unify_...* will unify the head and the rest of the list.
get_list A unify_... (Car) unify_... (Cdr)	

structure F/N	These instructions dereference A; if the dereference word is an unbound variable, it is tied to a structure created in the Heap; if the word is a structure the instructions unify_... will unify the sub-terms.
get_structure F/N,A unify_... .... unify_...	

The code produced in the compilation of a sub-term depends of its nature, as follows:

first occurrence of variable V	This instruction ties V to the cell pointed by S in Read mode and ties V to an unbound variable putted in the Heap, in Write mode.
If V is not singleton[1] : unify_variable else K[2] :unify_void(K)	

The instruction *unify_void* allows to optimize the singleton variables in the structures. In Read mode, adds K to S and in Write mode puts K unbound variables in the Heap.

other occurrence of V	To use the instruction *unify_local_value* we must first be sure that a connection will not occur from the Heap to the Stack. In Read mode: *unify_local_value* behaves exactly as the instruction *unify_value*. In Write mode : if the dereference word of V is an unbound permanent variable do a globalization of the last, else put the word in the Heap.
If it is possible to know if in the first occurrence V was tied with the Heap[3] : unify_value(V) else: unify_local_value(V)	

constant C	In Read mode this instruction behaves like *get_constant*, but instead of S is the word pointed by *S*. In write mode, puts a constant in the Heap.
unify_constant C	

The instruction for the integers is like the one for constants

empty list	A shortcut to unify_constant '[]'.
unify_nil	

As we can see, there are no unification instructions for recovering composed terms, they are unified with temporary variables X using the instruction *unify_variable X*, these instructions are decomposed by the *get_...X* instructions.

[1] A variable occurring only once in the clause.
[2] Number of successive singleton variables.
[3] First occurrence of V in a structure, or if temporary, first occurrence in the body.

Example 12

A[i] → f(h(b,Y)) becomes A[i] → f(X) and X → h(b,Y)

In the registers setting, the new temporary variables are set by the *put_...X* instructions before being unified by *unify_value X*.

Example 13

A[i] ← f(h(b,Y)) becomes X ← h(b,Y) and A[i] ← f(X)

Registers setting instructions

Here there are also different instructions depending of the argument to set. As before, *A* references A[i] and V a temporary or permanent variable.

first occurrence of V put_variable V,A	If V is permanent then sets V and A with an unbound variable else *V* and *A* are tied to an unbound variable putted in the Heap.
other occurrence of V If V is a safe variable and the current goal is not the last: put_value V,A else : put_unsafe_value V,A	The instruction *put_value* copies *V* to *A*. If is in the last goal, we must confirm that the copy will not link A to the current environment. The instruction *put_unsafe_value* takes care of the dangling references. If the dereferenced word of V belongs to the current environment then the variable is made global, else copy the word to A.

An unsafe variable with *n* occurrences in the last goal will need *n* put_unsafe_value instructions. The first will make the probable globalization and the others will copy the dereferenced word.

constant C put_constant C,A	Put the constant C in A.

The instruction *put_integer* is similar to the constant instruction.

empty list put_nil A	Abbreviation of put_constant '[]',A.
list not empty put_list A	Sets *A* with <LST,H> and having set the mode to Write so that the *unify* instructions that follows may copy again the Car and Cdr.

The instruction *put_structure F/N,A* works also in this way.

Control instructions

The control instructions manages the call and return from procedures as well as the environments. The call to the last goal has a different instruction since it must do the return. Coming next are the prototypes for facts and rules.

- for a fact *p(...).*

```
registers recovery
proceed
```

- for a clause *p(...):-q(...).*

```
registers recovery
registers setting for q
execute(q)
```

- for a clause $p(...):-q_1(...), q_2(...), ..., q_n(...).$

```
allocate(N)
registers recovery
registers setting for the goal q1
call(q1)
registers setting for the goal q2
call(q2)
.
.
.
registers setting for the goal qn
deallocate
execute(qn)
```

The instruction **allocate(N)** creates an environment for N variables and the **deallocate** removes it from memory.

The instruction **call(P/N)** sets the base register CP to the code address that follows the call and gives control[1] to the predicate P/N. The instruction **execute(P/N)** does the same but without changing the register CP.

The return from a procedure is done by the instruction **proceed** that only sets the register P to the value of register PC.

Indexing instructions

These instructions are used to group the code for the each clause of a predicate and because of this they are the top level instructions. They manage the choice points. They

[1] By setting the register P to the address of the first instruction of predicate P/N.

may create up two choice points in the beginning of a predicate, that's why we can speak of two levels of indexing. The two levels exist because a variable that appears in the heads first argument can not be used as a key type in the indexing.

For the clauses $C_1,...,C_n$ several groups ($G_0,...,G_m$) are created. If the first argument of the head is a variable then a level 1 type code is generated, otherwise a level 2 code is produced. In level one, each group contains only the code for a clause.

- **Level 1**

```
if m=0 | code for G0

       |      | try_me_else L1
       |      | code for G0
       |
       | L1:  | retry_me_else L2
else   |      | code for G1
       |      .
       |      .
       |      .
       | Lm:  | trust_me_else_fail
       |      | code for Gm
```

The instruction **try_me_else L_{else}** has the mission of creating the choice points where L_{else} is the next clause code address. The **retry_me_else L_{else}** instruction resets the base registers, as the previous instruction it sets the next alternative's code. The final instruction resets the base registers and removes the choice point.

- **Level 2**

  In level two, each group contains the following instructions:

  switch_on_term(L_{var}, L_{cte}, L_{int}, L_{lst}, L_{str})

  Accordingly to the argument type a L alternative is chosen.

For instance, if there are no constants then L_{cte} = fail, else

L_{cte}: switch_on_constant(N,[(cte_1, L_{cte1}),...,(cte_N, L_{cte_N})])

For each constant cte_j (j = 1,2,...,N) the code produced is the following:

if there is only a clause which has the constant cte_j as first argument then

L_{cte_j}=L_{j1} (being j_i the number of the clause having cte_j as first argument), else:

```
Lcte_j : | try (Lj1)
         | retry (Lj2)
         | .
         | .
         | .
         | trust (Ljk)
```

The integers and structures produce the same code.

If there is no list L_{lst} = fail else the following code is generated:

```
Llst : | try (Lj1)
       | retry (Lj2)
       | .
       | .
       | .
       | trust (Ljk)
```

where j_i is the clause number I.

If G_i has only one clause then $L_{var} = L_1$, else :

```
Lvar  : | try_me_else (Lvar2)
L1    : | code for clause 1

Lvar2 : | retry_me_else(Lvar3)
L2    : | code for clause 2
        | .
        | .
        | .
Lvarp : | trust_me_else_fail
Lp    : | code for clause p
```

The instruction `switch_on_term(`L_{var}, L_{cte}, L_{int}, L_{lst}, L_{str}`)` gives control to the L address according to which word type is register A[0] tied.

The `switch_on_constant(N,[(`cte_1,L_{cte1}`),...,(`cte_N,L_{cte_N}`)])` instruction associates to a constant, using a table, an address L. L is the address of the clause in which it's first argument is `cte` or to a level 2 code[1] when several clauses have `cte` as the first argument of the head. The same holds for the integers and structures.

[1] Try, retry and trust.

The instructions try(L), retry(L) and trust(L) behave like try_me_else(L_{else}), retry_me_else(L_{else}) and trust_me_else_fail respectively; the only difference is that in the cell of the choice points, it is the next instruction and not the L_{else} that is stored there. The next code to be executed is the one lying at $L_{address}$.

Memory release and savings

Saving memory in Prolog is essential due to the enormous memory needed, since the solutions choice is made at runtime. There are several ways of releasing memory, for instance, in the process of backtracking a part of the Heap is released when the top is reset to it's former position; Stack frames are also released through this process.

Other processes are used to save memory, for instance, facts and rules with only a body goal do not need environments[1]. Rules with more than a body goal need an environment until the end of the clause.

Example 14

Rule with more than a body goal

p(A,B):-q(B,A),r(A,B).

is transformed into

```
p/2:    allocate 2
        get_variable Y2,A1
        get_variable Y1,A2
        put_value Y1,A1
        put_value Y2,A2
        call q/2
        put_value Y2,A1
        put_value Y1,A2
        deallocate
        execute r/2.
```

As we can see, the *deallocate* instruction (that releases the environment) is put before those of the continuation (i.e. proceed, execute) since the argument registers are already loaded, no further references to the environment will be made. Obviously, the release of an environment has only an interest as long as no choice points have been created above it, in which case the environment will only be released after the disposal of the choice point.

☒ If a reference to the released environment exist it is a potential danger.

[1] Since putting the {allocate 0, ..., deallocate} instructions is useless.

To prevent situations of this kind, some rules are imposed:

WAM Binding Rules

❶ Always make the variable of higher address reference that of lower address;

❷ Heap variables must never be set to a reference into the Stack.

❸ The Stack must be allocated at higher addresses than the Heap, in the same global address space.

WAM binding rule number 3 is a logical consequence of the first two. The rules mentioned above cover two possibilities of variable bindings, namely heap-heap and heap-stack, but unfortunately this rules are not sufficient to prevent dangling references in a stack-stack binding, so unsafe variables appear.

Unsafe variables

Any permanent variables initialized with a *put_variable* instruction are called unsafe variables. The explanation for this, is that since the variable is permanent it must be in the Stack (environment frames) but to spare space the environment is released before the call to the last goal in which the permanent variable may appear and the danger is that the predicate may reference that variable.

Example 15

```
p/1: allocate n                    r/2: allocate m
     ...                                 get_variable Yk,A1
     put_variable Yi,A1                  ....
     ...
     deallocate
     execute r/2
```

If the predicate p/1 is executed first, the register A_1 will have the reference to the permanent variable Y_i and let suppose that nothing changes Y_i or A_1, then the environment is deallocate and r/2 is called; Y_i will be set with the contents of register A_1, in other words, there will be in the environment of r/2 an reference to another environment that as already been disposed off. To prevent it, the variable of Y_i is put if necessary in the Heap, which has the effect of making the value of Y_i global so as to guarantee that it may be discarded without leaving a nonsensical reference in A_1. To accomplish this the instruction *put_unsafe_value* is used at the place of *put_value* in the *first occurrence of the variable in the Last goal.*

Another source of danger is when a permanent variable appears in the last goal nested in a structure whether or not it is also an argument. To solve it, it is only necessary to set the permanent variable to point to an unbound variable created in the Heap.

The basic idea of CLP is to replace unification by constraint solving over a particular domain of interest[1] .

3.1 Introduction to CLP

The declarative nature of Logic Programming combined with the ability to reason and compute with partial information on specific domains allows Constraint Logic Programming to be used in a wide range of real life applications. One usual class of domains found in CLP are finite domains which where first introduced by Pascal van Hentenryck at the end of the eighties.

Finite domains are simply a set of values containing numbers or symbols with a finite cardinality, like for instance {1,2,3,20,100}, {sun, moon, earth} or 1..100.

The constraints to be used on finite domains are arithmetic constraints (equations, inequations, disequations) between linear terms, as well as symbolic constraints. For instance the relation *atmost(N, [X_1,...,X_m], V)* means that at most N variables X_i are equal to the integer V. In finite domains, constraint solving is done by propagation and consistency techniques belonging to Artificial Intelligence originated from Constraint Satisfaction Problems.

The general idea is to build a network of constraints between a finite number of variables, each with a number in a finite domain. It is usual to represent this scheme as a graph where variables are represented by nodes and constraints by arcs. The satisfiability of the set of constraints is assured by the propagation from neighbors nodes (local propagation), the possible values of the variables between the connected constraints.

The resolution of the constraints is done by several techniques, like for instance, the broadcast of the domains of variables through the network (arc consistency) or by a more general technique, the propagation of the relations between variables (k-consistency).

The responsability for checking the consistency of a set of constraints and, possibly for reducing it into some normal form is done by the constraint solver considered as a "black box". The black box approach does not give any control to programmers about the execution of the constraints.

A "glass box" approach was proposed[2] to give programmers a better control of the complexity of the methods needed to ensure the consistency of the constraints. It is based in a limited number of simple primitive constraints. The basic idea is to have a single constraint *X in r*, where X is the finite domain variable and r is a set of integers.

---

[1] Quoted from [11]

[2] In [R 12].

Complex constraints are translated at the compilation time in a set of primitive constraints.

The semantic of this constraint enforces *X* to belong to the domain denoted by *r*. The constraint *X in r* can be seen as a way of specifying the propagation mechanism. More precisely, it allows to specify what is to be propagated.

Any modification in a constraint is propagated in one of two ways through the network of constraints, by partial lookahead or by full lookahead. Partial lookahead occurs when the changed limits of variables are propagated; if a hole occurs in the middle of the domain, the hole is not propagated. Full lookahead occurs when any modification in the domain of the variables are propagated.

Example 16

The equation X=Y+C is define in CLP as:

```
'x=y+c'(X,Y,Z):- X in min(Y)+C .. max(Y)+C,
                 Y in min(X)-C .. max(X)-C.
```

Example 17

The equation Z=X+Y is define as:

```
'x+y=z'(X,Y,Z):- X in min(Z)-max(Y) .. max(Z)-min(Y),
                 Y in min(Z)-max(X) .. max(Z)-min(X),
                 Z in min(X)+min(Y) .. max(X)+max(Y).
```

The last example is a full lookahead example. The equation X=Y+C which is defined in partial lookahead can be also define in full lookahead.

Example 18

```
'x=y+c'(X,Y,Z):- X in dom(Y)+C,
                 Y in dom(X)-C.
```

An extension to the WAM for finite domains based in the *X in r* constraint is developed subsequently without touching the WAM's architecture and data structures. The constraint solving is done by the language *X in r*.

3.2 Language

The notions of finite domains and constraints are first formally defined. The syntax of this constraint system (X in r) follows.

✓ **Finite Domains** **49**

- A finite domain is a (non empty) set of natural numbers (i.e. a range). A range is a subset of {0,1,...,*infinity*} where infinity denotes the greatest integer that a variable can take.

We use the interval notation $k_1,...,k_2$ as a shorthand for the set $\{K_1,K_1+1,...,K_2\}$. In a range *r*, *min(r)* (resp. *max(r)*) is defined as the lower (resp. upper) bound of *r*.

Dom is the set of all domains. V_d is the set of FD variables.

As usual all the standard operations on sets (e.g. union, intersection, etc.) are defined as well as the following pointwise operations.

✓ **Pointwise operations on sets** **50**

- A pointwise operation (+,-,*,/) between a range *r* and an integer *i* is defined as the set obtained by applying the corresponding operation on each element of *d* i.e. develop for +, -, *, /.

Being *r* a range and *i* an integer, then the range of $r \bullet i$ with $\bullet \in \{+,-,*\}$ is defined as $r \bullet i = \{k = k' \bullet i, k' \in r\}$. The special case *r/i* is equal to $\{k = \lfloor k'/i \rfloor \in r\}$[1].

Syntax of the X in r Constraint

In the FD (finite domain) constraint system, there are three kind of syntactic objects:

- ✧ constraints;
- ✧ ranges;
- ✧ arithmetic terms.

The following sets are also define.

Constr	Set of syntactic constraints
SynDom	Set of syntactic domains
SynTerm	Set of syntactic terms

Table 5 : Definition of Sets

[1] An integer *i* resulting from a division is surrounded by the symbols $\lfloor \rfloor$, which indicates a lower rounding.

The definition of the constraints comes next.

✓ **Constraint** **51**

- A constraint is a formula of the form X *in* r where $X \in V_d$ and $r \in$ SynDom.

The notation X=n is a shorthand for X *in* $n..n$. A range r can have a constant range (e.g. 1..10) or an indexical range as those listed in the next table:

dom(Y)	represents the current domain of Y
min(Y)	represents the minimal value of the current domain of Y
max(Y)	represents the maximal value of the current domain of Y

Table 6 : Definition of Indexical ranges

A check must be made each time the domain of a variable Y is updated if a constraint X *in* r uses an indexical of such variable Y (e.g. X in dom(Y)). A reason for the necessity of the checking is given by the next example.

Example 19

The constraint X≠Y is define in CLP as:

```
'x≠y'(X,Y):- X in -dom(Y),
             Y in -dom(X).
```

The domain given by *-dom(Y)* grows according to the domain of Y; when the domain of Y decreases the complement *-dom(Y)* grows accordingly. A problem arises in the implementation since an inconsistent value of X (i.e. not belonging to the complement of the domain of Y) may became consistent if Y is reduced. Such constraints (X in -dom(Y)) must wait then for Y to be instantiated. The process that checks if a domain of a variable must be updated is named *forward checking*.

In addition to the indexical range, a constraint can also use run-time constant values as parameters. The constraint $\neg X$ *in* r is just X *in* -r, due to the fact that the FD system is closed under negation.

The following definitions are important to the FD constraint system.

✓ **store** **52**

- A store is a finite set of constraints.

✓ **Normal form of a store** **53**

- A store is in normal form if, and only if, it contains at most one constraint X *in* r for each variable $X \in V_d$.

c::= X in r	
r::= $t_1..t_2$ {t} R dom(Y) $r_1 : r_2$ r_1 & r_2 -r r + ct r - ct r * ct r / ct	interval singleton range parameter indexical domain union intersection complementation pointwise addition pointwise subtraction pointwise multiplication pointwise division
t::= min(Y) max(Y) ct $t_1 + t_2 \mid t_1 - t_2 \mid t_1 * t_2 \mid t_1 /< t_2 \mid t_1 /> t_2$	indexical term min indexical term max constant term integer operations
ct::= C n | infinity $ct_1 + ct_2 \mid ct_1 - ct_2 \mid ct_1 * ct_2 \mid ct_1 /< ct_2 \mid ct_1 /> ct_2$	term parameter greatest value

Table 7 : Syntax of the X in r constraint

Any store *S* is transformed into its normal form when all constraints *X in* r_1, *X in* r_2, ... , *X in* r_n on *X* are replaced by a single constraint of the form *X in* r_1 & r_2 & r_n. The resulting set is equivalent to the others since they have the same tuples of solutions. All the stores that will be used from this point forward are considered to be in normal form. Furthermore, the expression S∪{c} is use to express the addition of the constraint *c* to the store *S*.

The set of all stores is: Store.

3.3 Semantics of the X in r constraint

An important aspect of the semantics *X in r* is the tell operation

SynDom	Syntactic domain	T : Constr → Store → Store
SynTerm	Syntactic terms	T' : Constr → Store → Store
Dom	Domains	$\mathcal{E}_r$: DomSyn → Store → Dom
N	Natural numbers	$\mathcal{E}_t$: TermSyn → Store → N
Constr	X in r constraints	
Store	stores	

T [c] s	= fix(λ s . $\cup_{c' \in s \cup \{c\}}$ T' [c'] s)
T' [x in r] s	= **let** d = [$\mathcal{E}_r$ [t_1..t_2] s] **in** s $\cup$ {x in d} $\cup${x in r}
$\mathcal{E}_r$ [t_1..t_2] s	= $\mathcal{E}_t$ [t_1] s .. $\mathcal{E}_t$ [t_2] s
$\mathcal{E}_r$ [{t}] s	= {$\mathcal{E}_t$ [t] s}
$\mathcal{E}_r$ [R] s	= lookup_range(R)
$\mathcal{E}_r$ [dom(Y)] s	= cur_domain(X,s)
$\mathcal{E}_r$ [r_1 : r_2] s	= $\mathcal{E}_r$ [r_1] s $\cup$ $\mathcal{E}_r$ [r_2] s
$\mathcal{E}_r$ [r_1 & r_2] s	= $\mathcal{E}_r$ [r_1] s $\cap$ $\mathcal{E}_r$ [r_2] s
$\mathcal{E}_r$ [-r] s	= 0 .. *infinity* \ $\mathcal{E}_r$ [r] s
$\mathcal{E}_r$ [r + ct] s	= $\mathcal{E}_r$ [r] s + $\mathcal{E}_t$ [ct] s
$\mathcal{E}_r$ [r - ct] s	= $\mathcal{E}_r$ [r] s - $\mathcal{E}_t$ [ct] s
$\mathcal{E}_r$ [r * ct] s	= $\mathcal{E}_r$ [r] s * $\mathcal{E}_t$ [ct] s
$\mathcal{E}_r$ [r / ct] s	= $\mathcal{E}_r$ [r] s / $\mathcal{E}_t$ [ct] s
$\mathcal{E}_t$ [n] s	= n
$\mathcal{E}_t$ [*infinity*] s	= *infinity*
$\mathcal{E}_t$ [C] s	= lookup_term(C)
$\mathcal{E}_t$ [min(Y)] s	= min(cur_domain(X,s))
$\mathcal{E}_t$ [max(Y)] s	= max(cur_domain(X,s)
$\mathcal{E}_t$ [t_1 + t_2] s	= $\mathcal{E}_t$ [r] s + $\mathcal{E}_t$ [ct] s
$\mathcal{E}_t$ [t_1 - t_2] s	= $\mathcal{E}_t$ [r] s - $\mathcal{E}_t$ [ct] s
$\mathcal{E}_t$ [t_1 * t_2] s	= $\mathcal{E}_t$ [r] s * $\mathcal{E}_t$ [ct] s
$\mathcal{E}_t$ [t_1 /< t_2] s	= $\mathcal{E}_t$ [r] s / $\mathcal{E}_t$ [ct] s
$\mathcal{E}_t$ [t_1 /> t_2] s	= $\mathcal{E}_r$ [r] s / $\mathcal{E}_t$ [ct] s
cur_domain(X,s)	= $\mathcal{E}_r$ lookup_store(X,s)
lookup_store(X,s)	= **if** $\exists$ *X in r* **then** *r* **else** 0..*infinity*
lookup_range(R)	returns the domain bound of R
lookup_term(C)	returns the integer bound to C

Table 8 : Denotational Semantics of the Tell operation

✓ **Tell operation** **54**

- A tell operation results in the addition of its argument (constraint) in the current store.

The semantics of a tell operation is expressed in table 8 by the function T[X in r]S which adds a constraint *X in r* to the store *S*. The tell operation updates *X*, with respect to *r* evaluated in *S* and reactivates all the constraints depending on *X* through the propagation mechanism. X is updated by intermediary function T′ [X in r] and the propagation is done using a fix point operator on the result of T[X in r] that reevaluates all the constraints in S∪{X in r} until a stable state is reached.

The function T′ [X in r] adds two versions of the constraint *X in r* to the given store which allows to take care of the indexical constraints. The versions are:

❶ *r* is evaluated in *S*;

❷ *r* is unchanged to allow future reconsideration of this constraint in presence of an indexical range.

An indexical range can only be evaluated thanks to the current domain of a variable. So it is possible with the help of the first version to obtain the constraint *X in r* associated to *X*; and to evaluate r in an empty store (to avoid the evaluation of the indexicals of *r*).

The following notations are used.

- X_S = cur_domain(X,S) (i.e. the value of the domain of *X in S*).
- $\min(X)_S = \min(X_S)$.
- $\max(X)_S = \max(X_S)$.
- $r_S = \varepsilon_r [r] S$ (i.e. domain denoted by *r in S*).
- $t_S = \varepsilon_t [r] S$ (i.e. integer denoted by *t in S*).

Some important definitions follows.

✓ **Consistency of a store** **55**

- A store *S* is consistent if, and only if, does not contain any empty domain (i.e. $\forall X \in V_d \; X_S \neq 0$).

✓ **Instance of a variable** **56**

- A variable *X* is instantiated to *n* in the store *S* if, and only if, $X_S = \{n\}$.

✓ **The relation stronger** **57**

- Let S and S' be two sets of constraints, S' is stronger than S ($S' \subseteq S$) if, and only if, $\forall X \in V_d \; X_{S'} \subseteq X_S$.

The *tell* operation needs to be a monotone operation. This ensures the existence of a fix-point because all domains are finite. So it is possible to remove impossible values as soon as they appear and when performing the operation *tell* the reconsideration of accumulated information is avoided.

The operation *tell* of a constraint *X in r* is monotone if the range denoted by r is monotone. In other words, the range can only decrease when there is an addition of more constraints.

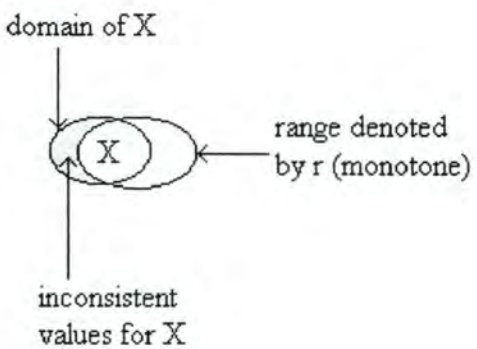
✓ **Monotone in a range** **58**

- A range r is monotone (resp. anti-monotone) if, and only if, $\forall S,S'$ $S' \subseteq S \Rightarrow r_{S'} \subseteq r_S$ (resp. $r_S \subseteq r_{S'}$).

✓ **Monotone in a constraint** **59**

- A constraint $c \equiv X \; in \; r$ is (anti-)monotone if, and only if, r is (anti-)monotone.

The removal of impossible values of X which do not belong to r in a *tell* operation, is accomplished by the intersection operation between X and r. This is schematized in the next figure.



To ensure that r is monotone it is necessary that a *tell* operation in a constraint containing an incorrect indexical term on X be delayed until X is instantiated (see next example). In clp(FD) this is done using a new indexical term *val(X)* which delays the activation of a constraint until is instantiated.

Example 20

'x≠y'(X,Y):- X in -{val(Y)},
Y in -{val(X)}.

3.4 Constraint Systems

A recent formalization has been proposed by V. Saraswat[1] to define constraints. It can be seen as a first-order generalization of Scott's information systems. The emphasis is put on the definition of an entailment relation between constraints, which is enough to define all the constraint system.. This allows to define the constraint system ex nihilo by verifying some properties as well the entailment relationship.

Entailment is a rule based relation that makes possible to define a kind of operational semantics of the entailment between constraints.

✓ **Constraint System** **60**

▫ A constraint system is a pair $(D, \vdash)$ satisfying the following conditions:

❶ D is a set of first-order formulas closed under conjunction and existential quantification.

❷ $\vdash$ is a entailment relation between a finite set of formulas and a single formula satisfying the following inference rules:

$$S, d \vdash d \ \textbf{(Struct)} \qquad \frac{S_1 \vdash d \quad S_2, d \vdash e}{S_1, S_2 \vdash e} \ \textbf{(Cut)}$$

$$\frac{S, d, e \vdash f}{S, d \wedge e \vdash f} \ (\wedge \vdash) \qquad \frac{S \vdash d \quad S \vdash e}{S \vdash d \wedge e} \ (\vdash \wedge)$$

$$\frac{S, d \vdash e}{S, \exists X.d \vdash e} \ (\exists \vdash) \qquad \frac{S \vdash d[t/X]}{S \vdash \exists X.d} \ (\vdash \exists)$$

In $(\exists \vdash)$, X is assumed not free in S,e.

❸ $\vdash$ is generic: that is $S[t/X] \vdash d[t/X]$ whenever $S \vdash d$, for any term t.

A definition of a pre-constraint system $(D, \vdash)$ satisfying only *Struct*, *cut* and the genericity condition, when existential quantification and conjunction are added, is enough to build constraint systems.

[1] In [R 15].

T Let (D', ⊢') be a pre-constraint system. Let *D* be the closure of *D'* under existential quantification and conjunction, and of ⊢' under basic inference rules. Then (D, ⊢) is a constraint system.

The entailment relation is defined next, which proves that Finite Domains is a Constraint System.

✓ **Entailment Relation** **61**

- A store *S* entails a constraint $c \equiv X \text{ in } r$ if, and only if *c* is true in any store *S'* stronger than *S*, i.e.

  $S \vdash c$ if, and only if, $\forall S' \quad S' \subseteq S \Rightarrow X_{S'} \subseteq r_{S'}$

A store *S* disentails a constraint $c \equiv X \text{ in } r$ if, and only if, *S* entails $\neg c$, i.e. $S \vdash X \text{ in } -r$.

(Constr, ⊢) is a pre-constraint system due to the following proposition.

✓ **Proposition** **62**

- ⊢ satisfies (**Struct**), (**Cut**) and is generic.

If *D* is defined as the existential closure and conjunction of *Constr* and using the basic inference rules, ⊢ is defined as the closure of the entailment relation, then *FD*=(D, ⊢) is a constraint system.

The next proposition proves that two constraints are equivalent when they share the same tuples of solutions.

✓ **Equivalence between constraints** **63**

- Two constraints c_1 and c_2 are equivalent if, and only if, $\forall S \quad S \vdash c_1 \Leftrightarrow S \vdash c_2$.

3.4 Clp(FD) Implementation

Implementing clp(FD) requires to study all the alterations to the WAM (in this particular case WAMCC) as well as the introduction of a new data type: FD variables. This type of variables is capable of storing a set of integers. FD variables will be stored

in the *heap* and are distinguished from other data types by a new tag (FDV). The necessary changes to the WAM will slightly affect data manipulation, unification, indexing and trailing instructions.

Data manipulation

The duplication process of a variable to a register for constants is done by copying the constant to the register; whereas unbound variables are bind to the register. A problem occurs in the loading of registers since FD variables can not be duplicated. There are two solutions to solve it:

- Using the same loading process of the WAM (i.e. same copy instruction + self reference); but this scheme has the disadvantage of slowing down the dereferentiation algorithm since it has to deal with a new tag word <FDV,α>, where α is the word's self address.

- In the second solution the algorithm is not modify. So FD variables are not copied; instead a binding is made from the destination word to the FD word.

In clp(FD), the second alternative was chosen since dereferentiation is an operation performed very often. A FD variable < FDV, α > is self referenced because the value α is used to obtain the address of the variable. The associated information with the FD variable follows the tagged word..

Unification

A FD variable X can be unified with the elements present in the next table.

Element	Comment
unbound variable *Z*	*Z* is just bound to *X*
integer *n*	equivalent to *X in n..n*
FD variable *Y*	equivalent to *X in dom(Y)* and *Y in dom(X)*.

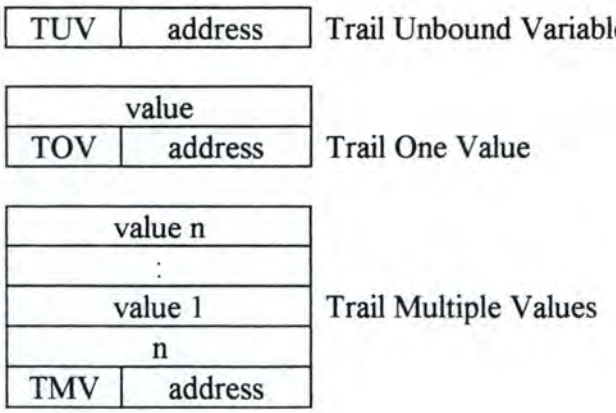
Table 9 : FD unifiable elements

Indexing

FD variables are managed like unbound variables. So all clauses are tried. A more complex indexing could be used based on the current value of the domain.

Trailing

A single entry trail is used in the WAM for unbound variables; however in clp(FD) a multiple entry trail is required to restore some values (min, max, etc.) of the FD variable. Trail frames are a composition of three form types, as shown next.



Applying the WAM standard criterion when trailing FD variables leads to useless trailing everytime a domain is reduced; instead in each choice point only a trailing is required for each FD variable. This requires the creation of a register (STAMP) that gives the number of the current choice points. Each time a choice point is created, the register increases by one and when it is removed the register is decreased. For each FD variable in the choice point, a record is used to store the number of the choice point where the variable was last put in the trail. A FD variable *X* is trailed (inclusive the stamp record) if Stamp(X)≠STAMP.

- How are the *X in r* constraints implemented ?

  - The *X in r* constraints are implemented thanks to a new data structure coupled with an instruction set to compile the constraints.

An execution of the *X in r* constraint is achieved by three operations, which are:

- **Evaluation of *r*** - computing the range *r*.

  To achieve this, the address of the compiled code responsible for the evaluation is stored. It is also necessary to record the context in which *r* must be evaluated since the range depends on some arguments (i.e. indexical terms or parameters). The context is called an environment where argument values, that need to be used by the code that computes the range *r*, are recorded.

- **Modification of X**.

  This operation updates *X* from the previous evaluation operation. The address of the variable *X* needs to be stored.

- **Propagation of the changes** - reexecute all constraints depending on *X*.

  A list of constraints depending on *X* as well as the its domain must be kept for that purpose.

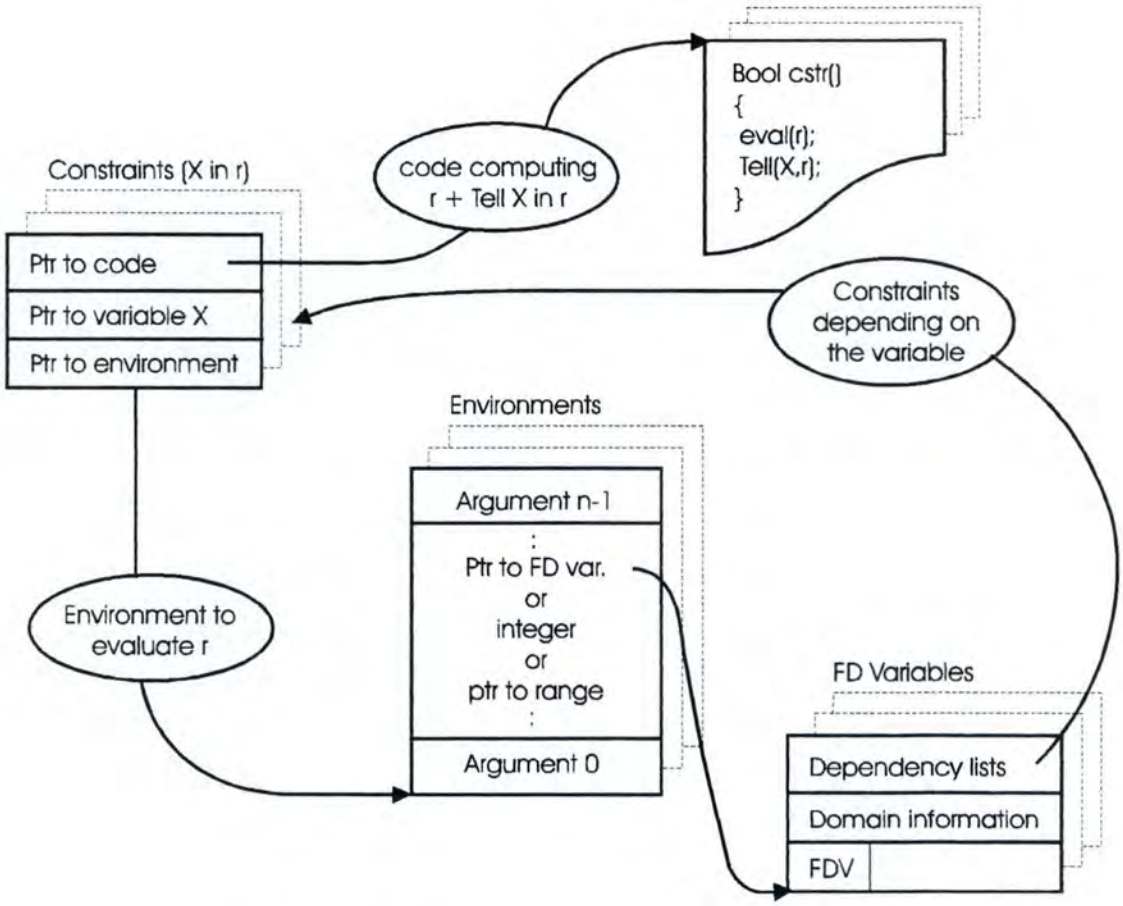


Figure 3 : Data Structures for the X in r constraint

Environments

An environment in which constraints are called are represented by an argument frame (A_Frame) where the address of the FD variables and parameter values are stored. All the constraints defined in a clause share the same A_Frame. The new register *AF* points to the current frame.

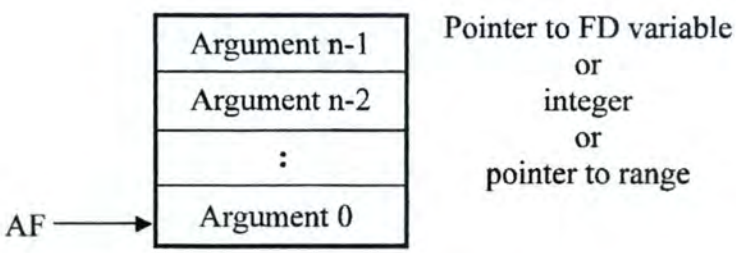


Figure 4 : Argument Frame

Constraints

A constraint frame (C_Frame) is created for each constraint. The following information is recorded inside a C_frame:

- pointer to the associated A_Frame;
- the address of the constrained FD variable;
- the address of the associated code.

The new register *CF* references the current C_Frame.

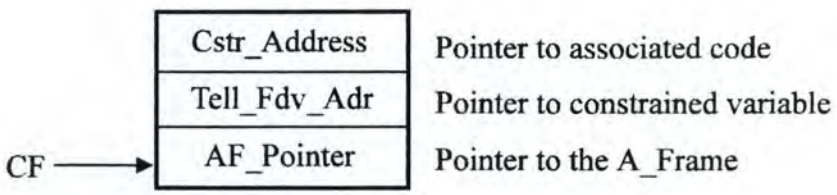


Figure 5 : Constraint Frame

Ranges

Two structures may be used to represent ranges.

- Min-Max. Thanks to the recorded min and max, intervals (included in a 0..*infinity*) are encoded.
- Sparse. Each value of the range can be record in a bit-vector from 0 to *vector-max*[1], where *vector_max* is redefined through an environment variable or through a built-in predicate.

A hole in an interval forces the initial representation (Min-Max) to switch to a sparse representation. In the transformation process from interval to sparse, values can be lost since *vector_max* is less than *infinity*. So a flag (extra_cstr) for a range exist to indicate that information has been lost due to the constrained operation done by the solver (via an imaginary constraint operation *X in* 0..*vector_max*). The user is informed, thanks to the flag, when there is an incompleteness in the solutions due to a variable that has been *extra-constrained*.

[1] Vector_max is by default 127.

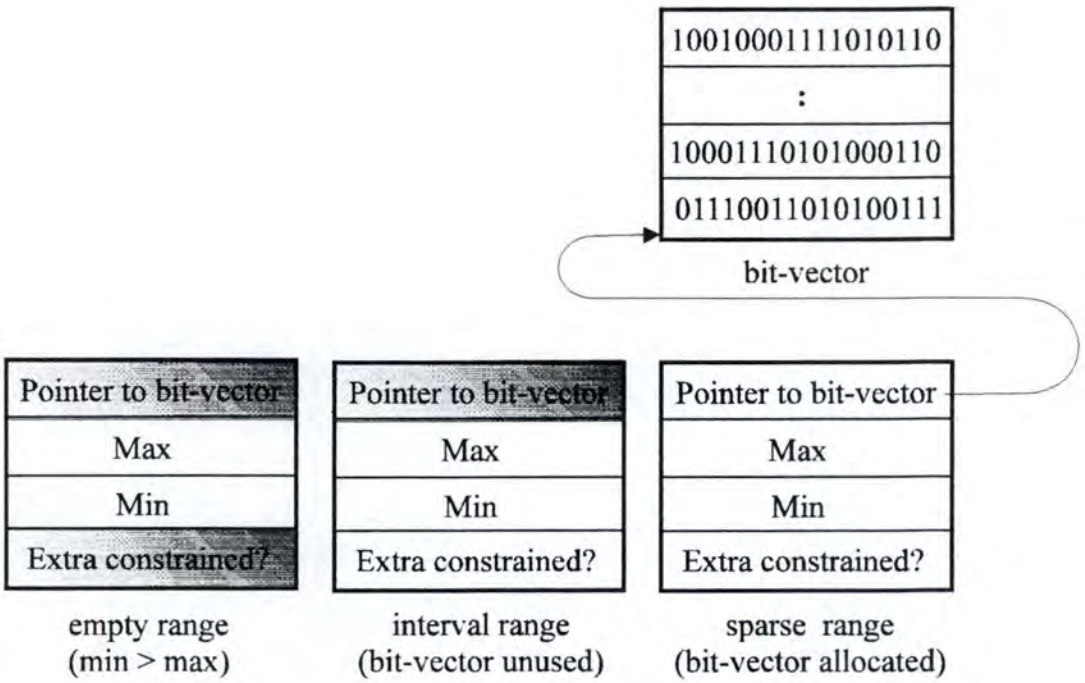


Figure 6 : Representations of a range

An empty range is represented by *min* > *max*.

Using the above notation it is possible to perform the intersection operation between two ranges because the result returned is *min* > *max* when R_1 or R_2 is empty.

Example 21

$\max(\min(R_1), \min(R_2)) .. \min(\max(R_1), \max(R_2))$

⊙ if for instance $R_1 = 0$ and $R_2 = \{1..5\}$ then

$\max(0,1) .. \min(0..5)$ is $\{1..0\}$ (*min* > *max*).

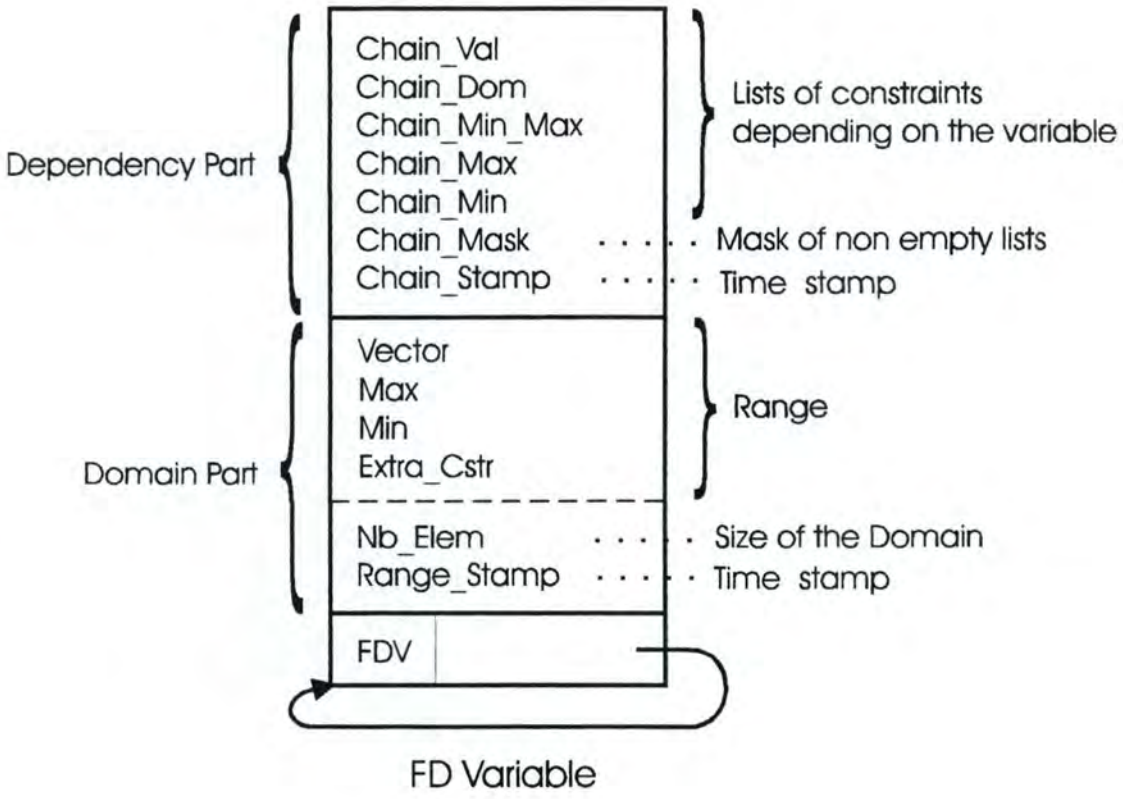


Figure 7 : DF variable frame

FD Variables

A FD variable frame has two main parts:

- the domain recording the range together with the number of elements present in the range;
- the dependency *X* constraints pointers to lists of constraints.

The domain is modified during execution whereas the lists of dependent constraints are created during the installation phase of the compilation. Both parts have their own stamps and it is possible to trail them independently. Several lists of constraints (see table below) are separated to avoid useless propagation.

Name	Definition
Chain_Min	list of constraints depending on *min(X)* and not on *max(X)*
Chain_Max	list of constraints depending on *max(X)* and not on *min(X)*
Chain_Min_Max	list of constraints depending on *min(X)* and on *max(X)*
Chain_Dom	list of constraints depending on *dom(X)*
Chain_Val	list of constraints depending on *val(X)*

Table 10 : Dependency X constraints pointers

Propagation Queue

Awaking and executing a set of constraints, in the propagation phase, adds new constraints to the set. For flexibility reasons, an explicit propagation queue is introduced which is feasible since the order of execution of the constraints is irrelevant. To manage the queue two new registers *BP* and *TP* (pointing to the base and top of the queue, respectively) are added to the system.

Instead of enqueuing all the constraints, a pair < X, mask > is just required; with *X* the variable that is updated and *mask* a bit-mask of the dependency lists to awake.

Registers

New registers are created to handle all the required DF data structures.

Registers	Definition
BP	Base Pointer to the propagation queue
TP	Top Pointer to the propagation queue
AF	Pointer to the current A_Frame
CF	Pointer to the current C_Frame
CC	Continuation after Constraint
STAMP	Choice Point number
T(t)	Term registers
R(r)	Range registers

Table 11 : Clp(FD) Registers

Remarks
The CC register points to the next instruction to execute after the call constraint. T(t) and R(r) are bank registers that contain the *min*, *max* and bit-vector.

Compilation scheme

The compilation of any clause having at least one *X in r* constraint creates the three groups of instructions found in the next table.

Group Name	Purpose
Interface with Prolog clauses	Create and load the A_Frame
Installation code	Install a Constraint
Constraint Code	Generate a procedure for each *X in r* constraint

Table 12 : Clp(FD) instructions type

Remarks
The necessary space for the A_Frame is reserved in the heap and the parameters values are loaded into the frame.
Code is produced to create and load the C_Frame. All the dependent list of constraints are initialized.
The procedure is composed of four parts: ▪ loading parameters, indexical terms and ranges into the appropriate registers. ▪ evaluating the range *r* through a syntactical tree. For each leaf and each node a instruction is created. ▪ Telling the constraint *X in r*. ▪ Returning.

Telling the constraint *X in r*

A constraint is told by the following algorithm.

If *X* is an integer, there are two possibilities:

❶ $X \in r$ ⇨ success

❷ $X \notin r$ ⇨ failure

else (*X* is an FD variable whose range is r_X) let r' be $r \cap r_X$:

- $r' = \varnothing$ ⇨ failure
- $r' = r_X$ (i.e. $r_X \subseteq r$) ⇨ success
- otherwise

  The domain of X is replaced by r' (X possibly becomes instantiated) and some propagation occurs. Since the domain has been modified, some constraints are required to be reexecuted. The current CC must be pushed into the stack to restore it after propagation.

A compiler implementing clp(FD) was developed at INRIA ([4] [10] [11])) in 1991 using the *X in r* language. It translates Prolog to C functions via the WAM. Predicates are translated to C macros. An extension to clp(FD) takes care of Boolean values by making C Boolean functions for the *X in r* constraints.

4.1 Introduction

In response to the great computational power needed to solve some problems, like for instance the *n*-queens problem, parallel systems are used. Parallel systems are grouped in two categories:

- ✧ Distributed systems, with a great number of processors interconnected;
- ✧ Centralized systems, with a limited number of processors (normally 2 to 4) sharing the same memory space and having for communication a common bus.

In distributed architectures at the software level, the communication mechanism used is message passing; instead centralized architectures use semaphores and shared variables for the same purpose.

Concurrent programming languages are needed to fully exploit the power offered by parallel systems. Logic programming languages are natural well suited for parallelism since the goals to succeed can be resolved independently. Our attention will be focused on the shared memory systems because they are easier to implement.

There are several frameworks whose aim is to make shared memory visible in a structured and safe way; the most popular of them all is certainly Linda which covers imperative, functional and logic programming languages. An instantiation of Linda to logic programming was proposed in [7], a combination of blackboards and logic programming which includes a description of basic control mechanisms. Its name: μLog[1].

The μLog framework

The μLog framework contain two categories:

- ✧ Active objects - logic programming goals - the resolution of each goal being interpreted as the behavior of a process.
- ✧ Passive objects - logic programming terms - they act as usual data.

[1] Pronounced as 'myülög' according to the Webster dictionary of pronunciation.

Three operations can be performed on them:

- ✧ Tell - puts objects on the blackboard;
- ✧ Get - removes objects from blackboard;
- ✧ Read - tests the presence of an object on the blackboard.

The above operations are the only possible communication mechanisms. Moreover, processes can not share variables and thus objects are first renamed before performing the operations.

In Active objects, two sorts of processes exist:

- ✧ Foreground processes - created at start up time - that correspond to the parallel resolution of some list of logic programming queries sharing no variables.
- ✧ Background processes - created at run time - that act as daemons on the blackboard.

The whole computation is successful when a successful termination is reached by all foreground processes.

Advantages of Blackboards

The main advantage of blackboards is the modular approach which allows the creation and testing of programs independently, but that will work together depending on the information existing in the blackboards. A common problem in parallel systems is the synchronization and mutual exclusion which can be avoid using blackboards since the *get* and *read* operations suspends execution avoiding in this manner the use of suspension rules. Moreover, mutual exclusion is obtained by the access to blackboards. Furthermore, variables may be shared by arguments regardless of their input and output positions.

In μLog, process goals and clause bodies may involve sequential and parallel composition operators denoted by ; and || respectively.

This framework (μLog) was conceived to be as general as possible so that would be possible to instantiate it with any logic programming language and any language using constraints.

4.2 Language

The μLog language contains the sets introduced in chapter 2:

Svar	Set of variables
Sfunct	Set of functions
Spred	Set of predicates
Sterm	Set of terms
Satom	Set of atoms
Ssubst	Set of substitutions

Table 13 : μLog sets

Several blackboard concepts are explain below.

✓ **Blackboard primitives and goals** **64**

- Blackboard primitives and goals are inductively defined as follows:

  ❶ blackboard primitives are constructs of the form:
  - tellt(t), readt(t), gett(t)
  - tellp(p), readp(p), getp(p)

  where *t* is a term and *p* a atom;

  ❷ any atom and any blackboard primitive is a goal;

  ❸ Δ is the empty goal;

  ❹ If G_1 and G_2 are goals, then $G_1;G_2$ is a goal and $G_1||G_2$ is also a goal.

The initial goals[1] or *igoal*'s, for short, are non-empty lists of goals $[G_1,...,G_m]$ sharing no variables. Programs are set of clauses of the form *H:-G*, where is an atom and G a goal. As seen before, the set of programs is called Sprog.

✓ **Foreground and background processes** **65**

- Foreground and background processes are constructs of the form ←G and ⌟G, respectively, where G is a goal.
- A process is either a foreground or a background process. When there is no concern of the qualification of the process it is represented by the arrow ⇐ followed by the goal.
- The set of processes is subsequently denoted by Sproc.
- The set {←,⌟} of background and foreground process arrows is referred to as Sarrow.

[1] The set of goals and the set of initial goals are referred as Sgoal and Sigoal, respectively.

✓ Other definitions **66**

- lproc - non-empty lists of processes.
- Slproc - Set of non-empty list of processes.
- Sbg - Set of possibly empty processes.

By convention, we denote an initial goal: *ig*
and the associated list of processes: $\overline{ig}$,
considering all of *ig* as foreground processes.

4.3 Operational Semantics

As seen in chapter 2, the operational semantics can be expressed by rules of the form:

$$\frac{\text{Assumptions}}{\text{Conclusion}} \text{ if Conditions}$$

In this transition system the configurations are in the form of triplets.

$< bt, l, \theta >$ where :

- *bt* is a list of terms representing the terms on the blackboard;
- *l* is a lproc representing the (background and foreground) processes currently running in the blackboard;
- θ is a substitution representing the values computed so far.

Some remarks are due.

Despite the fact that *l* is a list there is no order in the selection of processes. The notation *l*[] denotes a list of processes where a place holder has been introduced at some places. So *l*[⇐G] is a list of processes *G* obtained from *l[]* by replacing the place holder by ⇐G.

The notation l_1+l_2 represents the concatenation of two lists.
Sbt represents the set of lists *bt*.

In μLog, transitions occur as a result of the reduction of atoms and blackboard primitives. These reductions are called transitions rules. The reduction of compound goals is made by the use of classical rules of composition. The classical rules here can also be expressed by the transition rules thanks to the notion of contexts.

A context is a reduction of atoms or blackboard primitives selected thanks to the place holder □. The reduced goal is obtained from the goal under consideration by the atoms and blackboard primitives by their corresponding subgoals according to their reduction.

✓ Contexts **67**

- The contexts are functions inductively defined on the goals by the following rules. They are typically represented by the letter *c*, possibly subscripted.

  ❶ A nullary context is associated with any goal. It is represented by the goal and is defined as the constant mapping from $Sgoal^0$ to the goal with the goal as value.

  ❷ □ is a unary context that maps any goal to itself. For any goal G, this application is subsequently referred to as $\square[G]$.

  ❸ If *c* is a n-ary context and if G is a goal, then (c;G) is a n-ary context. Its application is defined as follows, for any goal $G_1,...,G_n$:

  $$(c;G)\ [G_1,...,G_n] = (c[G_1,...,G_n];G).$$

  ❹ If c_1 and c_2 are m-ary and n-ary contexts respectively, then $c_1||c_2$ is an (m+n)-ary context. Its application is defined as follows, for any goal $G_1,...,G_n$:

  $$(c_1||c_2)\ [G_1,...,G_{m+n}] = (c[G_1,...,G_m]) \,||\, (c_2[G_m,...,G_n]).$$

✓ Transition relation **68**

- Define the transition relation $\rightarrow$ as the smallest relation of (Sbt × Slproc × Ssubst) × (Sbt × Slproc × Ssubst) satisfying the following rules. As usual, for the ease of reading, the more suggestive notation $< bt, l, \theta > \rightarrow < bt', l', \theta' >$ is subsequently employed instead of $(bt, l, \theta, bt', l', \theta')$.

Atom reduction	
Rule A	$< bt, l[\Leftarrow[A]], \theta > \rightarrow < bt, l[\Leftarrow[B]], \theta\,\gamma >$ **if**
	{ (H:-B) is a fresh renaming of a clause of P H and $A\theta$ unify with $mgu\ \gamma$.
Comments	The atom A is reduced to the body goal B, inside a process, if $A\theta$ unifies with H with $mgu\ \gamma$.

Tell reduction	
Rule Tt	$< bt, l[\Leftarrow c[tellt(t)]], \theta > \rightarrow < bt+[u], l[\Leftarrow c[\Delta]], \theta >$ **if**
	{ u is a fresh renaming of $t\theta$
Comments	A term u is putted in the blackboard bt. $t\theta$ is renamed to u to ensure that the processes communicate only via the writing and reading of terms on the blackboard and not implicitly by means of shared variables.

Tell reduction	
Rule Tp	$< bt, l[\Leftarrow c[tellp(p)]], \theta > \rightarrow < bt, l[\Leftarrow c[\Delta]]+[\lrcorner q], \theta >$ **if**
	{ q is a fresh renaming of $p\theta$
Comments	A background process p is putted in the list of processes with the computed variables θ. The renaming is needed to prevent the appearance of shared variables.

Read reduction	
Rule Rt	$< bt, l[\Leftarrow c[readt(t)]], \theta > \rightarrow < bt, l[\Leftarrow c[\Delta]], \theta\gamma >$ **if**
	{ $\exists\ v \in bt$: any fresh renaming of v' of v unifies with $t\theta$. γ is the mgu corresponding to the unification of $t\theta$ and some fresh renaming of such a term v.
Comments	The process *readt(t)* checks for the presence of a term t with θ, leaving the blackboard and the list of processes unchanged.

Read reduction	
Rule Rp	$< bt, l[\Leftarrow c[readp(p)]], \theta > \rightarrow < bt, l[\Leftarrow c[\Delta]], \theta\gamma >$ **if**
	{ $\exists\ (\lrcorner A) \in l[\Leftarrow c[readp(p)]]\theta$: any fresh renaming A' of A unifies with $p\theta$. γ is the mgu corresponding to the unification of $p\theta$ and some fresh renaming of such a process A.
Comments	The process *readp(p)* checks for the presence of a process p with θ in the list of processes l.

Get reduction	
Rule Gt	$< bt, l[\Leftarrow c[gett(t)]], \theta > \rightarrow < bt', l[\Leftarrow c[\Delta]], \theta\gamma >$ **if**
	$\exists\, v \in bt$: any fresh renaming of v' of v unifies with $t\theta$. u is such a term v in bt. γ is the mgu corresponding to the unification of $t\theta$ and of some fresh renaming u. bt' is bt where u has been removed
Comments	If the blackboard bt contains a term which unifies with $t\theta$, then the term is removed, leaving the blackboard with a new configuration bt'.

Read reduction	
Rule Rp	$< bt, l[\Leftarrow c[getp(p)]], \theta > \rightarrow < bt, l'[\Leftarrow c[\Delta]], \theta\gamma >$ **if**
	$\exists\, (\hookleftarrow A) \in l[\Leftarrow c[readp(p)]]\theta$: any fresh renaming A' of A unifies with $p\theta$. $\hookleftarrow G$ is such a process $\hookleftarrow A$ in $l[\Leftarrow c[readp(p)]]\theta$. γ is the mgu corresponding to the unification of $p\theta$ and some fresh renaming of G. l' is l where the process corresponding to $\hookleftarrow G$ has been removed.
Comments	If the list of processes l contains a background process A which unifies with $p\theta$, then the process is removed from the list of processes, resulting in this way a new list l'.

The rules $(\Delta;G)$, $(\Delta\|G)$ and $(G\|\Delta)$ are interpreted as G. In any rule, if an occurrence of the arrow "$\Leftarrow$" exist a replacement must be made by the arrow "$\hookleftarrow$" or by the arrow "$\leftarrow$".

A successful computation is one for which all foreground processes have been reduced to the empty conjunction, while some background processes are possibly running on the background.

A derivation relation can be derived directly from the transition system.

✓ **Derivation relation** **69**

- Define the derivation relation $P \vdash ig$ with θ as the following relation on Sprog × Sigoal × Ssubst: for any $P \in$ Sprog, any $ig \in$ Sigoal, any $\theta \in$ Ssubst, $P \vdash ig$ with θ holds if, and only if, there exist $m \geq 0$, $bt_0,\ldots,bt_m \in$ Sbt, $l_0,\ldots,l_m \in$ Slproc and $\theta_0,\ldots,\theta_m \in$ Ssubst such that:

  ❶ $< bt_0, l_0, \theta_0 > \rightarrow \cdots \rightarrow < bt_m, l_m, \theta_m >$
  ❷ $< bt_0, l_0, \theta_0 > = < [], \text{~~ig~~}, \varepsilon >$

❸ $\theta_m = \theta$

❹ l_m is successful terminated

❺ $l_1,\ldots,l_{m-1}$ are not successful terminated.

✓ **Operational semantics** **70**

▫ Define the operational semantics as the following function $\mathcal{O}$: Sprog $\rightarrow$ Sigoal $\rightarrow \mathcal{P}$ (Ssubst): for any $P \in$ Sprog, any $ig \in$ Sigoal, $\mathcal{O}$ (P)(ig) = $\{\theta_{ig} :$ P $\vdash$ ig with $\theta\}$.

4.4 Declarative Semantics

Declarative semantics is concerned with truth, but truth in μlog depends, in general, of the current state of the blackboard. The actions performed in the blackboards are called events. There are three types of events: addition, removal and check (for the presence) of objects; where objects may be either terms or goals. The history of blackboard actions is modeled by sequences of events called blackboard traces.

✓ **Events** **71**

▫ The set of blackboard events is defined as (Sterm $\cup$ Sgoal) $\times$ {+, -, $*$}. Each of them is associated to a partial function which modifies the blackboard (composed of terms and goals) in the above associated way. A blackboard trace is a possibly empty sequence of blackboard events. The empty sequence is referred to as Λ. The set of blackboard traces is referred to as Str.

✓ **Validity of traces** **72**

▫ A trace $a_1.\cdots.a_m$ is valid if, and only if, it is either empty (m=0) or the composition of functions $a_m \circ \cdots \circ a_1$ is defined on the empty blackboard.

📖

Let t_1, t_2 be two traces. Their concatenation is represented by $t_1 \oplus t_2$ and their merge by $t_1 \otimes t_2$.

An interpretation, in μLog, depends on the status of the blackboard and consists of a set of triplets of the form (trace, goal, goal). These triplets describe the transition traces between the first goal to the second.

The set of all interpretations is called *μbase*.

✓ **μbase** 73

- The μbase set is defined as the set $\mathcal{P}$ (ground(Str) × (ground(Sgoal) × (ground(Sgoal)), where for any set *S*, ground(S) denotes the set of all ground instances of *S*. An interpretation is a member of μbase.

Truth is defined with respect to an interpretation and a trace.

✓ **Definition** 74

- Given a trace *t*, an interpretation *I* and a formula *f*, the fact that *f* is true with respect to *t* and *I*, denoted by $t \models_I f$, is defined by the cases below.
  - Formula: $t \models_I f$ if, and only if, $t^0 \models_I f^0$, for any ground instance (t^0, f^0) of (t, f).
  - Ground Goal: $t \models_I G$ if, and only if, $(t, G, \Delta) \in I$.
  - Ground Clause: $t \models_I (H:\text{-}B)$ if, and only if, $t \models_I H$ whenever $t \models_I B$.
  - Ground initial goals: $t \models_I [G_1,...,G_m]$ if, and only if, there exist $t_1,...,t_m$ and $u_1,...,u_m \in$ ground(Str); $p_1,...,p_n, r_1,...,r_n \in$ ground(Sgoal) such that:
    1. $t \models_I G_i$, i=1,...,m;
    2. $(u_i, p_i, r_i) \in I$, i=1,...,m;
    3. $t \in (t_1 \otimes \cdots \otimes t_2) \otimes (p_1^-.u_1.r_1^+) \otimes \cdots \otimes (p_n^-.u_n.r_n^+)$.

Thanks to the previous definition, truth can be directly defined for an interpretation.

✓ **Definition** 75

- Given an interpretation *I* and a formula *f*, the fact that *f* is true with respect to *I*, denoted by $\models_I f$, is defined by the cases below.
  - Set of formulae: $\models_I \{f_1,...,f_n\}$ if, and only if, for any f_i, $\models_I f_i$.
  - Clauses: $\models_I (H:\text{-}B)$ if, and only if, for any $t \in$ Str, $t \models_I (H:\text{-}B)$.

- Initial Goals: $\models_I [G_1,...,G_m]$ if, and only if, there is t valid such that $t \models_I [G_1,...,G_m]$.

The next definition presents the concept of satisfiability of an interpretation.

✓ **Satisfiability of an interpretation** **76**

- Let *Sg* be the set of atoms and blackboard primitives occurring in the bodies of the ground instances of the clauses of *P*. The interpretation *I* satisfies the program *P* if, and only if, the following properties hold.

  - Empty trace: $(\Lambda, G, G) \in I$ for any $G \in$ ground(Sgoal).

  - Transitive closure: if $(t_1, G_1, G_2) \in I$ and $(t_2, G_2, G_3) \in I$, then $(t_1 \oplus t_2, G_1, G_3)$.

  - Ground atom: if (A:-B) is a ground instance of a clause of *P* such that $(t, B, G) \in I$, then $(t, A, G) \in I$.

  - Ground tells, reads, gets: for any tellt(t), tellp(p), readt(t), readp(p), gett(t), getp(p) of *Sg*,

$(t^+, \text{tellt}(t), \Delta) \in I$ $(p^+, \text{tellp}(p), \Delta) \in I$	$(t^*, \text{readt}(t), \Delta) \in I$ $(p^*, \text{readp}(p), \Delta) \in I$	$(t^-, \text{gett}(t), \Delta) \in I$ $(p^-, \text{getp}(p), \Delta) \in I$

  - Ground sequential composition:

    ❶ if $(t, G_1, G_1') \in I$, then $(t, (G_1;G_2), (G_1';G_2')) \in I$, for any $G_2 \in$ Sgoal;

    ❷ if $(t_1, G_1, \Delta) \in I$, and $(t_2, G_2, G_2') \in I$, then $(t_1 + t_2, (G_1;G_2), G_2')$.

  - Ground parallel composition:

    ❶ if $(t, G_1, G_1') \in I$, then $(t, (G_1||G_2), (G_1'||G_2)) \in I$, for any $G_2 \in$ Sgoal;

    ❷ if $(t, G_2, G_2') \in I$, then $(t, (G_1||G_2), (G_1||G_2')) \in I$, for any $G_1 \in$ Sgoal.

Next comes the notion of model and logic consequence.

✓ **Model** **77**

- A model of a set of formulae *S* is an interpretation *I* such that $\models_I S$.

✓ **Logical consequences** **78**

- The initial goal *ig* is a logical consequence of the program *P* if, and only if, every model of *P* is a model of *ig*. This is subsequently denoted by P $\models$ ig.

✓ **Satisfiable model** **79**

- For any program *P*, the set of ground(Str) × ground(Sgoal) × ground(Sgoal) is a satisfiable model of *P*.

✓ **Intersection of satisfiable models** **80**

- Given a set $\{I_m\}_{m \in M}$ of satisfiable models of program *P*, the intersection $\cap_{m \in M} I_m$ is a satisfiable model of program *P*.

✓ **Minimal model** **81**

- Given a program *P*, the minimal model *Mp* is defined as the intersection of all satisfiable models of *P*.

✓ **Proposition** **82**

- For any $P \in$ Sprog, any $ig \in$ Sgoal, one has $P \models ig$ if, and only if, $\models_{Mp} ig$.

The declarative semantics based in the model theory is defined as follows.

✓ **Declarative Model Semantics** **83**

- The declarative model semantics $Decl_m$: Sprog → Sigoal → $\mathcal{P}$ (Ssubst) is defined as the following function: $Decl_m$ (P)(ig)=$\{\theta_{ig}: P \models ig\theta\}$, for any $P \in$ Sprog and any $ig \in$ Sigoal.

The models of a program can be characterized as the prefixed points of a continuous operator Tp:μbase → μbase.

✓ **Immediate consequence operator** **84**

- Let *P* be a program. The immediate consequence operator is defined as the function Tp: μbase → μbase, for any $I \in$ μbase:

Tp(I)={(Λ, G, G): G ∈ ground(Sgoal)}
∪{(t^+, tellt(t), Δ) : tellt(t) ∈ Sg} ∪ {(p^+, tellp(p), Δ) : tellp(p) ∈ Sg}
∪ {(t^*, readt(t), Δ): readt(t) ∈ Sg} ∪ {(p^*, readp(p), Δ): readp(p) ∈ Sg}
∪ {(t^-, gett(t), Δ) : gett(t) ∈ Sg} ∪ {(p^-, getp(p), Δ) : getp(p) ∈ Sg}
∪ {(t, A, G): (A:-B) is a ground instance of a clause in *P*, (t, B, G) ∈ I}
∪ {(t, (G_1;G_2), (G_1';G_2)): (t, G_1, G_1') ∈ I, G_2 ∈ ground(Sgoal) }

$\cup \{(t_1 \oplus t_2, (G_1;G_2), G_2'): (t_1, G_1, \Delta) \in I, (t_2, G_2, G_2') \in I\}$
$\cup \{(t, (G_1||G_2), (G_1'||G_2)): (t, G_1;G_1') \in I, G_2 \in \text{ground(Sgoal)}\}$
$\cup \{(t, (G_1||G_2), (G_1||G_2')): (t, G_2;G_2') \in I, G_1 \in \text{ground(Sgoal)}\}$
$\cup \{(t_1 \oplus t_2, G_1, G_3): (t_1, G_1, G_2) \in I, (t_2, G_2, G_3) \in I\}$.

✓ **Proposition** **85**

- For any program P, $Mp = lfp(Tp) = Tp \uparrow \omega$.

The declarative fixed point semantics is defined as follows, which is identical to the declarative model semantics.

✓ **Declarative fixed point semantics** **86**

- The declarative fixed point semantics $Decl_f : Sprog \rightarrow Sigoal \rightarrow \mathcal{P}(Ssubst)$ is defined as the following function: $Decl_f(P)(ig) = \{\theta_{|ig}: \models_{lfp(Tp)} ig\theta\}$, for any $P \in Sprog$, any $ig \in Sigoal$.

✓ **Proposition** **87**

- For any $P \in Sprog$, any $ig \in Sigoal$, any $\theta \in Ssubst$, $P \models ig\theta$ if, and only if, $\models_{lfp(Tp)} ig\theta$. In particular, $Decl_m = Decl_f$.

4.5 Relating the Operational and Declarative Semantics

The following proposition establishes the relation from the operational to the declarative semantics. Soundness establishes that any successful operational reduction induces logical consequences. The second proposition establishes a relation from the declarative to the operational semantics. Completeness asserts that any substitution that instantiates an initial goal to a logical consequence of a program is less general than a computed answer substitution for the initial goal and the program.

✓ **Soundness** **88**

- For any $P \in Sprog$, any $ig \in Sigoal$, any $\theta \in Ssubst$, if $P \vdash ig$ with θ, then, for any $\gamma \in Ssubst$, $P \models ig\theta\gamma$.

✓ **Completeness** **89**

- For any $P \in Sprog$, any $ig \in Sigoal$, any $\theta \in Ssubst$, if $P \models ig\theta$ then there exists $\gamma \in Ssubst$ such that $ig\gamma \geq ig\theta$ and $P \vdash ig$ with γ.

Part III

Chapter 5 -- Blackboard Client Server Application

5.1 Introduction

Using backboards in a client-server application has only one goal: sharing data among several clients. The blackboard concepts discussed in chapter four were adapted to fulfill this mission thanks to data encapsulation. Each blackboard has its own name and structure. The three classical operations, seen in the previous chapter, can be performed on them: *Tell*, *Get* and *Read*. Since blackboards are just an abstraction for data lying in the server, they are well suited to implement concurrency between programs. Complex programs may be written in more simple programs to take advantage of blackboards. Data calculated in one program is at hand for others that may require it, to continue their own calculations.

Implementing this kind of application requires a communication mechanism - InterProcessing Communication - capable of efficiently handling huge amounts of data since the server must be able to deal with a great number of client requests. From the different mechanisms available only the FIFO was sufficiently spread through the different operating systems and POSIX 1 compliant. Moreover, a typical use for FIFOs is to send data between a client and a server. The usual way to work with FIFOs and that was used in this application, was to use special files created by the server to listen to requests and another for each client. The special files, also called *named pipes*, have a know name and pathname to both server and clients; for example, the server can create a FIFO with the name */tmp/serv1* to listen to requests and others with */tmp/client.xxx*, where *xxx* is the process ID of the client.

To prevent the continuous reading of the FIFO in the server, a signal (SIGUSR2) is sent from the client to awake the server.

Since the server must be implemented as a background process and due to the fact that it may live for a long time, the best way to code it is as a daemon. Daemons do not have access to controlling terminals, so when errors or when the need for information arises the system log must be used to report it.

5.2 Data structures and data communication

The storage of blackboards is done in the server after a *tell* operation performed in the client. This type of action requires several data types in both the client and server. The data structures are described next.

Data structures in the Server

The first structure of data present in the server is obviously needed to store blackboards. It contains the blackboard name, its data size, the type of each component of the blackboard along with its size. They are represented in figure 8.

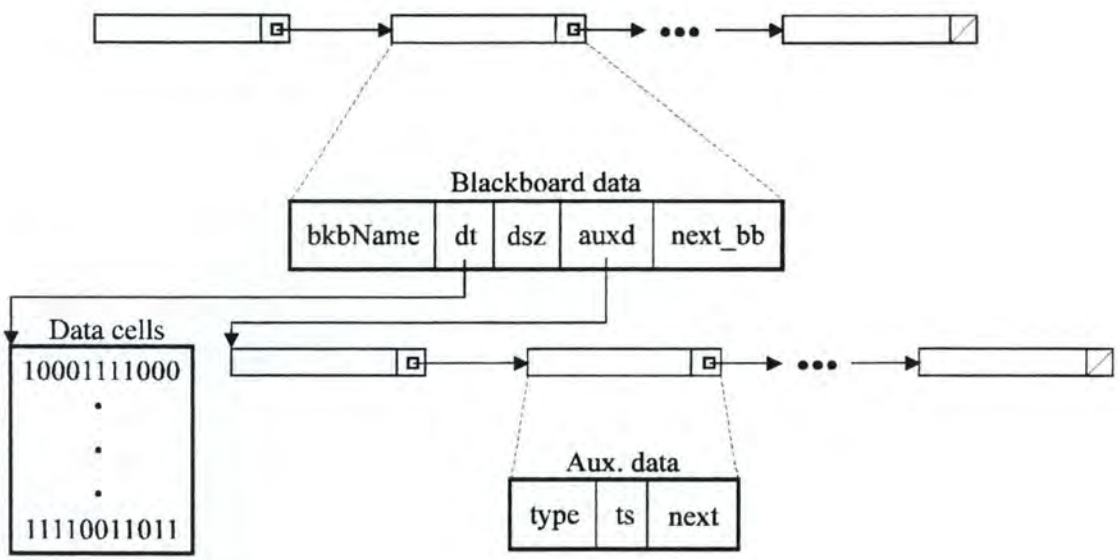


Figure 8 : Blackboard structures

The blackboard data are connected as a linked list containing pointers to both the next element in the list as well as a pointer to the first data cells and another to an auxiliary list. The auxiliary list contains the type and size of each element present in the data cells. The full description of all elements of the blackboard data structure follows.

Blackboard data	
Data name	Description
bkbName	Blackboard name.
dt	Pointer to the memory zone containing the Blackboard data.
dsz	Blackboard data size.
auxd	Pointer to first element of the auxiliary list.
next_bb	Pointer to the next element in the blackboard list.

Auxiliary data	
Data name	Description
type	Type (Number/String) of the corresponding element in the data cells.
ts	Size of the corresponding element in the data cells.
next	Pointer to the next element in the list.

Another structure, show in figure 9, is needed when managing clients since the server must know where to send the answers back to a given request.

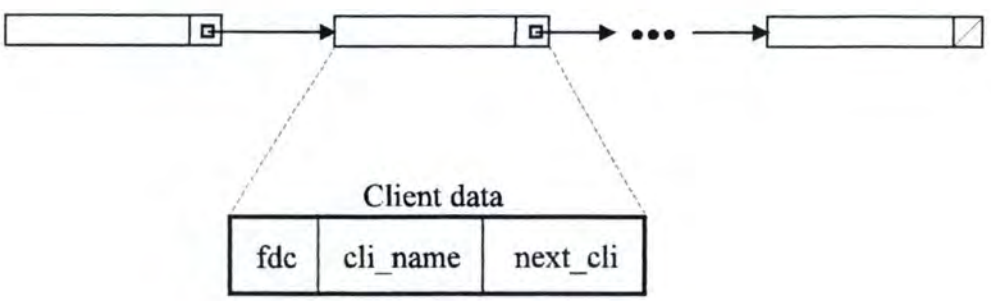


Figure 9 : Client Structure

This simple list contains all necessary information to contact the clients. The structure is composed of three records.

Client data	
Data name	Description
fdc	Client file descriptor.
cli_name	Name of Client.
next_cli	Next element of the client list.

To support both data structures a few records are needed.

Global data		
Data name	Description	Comment
fd	Server descriptor	Used to listen to requests.
fdp	Server auxiliar file descriptor	The aux. file holds server's PID.
buffer[]	FIFO's buffer	Holds the read, write information of the server and client, respectively.
proc_id	Server PID	

Data structure in the Client

The client data structure is similar to the server, only the record names in blackboard changes. Their names are, respectively: bname, bbf, bcsz, bst and bnext.

The blackboard list in the client allows the building of several blackboards inside the client (private blackboards) and when necessary they can be sent to the server; they will be globalized so as to be disponible to any other client. From the list only one is send each time and any blackboard can be send.

There is no counterpart to the client structure of the server. To support the client structures the following records were added to the program.

Client Global Data		
Data name	Description	Comments
buffer[]	Array of characters for FIFO	Buffer to send and receive data to server and from server, respectively.
pid	Client PID	PID used in the creation by the server of client's FIFO.
pids	Server PID	PID used when a signal is send to the server.
cli_name	Name of client	
size_str	Size of buffer	Client buffer current size.
GP	Global Pointer	If needed, points to a memory zone containing the unformated data as well as its atributes: type and size.

Data communication

The communication between server and client requires some communication protocols to be understood by both server and client. The differents protocols are a reflexion of the different purpose aimed by the communication data, as we can see later. All the above operations can succeed or fail; when they succeed the required information is returned whereas the failing returns an error code to the client. The protocol for all operations as well as the returned error code are presented next.

Client Message Format

- Creating Clients

@name_of_client_pid_of_client.

- Destroying Clients

#name_of_client_pid_of client.

- Creating Blackboards

$name_of_blackboard**[**size_of_blackboard**;***data***]**.

where *data* may be either:

- Numbers

... **;**size_of_number_in_bytes**,**number**;** ...

- Strings

... **;***string***;** ...

- Reading blackboards

?name_of_client**[**name_of_blackboard**;***blackboard_structure***]**.

- Getting blackboards

*name_of_client**[**name_of_blackboard**;***blackboard_structure***]**.

where *blackboard_structure* is composed of two connect fields:

... Type|Size ...

being Type the type of element (number or string) and being Size the size of the element in bytes.

Both *data* and *blackboard_structure* may have more than one recorded in which case they are separated by a semicolon.

Server Message Format

The server only sends information when there is a request to *read* or to *get* a blackboard. The message has the following format:

```
?name_of_blackboard{number_of_elements;Element,
first_element; ... ;Element,last_element}.
```

where `Element` is composed of two connected fields:

```
Type_of_element|Size_of_element
```

The server can also send error messages with the following format:

```
Error_code.
```

Error codes and all other messages identifiers can be customized in the *defs.h* header file.

5.3 Integrating the application with user code

The client-server application can be used by any C program or by a program that can call C code. The principle of this application is to integrate the client functions into a user program. Each program is them a client of the server.

To make less dificult using the client functions inside a user program, several macros were added. They are explained next.

- EXPORT

  Macro used to obtain the name of the program (client).

- CREATE_CLIENT

  Sends a request for client creation to the server.

- ADDB(x, y, w, z)

  Macro used to create a blackboard with some information or if the blackboard already exists, then simply add information to it. The macro parameters have the following meaning:

Parameter	Description
x	Blackboard name;
y	Data to put in the blackboard;
w	Type of data (number or string);
z	Size of data to add.

- SEND_BLACKBOARD(*name*)

  Sends the blackboard *name* to the server.

A blackboard may be read by the client that creates it or by another client. The first case usually uses the next macro and the second case must use the next function.

- READB(n, t, w, p)

  The meaning of the macro parameters is as follows.

Parameter	Description
n	Name of blackboard.
t	Type of action to perform - *read* or *get*.
w	Does the reading (getting) of the blackboard suspends the program or not?
p	Address of the pointer that contains the adress of the array of pointers that point to the individual elements of the blackboard.

The action *read, get* (parameter *t*) uses the symbol T_NORMAL and T_KILL, respectively. Parameter *w* requires either T_WAIT or T_NOWAIT. Parameter *p* is used by the client functions to store inside a pointer the address of the structure created. Figure 10 is a scheme of the final structure obtained after a *read* or *get* operation.

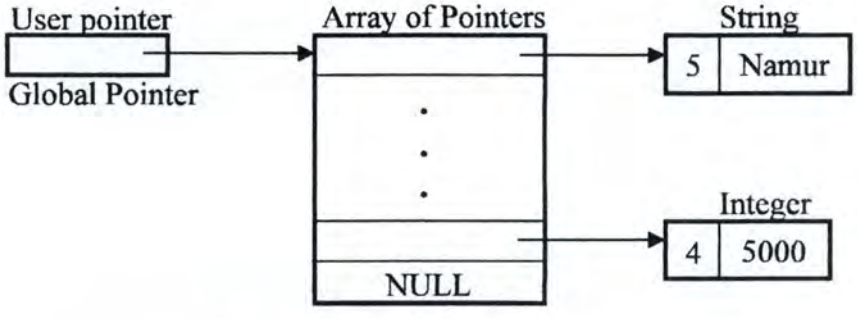


Figure 10 : Ending structure

It is possible to use the last macro with parameter *p* being NULL. A global pointer GP is used instead to store the location of the array of pointers.

The following function is used whenever we wish to *read* or *get* a blackboard. It is defined in uppercase so as to be used like a macro.

- DEF_STR(n, b, t, structure);

where:

- *n* is the number of arguments of the function[1];
- *b* is the name of the blackboard;
- *t* indicates if this action will suspend the program until the server returns an answer;
- *structure* is a set of argument parameters in the form:

... , S_INT or S_STR, ...

- GET_DATA(p, i, d).

The meaning of the macro parameters follows.

Parameter	Description
p	Pointer to array of pointers to data.
i	Number of requested element.
d	Pointer to requested data.

To better understand how all macros work two examples are given.

Example 22

Sending a blackboard

```
# include "client.c"

int main(int argc, char *argv[])
{
int i=007;
EXPORT
CREATE_CLIENT
ADDB("SS", "James", S_STR, strlen("James"))
ADDB("SS", &i, S_INT, sizeof(i))
ADDB("SS", "Bond", S_STR, strlen("Bond"))
SEND_BLACKBOARD("SS")
KILL_CLIENT
return(0);
}
```

[1] The number of arguments are needed because this is a function with a variable number of arguments.

Example 23

Reading and printing a blackboard

```
# include "client.c"

int main(int argc, char *argv[])
{
void ** p;
void * dt;
EXPORT
CREATE_CLIENT
DEF_STR(6, "SS", T_NORMAL, S_STR, S_INT, S_STR);
READ_BLACKBOARD(&p, T_WAIT)
GET_DATA(p, 1, dt)
 printf("\n %s", dt);
GET_DATA(p, 2, dt)
printf("\n %d", Ri[1] (dt));
GET_DATA(p, 3, dt)
printf("\n %s", dt);
KILL_CLIENT
return(0);
}
```

Through the C macros it is possible to use the client-server application in constraint logic programming thanks to clp(FD) which transforms Prolog code into C code. This allows to edit the C code to make the necessary changes to use the client-server application. To support this scheme it was necessary to slightly change two clp(FD) header files by adding new code. Interface functions were created to be called from the clp(FD) C code originated from Prolog code. Figure 11 presents this scheme.

The changes needed to be made in the C code are only two. The first is done inside the *main* function: insert in, the macro EXPORT as the first instruction. The second is made after the line of code that deals with putting values, lists or constants in the heap.

Example 24

```
allocate(1)
put_y_variable(0,0)
call(...)
put_y_value(0,0)
put_in_server("Example")
call(...)
deallocate
proceed
```

[1] Ri( ) is one of the created functions to get numbers through the pointer *dt*; the others are named: Rs, Rl, Rf, Rd, Rld for the types short, long, float, double and long double, respectively.

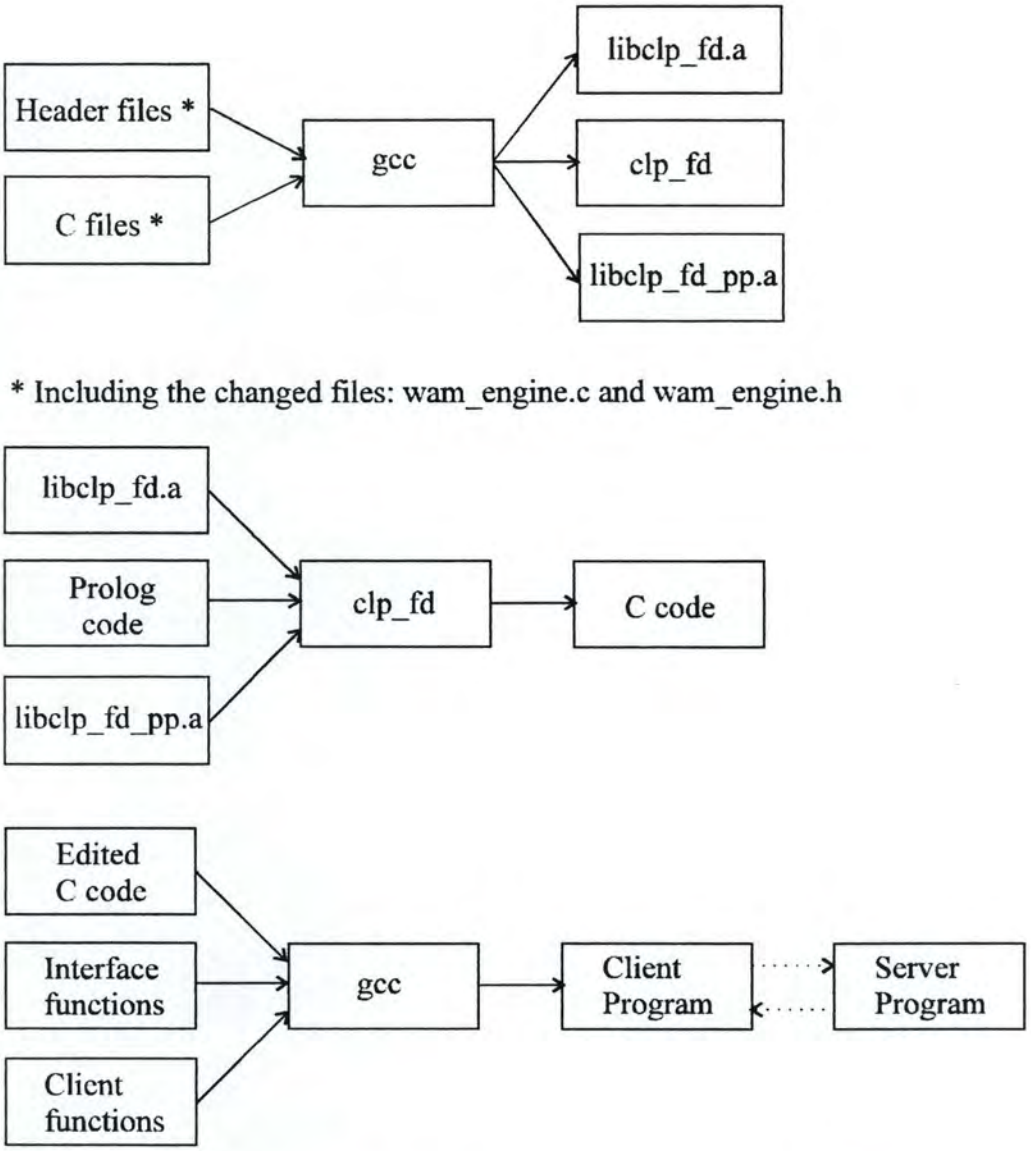


Figure 11 : Linking the Application and clp(FD)

The macro *put_in_server("Example")* puts the value of argument register A0 in the blackboard and sends the blackboard to the server.

5.4 Ending notes

Despite the fact that the application works well in a reliable way several points are worth observing. Due to the nature of the communication channel (FIFO) the overall speed of the application is affected. To increase the performance, other forms of communication are best suited like for instance shared memory, messages queues and semaphores. To increase the power of the application, the clients could be anywhere in the local network requesting information from the server in there machine and if, there were no requested blackboards, then the server could contact the other servers present in the network as seen in figure 12.

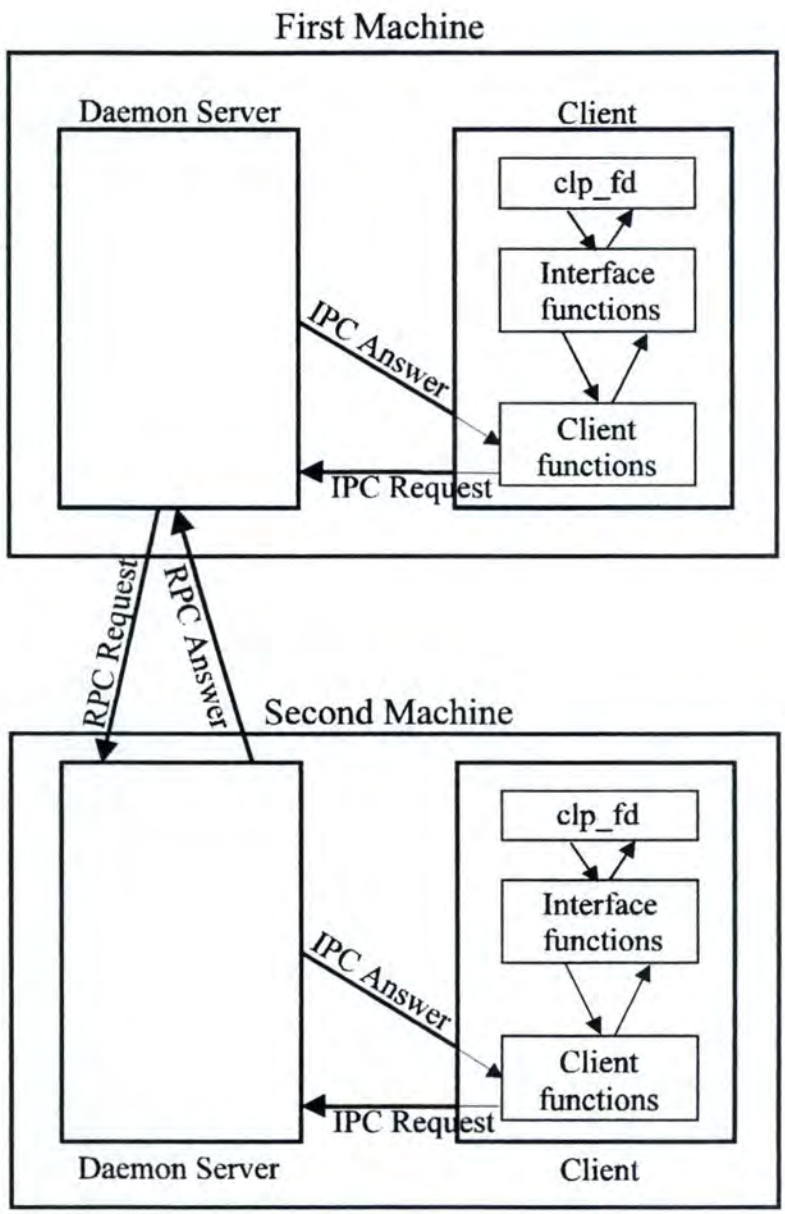


Figure 12 : A possible extension to the application

A powerfull communication mechanism to explore is Remote Procedure Calling (RPC, for short). Another way to improve the application is to increase the data types currently supported by the interface routines.

There is plenty of possibilities to explore in the course of a future work ...

Part IV

Chapter 6 -- Conclusion

6.1 Summary of the work

The end of this type of paper is not complete without a reference to the main aspects that make up the thesis. After the overview on logic programming, with its unification mechanism, the constraint programming is explained. Its main features are the introduction of a new language *X in r* that makes the constraint solver more transparent to programmers. The next presentation step is a new framework - μLog - in concurrent programming. It is caracterized by the use of blackboards as a new communication mechanism for parallel systems.

6.2 Main features

The simplicity in the use of the application, namely when manipulating blackboards, through the use of macros makes the Client Server Application a powerful one. Moreover, there is a real possibility in extending the application into the area of constraint logic programming. To exemplify, functions were created with the purpose of allowing Prolog language, when transformed in C code, to send blackboards to the server. The client-server design of the application centralizes the information, making more easily accessible to any client that may request it.

To prevent the possibility of slowing down other applications, due to the continuous reading of the FIFO by the server, a signal system was implemented. Each time a client has a request, it sends a signal to awake the server.

Flexibility is given by the structure built each time a requested blackboard arrives to the client since each data element can them be treated separately. More flexibility is

provided when the need arises to change both the path and name of the server FIFO and the path of the client FIFO. They can both be changed in a header file.

An effort was put in making the application portable, capable of running on different machines with different operating systems.

6.3 Problems and future work

Problems may arise when the communication protocols and clients are using the same codes; to overcome this situation it is possible to change them in a header file. The application problems are reported in the file created by the system log to that purpose.

Future versions of the application could include a multi-machine system capable of delivering to clients the answers to their requests, independently of the machine in which clients and servers are running.

A brief reference to the complete set is given here. The algorithms used by the instructions goes next. The notation *Vn* represents either a temporary or permanent variable indifferently.

Put instructions

put_variable Xn,Ai
put_variable Yn,Ai
put_value Vn, Ai
put_unsafe_value Yn,Ai
put_structure f,Ai
put_list Ai
put_constant c,Ai

Get instructions

get_variable Vn,Ai
get_value Vn,Ai
get_structure f,Ai
get_list Ai
get_constant c,Ai

Set instructions

set_variable Vn
set_value Vn
set_local_value Vn
set_constant Vn
set_void n

Unify instructions

unify_variable Vn
unify_value Vn
unify_local_value Vn
unify_constant c
unify_void n

Control instructions

allocate
deallocate
call P,N
execute P
proceed

Choice instructions

try_me_else L
retry_me_else L
trust_me
try L
retry L
trust L

Indexing instructions

switch_on_term V,C,L,S
switch_on_constant N,T
switch_on_structure N,T

Cut instructions

neck_cut
get_level Yn
cut Yn

Put instructions

put_variable Xn,Ai

Push a new unbound REF cell onto the heap and copy it into both register Xn and register Ai. Continue execution with following instruction.	HEAP[H] ← <REF,H> Xn ← HEAP[H] Ai ← HEAP[H] H ← H +1 P ← P + *instruction_size*(P)

put_variable Yn,Ai

Initialize the *n*-th stack variable in the current environment to 'unbound' and let Ai point to it. Continue execution with the following instruction.	addr ← E + n +1 STACK[addr] ← <REF,addr> Ai ← STACK[addr] P ← P + *instruction_size*(P)

put_value Vn,Ai

Place the contents of Vn into register Ai. Continue execution with the following instruction.	Ai ← Vn P ← P + *instruction_size*(P)

put_unsafe_value Yn,Ai

If the dereference value of Yn is not an unbound stack variable in the current environment, set Ai to that value.
Otherwise, bind the referenced stack variable to a new unbound variable cell pushed on the heap, and set Ai to point to that cell. Continue execution with the following instruction.
	addr ← `deref`(E+n+1)
	if addr < E
	then Ai ← STORE[addr]
	else
	  begin
	  HEAP[H] ← <REF,H>
	  `bind`(addr,H)
	  Ai ← HEAP[H]
	  H ← H +1
	  end
	P ← P + *instruction_size*(P)

put_structure *f*,Ai

Push a new functor cell containing *f* onto the heap and set register Ai to an STR cell pointing to that functor cell. Continue execution with the following instruction.	HEAP[H] ← *f*/n Ai ← <STR,H> H ← H +1 P ← P + *instruction_size*(P)

put_list Ai

Set register Ai to contain a LIS cell pointing to the current top of the heap. Continue execution with the following instruction.	Ai ← <LIS,H> P ← P + *instruction_size*(P)

put_constant c,Ai

Place a constant cell containing *c* into register Ai. Continue execution with the following instruction.	Ai ← <CON,H> P ← P + *instruction_size*(P)

Get instructions

get_variable Vn,Ai

Place the contents of register Ai into variable Vn. Continue execution with the following instruction.	Vn ← Ai P ← P + *instruction_size*(P)

get_value Vn,Ai

Unify variable Vn and register Ai. Backtrack on failure, otherwise continue execution with following instructions.	`unify`(Vn,Ai)
	if fail
	  then *backtrack*
	  else P ← P + *instruction_size*(P)

get_structure *f*,Ai

If the dereferenced value of register Ai is an unbound variable, then bind that variable to a new STR cell pointing to *f* pushed on the heap and set mode to *write*; otherwise, if it is a STR cell pointing to functor *f*, then set register *S* to the heap address following that functor cell's and set mode to *read*. If it is not a STR cell or if the functor is different than *f*, fail. Backtrack on failure, otherwise continue execution with the following instruction.	addr ← `deref`(Ai) **case** STORE[addr] **of** <REF,_> : HEAP[H]← <STR,H+1> HEAP[H+1] ← *f* `bind`(addr,H) H ← H + 2 *mode* ← write <STR,*a*> : **if** HEAP[*a*] = *f* **then** **begin** S ← *a* +1 *mode* ← read **end** **else** *fail* ← **true** **other** : *fail* ← **true** **endcase** **if** *fail* **then** *backtrack* **else** P ← P + *instruction_size*(P)

get_list Ai

If the dereferenced value of register Ai is an unbound variable, then bind that variable to a new LIS cell pushed on the heap and set mode to *write*; otherwise, if it is a LIS cell, then set register *S* to the heap address it contains and set mode to *read*. If it is not a LIS cell, fail. Bactrack on failure, otherwise continue execution with the following instruction.	addr ← `deref`(Ai) **case** STORE[addr] **of** <REF,_> : HEAP[H]← <LIS,H+1> `bind`(addr,H) H ← H + 1 *mode* ← write <LIS,*a*> : S ← *a* *mode* ← read **other** : *fail* ← **true** **endcase** **if** *fail* **then** *backtrack* **else** P ← P + *instruction_size*(P)

get_constant *c*,Ai

If the dereferenced value of register Ai is an unbound variable, bind that variable to constant *c*. Otherwise, fail if it is not the constant *c*. Bactrack on failure, otherwise continue execution with the following instruction.	addr ← `deref`(Ai) **case** STORE[addr] **of** <REF,_> :STORE[addr]←<CON,*c*> `trail`(addr) <CON,*c*'> : *fail* ← (c ≠ c') **other** : *fail* ← **true** **endcase** **if** *fail* **then** *backtrack* **else** P ← P + *instruction_size*(P)

Set instructions

set_variable Vn

Push a new unbound REF cell onto the heap and copy it into variable Vn. Continue execution with the following instruction.	HEAP[H] ← <REF, H> Vn ← HEAP[H] H ← H + 1 P ← P + *instruction_size*(P)

set_value Vn

Push Vn's value onto the heap. Continue execution with the following instruction.	HEAP[H] ← Vn H ← H + 1 P ← P + *instruction_size*(P)

set_local_value Vn

If the dereferenced value of Vn is an unbound heap variable, push a copy of it onto the heap. If the dereferenced value is an unbound stack address, push a new unbound REF cell onto the heap and bind the stack variable to it. Continue execution with the following instruction.	addr ← `deref`(Vn)
	if addr < H
	then HEAP[H] ← HEAP[addr]
	else
	  begin
	  HEAP[H] ← <REF,H>
	  `bind`(addr,H)
	  end
	H ← H + 1
	P ← P + *instruction_size*(P)

set_constant *c*

Push the constant *c* onto the heap. Continue execution with the following instruction.	HEAP[H] ← <CON, *c*> H ← H + 1 P ← P + *instruction_size*(P)

set_void *n*

Push *n* new unbound REF cells onto the heap. Continue execution with the following instruction.	**for** *i* ← H + n - 1 **do** HEAP[H] ← <REF, *i*> H ← H + 1 P ← P + *instruction_size*(P)

Unify instructions

unify_variable Vn

In *read* mode, place the contents of heap address *S* into variable Vn; in *write* mode, push a new unbound REF cell onto the heap and copy it into Xi.
In either mode, increment *S* by one. Continue execution with the following instruction.
	case *mode* **of**
	  read : Vn ← HEAP[S]
	  write : HEAP[H] ← <REF,H>
	  Vn ← HEAP[H]
	  H ← H + 1
	endcase
	S ← S + 1
	P ← P + *instruction_size*(P)

unify_value Vn

In *read* mode, unify variable Vn and heap address *S*; in *write* mode, push the value of Vn onto the heap. In either mode, increment *S* by one. Backtrack on failure, otherwise continue execution with the following instruction.	**case** *mode* **of** read : `unify`(Vn, S) write : HEAP[H] ← Vn H ← H + 1 **endcase** S ← S + 1 **if** *fail* **then** *backtrack* **else** P ← P + *instruction_size*(P)

unify_local_value Vn

In *read* mode, unify variable Vn and heap address *S*. In *write* mode, if the dereferenced value of Vn is an unbound heap variable, push a copy of it onto the heap. If the dereferenced value is an unbound stack address, push a new unbound REF cell onto the heap and bind the stack variable to it. In either mode, increment *S* by one. Backtrack on failure, otherwise continue execution with the following instruction.	**case** *mode* **of**
	read : `unify`(Vn, S)
	write : addr ← `deref`(Vn)
	if addr < H **then**
	HEAP[H]←HEAP[addr]
	else
	begin
	HEAP[H] ← <REF,H>
	end
	H ← H + 1
	endcase
	S ← S + 1
	if *fail*
	then *backtrack*
	else P ← P + *instruction_size*(P)

unify_constant *c*

In *read* mode, dereference the heap address *S*. If the result is an unbound variable, bind that variable to the constant *c*; otherwise, fail if the result is different than constant *c*. In *write* mode, push the constant *c* onto the heap. Backtrack on failure, otherwise continue execution with the following instruction.	**case** *mode* **of** read : addr ← `deref`(S) **case** STORE[addr] **of** <REF,_> : STORE[addr]←<CON,*c*> `trail`(addr) <CON,*c*'> : *fail* ← (c ≠ c') **other** : *fail* ← **true** **endcase** write : HEAP[H] ← <CON, *c*> H ← H + 1 **endcase** **if** *fail* **then** *backtrack* **else** P ← P + *instruction_size*(P)

unify_void *n*

In *write* mode, push *n* new unbound REF cells onto the heap. In *read* mode, skip the next *n* heap cells starting at location *S*. Continue execution with the following instruction.	**case** *mode* **of** read : S ← S + n write : **for** *i* ← H **to** H + n -1 **do** HEAP[*i*] ← <REF, *i*> H ← H + n **endcase** P ← P + *instruction_size*(P)

Control instructions

allocate

Allocate a new environment on the stack, setting its continuation environment and continuation point fields to current *E* and *CP*, respectively. Continue execution with the following instruction.	**if** E > B **then** newE←E+CODE [STACK[E+1]-1]+2 **else** newB ← B + STACK[B] + 8 STACK[newE] ←E STACK[newE + 1] ←CP E ← newE P ← P + *instruction_size*(P)

deallocate

Remove the environment frame at stack location *E* from the stack by resetting *E* to the value of its *CE* field and the continuation pointer *CP* to the value of its *CP* field. Continue execution with the following instruction.	CP ← STACK[E + 1] E ← STACK[E] P ← P + *instruction_size*(P)

call P,N

If *P* is defined, then save the current choice point's address in *B0* and the value of current continuation in *CP*, and continue execution with instruction labeled *P*, with *N* stack variables remaining in the current environment; otherwise backtrack.	**if** *defined*(P) **then**
	begin
	CP ← P + *instruction_size*(P)
	num_of_args ← *arity*(P)
	B0 ← B
	P ← @(P)
	end
	else *backtrack*

execute P

If *P* is defined, then save the current choice point's address in *B0* and continue execution with instruction labeled *P*; otherwise backtrack.	**if** *defined*(P) **then**
	  begin
	  num_of_args ← *arity*(P)
	  B0 ← B
	  P ← @(P)
	  end
	else *backtrack*

proceed

Continue execution at instruction whose address is indicated by the continuation register *CP*.	P ← CP

Choice instructions

try_me_else L

Allocate a new choice point frame on the stack setting its next clause field to L and the other fields according to the current context, and set *B* to point to it. Continue execution with the following instruction.	**if** E > B **then** newB ← E + CODE[STACK[E + 1] -1]+2 **else** newB ← B + STACK[B] + 8 STACK[newB] ← num_args *n* ← STACK[newB] **for** *i* ← 1 **to** *n* **do** STACK[newB + *i*] ← A*i* STACK[newB + *n* + 1] ← E STACK[newB + *n* + 2] ← CP STACK[newB + *n* + 3] ← B STACK[newB + *n* + 4] ← L STACK[newB + *n* + 5] ← TR STACK[newB + *n* + 6] ← H STACK[newB + *n* + 7] ← B0 B ← newB HB ← H P ← P + *instruction_size*(P)

retry_me_else L

Having backtracked to the current choice point, reset all the necessary information from it and update its next clause field to L. Continue execution with the following instruction.	$n \leftarrow$ STACK[B] **for** $i \leftarrow 1$ **to** n **do** A$i \leftarrow$ STACK[B + i] E $\leftarrow$ STACK[B + n + 1] CP $\leftarrow$ STACK[B + n + 2] STACK[B + n + 4] $\leftarrow$ L `unwind_trail`(STACK[B + n + 5], TR) TR $\leftarrow$ STACK[B + n + 5] H $\leftarrow$ STACK[B + n + 6] HB $\leftarrow$ H P $\leftarrow$ P + *instruction_size*(P)

trust_me

Having backtracked to the current choice point, reset all the necessary information from it, then discard it by resetting B to its predecessor. Continue execution with the following instruction.	$n \leftarrow$ STACK[B] **for** $i \leftarrow 1$ **to** n **do** A$i \leftarrow$ STACK[B + i] E $\leftarrow$ STACK[B + n + 1] CP $\leftarrow$ STACK[B + n + 2] `unwind_trail`(STACK[B + n + 5], TR) TR $\leftarrow$ STACK[B + n + 5] H $\leftarrow$ STACK[B + n + 6] B $\leftarrow$ STACK[B + n + 3] HB $\leftarrow$ STACK[B + n + 6] P $\leftarrow$ P + *instruction_size*(P)

try L

Allocate a new choice point frame on the stack setting its next clause field to the following instruction and the other fields according to the current context, and set B to point to it. Continue execution with instruction labeled L.	**if** E > B **then** newB $\leftarrow$ E + CODE[STACK[E + 1] -1]+2 **else** newB $\leftarrow$ B + STACK[B] + 8 STACK[newB] $\leftarrow$ num_args $n \leftarrow$ STACK[newB] **for** $i \leftarrow 1$ **to** n **do** STACK[newB + i] $\leftarrow$ Ai STACK[newB + n + 1] $\leftarrow$ E STACK[newB + n + 2] $\leftarrow$ CP STACK[newB + n + 3] $\leftarrow$ B STACK[newB + n + 4] $\leftarrow$ P +*instruction_size*(P) STACK[newB + n + 5] $\leftarrow$ TR STACK[newB + n + 6] $\leftarrow$ H STACK[newB + n + 7] $\leftarrow$ B0 B $\leftarrow$ newB HB $\leftarrow$ H P $\leftarrow$L

retry L

Having backtracked to the current choice point, reset all the necessary information from it and update its next clause field to the following instruction. Continue execution with instruction labeled L.	$n \leftarrow$ STACK[B] **for** $i \leftarrow 0$ **to** n -1 **do** A$i \leftarrow$ STACK[B + i] E $\leftarrow$ STACK[B + n + 1] CP $\leftarrow$ STACK[B + n + 2] STACK[B + n + 4] $\leftarrow$ P + *instruction_size*(P) `unwind_trail`(STACK[B + n + 5], TR) TR $\leftarrow$ STACK[B + n + 5] H $\leftarrow$ STACK[B + n + 6] HB $\leftarrow$ H P $\leftarrow$ L

trust L

Having backtracked to the current choice point, reset all the necessary information from it, then discard it by resetting *B* to its predecessor. Continue execution with instruction labeled L.	$n \leftarrow$ STACK[B] **for** $i \leftarrow 1$ **to** n **do** A$i \leftarrow$ STACK[B + i] E $\leftarrow$ STACK[B + n + 1] CP $\leftarrow$ STACK[B + n + 2] `unwind_trail`(STACK[B + n + 5], TR) TR $\leftarrow$ STACK[B + n + 5] H $\leftarrow$ STACK[B + n + 6] B $\leftarrow$ STACK[B + n + 3] HB $\leftarrow$ STACK[B + n + 6] P $\leftarrow$ L

Indexing instructions

switch_on_term V,C,L,S

Jump to instruction labeled, respectively, V, C, L or S, depending on whether the dereferenced value of argument register A1 is variable, a constant, a non-empty list, or a structure, respectively.	**case** STORE[deref(A1)] **of** <REF, _> : P $\leftarrow$ V <CON, _> : P $\leftarrow$ C <LIS, _> : P $\leftarrow$ L <STR, _> : P $\leftarrow$ S **endcase**

switch_on_constant N,T

The dereferenced value of register A1 being a constant, jump to the instruction associated to it in hash-table T of size N. If the constant found in A1 is not one in the table, backtrack.	< tag, val > $\leftarrow$ STORE[`deref`(A1)] < found, inst > $\leftarrow$ get_hash(val,T, N) **if** found **then** P $\leftarrow$ inst **else** *backtrack*

switch_on_structure N,T

The dereferenced value of register A1 being a constant, jump to the instruction associated to it in hash-table T of size N. If the functor of the structure found in A1 is not one in the table, backtrack.	< tag, val > ← STORE[`deref`(A1)]
	< found, inst > ← get_hash(val,T, N)
	if found
	  then P ← inst
	  else *backtrack*

Cut instructions

neck_cut

If there is a choice point after that indicated by B0, discard it and tidy the trail up to that point. Continue execution with following instruction.	**If** B > B0 **then**
	  begin
	  B ← B0
	  `tidy_trail`
	  end
	P ← P + *instruction_size*(P)

get_level Yn

Set Yn to the current value of B0. Continue execution with following instruction.	STACK[B + *n* + 2] ← B0 P ← P + *instruction_size*(P)

cut Yn

Discard all (if any) choice points after that indicated by Yn, and tidy the trail up to that point. Continue execution with following instruction.	**If** B > STACK[B + *n* + 2] **then**
	  begin
	  B ← STACK[B + *n* + 2]
	  `tidy_trail`
	  end
	P ← P + *instruction_size*(P)

WAM ancillary operations

The backtrack operation

```
procedure backtrack
  if B = bottom_of_stack
    then fail_and_exit_program
    else
      begin
        B0 ← STACK[B + STACK[B] + 7]
        P ← STACK[B + STACK[B] + 7]
      end
end backtrack
```

The deref operation

```
function deref(a : address) : address
    begin
      < tag, value > ← STORE[a]
      if (tag = REF) ∧ (value ≠ a)
        then return deref(value)
        else return a
    end deref
```

The bind operation

```
procedure bind(a1, a2 : address)
  < t1 , _ > ← STORE[a1]
  < t2 , _ > ← STORE[a2]
  if (t1 = REF) ∧ ((t2 = REF) ∨ (a2 < a1)) then
    begin
      STORE[a1] ← STORE[a2]
      trail(a1)
    end
  else
    begin
          STORE[a2] ← STORE[a1]
      trail
    end
end bind
```

The trail operation

```
procedure trail(a :address)
  if (a < HB) ∨ ((H < a) ∧ (a < B))  then
    begin
       TRAIL[TR] ← a
       TR ← TR + 1
    end
end trail
```

The unwind_trail operation

```
procedure unwind_trail(a1 , a2 : address)
  for i ← a1 to a2 -1 do
     STORE[TRAIL[i]] ← < REF, TRAIL[i] >
end unwind_trail
```

The tidy_trail operation

```
procedure tidy_trail
  i ← STACK[B + STACK[B] + 5]
  while  i < TR do
    if (TRAIL[i] < HB) ∨ ((H < TRAIL[i]) ∧ (TRAIL[i] < B))
    then i ← i + 1
    else
      begin
         TRAIL[i] ← TRAIL[TR + 1]
         TR ← TR - 1
      end
end tidy_trail
```

The complete unify operation

```
procedure unify(a1 , a2 : address)
   push(a1 , PDL)
   push(a2 , PDL)
   fail ← false
   while ¬(empty(PDL) ∨ fail) do
      begin
         d1 ← deref(pop(PDL))
         d2 ← deref(pop(PDL))
         if d1 ≠ d2 then
            begin
               < t1, v1 > ← STORE[d1]
               < t2, v2 > ← STORE[d2]
         else
            case t2 of
                  REF : bind(d1 , d2)
                  CON : fail ← (t1 ≠ CON) ∨ (v1 ≠ v2)
                  LIS : if t1 ≠ LIS then fail ← true
                        else
                           begin
                              push(v1 , PDL)
                              push(v2 , PDL)
                              push(v1 + 1, PDL)
                              push(v2 + 1, PDL)
                           end
                  STR : if t1 ≠ STR then fail ← true
                        else
                          begin
                             f1/n1 ← STORE[v1]
                             f2/n2 ← STORE[v2]
                             if (f1 ≠ f2) ∨ (n1 ≠ n2) then fail ← true
                             else
                                for i ← 1 to n1 do
                                   begin
                                      push(v1 + i, PDL)
                                      push(v2 + i, PDL)
                                   end
                          end
            endcase
         end
      end
end unify
```

Interfacing with Prolog Clause

These instructions are responsible for creating and loading the A_Frame. Mainly, the space is reserved at the top of the heap, the addresses of FD variables and values of parameters are loaded into this new frame.

Instruction	Purpose
fd_set_AF(nb_arg, Vi)	Reserves space on the top of the heap, for A_Frame, whose size is nb_arg. AF and the Vi variable point to the start of the A_Frame.
fd_variable_in_A_frame(Vj)	Binds Vj to an FD variable created on top of the heap (whose range is 0..infinite). Puts its address into the cell pointed by AF. AF is incremented.
fd_value_in_A_frame	Let w be the dereferenced word of Vj, if it is: ▪ an unbound variable: similar to fd_variable_in_A_frame(w). ▪ an integer: it is pushed on the heap and its address is stored into the cell pointed by AF. AF is incremented. ▪ an FD variable: its address is stored into the cell pointed by AF. AF is incremented.
fd_range_parameter_in_A_frame(Vj)	The dereferenced of Vj must be a list of integers and a corresponding range is created on top of the heap whose address is copied into the cell pointed by AF. AF is incremented.
fd_term_parameter_in_A_frame(Vj)	The dereferenced of Vj must be an integer and its value is copied into the cell pointed by AF. AF is incremented.
fd_install_constraint(install_proc, Vi)	Restores AF with Vi, sets CC to the next instruction and gives control to the install procedure.
fd_call_constraint	Sets CC to the next instruction and gives control to the code of the constraint pointed by CF.

The last two instructions are produced for every constraint.

Installing Constraints

For every constraint, an installation procedure is generated. It is responsible for creating and loading the C_Frame. It also initializes the appropriate chain lists for all FD variables used by this constraint.

Instruction	Purpose
fd_create_C_frame (constraint_proc, tell_fv)	Creates, on top of the heap, a C_Frame associated to the constraint whose code is located at the address *constraint_proc* and whose constrained variable is *tell_fv*. CF points to this C_Frame.
fd_install_ind_min(fv) fd_install_ind_max fd_install_ind_min_max(fv) fd_install_ind_dom(fv) fd_install_dly_val(fv)	These are used when the constraint (currently pointed by CF) uses the min(or max, or both the min and max, etc) of the fv^{th} variable.
fd_proceed	Gives control to the address pointed by CC.

Computing Constraint

For every constraint *X in r* a constraint procedure is generated which is decomposed into four parts:

- loading parameters, indexical terms and ranges into appropriate registers;
- computing the range *r*;
- telling the constraint *X in r*;
- returning.

Loading parameters, indexical terms and ranges

Instruction	Purpose
fd_range_parameter(R(r), fp)	Loads the range pointed by fp^{th} parameter into R(r).
fd_term_parameter(T(t), fp)	Loads the value pointed by fp^{th} parameter into T(t).
fd_ind_min(T(t), fv) fd_ind_max(T(t), fv)	Loads the min, max of the fv^{th} variable into T(t).
fd_ind_min_max($T(t_1)$,$T(t_2)$,fv)	Loads the min and the max of the fv^{th} variable in $T(t_1)$ and $T(t_2)$.
fd_ind_dom(R(r), fv)	Loads the domain (a range) of the fv^{th} variable into R(r).
fd_dly_val(T(t),fv, lab_else)	If the fv^{th} variable is an integer, it is copied in T(t), or else the control is given to the label *lab_else*.

Computing the range

Instruction	Purpose
fd_interval_range(R(r), T(t_1), T(t_2))	
fd_union(R(r), R(r_1))	Execute R(r) ← R(r) ∪ R(r_1).
fd_union(R(r), R(r_1))	Execute R(r) ← R(r) ∩ R(r_1).
fd_compl(R(r))	Execute R(r) ← 0..∝ \ R(r).
fd_compl_of_singleton(R(r), T(t))	Execute R(r) ← 0..∝ \ {T(t)}.
fd_add(R(r), T(t))	Execute R(r) ← R(r) +pointwise T(t).
fd_sub(R(r), T(t))	Execute R(r) ← R(r) - pointwise T(t).
fd_mul(R(r), T(t))	Execute R(r) ← R(r) * pointwise T(t).
fd_floor_div(R(r), T(t))	Execute R(r) ← R(r) / pointwise T(t).
fd_range_copy(R(r), R(r_1))	Execute R(r) ← R(r_1).
fd_integer(T(t), n)	Execute T(t) ← n.
fd_add(T(t), T(t_1))	Execute T(t) ← T(t) + T(t_1).
fd_sub(T(t), T(t_1))	Execute T(t) ← T(t) - T(t_1).
fd_mul(T(t), T(t_1))	Execute T(t) ← T(t) * T(t_1).
fd_floor_div(T(t), T(t_1))	Execute T(t) ← T(t) $\lfloor/\rfloor$ T(t_1).
fd_ceil_div(T(t), T(t_1))	Execute T(t) ← T(t) $\lceil/\rceil$ T(t_1).
fd_term_copy(T(t), T(t_1))	Execute T(t) ← T(t_1).

Telling the constraint X in r

The current constraint is pointed by CF and X can be reached from the C_Frame. So only *r* must be provided to *tell*. In order to optimize the execution two particular cases can be distinguish: X in $t_1..t_2$ and X in r. The complete description of the tell operation has already been given in chapter 3.

Instruction	Purpose
fd_tell_range(R(r))	Tells *X in r* where *r* is a range.
fd_tell_interval(T(t_1), T(t_2))	Tells *X in* $t_1..t_2$ (i.e. *r* is an interval).

The following functions can be used by any client program to communicate with the server.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <errno.h>
#include <signal.h>
#include <stdarg.h>
#include "conv.h"
#include "defs.h"

/* Macros definition */
#define EXPORT strcpy(cli_name,argv[0]);
#define CREATE_CLIENT create_client();
#define KILL_CLIENT   kill_client();
#define ADDB(x,y,w,z) add_to_blackboard(x,y,w,z);
#define SEND_BLACKBOARD(n) send_blackboard(n);
#define FREE_BB(b) free_blackboard(n);
#define READB(n,t,w,p) (t?(read_and_kill_bb(n,w,(void ***) p)):
                        read_bb(n,w,(void ***) p));
#define READ_BLACKBOARD(p,w) (w?(nowaiting_read((void ***) p)):
                                waiting_read((void ***) p));
#define GET_DATA(p,i,d) get_data(p,i,&d);

#define POINT ".\0"
#define USCR  "_\0"
#define AT    "@\0"
#define CS    "#\0"
#define MONEY "$\0"
#define LSB   "[\0"
#define RSB   "]\0"
#define IMS   "?\0"
#define AST   "*\0"
#define BFSZ  size_str=strlen(buffer)
#define BUF   &buffer[size_str]
#define CB(x) BFSZ;strcpy(BUF,x)
#define SZT   sizeof(size_t)
```

```
/* Aux. data */

  struct aux
        {
        char type;
        int  ts;
        struct aux * next;
        };


/* Blackboard Representation */

  struct bkb
        {
        char bname[12];
        void * bbf;
        size_t bcsz;
        struct aux * bst;
        struct bkb * bnext;
        };

  struct bkb * first_out,first_in;


const char Cts=S_BS; /* Constant type separator */
const char Cds=S_DS; /* Constant dec. separator */

/* Functions prototype */

void  DEF_STR(int,...);
static void sig_usr(int);
pid_t  get_server_pid(void);
void  tell_server(size_t);
void  open_server(void);
int  open_client(void);  /* Open client to check it was create by server    */
void open_client1(void); /* Open client to unblock server when it opens ... */
                  /* ... client's FIFO.                              */
void  create_client(void);
void  kill_client(void);
void  send_blackboard(char *);
void  read_and_kill_bb(char *,char, void ***);
void  read_bb(char *,char, void ***);
void  create_bbint(void *,size_t);
void  create_bbstring(void * ,size_t);
void  add_to_blackboard(char * ,void * ,char ,size_t);
void  adds_bb(void *,struct bkb *,size_t);
void  addi_bb(void *,struct bkb *,size_t);
```

```
void   clear_buffer(void);
void   remove_bkb(struct bkb *);
void   remove_aux(struct aux *);
void   remove_all_bkb(void);
void   put_info(struct bkb *);
void   putf_info(char *,char *);
void   waiting_read(void ***);
void   nowaiting_read(void ***);
int    get_info(int);
void   decode_info(void ***);
void   put_bkb(char *, void ***);
size_t get_nelem(char *, struct aux **);

 /* Functions that return an number of special type ( int, float ...) */

short int   Rs(void *);
int         Ri(void *);
long int    Rl(void *);
float       Rf(void *);
double      Rd(void *);
long double Rld(void *);

/* Global Data */

void ** GP;
char   buffer[PIPE_BUF];
int    fd=-1;
pid_t  pid,pids;
char   cli_name[256];
size_t  size_str;

pid_t get_server_pid(void)
{
 int fdp;
 if((fdp=open(SVR_ID, O_RDONLY))!=-1)
    {
    read(fdp,&pids,sizeof(pid));
    close(fdp);
    return(pids);
    }
 else
   return(-1);
}

void tell_server(size_t sz)
{
if(fd>-1)
```

```
  {
  if((write(fd,buffer,sz))!=sz)
     printf("Error while writing to server FIFO\n");
  sleep(2);
  kill(pids,SIGUSR2);
  }
}

void open_server(void)
{
pids=get_server_pid();
if(pids>0)
  {
  fd=open(SVR_FF, O_WRONLY | O_APPEND | O_NONBLOCK);
  }
}

int open_client(void)
{
int fda=-1;
char s[256];
char sa[20];
if(pid>0 && fd>-1)
  {
  strcpy(s,CNT_DIR);
  strcat(s,cli_name);
  strcat(s,USCR);
  itoa(pid,sa);
  strcat(s,sa);
  if((fda=open(s,O_RDONLY | O_NONBLOCK))!=-1)
    {
    close(fda);
    return(1);
    }
  }
return(0);
}

void open_client1(void)
{
int fda=-1;
char s[256];
char sa[20];
if(pid>0 && fd>-1)
  {
  strcpy(s,CNT_DIR);
  strcat(s,cli_name);
```

```
  strcat(s,USCR);
  itoa(pid,sa);
  strcat(s,sa);
 while((fda=open(s,O_RDONLY))==-1) {}
 close(fda);
  }
}

void create_client(void)
{
char s[20];

pid=(int) getpid();
pids=get_server_pid();
strcpy(buffer,AT);
CB(cli_name);        /* Copy to current location in buffer array, cli_name */
CB(USCR);            /* same thing for underscore                          */
itoa(pid,s);         /* put pid of this process in string s                */
CB(s);               /* put string s in current buffer location            */
CB(POINT);           /* same thing for a . (point)                         */
open_server();       /* Is server runing ?                                 */
BFSZ;
tell_server(size_str); /* tell server that this client has a request       */
open_client1();
}

void kill_client(void)
{
char s[20];
if(open_client())
  {
  strcpy(buffer,CS);
  CB(cli_name);
  CB(USCR);
  itoa(pid,s);
  CB(s);
  CB(POINT);
  BFSZ;
  tell_server(size_str);
  remove_all_bkb();
  }
}

void create_bbint(void *pt,size_t sz)
{
size_t a;
if(pt!=NULL && sz!=0)
```

```
 {
 a=sz+(2*sizeof(size_t))+2;
 first_out->bbf=(void *) calloc(1,a);
 first_out->bst=(struct aux *) calloc(1,sizeof(struct aux));
 memcpy(first_out->bbf,&sz,sizeof(size_t));
 memcpy(first_out->bbf+SZT,&Cts,1);
 memcpy(first_out->bbf+SZT+1,&sz,sizeof(size_t));
 memcpy(first_out->bbf+(2*SZT)+1,&Cds,1);
 memcpy(first_out->bbf+(2*SZT)+2,pt,sz);
 (first_out->bst)->type=S_INT;
 (first_out->bst)->ts=sz;
 (first_out->bst)->next=NULL;
 first_out->bcsz=a;
 }
}

void create_bbstring(void *pt,size_t sz)
{
size_t a;
if(pt!=NULL && sz!=0)
 {
  a=sz+1+SZT;
  first_out->bbf=(void *) calloc(1,a);
  first_out->bst=(struct aux *) calloc(1,sizeof(struct aux));
  memcpy(first_out->bbf,&sz,SZT);
  memcpy(first_out->bbf+SZT,&Cts,1);
  memcpy(first_out->bbf+SZT+1,pt,sz);
  (first_out->bst)->type=S_STR;
  (first_out->bst)->ts=sz;
  (first_out->bst)->next=NULL;
  first_out->bcsz=a;
 }
}

void add_to_blackboard(char *np,void *bp,char t,size_t sz)
{
struct bkb *bkp;
int found;
if(first_out==NULL)
 {
  first_out=(struct bkb *) calloc(1,sizeof(struct bkb));
  if(first_out!=NULL && strlen(np)!=0)
   {
    strcpy(first_out->bname,np);
    if(t)
     create_bbint(bp,sz);
    else
```

```
      create_bbstring(bp,sz);
    }
  }
else
  {
  bkp=first_out;
  found=0;
  while(bkp!=NULL && found==0)
      {
      if(strcmp(np,bkp->bname)==0)
        {
        if(t)
          addi_bb(bp,bkp,sz);
        else
          adds_bb(bp,bkp,sz);
        found=1;
        }
      else
        bkp=bkp->bnext;
      }
  if(bkp==NULL) /* or found is false */
    {
    bkp=first_out;
    first_out=(struct bkb *) calloc(1,sizeof(struct bkb));
    if(first_out!=NULL && strlen(np)!=0)
      {
      first_out->bnext=bkp;
      if(t)
        create_bbint(bp,sz);
      else
        create_bbstring(bp,sz);
      }
    else
      first_out=bkp;
    }
  }
}

void adds_bb(void *bp,struct bkb *bkp,size_t sz)
{
void *nb;
size_t t;
struct aux * f;

if(bkp->bbf!=NULL && bp!=NULL && sz!=0)
  {
  f=first_out->bst;
```

```
  while(f->next!=NULL)
      f=f->next;
  f->next=(struct aux *) calloc(1,sizeof(struct aux));
  f=f->next;
  f->type=S_STR;
  f->ts=sz;
  f->next=NULL;
  memcpy(&t,bkp->bbf,SZT);
  t+=sz;
  memcpy(bkp->bbf,&t,SZT);
  nb=(void *) calloc(1,(bkp->bcsz)+sz+1);
  memcpy(nb,bkp->bbf,bkp->bcsz); /* copy the previous contents to new location */
  memcpy(nb+(bkp->bcsz),&Cts,1);   /* add a ; to new location */
  memcpy(nb+(bkp->bcsz)+1,bp,sz); /* add string to new location */
  bkp->bcsz+=sz+1;
  free(bkp->bbf);
  bkp->bbf=nb;
 }
}

void addi_bb(void *bp,struct bkb *bkp,size_t sz)
{
void *nb;
size_t t;
struct aux *f;
if(bkp->bbf!=NULL && bp!=NULL && sz!=0)
 {
  f=first_out->bst;
  while(f->next!=NULL)
     f=f->next;
  f->next=(struct aux *) calloc(1,sizeof(struct aux));
  f=f->next;
  f->type=S_INT;
  f->ts=sz;
  f->next=NULL;
  memcpy(&t,bkp->bbf,sizeof(t));
  t+=sz;
  memcpy(bkp->bbf,&t,SZT);
  nb=(void *) calloc(1,(bkp->bcsz)+SZT+2+sz);
  memcpy(nb,bkp->bbf,bkp->bcsz);
  memcpy(nb+(bkp->bcsz),&Cts,1);
  memcpy(nb+(bkp->bcsz)+1,&sz,SZT);
  memcpy(nb+(bkp->bcsz)+SZT+1,&Cds,1);
  memcpy(nb+(bkp->bcsz)+SZT+2,bp,sz);
  bkp->bcsz+=sz+2+SZT;
  free(bkp->bbf);
  bkp->bbf=nb;
```

```
  }
}

void send_blackboard(char *np)
{
struct bkb *p;
char found;
 if(strlen(np)!=0)
   {
    found=0;
    p=first_out;
    while(p!=NULL && found==0)
        {
         if(strcmp(np,p->bname)==0 && p->bbf!=NULL)
           {
            clear_buffer();
            CB(MONEY);
            CB(p->bname);
            CB(LSB);
            BFSZ;
            size_str=strlen(np)+2;
            memcpy(&buffer[size_str],p->bbf,p->bcsz);
            strcpy(&buffer[p->bcsz+strlen(np)+2],RSB);
            strcpy(&buffer[p->bcsz+strlen(np)+3],POINT);
            found=1;
            size_str=p->bcsz+strlen(np)+4;
            tell_server(size_str);
            free(p->bbf);
           }
         else
           p=p->bnext;
        }
   }
}

void read_and_kill_bb(char *n, char t, void *** mp)
{
if(n!=NULL && strlen(n)!=0 && open_client())
  {
   if(t==T_WAIT)
     {
      putf_info(n,AST);
      tell_server(size_str);
      waiting_read(mp);
     }
   else
     {
```

```
    putf_info(n,AST);
    tell_server(size_str);
    nowaiting_read(mp);
   }
  }
}

void read_bb(char *n, char t, void *** mp)
{
struct bkb *p;
char found;

if(n!=NULL && strlen(n)!=0 && open_client())
  {
  if(t==T_WAIT)
    {
    putf_info(n,IMS);
    tell_server(size_str);
    waiting_read(mp);
    }
  else
    {
    putf_info(n,IMS);
    tell_server(size_str);
    nowaiting_read(mp);
    }
  }
}

void waiting_read(void *** mp)
{
int fdc=-1;
char s[256];
char sa[20];
strcpy(s,CNT_DIR);
strcat(s,cli_name);
strcat(s,USCR);
itoa(pid,sa);
strcat(s,sa);
if((fdc=open(s,O_RDONLY | O_NONBLOCK))>-1)
  {
  sleep(20);
  get_info(fdc);
  decode_info(mp);
  close(fdc);
  }
else
```

```
 fprintf(stderr,"Can't open client's fifo\n");
}

void nowaiting_read(void *** mp)
{
int fdc=-1;
char s[256];
char sa[20];
strcpy(s,CNT_DIR);
strcat(s,cli_name);
strcat(s,USCR);
itoa(pid,sa);
strcat(s,sa);
if((fdc=open(s,O_RDONLY | O_NONBLOCK))>-1)
  {
   errno=0;
   sleep(20);
   get_info(fdc);
   if(errno==0)
     decode_info(mp);
   else
     fprintf(stderr,"Error number:%d\n",errno);
   close(fdc);
  }
else
 fprintf(stderr,"Can't open client's fifo\n");
}

int get_info(int fdc)
{
char c;
ssize_t n=0;
int i=0;
clear_buffer();
do {
    n=read(fdc,&c,1);
    if(n>0)
      {
       buffer[i]=c;
       i++;
      }
   } while(c!=S_END && n>-1 && i<PIPE_BUF);
    if(n==-1)
      fprintf(stderr,"Error while reading fifo\n");
    if(c==S_END && n>0)
      {
       buffer[i-1]=NULLC;
```

```
      return(1);
     }
    else
     {
     buffer[0]=NULLC;
     return(0);
     }
}

void decode_info(void *** mp)
{
switch(buffer[0])
     {
     case S_RB:put_bkb(&buffer[1],mp);
              break;
     case S_KB:put_bkb(&buffer[1],mp);
              break;
     case E_NOB:fprintf(stderr,"The blackboard does not exist\n");
              break;
     case E_CNT:fprintf(stderr,"The client does not exist\n");
              break;
     }
}

void put_bkb(char *buf,void *** mp)
{
int i,a;
size_t sz=0;
char *sa;
size_t tam=0;
char ch;
size_t nb=0;
size_t count=0;
void ** pt;
i=a=0;
while(buf[i]!=S_BD && i<PIPE_BUF) i++;
if(buf[i]==S_BD)
  {
   sa=(char *) calloc(1,i-a+1);
   memcpy(sa,buf,i-a);
   memcpy(&sa[i-a+1],"\0",1);
   free(sa);
   i++;
   memcpy(&nb,&buf[i],SZT);
   i+=SZT;
   pt=(void **) calloc(nb+1,sizeof(void *));
   *(pt+nb)=NULL;
```

```
if(buf[i]==S_BS)
  {
  count=0;
  while(buf[i]!=S_ED && i<PIPE_BUF)
      {
      i++;
      memcpy(&ch,&buf[i],sizeof(char));
      i++;
      memcpy(&tam,&buf[i],SZT);
      i+=SZT;
      if(buf[i]==S_DS)
        {
        a=++i;
        while(buf[i]!=S_BS && buf[i]!=S_ED && i<PIPE_BUF)
            i++;
        if(buf[i]==S_BS || buf[i]==S_ED)
          {
          *(pt+count)=(void *) calloc(tam+SZT+sizeof(char),1);
          if(*(pt+count)!=NULL)
            {
            memcpy(*(pt+count),&ch,sizeof(char));
            memcpy((*(pt+count))+sizeof(char),&tam,SZT);
            memcpy((*(pt+count))+SZT+sizeof(char),&buf[a],i-a);
            count++;
            }
          }
        }
      }
  }
if(buf[i]==S_ED)
  {
  if(mp!=NULL)
    *mp=pt;
  else
    GP=pt;
  }
 else
  {
  i=0;
  while(i<=nb)
      {
      free(*(pt+i));
      i++;
      }
  free(pt);
  }
}
```

```
}

size_t get_nelem(char * sa,struct aux ** pdt)
{
struct bkb *p;
struct aux *ap;
size_t sz=0;
int found=0;
p=first_out;
while(p!=NULL && found==0)
    {
    if(strcmp(sa,p->bname)==0)
      {
       found=1;
       *pdt=p->bst;
       ap=p->bst;
       while(ap!=NULL)
           {
            sz++;
            ap=ap->next;
           }
      }
    else
      p=p->bnext;
  }
return(sz);
}

void putf_info(char *np,char *c)
{
struct bkb *p;
char found;
found=0;
p=first_out;
while(p!=NULL && found==0)
    {
    if(strcmp(np,p->bname)==0 && p->bst!=NULL)
      {
       clear_buffer();
       strcpy(buffer,c);
       put_info(p);
       found=1;
      }
    else
      p=p->bnext;
    }
}
```

```
void put_info(struct bkb *p)
{
struct aux *a;
char s[20];
strcat(buffer,cli_name);
strcat(buffer,USCR);
itoa(pid,s);
strcat(buffer,s);
strcat(buffer,LSB);
strcat(buffer,p->bname);
BFSZ;
memcpy(BUF,&Cts,sizeof(Cts));
size_str++;
a=p->bst;
BFSZ;
while(a!=NULL)
    {
    if(a->type==S_STR || a->type==S_INT)
      {
       memcpy(BUF,&(a->type),sizeof(a->type));
       size_str++;
       memcpy(BUF,&(a->ts),sizeof(a->ts));
       size_str+=sizeof(a->ts);
      }
   a=a->next;
    }
strcpy(BUF,RSB);
size_str++;
strcpy(BUF,POINT);
size_str++;
}

/* removes one blackboard and the needed structures */

void remove_bkb(struct bkb *p)
{
 struct bkb *ax;

ax=first_out;
while(ax->bnext!=NULL && ax->bnext!=p)
    ax=ax->bnext;

if(p==ax->bnext)
  {
  ax->bnext=p->bnext;
  remove_aux(p->bst);
  free(p->bbf);
```

```
  free(p);
 }
}

void remove_aux(struct aux *x)
{
struct aux *a;
a=x;
while(a!=NULL)
    {
    x=x->next;
    free(a);
    a=x;
    }
}

void remove_all_bkb(void)
{
struct bkb *a;
a=first_out;
while(a!=NULL)
    {
    free(a->bbf);
    remove_aux(a->bst);
    a=first_out->bnext;
    free(first_out);
    }
first_out=NULL;
}

void clear_buffer(void)
{
int i;
for(i=0;i<PIPE_BUF;i++)
  buffer[i]=NULLC;
}

void DEF_STR(int n,...)
{
int w,i;
size_t vi;
char * s;
char t,c,spid[20];
va_list vl;

va_start(vl,n);
s=va_arg(vl,char *);
```

```
t=va_arg(vl,char);
clear_buffer();
if(t==T_KILL)
  buffer[0]=S_KB;
else
  buffer[0]=S_RB;
strcat(buffer,cli_name);
strcat(buffer,USCR);
itoa(pid,spid);
strcat(buffer,spid);
BFSZ;
buffer[size_str]=S_BM;
strcat(buffer,s);
BFSZ;
for(i=3;i<=n;i++)
  {
   memcpy(&buffer[size_str],&Cts,1);
   size_str++;
   c=va_arg(vl,char);
   memcpy(&buffer[size_str++],&c,1);
   if(c==S_INT)
     {
      vi=(size_t) va_arg(vl,int);
      memcpy(&buffer[size_str],&vi,sizeof(size_t));
      size_str+=sizeof(size_t);
      i++;
     }
  }
buffer[size_str++]=S_EM;
buffer[size_str++]=S_END;
tell_server(size_str);
va_end(vl);
}

void get_data(void ** mp, size_t ind, void ** data)
{
static char cr;
static short int si;
static int it;
static long int li;
static float ft;
static double de;
static long double ld;
static char * sg;

char tp;
size_t ts=0;
```

```
ind--;
if(mp!=NULL && ind >=0)
 {
  memcpy(&tp,*(mp+ind),sizeof(char));
  memcpy(&ts,*(mp+ind)+sizeof(char),sizeof(size_t));
  if(tp==S_STR && ts>0)
   {
    if(ts==1)
      {
       memcpy(&cr,*(mp+ind)+sizeof(char)+sizeof(size_t),sizeof(char));
       *data=(void *) (&cr);
      }
    else
      {
       if(ts>1)
         {
          sg=(char *) ((*(mp+ind))+sizeof(char)+sizeof(size_t));
          *data=(void *) sg;
         }
       else
         *data=(void *) NULL;
      }
   }
  else
   {
    if(tp==S_INT && ts>0)
      {
       switch(ts)
         {
          case sizeof(short int):
              memcpy(&si,*(mp+ind)+sizeof(char)+SZT,sizeof(int));
              *data=(void *) (&si);
              break;
          case sizeof(int):
              memcpy(&it,*(mp+ind)+sizeof(char)+SZT,sizeof(int));
              *data=(void *) (&it);
              break;
/*        case sizeof(long int):
              memcpy(&li,*(mp+ind)+sizeof(char)+SZT,sizeof(long int));
              *data=(void *) (&li);
              break;
          case sizeof(float):
              memcpy(&ft,*(mp+ind)+sizeof(char)+SZT,sizeof(float));
              *data=(void *) (&ft);
              break;
*/
```

```
            case sizeof(double):
                memcpy(&de,*(mp+ind)+sizeof(char)+SZT,sizeof(double));
                *data=(void *) (&de);
                break;
            case sizeof(long double):
                memcpy(&ld,*(mp+ind)+sizeof(char)+SZT,sizeof(long double));
                *data=(void *) (&ld);
                break;
          }
        }
      else
        *data=(void *) NULL;
    }
  }
}


short int Rs(void * pt)
{
short int s;
memcpy(&s,pt,sizeof(short int));
return(s);
}

int Ri(void * pt)
{
int i;
memcpy(&i,pt,sizeof(int));
return(i);
}

long int Rl(void * pt)
{
long int l;
memcpy(&l,pt,sizeof(long int));
return(l);
}

float Rf(void * pt)
{
float f;
memcpy(&f,pt,sizeof(float));
return(f);
}

double Rd(void * pt)
{
```

```
int d;
memcpy(&d,pt,sizeof(double));
return(d);
}

long double Rld(void * pt)
{
int ld;
memcpy(&ld,pt,sizeof(long double));
return(ld);
}
```

Appendix D - Server functions

The following functions are responsible for the management of clients and blackboards.

```
/* Needed header files */

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <syslog.h>
#include <limits.h>
#include <string.h>
#include "m_signal.h"
#include "conv.h"
#include "defs.h"

/* Functions prototype */

int    daemon_init(void);
void   error_log(int ,char *);
void   error_logint(int);
static void sig_usr(int);
static void sig_term(int);
void   creat_fifo(void);
void   read_fifo(char *);
void   daemon_install(void);
void   create_client(char *);
void   decode_info(char *);
int    decode(char *);
int    mem_client(char *);
void   destroy_client(char *);
void   create_blackboard(char *);
void   read_kill_blackboard(char *);
void   read_blackboard(char *);
void   clear_buffer();
void   remove_blackboard(char *);
void   send_error_to_client(char *, char );
```

```
/* Global data definition */

int fd;  /* File server's descriptor witch is used to listen to resquest */
int fdp; /* File holding daemon's pid  */
char buffer[PIPE_BUF]; /* Buffer for FIFO */
pid_t proc_id;

 /* Constant data */

 const char Cbs=S_BS;
 const char Cds=S_DS;
 const char Cim=S_RB;
 const char Cbd=S_BD;
 const char Cend=S_END;
 const char Ced=S_ED;

  /* List of clients */

    struct list_cli
         {
          int fdc;
          char cli_name[MaxBBname+3];
          struct list_cli * next_cli;
         };

    struct list_cli *first_cli; /* pointer to first client list member */

  /* Aux. data */

   struct aux
        {
         char type;
         int  ts;
         struct aux * next;
        };

  /* Blackboard Data Representation */

    struct list_bb
         {
          char   bkbName[12];
          void * dt;
          size_t dsz;
          struct aux * auxd;
          struct list_bb * next_bb;
         };
```

```
    struct list_bb * first_bb; /* pointer to first backboard list member */

void   remove_all_aux(struct aux *);
int    cmp_structure(struct aux *,char *);
void   sendb_client(char *, struct list_bb *);
void   put_answer_in_fifo(int, struct list_bb *);
size_t get_nelem(struct aux *);

/* Daemon self installation function */

int daemon_init(void)
{
pid_t pid;

if((pid=fork())<0)
  return(-1);
else
  if(pid!=0)
    exit(0); /* The father exits */

/* The son goes on */

setsid();

if((pid=fork())<0)
   return(-1);
else
  if(pid!=0)
    exit(0); /* The son exits */

/* The son's son continues */

proc_id=getpid();
if((fdp=open(SVR_ID, O_WRONLY | O_CREAT | O_TRUNC,
S_IWUSR|S_IRUSR))==-1)
  {
   error_log(2,"Can't create file with Daemon's Pid");
   exit(1);
  }
else
  {
   if(write(fdp,&proc_id,sizeof(proc_id))==-1)
     {
      error_log(2,"Can't write PID to file");
      exit(0);
     }
  }
```

```
openlog("wd",LOG_NDELAY,LOG_USER);
chdir("/");
umask(0);
return(0);
}

/* Error handling routine */

void error_log(int priority,char * error_message)
{
 switch(priority)
      {
      case 0: /* Information */
           syslog(LOG_INFO,"Information message - %s.",error_message);
           break;
      case 1: /* Warning     */
           syslog(LOG_WARNING,"Warning message - %s.",error_message);
           break;
      case 2: /* Error       */
           syslog(LOG_ERR,"Error message - %s.",error_message);
           break;
      }
}

void error_logint(int i)
{
char s[20];
itoa(i,s);
error_log(0,s);
}

void creat_fifo(void)
{
if(mkfifo(SVR_FF, S_IRUSR | S_IWOTH | S_IWGRP | S_IWUSR)==0)
  {
  if((fd=open(SVR_FF, O_RDONLY))==-1)
      error_log(2,"Can't open fifo file");
  }
else
   error_log(2,"Can't make fifo file");
}

void read_fifo(char * buf)
{
char c;
ssize_t n=0;
int i=0;
```

```
  do{
    n=read(fd,&c,1);
    if(n==-1)
      error_log(1,"Error while reading");
    else
      {
      buf[i]=c;
      i++;
      }
  } while(c!='.' && n>0 && i<PIPE_BUF);
   if(c=='.' && n>0 && i<PIPE_BUF)
     {
     buf[i-1]='\0';
     decode_info(buf);
     clear_buffer();
     i=0;
     }
   else
     buf[0]='\0';

}

void decode_info(char *buf)
{
switch(buf[0])
     {
     case S_CC:if(!mem_client(&buf[1])) create_client(&buf[1]);
               else error_log(0,"Client is already know to server");
               break;
     case S_KC:destroy_client(&buf[1]);
               break;
     case S_CB:create_blackboard(&buf[1]);
               break;
     case S_KB:read_kill_blackboard(&buf[1]);
               break;
     case S_RB:read_blackboard(&buf[1]);
               break;
     }
}

void read_kill_blackboard(char *buf)
{
int i=0;
int a=0;
int f_test=0;
int f_test1=0;
char *s;
```

```
char sc[12];
s=(char *) calloc(1,MaxBBname+3);
if(first_bb!=NULL)
  {
  while(buf[i]!=S_BM && buf[i]!=NULLC && i<PIPE_BUF) i++;
  if(buf[i]==S_BM)
    {
    strncpy(sc,buf,i);
    if(mem_client(sc)) /* Is there such a client ? */
      f_test1=1;
    else
      f_test1=0;
    if(buf[i]==S_BM)
      {
      a=i;
      while(buf[i]!=S_BS && buf[i]!=NULLC && i<PIPE_BUF) i++;
      if(buf[i]==S_BS)
        {
        strncpy(s,&buf[a],i-a);
        f_test=1;
        }
      else
        f_test=0;
      }
    }
  }

read_blackboard(buf);
if(f_test1)
  remove_blackboard(s);
else
  {
  if(f_test)
    send_error_to_client(sc,E_NOB);
  error_log(1,"Can't remove blackboard");
  }
free(s);
}

void read_blackboard(char *buf)
{
struct list_bb * aux;
struct list_bb * x;
int i=0;
int a=0;
int f_test=0;
int found=0;
```

```
char *s;
s=(char *) calloc(1,MaxBBname+3);
aux=first_bb;
x=NULL;
if(first_bb!=NULL)
 {
  while(buf[i]!=S_BM && buf[i]!=NULLC && i<PIPE_BUF) i++;
  if(buf[i]==S_BM)
    {
     strncpy(s,buf,i);
     if(mem_client(s)) /* Is there such a client ? */
       f_test=1;
     else
       f_test=0;
    }
  a=++i;
  while(f_test && buf[i]!=S_BS && buf[i]!=NULLC && i<PIPE_BUF) i++;
  if(buf[i]==S_BS)
    {
     while(aux->bkbName!=NULL && found==0)
         {
          if(strncmp(&buf[a],aux->bkbName,i-a)==0)
            {
             if(cmp_structure(aux->auxd,&buf[++i]))
               {
                x=aux;
                f_test=1;
               }
             else
               {
                error_log(0,"Structure not equal");
                send_error_to_client(s,E_NOB);
                f_test=0;
               }
             found=1;
            }
          else
           aux=aux->next_bb;
         }
    }
  else
    f_test=0;
 }

if(f_test)
  sendb_client(s,x);
free(s);
```

```
}

void sendb_client(char *cname, struct list_bb *pt)
{
struct list_cli *p;
p=first_cli;
while(p!=NULL)
   {
   if(strcmp(cname,p->cli_name)==0)
    {
     put_answer_in_fifo(p->fdc,pt);
     return;
    }
   p=p->next_cli;
   }
}

void put_answer_in_fifo(int fdc, struct list_bb *pt)
{
struct aux *x;
size_t n=0;
size_t scale;
size_t sz=0;
int i;
clear_buffer();
memcpy(buffer,&Cim,1);
n=strlen(pt->bkbName);
memcpy(&buffer[++sz],pt->bkbName,n);
sz+=n;
memcpy(&buffer[sz],&Cbd,1);
sz++;
n=get_nelem(pt->auxd);
memcpy(&buffer[sz],&n,sizeof(size_t));
sz+=sizeof(size_t);
x=pt->auxd;
scale=0;
do {
   memcpy(&buffer[sz],&Cbs,1);
   sz++;
   memcpy(&buffer[sz],&(x->type),sizeof(char));
   sz++;
   memcpy(&buffer[sz],&(x->ts),sizeof(size_t));
   sz+=sizeof(size_t);
   memcpy(&buffer[sz],&Cds,1);
   sz++;
   memcpy(&buffer[sz],(pt->dt)+scale,x->ts);
   scale+=(x->ts);
```

```
    sz+=(x->ts);
    x=x->next;
  } while(x!=NULL);
memcpy(&buffer[sz++],&Ced,1);
memcpy(&buffer[sz++],&Cend,1);
if(write(fdc,buffer,sz)!=sz)
  error_log(1,"Can't write to client's fifo");
}

size_t get_nelem(struct aux *p)
{
int count;
count=0;
while(p!=NULL)
    {
     p=p->next;
     count++;
    }
return count;
}

int cmp_structure(struct aux *p,char *buf)
{
int r=0;
int ok=1;
size_t t=0;
if(p==NULL)
  return(0);
while(p!=NULL && ok && buf[r]!=S_EM)
    {
     if(strncmp(&buf[r],&(p->type),1)==0)
       {
        ok=1;
        if(p->type==S_INT)
          {
           r++;
           memcpy(&t,&buf[r],sizeof(p->ts));
           if(t=p->ts)
             {
              ok=1;
              r+=sizeof(p->ts);
             }
           else
             ok=0;
          }
        else
          r++;
```

```
      if(buf[r]==S_BS)
        r++;
      if(ok)
        p=p->next;
      }
    else
      ok=0;
   }
if(p==NULL && ok && buf[r]==S_EM)
  return(1);
else
  return(0);
}

void send_error_to_client(char *s, char error)
{
struct list_cli *b;
char c[2];
c[0]=error;
c[1]=S_END;
b=first_cli;
while(b!=NULL)
   {
   if(strcmp(b->cli_name,s)==0)
     {
     write(b->fdc,c,2);
     return;
     }
   b=b->next_cli;
   }
}

void remove_blackboard(char *nclient)
{
struct list_bb *b;
struct list_bb *a;
b=first_bb;
while(b!=NULL && nclient!=NULL && strlen(nclient)!=0)
   {
   a=b;
   if(strcmp(nclient,b->bkbName)==0)
     {
     if(b==first_bb)
       first_bb=NULL;
     else
       a->next_bb=b->next_bb;
     free(b->dt);
```

```
        remove_all_aux(b->auxd);
        free(b);
        return;
       }
      else
       b=b->next_bb;
    }
}

/* Appends a new client to the client's list */

void create_client(char *buf)
{
char s[50];
int fda;
struct list_cli *p;

strcpy(s,CNT_DIR);
strcat(s,buf);
if(mkfifo(s,S_IRUSR | S_IROTH | S_IRGRP | S_IWUSR)==0)
  {
  if((fda=open(s,O_WRONLY | O_CREAT | O_TRUNC))==-1)
    {
    strcpy(s,"Can't create file for client - ");
    strcat(s,buf);
    error_log(2,s);
    }
  else
    {
    if(first_cli==NULL) /* First client ? */
      {
      first_cli=(struct list_cli *) calloc(1,sizeof(struct list_cli));
      p=first_cli;
      }
    else            /* There are others */
      {
      p=first_cli;
      while(p->next_cli!=NULL)
          p=(p->next_cli);
      p->next_cli=(struct list_cli *) calloc(1,sizeof(struct list_cli));
      p=p->next_cli;
      }
    strcpy(p->cli_name,buf); /* copy client's name to client's list */
    p->fdc=fda;          /* same thing for file descriptor     */
    }
  }
else
```

```
 error_log(1,"Can't make fifo for client");
}

/* Search memory to see if a client has already been created */

int mem_client(char *buf)  /* Is client in memory ? */
{
struct list_cli *p;
p=first_cli;
while(p!=NULL)
   {
   if(strcmp(buf,p->cli_name)==0)
     return(1);
   p=p->next_cli;
   }
return(0);
}


void destroy_client(char *buf)
{
struct list_cli *p;
struct list_cli *a;
char s[50];

p=first_cli;
a=NULL;
while(p!=NULL)
    {
    if(strcmp(buf,p->cli_name)==0)
      {
       error_log(0,"Found Client, removing...");
       strcpy(s,CNT_DIR);
       strcat(s,p->cli_name);
       if(p==first_cli)
         {
          free(p);
          first_cli=NULL;
         }
       else
         {
          a->next_cli=p->next_cli;
          free(p);
         }
       if(unlink(s)<0)
         {
          strcpy(s,"Can't remove file of client ");
```

```
          strcat(s,buf);
          error_log(1,s);
         }
        return;
       }
     a=p;
     p=p->next_cli;
    }
 error_log(0,"Can't find client");
}

void create_blackboard(char *buf)
{
if(decode(buf)==0)
  error_log(1,"Can't create blackboard");

}

int decode(char *buf)
{
struct list_bb * aux;
int i=0;
int a=0;
int scale;
size_t t_size;
size_t bb_size;
int f_test=0;
size_t auxdata=0;
struct aux * ax;
struct aux * bx;
aux=first_bb;
bx=NULL;
first_bb=(struct list_bb *) calloc(1,sizeof(struct list_bb));
if(first_bb!=NULL)
  {
   f_test=1;
   while(buf[i]!=S_BM && buf[i]!=NULLC && i<PIPE_BUF) i++;
   if(buf[i]==S_BM)
     {
      if(i+1>MaxBBname)
        strncpy(first_bb->bkbName,buf,MaxBBname);
      else
        strncpy(first_bb->bkbName,buf,i);
      f_test=1;
      a=++i;
      while(buf[i]!=S_BS && i<PIPE_BUF) i++;
      if(buf[i]==S_BS) /* Get blackboard size */
```

```
    {
     memcpy(&bb_size,&buf[a],sizeof(size_t));
     first_bb->dt=calloc(1,bb_size);
     if(first_bb->dt==NULL)
       f_test=0;
     }
    else
     f_test=0;
    scale=0;
    bx=NULL;
    while(buf[i]!=S_EM && f_test && i<PIPE_BUF)
     {
     a=++i;
     while(buf[i]!=S_DS && buf[i]!=S_BS && buf[i]!=S_EM && i<PIPE_BUF)
       i++;
     if(buf[i]==S_BS || buf[i]==S_EM) /* String to copy */
       {
       ax=(struct aux *) calloc(1,sizeof(struct aux));
       memcpy((first_bb->dt+scale),(void *) (&buf[a]),i-a);
       ax->ts=i-a;
       ax->type=S_STR;
       scale=scale+i-a;
       if(first_bb->auxd==NULL)
         first_bb->auxd=ax;
       else
         bx->next=ax;
       bx=ax;
       }
     if(buf[i]==S_DS) /* Number to copy */
       {
       ax=(struct aux *) calloc(1,sizeof(struct aux));
       memcpy(&t_size,&buf[a],sizeof(size_t));
       ax->type=S_INT;
       ax->ts=t_size;
       memcpy((first_bb->dt+scale),(void *) (&buf[++i]),t_size);
       scale+=t_size;i=i+ ((int) t_size);
       if(first_bb->auxd==NULL)
         first_bb->auxd=ax;
       else
         bx->next=ax;
       bx=ax;
       }
     if(i==PIPE_BUF)
       f_test=0;
    }
  }
}
```

```
else
 f_test=0;

if(!f_test)
  {
  free(first_bb->dt);
  remove_all_aux(first_bb->auxd);
  free(first_bb);
  first_bb=aux;
  }
else
  {
  first_bb->dsz=bb_size;
  first_bb->next_bb=aux;
  }
return(f_test);
}

void remove_all_aux(struct aux *p)
{
struct aux * x;
x=NULL;
while(p!=NULL)
    {
    x=p;
    p=p->next;
    free(x);
    }
}

void clear_buffer(void)
{
int i;
for(i=0;i<PIPE_BUF;i++)
  buffer[i]=NULLC;
}

static void sig_usr(int signum)
{
if(signum==SIGUSR1 && first_bb!=NULL)
  {
  error_log(0,first_bb->bkbName);
  }
if(signum==SIGUSR2)
  read_fifo(&buffer[0]);
}
```

```
static void sig_term(int signum)
{
struct list_cli *p;
struct list_bb *b;
void *v;
char s[50];

 if(unlink(SVR_FF)<0)
   error_log(1,"Can't unlink FIFO");
 if(unlink(SVR_ID)<0)
   error_log(1,"Can't unlink file with PID");
p=first_cli;
if(p==NULL)
  error_log(0,"No clients to remove");
while(p!=NULL)
    {
    strcpy(s,CNT_DIR);
    strcat(s,p->cli_name);
    if(unlink(s)<0)
      {
      strcpy(s,"Can't unlink file ");
      strcat(s,p->cli_name);
      error_log(1,s);
      }
     p=p->next_cli;
     free(first_cli);
     first_cli=p;
    }
b=first_bb;
if(b==NULL)
  error_log(0,"No Blackboards to remove");
while(b!=NULL)
    {
     free(b->dt);
     b=b->next_bb;
     remove_all_aux(first_bb->auxd);
     free(first_bb);
     first_bb=b;
    }
 closelog();

 exit(0);
}

void daemon_install(void)
{
int ff;
```

```
if((ff=open(SVR_ID,O_RDONLY))!=-1)
  {
   printf("\nDaemon already in memory. Exit ...\n");
   exit(0);
  }
}

static void sig_pipe(int signum)
{
error_log(1,"Client is now longer active");
error_log(1,"It didn't read back the answer");
}

void main(void)
{
daemon_install();
if(daemon_init()!=0)
   exit(1);
creat_fifo();

if(signal(SIGUSR1,sig_usr)==SIG_ERR)
  {
   error_log(2,"Can't catch SIGUSR1");
   exit(1);
  }

if(signal(SIGUSR2,sig_usr)==SIG_ERR)
  {
   error_log(2,"Can't catch SIGUSR2");
   exit(1);
  }

if(signal(SIGTERM,sig_term)==SIG_ERR)
  {
   error_log(2,"Can't catch SIGTERM");
   exit(1);
  }

if(signal(SIGPIPE,sig_pipe)==SIG_ERR)
  {
   error_log(2,"Can't catch SIGPIPE");
   exit(1);
  }

while(1) { }
}
```

These are intermediary functions between the clp(FD) program and the client functions.

Interface.h

```
void * SuPointer;
size_t SuSize;
int SuType;
unsigned long var_ulong;
AtomInf * atom;
```

Interface.c

```
/* This is the interface C file that must be included in the first line          */
/* of the progam.usr that is generated by clp_fd for every program.pl file */

#include "client.c"

/* Functions definition */

void put_int_in_server(char *, int);
void put_string_in_server(char *, char *);

/* GLOBAL DATA */

/* External data */

extern void * SuPointer;
extern size_t SuSize;
extern int SuType;
extern unsigned long var_ulong;
extern AtomInf * atom;

/* Local data */

unsigned long vt; /* Var Tag */
char * sp;
void * Car;
unsigned int si;
```

```
/* Constants definition */

#define CONSTANT 1
#define LIST     2
#define VALUE    3
#define ATOMINF (AtomInf *)

/* Macros definition */

#define put_constant_in_server(Name)                                    \
      {                                                                 \
      sp=atom->name;                                                    \
      sp=atom->name;                                                    \
      }

#define put_value_in_server(Name)                                       \
      {                                                                 \
      memcpy((void *) &var_ulong,SuPointer,SuSize);                     \
      vt=var_ulong & 7;                                                 \
      var_ulong=var_ulong/8;                                            \
      si=(int) var_ulong;                                               \
      switch(vt)                                                        \
          {                                                             \
          case 0: /* INT */                                             \
               put_int_in_server(Name,si);                              \
               break;                                                   \
          case 3: /* CST */                                             \
               atom=ATOMINF var_ulong;                                  \
               sp=atom->name;                                           \
               put_string_in_server(Name,sp);                           \
               break;                                                   \
          }                                                             \
      }

#define put_in_server(Name)                                             \
      {                                                                 \
       switch(SuType)                                                   \
          {                                                             \
          case CONSTANT:put_constant_in_server(Name)                    \
              break;                                                    \
          case VALUE:put_value_in_server(Name)                          \
              break;                                                    \
          }                                                             \
      }

/* Local Functions */
```

```
void put_int_in_server(char * name,int a)
{
if(name!=NULL)
  {
  CREATE_CLIENT
  ADDB(name,&a,S_INT,sizeof(int))
  SEND_BLACKBOARD(name)
  KILL_CLIENT
  }
}

void put_string_in_server(char * name, char * str)
{
if(name!=NULL && str!=NULL)
  {
  CREATE_CLIENT
  ADDB(name,str,S_STR,strlen(str))
  SEND_BLACKBOARD(name)
  KILL_CLIENT
  }
}
```

These header files are needed to convert values, to handle signals and to define the symbols used in the communication protocols as well as the pathname and name, of both server and client program.

defs.h

```
#define NULLC 0
#define S_STR  0
#define S_INT  1

/* Symbols used in FIFO's messages */

#define S_CC  64    /* Create client          */
#define S_KC  35    /* Destroy client         */
#define S_CB  36    /* Create blackboard      */
#define S_KB  42    /* Read and kill blackboard */
#define S_RB  63    /* Read blackboard        */
#define S_BM  91    /* Begin message          */
#define S_EM  93    /* End message            */
#define S_BD  123   /* Begin data             */
#define S_ED  125   /* End data               */
#define S_BS  59    /* Block separator        */
#define S_DS  44    /* decimal separator      */
#define S_END  46    /* End fifo message      */

/* Constant Error's */

#define E_NOB 92
#define E_CNT 47

/* Contants used in macros */

#define T_WAIT  0
#define T_NOWAIT 1
#define T_NORMAL 0
#define T_KILL  1
```

```
/* Other Constants */

#define MaxBBname 12
#define CNT_DIR   "/tmp/"          /* Client's directory for fifo              */
#define SVR_ID    "/tmp/wdaemon"   /* Server's path where is stored the ID */
#define SVR_FF    "/tmp/wd"        /* Server's path for fifo                   */
```

conv.h

```
#include <stdio.h>
void itoa(int,char s[]);
void rvs(char s[]);

void itoa(int n,char s[])
{
 int i, sign;
 if((sign=n)<0)
   n=-n;
 i=0;
do {
   s[i++]=n % 10 + '0';
  } while((n/=10)>0);
 if(sign<0)
    s[i++]='-';
 s[i]='\0';
 rvs(s);
}

void rvs(char s[])
{
 int c,i,j;
for(i=0,j=strlen(s)-1;i<j;i++,j--)
  {
   c=s[i];
   s[i]=s[j];
   s[j]=c;
  }
}
```

m_signal.h

```
#include <signal.h>

typedef void Sigfunc(int);

Sigfunc * signal(int, Sigfunc *);

Sigfunc * signal(int signo, Sigfunc *func)
{
struct sigaction act, oact;

act.sa_handler=func;
sigemptyset(&act.sa_mask);
act.sa_flags=0;
if(signo==SIGALRM)
  {
  #ifdef SA_INTERRUPT
       act.sa_flags|= SA_INTERRUPT; /* SunOS */
  #endif
  }
else
  {
  #ifdef SA_RESTART
       act.sa_flags|= SA_RESTART;
  #endif
  }

if(sigaction(signo,&act,&oact)<0)
  return(SIG_ERR);
return(oact.sa_handler);

}
```

Appendix G - The max Example

An example is presented next to exemplify the use of the client functions altogether with the interface functions and clp(FD).

An implementation in Prolog[1] of the greatest of two numbers is as follows:

max.pl

```
:-main.

max:-write('First number:'),
     read(X),nl,
     write('Second number:'),
     read(Y),nl,
     max(X,Y,Max),
     write('The greatest number is '),
     write(Max),nl.

max(X,Y,X):- X>=Y.

max(X,Y,Y):- X<Y.
```

This is translated by clp(FD) to the following C files[2]:

max.usr

```
/*The following instruction must be added to use the Blackboard Application*/

#include "interface.c"

/* Above this line, put your first macros (these included by pragma_c) */

#undef  fail
#define fail Fail_Like_Bool

/* Below this line, put your others macros and your functions */


static void Initialize_Usr(void)
```

---

[1] Using clp(FD):

[2] The instructions in bold are those that must be putted there when editing the C code.

```
{
}

/* end of user file */

#undef  fail
#define fail Fail_Like_Wam
```

max.h

```
#define NB_OF_PRIVATE_PREDS    3

static char    *module_name="max";
static int     module_nb;

static AtomInf *X5B5D;
static AtomInf *X6D6178;
static AtomInf *X4669727374206E756D6265723A;
static AtomInf *X5365636F6E64206E756D6265723A;
static AtomInf *X5468652067726561746572206E756D62657220697320;
static AtomInf *X246578655F31;
static AtomInf *X74727565;
```

max.c

```
#define DEBUG_LEVEL    0

#include "wam_engine.h"
#include "fd_engine.h"

#include "max.h"
#include "max.usr"


#define ASCII_PRED "max"
#define PRED       X6D6178
#define ARITY      0

Begin_Private_Pred
    allocate(3)
    put_constant(X4669727374206E756D6265723A,0,"First number:")
    call(Pred_Name(X7772697465,1),0,1,"write",1)        /* begin sub 1 */
    put_y_variable(2,0)
    call(Pred_Name(X72656164,1),0,2,"read",1)        /* begin sub 2 */
```

```
    call(Pred_Name(X6E6C,0),0,3,"nl",0)        /* begin sub 3 */
    put_constant(X5365636F6E64206E756D6265723A,0,"Second number:")
    call(Pred_Name(X7772697465,1),0,4,"write",1)        /* begin sub 4 */
    put_y_variable(1,0)
    call(Pred_Name(X72656164,1),0,5,"read",1)        /* begin sub 5 */
    call(Pred_Name(X6E6C,0),0,6,"nl",0)        /* begin sub 6 */
    put_y_value(2,0)
    put_y_value(1,1)
    put_y_variable(0,2)
    call(Pred_Name(X6D6178,3),1,7,"max",3)        /* begin sub 7 */
    put_constant(X546865206772656174657374206E756D62657220697320,0,"The
greatest number is ")
    call(Pred_Name(X7772697465,1),0,8,"write",1)        /* begin sub 8 */
    put_y_value(0,0)
    put_in_server("max")
    call(Pred_Name(X7772697465,1),0,9,"write",1)        /* begin sub 9 */
    deallocate
    execute(Pred_Name(X6E6C,0),0,"nl",0)

End_Pred

#undef ASCII_PRED
#undef PRED
#undef ARITY



#define ASCII_PRED "max"
#define PRED      X6D6178
#define ARITY     3

Begin_Private_Pred
    try_me_else(1)
    get_x_value(0,2)
    math_load_x_value(0,0)
    math_load_x_value(1,1)
    builtin_2(gte,0,1)
    proceed

label(1)
    trust_me_else_fail
    get_x_value(1,2)
    math_load_x_value(0,0)
    math_load_x_value(1,1)
    builtin_2(lt,0,1)
    proceed
```

```
End_Pred

#undef ASCII_PRED
#undef PRED
#undef ARITY



#define ASCII_PRED "$exe_1"
#define PRED      X246578655F31
#define ARITY     0

Begin_Private_Pred
    put_constant(X74727565,0,"true")
    put_constant(X74727565,1,"true")
    execute(Pred_Name(X746F705F6C6576656C,2),0,"top_level",2)

End_Pred

#undef ASCII_PRED
#undef PRED
#undef ARITY


Begin_Init_Tables(max)

 Define_Atom(X5B5D,"[]")
 Define_Atom(X6D6178,"max")
 Define_Atom(X4669727374206E756D6265723A,"First number:")
 Define_Atom(X5365636F6E64206E756D6265723A,"Second number:")
 Define_Atom(X546865206772656174657374206E756D62657220697320,"The
greatest number is ")
 Define_Atom(X246578655F31,"$exe_1")
 Define_Atom(X74727565,"true")


 Define_Pred(X6D6178,0,0)

 Define_Pred(X6D6178,3,0)

 Define_Pred(X246578655F31,0,0)

 Init_Usr_File

End_Init_Tables
```

```
Begin_Exec_Directives(max)


 Exec_Directive(1,Pred_Name(X246578655F31,0))

End_Exec_Directives


/*** MAIN ***/

int main(int argc,char *argv[])

{
 EXPORT
 unix_argc=argc;
 unix_argv=argv;


 Init_Wam_Engine();

 Init_Tables_Of_Module(Builtin)
 Init_Tables_Of_Module(max)

 Exec_Directives_Of_Module(Builtin)
 Exec_Directives_Of_Module(max)

 Term_Wam_Engine();

 return 0;
}
```

Appendix H - A Client program

A client program using the client functions was developed to help users, to send blackboards to the server more easily.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ncurses/curses.h>
#include "client.c"


/* Pseudo functions */

#define put_menu_elem(a,b,c,d) put_string(a,b,c,d);


/* Functions Prototype */

void main_menu(void);
void mmenu(void);
void cmenu(void);
void put_string(int, int, chtype, char *);
void put_integer(int, int, chtype, int *);
void cblackboard(void);
void dblackboard(void);
void sblackboard(void);
void rblackboard(void);
void gblackboard(void);
void test_color(void);
void col_mmenu(void);
char * get_string(int);
int  * get_integer(int);
char * get_bb_name(void);
void get_bb_data(int);
void clear_lines(int, int);
void clear_area(int, int, int, int);
void put_empty_line(void);
char * change_bb_name(char *);
void insert_record(char *str);
void delete_record(void);
void change_record(void);
char get_record_type(void);
void tmenu(void);
```

```
void def_menu(void);
void define_structure(char, char *);
void send_request(void);
void add_int_to_struct(void);
void add_str_to_struct(void);
void add_struct(void);
void forget_choice(void);
void restart_over(char);
void put_info_in_buffer(void);
void present_data(void);
char get_data_type(void **,int);

/* Global Vars */
char fcol=0;
char fbname=0;
char fir=0;
chtype attrib[4];
void **mp;
void *dt;

struct
  {
   char *bbname;
   char tp;      /* Type of data    */
   size_t size;  /* Size of data    */
   void * bdata; /* Blackboard data */
  } bbs;

struct
 {
  char fis;
  size_t ne;
  char lt;
  size_t lsz;
 } ls;

int main(int argc,char **argv)
{
EXPORT
CREATE_CLIENT
initscr();
cbreak();
noecho();
test_color();
col_mmenu();
erase();
refresh();
```

```
main_menu();
echo();
KILL_CLIENT
return(0);
}


void col_mmenu(void)
{
init_pair(1,COLOR_BLUE,COLOR_BLACK);
init_pair(2,COLOR_RED,COLOR_BLACK);
init_pair(3,COLOR_YELLOW,COLOR_BLACK);
init_pair(4,COLOR_GREEN,COLOR_BLACK);
if(fcol)
  {
   attrib[0]=COLOR_PAIR(2);
   attrib[1]=COLOR_PAIR(1) | A_BOLD;
   attrib[2]=COLOR_PAIR(3) | A_BOLD;
   attrib[3]=COLOR_PAIR(4);
  }
else
  {
   attrib[0]=A_BOLD;
   attrib[1]=A_NORMAL;
   attrib[2]=A_NORMAL;
   attrib[3]=A_BLINK;
  }
}


void main_menu(void)
{
char option=1;
char lop=0;
char ext=0;
int ch=0;

mmenu();
refresh();
do{
  if(ext)
    {
     switch(ch)
          {
          case 65:lop=option--;
                break;
          case 66:lop=option++;
```

```
                break;
            }
        }
if(option==0)
 option=6;
else
 if(option==7)
   option=1;
refresh();

  switch(lop)
      {
      case 1:
           put_menu_elem(10,31,attrib[1],"Create Blackboard");
           break;
      case 2:
           put_menu_elem(11,31,attrib[1],"Delete Blackboard");
           break;
      case 3:
           put_menu_elem(12,31,attrib[1],"Send   Blackboard");
           break;
      case 4:
           put_menu_elem(13,31,attrib[1],"Read   Blackboard");
           break;
      case 5:
           put_menu_elem(14,31,attrib[1],"Get    Blackboard");
           break;
      case 6:
           put_menu_elem(16,38,attrib[1],"Exit");
           break;
      }

  switch(option)
      {
      case 1:
           put_menu_elem(10,31,attrib[2],"Create Blackboard");
           break;
      case 2:
           put_menu_elem(11,31,attrib[2],"Delete Blackboard");
           break;
      case 3:
           put_menu_elem(12,31,attrib[2],"Send   Blackboard");
           break;
      case 4:
           put_menu_elem(13,31,attrib[2],"Read   Blackboard");
           break;
      case 5:
```

```
            put_menu_elem(14,31,attrib[2],"Get    Blackboard");
            break;
        case 6:
            put_menu_elem(16,38,attrib[2],"Exit");
            break;
       }
if((ch=getch())==27)
  if(getch()==91)
    {
    ch=getch();
    ext=1;
    }
  else
    ext=0;
else
  {
  refresh();
  ext=1;
  if(ch==10)
    {
    refresh();
    switch(option)
         {
         case 1:if(!fbname)
                  cblackboard();
                break;
         case 2:dblackboard();
                break;
         case 3:sblackboard();
                break;
         case 4:rblackboard();
                break;
         case 5:gblackboard();
                break;
         case 6:clear();refresh();return;
         }
     mmenu();
    }

  }

}while(ch!=27 && ext);
clear();
refresh();
}
```

```
void put_string(int y, int x,chtype atr,char * str)
{
move(y,x);
attron(atr);
waddstr(stdscr,str);
attroff(atr);
refresh();
}

void put_integer(int y, int x, chtype atr, int * i)
{
char s[20];
int a;
move(y,x);
attron(atr);
memcpy((void *) &a,(void *) i,sizeof(int));
itoa(a,s);
waddstr(stdscr,s);
attroff(atr);
refresh();
}

void cblackboard(void)
{
char option=1;
char lop=0;
int ch=0;

clear();
move(8,0);
hline(95,80);
move(16,0);
hline(95,80);
put_menu_elem(7,32,attrib[0],"Create Blackboard");
do {
bbs.bbname=get_bb_name();
} while(bbs.bbname==NULL);
cmenu();
refresh();
do{
  switch(ch)
       {
       case 65:lop=option--;
            break;
       case 66:lop=option++;
            break;
       }
```

```
if(option==0)
 option=5;
else
 if(option==6)
   option=1;

 switch(lop)
      {
      case 1:
           put_menu_elem(10,29,attrib[1],"Change Blackboard Name");
           break;
      case 2:
           put_menu_elem(11,29,attrib[1],"Insert Blackboard Record");
           break;
      case 3:
           put_menu_elem(12,29,attrib[1],"Delete Blackboard Record");
           break;
      case 4:
           put_menu_elem(13,29,attrib[1],"Change Blackboard Record");
           break;
      case 5:
           put_menu_elem(15,38,attrib[1],"Exit");
           break;
      }

 switch(option)
      {
      case 1:
           put_menu_elem(10,29,attrib[2],"Change Blackboard Name");
           break;
      case 2:
           put_menu_elem(11,29,attrib[2],"Insert Blackboard Record");
           break;
      case 3:
           put_menu_elem(12,29,attrib[2],"Delete Blackboard Record");
           break;
      case 4:
           put_menu_elem(13,29,attrib[2],"Change Blackboard Record");
           break;
      case 5:
           put_menu_elem(15,38,attrib[2],"Exit");
           break;
      }
if((ch=getch())==27)
  {
  if(getch()==91)
    ch=getch();
```

```
  }
else
  {
  refresh();
  if(ch==10)
    {
     switch(option)
          {
          case 1:bbs.bbname=change_bb_name(bbs.bbname);
               break;
          case 2:insert_record(bbs.bbname);
               break;
          case 3:delete_record();
               break;
          case 4:change_record();
               break;
          case 5:clear();refresh();return;
          }
     cmenu();
    }

  }
  refresh();
 }while(1);
}

void dblackboard(void)
{
if(fbname)
  {
  clear_buffer();
  bbs.bbname=NULL;
  bbs.size=0;
  bbs.tp=0;
  bbs.bdata=NULL;
  fbname=0;
  }
else
  {
  put_string(19,25,A_BLINK,"The blackboard must be first created!");
  sleep(3);
  clear_lines(18,18);
  }
}

void sblackboard(void)
{
```

```
int r;
if(fbname && fir)
  {
   ADDB(bbs.bbname,bbs.bdata,bbs.tp,bbs.size)
   SEND_BLACKBOARD(bbs.bbname)
  }
else
  {
   put_string(19,25,A_BLINK,"The blackboard must be first created!");
   sleep(3);
   clear_lines(18,18);
  }
}

void rblackboard(void)
{
clear_buffer();
define_structure(S_RB,"Read Blackboard");
send_request();
}

void gblackboard(void)
{
clear_buffer();
define_structure(S_KB,"Get Blackboard");
send_request();
}


void mmenu(void)
{
clear();
move(8,0);
hline(95,80);
move(17,0);
hline(95,80);
put_menu_elem(7,35,attrib[0],"Main Menu");
put_menu_elem(10,31,attrib[1],"Create Blackboard");
put_menu_elem(11,31,attrib[1],"Delete Blackboard");
put_menu_elem(12,31,attrib[1],"Send   Blackboard");
put_menu_elem(13,31,attrib[1],"Read   Blackboard");
put_menu_elem(14,31,attrib[1],"Get    Blackboard");
put_menu_elem(16,38,attrib[1],"Exit");
refresh();
}

void cmenu(void)
```

```
{
put_menu_elem(10,29,attrib[1],"Change Blackboard Name");
put_menu_elem(11,29,attrib[1],"Insert Blackboard Record");
put_menu_elem(12,29,attrib[1],"Delete Blackboard Record");
put_menu_elem(13,29,attrib[1],"Change Blackboard Record");
put_menu_elem(15,38,attrib[1],"Exit");
refresh();
}

void test_color(void)
{
 fcol=(char) has_colors();
 if(fcol)
   start_color();
}

char * get_bb_name(void)
{
char *s;
put_string(11,25,attrib[2],"Chose a name to the blackboard");
move(13,0);
attron(A_REVERSE);
put_empty_line();
move(13,0);
s=get_string(80);
if(s!=NULL) fbname=TRUE;
attroff(A_REVERSE);
clear_lines(11,14);
return(s);
}

char * change_bb_name(char *str)
{
char *s;
clear_lines(10,15);
put_string(11,24,attrib[2],"Chose a new name to the blackboard");
move(13,0);
attron(A_REVERSE);
put_empty_line();
move(13,0);
s=get_string(80);
attroff(A_REVERSE);
if(s==NULL)
   s=str;
else
   free(str);
clear_lines(11,14);
```

```
return(s);
}

void insert_record(char *str)
{
if(fbname)
  {
  if(fir)
     ADDB(str,bbs.bdata,bbs.tp,(size_t) bbs.size);
  get_record_type();
  }
}


void delete_record(void)
{
char ch;
if(fbname && fir)
  {
  put_string(18,20,attrib[2],"Do you wish to delete last record (Y/N) ? ");
  attron(A_REVERSE);
  if((ch=getchar()=='y') || ch=='Y')
    {
    free(bbs.bdata);
    bbs.bdata=NULL;
    bbs.size=0;
    bbs.tp=0;
    fir=0;
    }
  attroff(A_REVERSE);
  clear_lines(18,21);
  }
else
  {
  put_string(18,25,A_BLINK,"There is no inserted record.");
  sleep(3);
  clear_lines(18,18);
  }
}

void change_record(void)
{
char *s;
int  *pi;
if(fbname && fir)
  {
  clear_lines(10,15);
```

```
  put_string(18,30,attrib[2],"Current data in record");
  put_string(11,32,attrib[2],"New data to store");
  if(bbs.tp) /* Integer */
    {
     put_integer(20,38,attrib[3],(int *) bbs.bdata);
     attron(A_REVERSE);
     clear_area(13,38,13,42);
     attroff(A_REVERSE);
     move(13,38);
     refresh();
     pi=get_integer(5);
     if(pi!=NULL)
       bbs.bdata=(void *) pi;
    }
  else      /* String  */
    {
     put_string(20,0,attrib[3],(char *) bbs.bdata);
     move(13,0);
     attron(A_REVERSE);
     put_empty_line();
     attroff(A_REVERSE);
     move(13,0);
     s=get_string(80);
     if(s!=NULL)
       bbs.bdata=(void *) s;
    }
  clear_lines(10,15);
  clear_lines(18,20);
 }
else
 {
  put_string(18,25,A_BLINK,"There is no inserted record.");
  sleep(3);
  clear_lines(18,18);
 }
}

char get_record_type(void)
{
int ch=0;
int cop=0;
tmenu();
do {
   if(ch==65 && cop==1)
     {
      put_menu_elem(14,63,attrib[1],"Integer");
      put_menu_elem(13,63,attrib[2],"String");
```

```
    cop=0;
    ch=0;
    }
  if(ch==65 && cop==0)
    {
    put_menu_elem(13,63,attrib[1],"String");
    put_menu_elem(14,63,attrib[2],"Integer");
    cop=1;
    ch=0;
    }
  if(ch==66 && cop==1)
    {
    put_menu_elem(14,63,attrib[1],"Integer");
    put_menu_elem(13,63,attrib[2],"String");
    cop=0;
    ch=0;
    }
  if(ch==66 && cop==0)
    {
    put_menu_elem(13,63,attrib[1],"String");
    put_menu_elem(14,63,attrib[2],"Integer");
    cop=1;
    }

  ch=getch();
  if(ch==27)
    {
    if(getch()==91)
      ch=getch();
    }
  else
    {
    if(ch==10)
      {
      get_bb_data(cop);
      clear_area(11,53,14,73);
      fir=1;
      break;
      }
    }
  }while(ch!=10);
}

void tmenu(void)
{
put_string(11,53,attrib[3],"------>");
put_menu_elem(11,60,attrib[0],"Type of Record");
```

```
put_menu_elem(13,63,attrib[2],"String");
put_menu_elem(14,63,attrib[1],"Integer");
}

void get_bb_data(int t)
{
switch(t)
    {
    case 0: /* A string was chosen   */
        put_string(18,32,attrib[2],"String to Record");
        move(20,0);
        attron(A_REVERSE);
        put_empty_line();
        move(20,0);
        bbs.bdata=(void *) get_string(80);
        bbs.size=strlen((char *) bbs.bdata);
        bbs.tp=S_STR;
        attroff(A_REVERSE);
        break;
    case 1: /* An integer was chosen */
        put_string(18,32,attrib[2],"Integer to Record");
        attron(A_REVERSE);
        clear_area(20,38,20,42);
        move(20,38);
        refresh();
        bbs.bdata=(void *) get_integer(6);
        bbs.tp=S_INT;
        bbs.size=sizeof(int);
        move(1,1);
        attroff(A_REVERSE);
        break;
    }
clear_lines(18,21);
}

int * get_integer(int n)
{
char *s;
int i;
int *pi;
s=get_string(n);
if(s!=NULL)
  {
  pi=(int *) calloc(sizeof(int),1);
  i=strtol(s,(char **)NULL,10);
  memcpy((void *) pi,(void *) &i,sizeof(int));
  }
```

```
else
 pi=NULL;
free(s);
return(pi);
}

char * get_string(int n)
{
char *s;
char ch;
int z=0;
s=NULL;
echo();
ch=getchar();
if(ch!=27 && n>0)
  {
   s=(char *) calloc(sizeof(char),n+1);
   *s=ch;
   fgets(s+1,n-1,stdin);
   z=strlen(s);
   move(5,5);refresh();
   *(s+z-1)=NULLC;
  }
else
   flushinp();
noecho();
return(s);
}

void define_structure(char c,char *str)
{
char option=1;
char lop=0;
int ch=0;

clear();
ls.ne=0;
ls.lt=0;
ls.lsz=0;
ls.fis=0;
mp=NULL;
dt=NULL;
move(8,0);
hline(95,80);
move(16,0);
hline(95,80);
put_menu_elem(7,32,attrib[0],str);
```

```
do {
bbs.bbname=get_bb_name();
} while(bbs.bbname==NULL);
buffer[0]=c;
put_info_in_buffer();
def_menu();
refresh();
do{
  switch(ch)
       {
       case 65:lop=option--;
            break;
       case 66:lop=option++;
            break;
       }
if(option==0)
 option=5;
else
 if(option==6)
   option=1;

  switch(lop)
       {
       case 1:
            put_menu_elem(10,37,attrib[1],"String");
            break;
       case 2:
            put_menu_elem(11,37,attrib[1],"Integer");
            break;
       case 3:
            put_menu_elem(12,31,attrib[1],"Forget last choice");
            break;
       case 4:
            put_menu_elem(13,31,attrib[1],"Restart over again");
            break;
       case 5:
            put_menu_elem(16,38,attrib[1],"Exit");
            break;
       }

  switch(option)
       {
       case 1:
            put_menu_elem(10,37,attrib[2],"String");
            break;
       case 2:
            put_menu_elem(11,37,attrib[2],"Integer");
```

```
                break;
        case 3:
                put_menu_elem(12,31,attrib[2],"Forget last choice");
                break;
        case 4:
                put_menu_elem(13,31,attrib[2],"Restart over again");
                break;
        case 5:
                put_menu_elem(16,38,attrib[2],"Exit");
                break;
        }
if((ch=getch())==27)
  {
  if(getch()==91)
    ch=getch();
  }
else
  {
  refresh();
  if(ch==10)
    {
    switch(option)
        {
        case 1:add_str_to_struct();
                break;
        case 2:add_int_to_struct();
                break;
        case 3:forget_choice();
                break;
        case 4:restart_over(c);
                break;
        case 5:clear();refresh();free(bbs.bbname);fbname=FALSE;return;
        }
    def_menu();
    }

  }
  refresh();
 }while(1);
clear_lines(1,7);
}


void send_request(void)
{
if(bbs.bbname!=NULL)
  {
```

```
  if(ls.fis)
    add_struct();
  buffer[size_str++]=S_EM;
  buffer[size_str++]=S_END;
  tell_server(size_str);
  READ_BLACKBOARD(&mp,T_WAIT);
  if(mp!=NULL)
   present_data();
  else
   {
   put_string(18,25,A_BLINK,"There is no blackboard with that structure.");
   sleep(3);
   clear_lines(18,18);
   }
 }
}

void add_int_to_struct(void)
{
int x,y;
getyx(stdscr,y,x);
if(ls.fis)
  add_struct();
ls.fis=1;
ls.lsz=sizeof(int);
ls.lt=S_INT;
move(y,x);
refresh();
}

void add_struct(void)
{
switch(ls.lt)
    {
    case 0: /* Add string to buffer */
          memcpy(&buffer[size_str++],&Cts,1);
          memcpy(BUF,&(ls.lt),sizeof(ls.lt));
          size_str+=sizeof(ls.lt);
          ls.ne++;
          break;
    case 1: /* Add int to buffer    */
          memcpy(&buffer[size_str++],&Cts,1);
          memcpy(BUF,&(ls.lt),sizeof(ls.lt));
          size_str+=sizeof(ls.lt);
          memcpy(BUF,&(ls.lsz),sizeof(ls.lsz));
          size_str+=sizeof(ls.lsz);
          ls.ne++;
```

```
            break;
        }
}

void add_str_to_struct(void)
{
int x,y;
getyx(stdscr,y,x);
if(ls.fis)
  add_struct();
ls.fis=1;
ls.lsz=0;
ls.lt=S_STR;
move(y,x);
refresh();
}

void forget_choice(void)
{
ls.fis=0;
ls.lsz=0;
ls.lt=0;
}

void restart_over(char c)
{
clear_buffer();
clear_lines(1,7);
put_info_in_buffer();
ls.fis=0;
ls.lsz=0;
ls.lt=0;
}

void put_info_in_buffer(void)
{
char spid[20];

strcat(buffer,cli_name);
strcat(buffer,USCR);
itoa(pid,spid);
strcat(buffer,spid);
BFSZ;
buffer[size_str++]=S_BM;
strcat(buffer,bbs.bbname);
BFSZ;
}
```

```
void present_data(void)
{
int i,a,integ;
char dtype;
dtype=0;
clear();
put_string(1,20,attrib[0],"Data received from the Blackboard");
put_string(2,20,A_NORMAL,"_____");
put_string(24,20,attrib[1],"Press ESC to return to main menu");
a=1;
for(i=1;i<=ls.ne;i++)
  {
   put_integer(3+a,1,A_BOLD,&i);
   GET_DATA(mp,i,dt);
   dtype=get_data_type(mp,i);
   switch(dtype)
       {
       case S_STR:
              put_string(3+a,3,attrib[3],(char *) dt);
              break;
       case S_INT:integ=Ri(dt);
              put_integer(3+a,3,attrib[3],&integ);
              break;
       }
   if(a==23)
    {
     clear_lines(3,23);
     a=1;
    }
   else
    a++;
  }
while(getch()!=27);
}

char get_data_type(void ** mp,int i)
{
int a=0;
char c;
while((*(mp+a))!=NULL && a!=(i-1))
    a++;
if((*(mp+a))!=NULL && a==i-1)
  memcpy(&c,(*(mp+a)),sizeof(char));
else
  c=2;
return(c);
```

```
}

void def_menu(void)
{
clear();
move(8,0);
hline(95,80);
move(17,0);
hline(95,80);
put_menu_elem(7,32,attrib[0],"Define Structure");
put_menu_elem(10,37,attrib[1],"String");
put_menu_elem(11,37,attrib[1],"Integer");
put_menu_elem(12,31,attrib[1],"Forget last choice");
put_menu_elem(13,31,attrib[1],"Restart over again");
put_menu_elem(16,38,attrib[1],"Exit");
refresh();
}

void clear_lines(int y1,int y2)
{
int i,a;
int x,y;

getyx(stdscr,y,x);
if(y2<y1)
  {
   a=y1;
   y1=y2;
   y2=y1;
  }
for(i=y1;i<=y2;i++)
   {
    move(i,0);
    clrtoeol();
   }
move(y,x);
refresh();
}

void clear_area(int y1, int x1, int y2, int x2)
{
int i,r,a;
int x,y;

getyx(stdscr,y,x);
if(y2<y1)
  {
```

```
  a=y1;
  y1=y2;
  y2=y1;
  }
if(x2<x1)
  {
  a=x1;
  x1=x2;
  x2=x1;
  }
for(i=y1;i<=y2;i++)
   {
    for(r=x1;r<=x2;r++)
       {
        move(i,r);
        waddstr(stdscr," ");
       }
   }
move(y,x);
refresh();
}

void put_empty_line(void)
{
 int i;
int x,y;

getyx(stdscr,y,x);
for(i=1;i<=80;i++)
   waddstr(stdscr," ");
move(y,x);
refresh();
}
```

Only two header files were changed to handle blackboards. Next comes the portion of code of the changed parts. Changes are written in bold.

wam_engine.h

```
/*--------------------------------------*/
/* Global Variables                     */
/* ( Includes the global variables      */
/* added to support the Blackboard */
/* Application )                        */
/*--------------------------------------*/

...
#include "/home/jpc/proj/interface.h"
...
#define put_y_value(y,a)                                \
   {                                                    \
    DBG_INST("put_y_value(" #y "," #a ")")              \
    A(a)=Y(E,y);                                        \
    SuType=3;                                           \
    SuPointer=(void *) &A(a);                           \
    SuSize=sizeof(A(a));                                \
   }
...
```

wam_engine.c

```
/*--------------------------------------*/
/* PUT_Y_UNSAFE_VALUE      */
/* Changed version                      */
/*--------------------------------------*/

void Put_Y_Unsafe_Value(int y,int a)
{

WamWord word,tag,*adr;

Deref(Y(E,y),word,tag,adr)
```

```
if (tag==REF && adr>=(WamWord *) EE(E))
   {
    A(a)=Tag_Value(REF,H);
    Globalize_Local_Unbound_Var(adr)
    SuType=0;
   }
 else
   {
    A(a)=(Global_UnMove(tag)) ? Tag_Value(REF,adr) : word;
    SuPointer=(void *) &A(a);
    SuSize=sizeof(A(a));
    SuType=3;
   }
}


/*----------------------------*/
/* PUT_CONSTANT       */
/* Changed version          */
/*----------------------------*/
void Put_Constant(AtomInf *atom,int a)

{
 A(a)=Tag_Value(CST,atom);
 SuType=1;
 SuPointer=(void *)&A(a);
 SuSize=sizeof(A(a));
}


/*-------------------------*/
/* PUT_LIST             */
/* Changed version      */
/*-------------------------*/
void Put_List(int a)

{
 A(a)=Tag_Value(LST,H);
 SuType=2;
 SuPointer=(void *)&A(a);
 SuSize=sizeof(A(a));
 S=WRITE_MODE;
}
```

[R 1] A. Martelli and U. Montanari. An Efficient Unification Algorithm. ACM Transactions on Programming Languages and Systems, 4(2):258-282, 1982.

[R 2] D. Chan Constructing Negation based on the Completed Database. In R.A. Symposium on Logic Programming., Series in LP, pages 111-125, Seattle, USA, August 1988. The MIT Press.

[R 3] D. Lugiez. A Deduction Procedure For First Order Programs. In G. Levi and M. Martelli, editors, Proc Sixth International Conference on Logic Programming, Series in Logic Programming, pages 585-876, Lisbon, Portugal, July 1989. The MIT Press.

[R 4] G. Huet. Résolution d'equations dans les langages d'ordre 1,2,...,ω. PhD thesis, Université Paris VII, France, September 1976.

[R 5] J. A. Robinson. A Machine-oriented Logic based on the Resolution Principle. Journal of the ACM, 12(1):23-41, 1965.

[R 6] J. Herbrand. Researches in the Theory of Demonstration. In J. van Heijenoort, editor, From Frege to Gödel: a Source Book in Mathematical Logic 1879-1931, pages 525-581. Harvard University Press, Cambridge, MA, USA, 1967.

[R 7] J. Maluszynski and T. Naslund. Fail Substitutions for Negation as Failure. In E. L. Lusk and R. Overbeek, editors, Proc. North American Conference on Logic Programming, Series in Logic Programming, pages 461-476, Cleveland, USA, October 1989. The MIT Press.

[R 8] K. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, Logic and Databases, pages 293-322. Plenum Press, New York, USA, 1978.

[R 9] L. Naish. Negation and Control in Prolog, volume 238 of Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 1986.

[R 10] M. S. Paterson and M. N. Wegman. Linear Unification. Journal of Computer and System Sciences, 16(2):158-167, 1978.

[R 11] N. Wallace. Negation by Constraints: a Sound and Efficient Implementation of Negation in Deductive Databases. In S. Haridi, editor, Proc. Fourth Symposium on Logic Programming, pages 253-263, San Francisco, USA, August - September 1987. The IEEE Computer Society Press.

[R 12] P. Van Henteryck, V. Saraswat and Y. Deville. Constraint Processing in cc(FD). Draft, 1991.

[R 13] R. Barbuti, P. Mancarella, D. Pedreschi, F. Turini. A Transformational Approach to Negation in Logic Programming. Journal of Logic Programming, 8(3):201-228, 1990.

[R 14] T. Khabaza. Negation as Failure and Parallelism. In S. Tärnlund, editor, Proc. Second International Conference on Logic Programming, pages 70-75, Uppsala, Sweden, July 1984. Upsala University Press.

[R 15] V. Saraswat. The Category of Constraint Systems is Cartesian-Closed. In Logic in Computer Science, IEEE Press. 1992.

[1] Brian W. Kernighan and Dennis M. Ritchie. The C programming language. Prentice Hall, 1988.

[2] Daniel Diaz. Wamcc 2.21 User's Manual. INRIA, Le Chesnay, France, 1994.

[3] Daniel Diaz. Clp(FD) 2.21 User's Manual. INRIA, Le Chesnay, France, 1994.

[4] Daniel Diaz. Étude de la compilation des langages logiques de programmation par contraintes sur les domaines finis: Le systeme clp(FD). PhD thesis, Orleans University, France, January 1995.

[5] H. Aït-Kaci. Warren's Abstract Machine: A Tutorial Reconstruction. Series in Logic Programming. The MIT Press, Cambridge, USA, 1991.

[6] Jean-Marie Jacquet. Constructing Logic Programs. In J-M Jacquet (ed). John Wiley & Sons, Lda, !993.

[7] Jean-Marie Jacquet and Koenraad De Bosschere. On the semantics of μLog. Future Generations Computer Systems, n° 10 (1994) pp. 93-135.

[8] Matt Welsh. Linux Installation and Getting Started. Linux Documentation Project, 1995.

[9] Peter Van Linden. Expert C programming. Prentice Hall, 1994

[10] Philippe Codognet and Daniel Diaz. Wamcc: Compiling Prolog to C. In 12th International Conference on Logic Programming, Tokyo, Japan, Mit Press, 1995.

[11] Philippe Codognet and Daniel Diaz. Compiling Constraints in clp(FD). The Journal of Logic Programming, 27 (3), pp. 185-226, June, 1196.

[12] Philippe Codognet and Daniel Diaz. Local Propagation Methods for Soving Boolean Constraints in constraints Logic Programming.

[13] W. Richard Stevens. Advanced Programming in the Unix Environment. Addison-Wesley Publishing Company, 1992.

R

S

T

U

V

W