DOMAINS IN LOGIC PROGRAMMING

P. Van Hentenryck and M. Dincbas

European Computer-industry Research Centre (E.C.R.C.) Arabellast. 17 D-8000 Munich 81 West-Germany

ABSTRACT.

When confronted with constraint satisfaction problems (CSP), the "*generate & test*" strategy of Prolog is *particulary inefficient*. Also, control mechanisms defined for logic programming languages fall short in CSP because of their restricted use of constraints. Indeed, constraints are used passively for testing generated values and not for actively pruning the search space by eliminating combinations of values which cannot appear together in a solution. One remedy is to introduce the **domain** concept in logic programming language. This allows for an active use of constraints. This extension which does not impede the declarative (logic) reading of logic languages, consists in a modification of the unification, the redefinition of the procedural semantics of some built-in predicates ($\neq$, $\leq$, $<$, $\geq$, $>$) and a new evaluable function and can be implemented efficiently. Without any change to the search procedure and without introducing a new control mechanism, look ahead strategies, more intelligent choices and consistency techniques can be implemented naturally in programs. Moreover, when combined with a delay mechanism, this leads directly to a strategy which applies active constraints as soon as possible.

1. Motivations

As Prolog is applied to more and more areas, inadequacies of its search procedure appear and, although there were substantial efforts to develop powerful control mechanisms, the proposed solutions are not entirely satisfactory for different kinds of problems. This is the case of constraint satisfaction problems (CSP). A constraint satisfaction problem can be defined as follows.

Assume the existence of a finite set J of variables $\{X_1, X_2, \ldots, X_n\}$. Suppose each variable X_i takes its values from a finite set U_i called the domain of the variable. A constraint C can be seen as a relation on a non-empty subset $I = \{Y_1, Y_2, \ldots, Y_M\}$ of J which defines a set of tuples $<u_1, \ldots, u_M>$. **The constraint satisfaction problem** *is to determine all the possible assignments f of values to variables such that the corresponding values assignments satisfy the constraints.* The class of CSP is related to many problems in AI like logical puzzles, scene labeling, graph isomorphisms, graph colouring and propositional theorem proving among others. The simple backtrack search (depth first search with chronological backtracking) is very inefficient for this kind of problems. Indeed Prolog uses a "generate and test" strategy and this leads to give values to all variables before testing the constraints. Moreover, although more powerful control mechanisms can reduce the search space, they fall short in CSP because of their restricted passive use of the constraints. Indeed, these mechanisms can only provide **coroutining** which is based on the "apply tests as soon as possible" heuristics which is not the best-suited one for this class of problems. With these mechanisms, a constraint is tested as soon as its variables have received their values. Thus, the search space is only reduced in an "a posteriori way" after the discovering that the generated values do not satisfy a constraint. The main drawbacks of such an approach are the continual rediscoverying of the same facts and the pathological behaviour of (chronological) backtracking. See (Mackworth, 1977) for a convincing example. Intelligent backtracking is a remedy to this state of affairs but does not attack the real cause of the problems and introduces an important overhead when not necessary.

There is another way to use constraints (we will speak about an active use of the constraints (Gallaire, 1985) which consists in reducing the search space in an "a priori" manner by removing inconsistencies, combinations of values which cannot appeared together in a solution (Freuder, 1978). This approach is the basis of consistency techniques (Mackworth, 1977), (Freuder, 1978), which has been used in refinements of the simple backtrack search (i.e. forward checking, looking ahead procedures) (Haralick and Elliot, 1980) and in Alice, a problem solver for combinatorial problems (Lauriere, 1978). When facing a constraint satisfaction problem, a logic program (which can be considered as a kind of meta-interpreter) can be written which implements, say, a forward checking strategy. It will generally be more efficient that the usual Prolog programs. However, this requires an important programming effort, leads to less readable and less maintenable programs and does not allow the full efficiency of these approaches because it creates a level above Prolog. As a matter of fact, logic programming languages lack primitives for an active treatment of constraints. It seems difficult to define new control mechanisms in order to use constraints more actively. This state of affairs comes from the fact that (first order) Horn clauses obscure sometimes properties of predicates and thus prevents the interpreter from using them. Consider for instance the case of variables. In logic programming, the variables range over the Herbrand universe which is generally infinite. However, in many applications and in CSP, the domain of variables is finite and much more restricted but this information is hidden in predicates like permutation(X,Y), monadic predicates to restrict the range of variables or more generally in generators. Introducing the **domain** concept into logic programming languages leads directly to an active use of constraints and the opportunity to implement naturally forward checking, consistency techniques and the like. The next section introduces domain declarations in logic programming and its interest for using constraints actively is discussed. For supporting this use, the procedural semantics is modified and this consists in a modification of the unification, the redefinition of the procedural semantics of some built-in predicates ($\neq$, $\leq$, $<$, $\geq$, $>$) and a new evaluable function. Next, some examples are given and compared with usual Prolog programs. Finally, it is shown how some heuristics and consistency techniques can be built from our basic extension and implementation issues are discussed. The reader is refered to (Van Hentenryck and Dincbas, 1986) for more details on all the features defined here.

2. Domain declarations.

It is often the case that the variables range over a finite domain but this information cannot be expressed clearly in logic programming languages. **Domain declarations** are introduced for taking this fact into account. A domain declaration for predicate p of arity n is an expression of the following form.

> **domain** p:<a_1,...,a_n>.
> where a_i is either H or D^m $1 \leq i \leq n$.

When a_i is equal to H, this means that argument i of p ranges over the Herbrand universe. Otherwise, it means that argument i is a set of m variables which ranges over D. As in CSP the number m is fixed, these declarations are fully satisfactory. In the following, the domain D are finite and explicit set of constants[*]. The domain declarations can be considered as a kind of meta-knowledge although quite different from the one proposed usually in logic programming. The effect of these definitions is to reduce the class of interpretations which must be considered for a logic program. In fact, the declarative logic semantics of the "extended" language is a particular case of the many-sorted logic defined in (Cohn, 1983). It is well-known that many-sorted logic can improve the efficiency of theorem-provers (Walther, 1984) and this kind of extensions has already been used in logic programming (see for instance (Mycroft and O'Keefe, 1984)). ***But the main new point of this paper is that using such a logic allows procedural uses which have no counterparts in an unsorted logic.*** The domain declarations can be used to determine the definition domain of each variable.[**] In the following, a variable with an explicit and finite definition domain is refered as a d-variable and its domain is noted D_X. Other variables are refered as h-variable. Moreover, at any time of the computation, the possible set (i.e. the set of values in which a d-variable takes its value) of a d-variable can be determined. Initially, this set is the definition domain but the constraints can reduce it. For instance, a constraint "X $\neq$ 3" can be used to remove "3" from the possible set of the variable. This is an active way to use the non-equality constraints contrarily to the passive use of Prolog which can only handle such a constraint when both arguments are instantiated. This use of constraints, combined with the oppportunity of generating only values in the possible set, increases substantially the efficiency of logic programming for solving CSP. Indeed, this allows to prune the search space in an "a priori" manner instead of the "a posteriori" manner of Prolog. Moreover, this can be used to reduce the gap between the declarative and procedural semantics for the built-in predicates and this is one of the directions of logic programming proposed by (Kowalski, 1985).

3. Procedural semantics.

The invariant of all proposed extensions is that the domain of a d-variable has a cardinality greater than 1. If domains of cardinality 1 are defined, the value can directly be assigned to the variables defined on this domain.

3.1. Unification.

It is clear that the unification must take into account the domain of the variables. The unification algorithm must be modified to handle the three following cases.

- If a h-variable and a d-variable must be unified, the h-variable must be bound to the d-variable.
- If a constant and a d-variable must be unified, the d-variable is bound to the constant if it is in the domain of the variable. Otherwise, the unification fails.
- If two d-variables must be unified, then let D_Z the intersection of their domains. If D_Z is empty, the unification fails. If D_Z={v} then v is bound to both

[*] The fact that we only considered constants is in no way restrictive. The definitions given here can be extended to arbitrary ground terms but the generalization is not considered here for clarity and brevity.

[**] The domain of a variable can be determined at compile time.

variables. Otherwise, both variables are bound to a new variable Z whose domain is D_Z.

3.2. Built-in predicates.

The real interest of the domain extension is not only in the logic part of the language but also in the procedural part, say, in the redefinition of some built-in predicates. These predicates can now take into account the domain of variables.

3.2.1. Non-equality predicate.

The declarative semantics of the predicate is " $X \neq Y$ holds if X is not equal to Y". However, the usual procedural semantics of this predicate is given by the following clause

$X \neq Y \leftarrow$
  $\text{not}(X = Y)$.

where not(G) is the "negation as failure" rule. Thus, the only use of such a constraint is a passive one. Moreover, there is a gap between the declarative and procedural semantics. A safe computation rule must be defined which only selects the non-equality predicates when they are ground. Moreover, a generator of values for X and Y must be provided in order to be complete. Our procedural semantics allows an active use of these predicates and also reduces the gap between the declarative and procedural semantics.

$X \neq Y$ is defined by

- If X is d-variable and Y is a constant, let D_Z be $D_X \setminus \{Y\}$. If $D_Z=\{v\}$, then X is unified with v and $X \neq Y$ succeeds. Otherwise, X is bound to Z (whose domain is D_Z) and $X \neq Y$ succeeds.
- The case where Y is a d-variable and X is a constant is similar to the previous one.
- If X and Y are constants, $X \neq Y$ succeeds if X and Y are distinct constants and fails otherwise.
- Its effect is undefined otherwise.

Note that this definition can easily be combined with a delay mechanism like wait, geler, freeze, (Naish, 1985), (Colmerauer, Kanoui and Van Caneghem, 1983) ,(Dincbas, 1984) in order to delay the constraint until one of the first three cases is fulfilled. In this case, there are no undefined cases and *this will lead to a very efficient way to handle non-equality constraints which consists in using the constraint in an active way as soon as possible. Indeed, a non-equality predicate can be considered as active in two different senses. First, it can remove a value from the possible set of a variable. Second, it can assign a value to a variable when only one consistent value is left for this variable.* The procedural semantics is equivalent to the declarative one in the first three cases. The gap between the semantics is in the fourth case. If a safe computation rule which only selects a non-equality predicate when one of the first three cases is fulfilled, the procedural semantics will be sound. However, in order to be complete, a generator of values for X and Y must be provided. Note that, in fact, this semantics is suboptimal. Indeed if, for instance, X is a d-variable with $d_X = \{1,2,3\}$ and Y is a d-variable with $D_Y = \{4,5,6\}$, then the predicate $X \neq Y$ should succeed because whatever the values that X and Y will take, they will be different.

3.2.2. Inequality predicates.

We consider here the predicates "$X \leq Y$" whose declarative semantics is given by "$X \leq Y$" holds if X and Y are integers and X is less than or equal to Y". The usual procedural semantics of this predicate is the following : "$X \leq Y$ succeeds if X and Y are instantiated to integers and X is less than or equal to Y". This time again, the procedural semantics entails only a passive use of this constraint. Also, there is a discontinuity between the declarative and the procedural semantics. The procedural semantics can be redefined as follows.

$X \leq Y$ succeeds in the following cases

- If X is a d-variable, Y is an integer then let sup $= \{V: V \in D_X \text{ and } V > Y\}$ and $D_Z = D_X \setminus$ sup. If D_Z is empty, then $X \leq Y$ must fail. If not, if $D_Z = \{v\}$, then X is bound to v and $X \leq Y$ succeeds. Otherwise, X is bound to Z and $X \leq Y$ succeeds.
- The case where X is an integer and Y is a d-variable is similar to the previous one.
- If X and Y are instantiated to integer values, $X \leq Y$ succeeds if X is less than or equal to Y. Otherwise, it fails.
- Otherwise, its effect is undefined.

Therefore, this predicate is an active predicate which removes 0 or 1,...or n values from the domain of a variable and which can assign a value to a variable. It is clear that it can be combined with a delay mechanism for handling the last case. Note also that this implementation is suboptimal. If $D_X = \{1,2,3\}$ and D_Y is $\{4,5,6\}$, $X \leq Y$ should succeed. In the same way, $Y \leq X$ should fail. The others inequality predicates can be defined in the same way.

3.2.3. Domain primitives.

The extensions presented so far can be improved by giving to the user an access to the possible set of a variable. We introduce a new evaluable function domain(X) (which holds if domain(X) returns the list of instances of a term which satisfy the domains) whose procedural semantics is

```
domain(X) = the list of all the values
            in D_X if X is a d-variable.
          = [X] if X is a ground term.
          is undefined otherwise
```

This function can be used to generate values for a d-variable by using, for instance, "member(X.domain(X))" where member(X,Y) holds if X is an element of the list Y.[***] This function is quite useful when the "first fail" principle and arc-consistency are to be used (see below).

4. Examples

In the following, we will give two examples of the basic mechanisms. The first one (N-queens problem) shows how a new search procedure (forward checking) can be implemented in a logical way without the need to rewrite a specific meta-interpreter. The second one is a logical puzzle which also shows that our extension combined with a delay mechanism can lead to a "data-driven computation" as, for instance, in the constraint language of (Sussman and Steele, 1980).

4.1. N-queens problem.

The following N-queens program implements a forward checking strategy (based on the "lookahead in the future in order not to worry about the past" heuristics). This means that the program chooses a possible value for a variable, removes the inconsistent values for the other variables and so on until all variables have a value. By moving along this way, there is no need to test the value assigned to the present variable against the values of already assigned variables. This is the most efficient heuristic for this problem (Haralick and Elliot, 1980). The program is the following.

```
queens([]).
queens([X|Y]) ←
   member(X.domain(X)) ,
   safe(X.Y.1) ,
   queens(Y).

safe(X,[].Nb).
safe(X,[F|T].Nb) ←
   noattack(X.F.Nb) ,
   Newnb is Nb + 1 ,
   safe(X.T.Newnb).
```

```
noattack(X,Y,Nb) ←
   Y ≠ X ,
   XmoinsNb is X - Nb ,
   Y ≠ XmoinsNb
   XplusNb is X + Nb ,
   Y ≠ XplusNB.
```

The five-queens program can be expressed by the following clause.

```
domain five-queens:<{1.2.3.4.5}^5>.
five-queens({X1,X2.X3.X4.X5}) ←
   queens([X1,X2.X3,X4,X5]).
```

The usual Prolog program consists in assigning to the n variables of the list a permutation of [1,...,n] and then in testing if the assignment satisfies the constraints. This is a very inefficient approach. Control mechanisms, based on control informations provided by the user, can be used to apply the tests as soon as possible and thus improve the efficiency of the search (see IC-Prolog (Clark and Mc Cabe, 1979), Metalog (Dincbas, 1984), MU-Prolog (Naish, 1985)). However, consider the first steps of our program. The "member(X.domain(X))" will choose a value for X in its domain. Let say 1. Then immediately, all the inconsistent values of X2,...,X5 are "removed from their possible set" by the safe predicate. Indeed, the noattack(X,Y,Nb) can be used with the following modes: noattack(+,+,+) and noattack(+,-,+). In the later case, "-" means that Y is a d-variable and the effect of the predicate is to remove X, X - Nb, X + Nb from the domain of Y. Therefore, at this time, the situation is given in figure 1 (O represents an assigned value and X an inconsistent value). In the next step, the values, 1 and 2 will not be considered for X2. In a coroutining program, these values would have been tested. The next step chooses 3 as value for X2.

	1	2	3	4	5
1	O	X	X	X	X
2		X			
3			X		
4				X	
5					X

	1	2	3	4	5
1	O	X	X	X	X
2		X	X		
3		O	X	X	X
4			X	X	
5				X	X

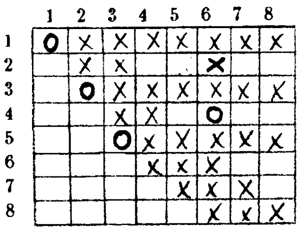
figure 1: 5-queens after 1 and 2 choices

Therefore, this instantiates immediately X3 and X4 (to 5 and 2 because their possible set is reduced to 1 element) and their safe predicates will reduce to one element the domain of X5 (i.e 4). The problem is solved with two choices and without any backtracking.

Now, consider, the search for another solution. In our case, the first backtrack point is X2 and the values assigned to X2,X3,X4 and X5 will no more be compared with X1. It's not the case for the logic program with a control mechanism: each time a value

[***] However, the unification will test if each value is in the domain of the variable. Therefore, a predicate "indomain(X)" can be introduced whose declarative semantics is "indomain(X) holds if member(X.domain(X)) holds" and which avoids the unification inefficiency.

is assigned to X2, X3, X4 and X5, this value must be compared with X1. This entails a lot of redundancy. Consider now the eight-queens problem after 3 choices.



	1	2	3	4	5	6	7	8
1	O	X	X	X	X	X	X	X
2		X	X			X		
3		O	X	X	X	X	X	X
4			X	X		O		
5			O	X	X	X	X	X
6				X	X	X		
7					X	X	X	
8						X	X	X

figure 2: 8-queens after 3 choices

X6 has already received a value as there is only one value left in its domain. Therefore, when choosing 2 as value for X4, the safe predicate will fail and this choice will be reviewed. This failure is detected as early as possible. Moreover, the real cause of the failure (X4) is detected. In a logic program with control mechanism, the failure will be detected only when assigning values for X6 and the backtracking will consider all the values (from 1 to 8) for X5, X6. *The point here is twofold: first, there exist powerful search procedures for CSP which detect failure earlier than control mechanisms, choose the right backtrack point without any overhead and avoid a lot of redundancy; second, a simple extension is sufficient to allow for fully declarative programs which implements such a procedure without the need of control informations.*

4.2. A combinatorial problem.

The problem is the following (Lauriere, 1978).

Six couples took part in a tennis match. Their names were Howard, Kress, McLean, Randolph, Lewis and Rust. The first names of their wives were Margaret, Susan, Laura, Diana, Grace and Virginia. Each of the ladies hailed from a different city: Forth Worth, Wichita, Mt Vernon, Boston, Dayton, Kansas City. Finally, each of the women had a different hair color, namely black, brown, gray, red, auburn and blond. Informations are given to state doubles and single which were played. For instance, Howard and Kress played against Grace and Susan or the gray hair lady played against Margaret. There is only one other fact we ought to know to be able to find the last names, home towns and hair colors of all six wives, and that is the fact that "No married couple ever took part in the same game". The following Prolog program solves the problem.

```
tennis([Ho,Ke,Mc,Ra,Le,Ru],[Fo,Wi,Mt,Bo,Da,Ka],
        [Bl,Br,Gr,Re,Au,Blo]) ←
  perm([Ho,Ke,Mc,Ra,Le,Ru], [ma,su,la,di,gr,vi]),
  Ho ≠ gr. Ho ≠ su. Ke ≠ gr. Ke ≠ su. Mc ≠ la.
  Mc ≠ su.Ra ≠ la. Ra ≠ su. Mc ≠ gr. Ra ≠ gr.
  Le ≠ gr. Ke ≠ la. Ke ≠ vi. Mc ≠ di. Mc ≠ vi.
  perm([Bl.Br.Gr.Re.Au,Blo].[ma.su.la.di.gr.vi]),
  Br ≠ vi.Br ≠ Ho. Br ≠ Mc. Ra ≠ Gr. Gr ≠ la.
  Blo ≠ la. Blo ≠ di. Le ≠ Blo. Blo ≠ ma.
  perm([Fo.Wi.Mt,Bo.Da.Ka].[ma.su.la.di.gr.vi]).
  Fo ≠ Ho.Fo ≠ Mc.Fo ≠ Ra.Wi ≠ Ho.Wi ≠ Mc.
  Da ≠ ma.Mt ≠ ma,Mt ≠ di. Da ≠ di. Mt ≠ vi.
  Wi ≠ Ra. Wi ≠ Ke.Ru ≠ Fo.Fo ≠ Ke.Gr ≠ Bo.
  Re ≠ Da. Gr ≠ Fo. Re ≠ Mt. Blo ≠ Da.
  Bl ≠ Bo. Bl ≠ Da. Ka ≠ ma.
```

where perm(L,Res) holds if the list Res is a permutation of the list L. In this program, the permutation predicates assign values to the variables. Next, the constraints are tested and if they are not satisfied, backtracking occurs in the permutations. Note also, that if a value for "Ho" generated in the first permutation cannot satisfy the constraint "Fo ≠ Ho" tested after the third permutation, the backtracking will generate all the possible values for all the variables. This time again, this is quite inefficient. These constraints can be used immediately in order to remove inconsistencies. The program becomes the following. Let D = {ma.su,la,di,gr,vi}

```
domain tennis:<D^6,D^6,D^6>.
tennis({Ho,Ke,Mc,Ra,Le,Ru},{Fo,Wi,Mt,Bo,Da,Ka}
        ,{Bl,Br,Gr,Re,Au,Blo}) ←
  Ho ≠ gr, Ho ≠ su, Ke ≠ gr, Ke ≠ su, Mc ≠ la.
  Mc ≠ su,Ra ≠ la. Ra ≠ su. Mc ≠ gr. Ra ≠ gr.
  Le ≠ gr. Ke ≠ la. Ke ≠ vi. Mc ≠ di. Mc ≠ vi.
  Mt ≠ ma,Mt ≠ di, Da ≠ di. Mt ≠ vi.
  Blo ≠ la, Blo ≠ di,Da ≠ ma. Ka ≠ ma.
  Br ≠ vi. Gr ≠ la, Blo ≠ ma.
  labeling([Ho,Ke,Mc,Ra,Le,Ru]),
  Fo ≠ Ho,Fo ≠ Mc,Fo ≠ Ra.Wi ≠ Ho.Wi ≠ Mc.
  Wi ≠ Ra, Wi ≠ Ke, Ru ≠ Fo. Br ≠ Ho.
  Br ≠ Mc, Le ≠ Blo, Ra ≠ Gr. Fo ≠ Ke.
  labeling([Bl,Br,Gr,Re,Au,Blo]),
  Gr ≠ Bo, Re ≠ Da, Gr ≠ Fo, Re ≠ Mt.
  Blo ≠ Da, Bl ≠ Bo, Bl ≠ Da,
  labeling([Fo,Wi,Mt,Bo,Da,Ka]).

labeling([]).
labeling([X|Y]) ←
   member(X,domain(X)),
   out-of(X.Y).
   labeling(Y).

out-of(X,[]).
out-of(X,[F|T]) ←
    X ≠ F,
    out-of(X,T).
```

The labeling procedure is used instead of the permutation procedure in order to assign to variables only values in their possible set. The procedure labeling(L) holds if all elements of the list L are different (which is insured by the "out-of" predicate). The list L must include only ground terms or d-variables. In the latter case, the domain of the variable is used as generator by the "member" predicate. The main difference between the two programs is that the second one immediately solves most of the constraints and therefore reduces immediately the search space. The constraints are solved once for all. Consider the case of the constraint "Fo $\neq$ Ho". This constraint is solved after the first labeling precedure instead of after three permutations. Moreover, when encountered, it will reduce the possible set of "Fo". Other constraints also reduce this set or assign values to variable. The choices are made in smaller domains and only a few constraints depend on them. No pathological behaviour (like in the case of simple backtracking) will arise. This allows us to move from a "generate and test" strategy towards a **"constrained-search"** strategy for problem solving.

If a delay mechanism is used for the non-equality predicates, in the first program, all the constraints can be written first and will be tested as soon as possible (i.e. in this case when the two variables are instantiated. but the above-mentionned problem remains as the constraints are used passively. *However, the program where labeling predicates are replaced by alldifferent predicates (i.e. alldifferent([Ho,Ke,Mc,Ra,Le,Ru]), alldifferent([Bl,Br,Gr,Re,Au,Blo]), alldifferent([Fo,Wi,Mt,Bo,Da,Ka])) will solve the problem if a delay mechanism is combined with our basic mechanism.* The predicate alldifferent(L) holds if all the elements of the list L are not equal. It can be defined by the following clauses.

```
alldifferent([]).
alldifferent([X|Y]) ←
 out-of(X,Y) ,
 alldifferent(Y).
```

This predicate is the same as the Colmerauer's one but it is used here in an active way instead of in a purely passive way in (Colmerauer, Kanoui and Van Caneghem, 1983). There, an non-equality predicate is selected as soon as both arguments are ground. In our case, it is selected as soon as one of these arguments is ground and can assign values to variables. It entails that this problem can be solved without generation of values and thus without choices (!): **the program just solves the constraints**. This is indeed a particular case but it shows how the search space can be reduced with a simple extension. This will be very important for interesting (NP-complete) problems. In this case, it is very important to reduce as soon and as much as possible the search space in order to avoid the combinatorial explosion. *The point here is twofold: first, active constraints are used to reduce in an "a priori" manner the search space and thus avoids the pathological behaviour of backtracking. Second, combined with a delay mechanism, it allows a data-driven computation. Constraints reduce the possible sets of the variables. As soon as the domain has a cardinality one, a value is assigned to this variable and this fact is propagated by allowing other constraints to be selected.*

5. Others features of the extensions.

Our extension can be considered as a set of primitives which can be used to build more sophisticated mechanisms and heuristics. An example is the "first fail principle" (Haralick and Elliot, 1980). Forward checking (and other search procedures) can be substantially improved by using the so-called "to succeed, try first where you are most likely to fail" heuristics. This heuristics can be implemented in CSP by choosing the most constrained variables to be instantiated first. Consider the labeling procedure seen before. It can be rewritten as

```
labeling([]).
labeling([X|Y]) ←
   choose-var([X|Y],Var,Other) ,
   member(Var,domain(Var)) ,
   out-of(Var,Other),
   labeling(Other).
```

The procedure choose-var(L,Var,Other) holds if Var is the element of L whose domain cardinality is the smallest one and Other is the list of other variables of L. The domain cardinality of a variable can be computed by a goal "← length(domain(V),Lg)" where the procedure length(L,Lg) holds if Lg is the length of the list L. It is clear that further efficiency can be obtained by building in the "choose-var(L,Var,Other)" predicate. This heuristics is particularly well suited for many problems like map (graph) coloring problems where many guidelines are known. In usual logic programs for CSP, the "efficiency" is greatly affected by the order of the litterals inside a clause or the order of arguments in predicates like permutation. Such an order must be determined statically and requires a deep analysis of the problem. With our extension, the order of instantiation can be determined dynamically and requires no analysis of the problem. It seems very difficult to get a similar effect in usual logic languages without rewriting all the program in order to manipulate explicitely the domains. In (Van Hentenryck and Dincbas, 1986), it is shown how arc-consistency and others more sophisticated mechanisms, like the reasoning on intervals of (Lauriere, 1978), can be implemented easily with the primitives presented here. *The point here is twofold: first, our basic mechanisms are sufficiently powerful to implement more sophisticated mechanisms which requires a lot of programming effort in usual logic language. This gives to the user the opportunity to define his own mechanisms if necessary. Also, the user is not restricted to a particular strategy for applying these mechanisms. Second, there exist specific mechanisms which are often used and which can substantially reduce the search space.*

Therefore, it may be worth to provide these mechanisms as primitives once the domain extension has been provided

6. Implementation issues.

The implementation of this extension entails no overhead when not used and can be implemented efficiently, two conditions stated by (Shapiro, 1983). What are the modifications required by our basic mechanisms ? In the variables environment, besides the usual information, a pointer to the domain (or more precisely the possible set) must be provided. In the following, we consider only the case where the domains are defined as a set of consecutive integers. This is in no way restrictive. Indeed, a correspondance can be made at implementation level between a finite set of constants and a set of consecutive integers. Then, it is clear that this can also be done for set of integers and that we have a direct access to elements of the domain. Therefore, only a boolean array "a" is necessary to represent the domain of a d-variable. At the beginning, all the booleans are true but the constraints can modify them. At any time of the computation, if a[i] = true then this means that i is in the possible set of the variable. Otherwise, it is not. However, it can be interesting to store the minimum and maximum indices. In this case, the possible set of a d-variable is given by all the values between the minimun and the maximun such that a[i] is true. The resolution of a non-equality predicate "$X \neq i$" consists in accessing a[i]. If it is true, a[i] must be set to "false". If necessary, the variable must be put on the trail with a pointer to i. When backtracking, the only thing to do is to reset a[i] to true. In general, the inequality case is more complicated as a list of values could need to be reset. However, if maximum and minimum values are stored, only these values must be modified and thus reset when backtracking occurs. However, a set of values must be stored if we unify two d-variables. In any case, this modification can be implemented efficiently (especially when combined with a delay mechanism).

7. Conclusion

An extension of logic programming languages has been proposed which increases their efficiency when solving CSP. It is based on domain declarations, a slight modification of unification, the redefinitions of some built-in predicates $<$, $\leq$, $>$, $\geq$, $\neq$ and a new evaluable function. Its main advantages are to bring active use of constraints into logic programming and to allow look ahead strategies, first fail heuristics, consistency techniques and the like to be implemented efficiently without programming effort and the need for extra control informations. The efficiency of logic programs for solving CSP is substantially improved by avoiding the pathological behaviour of backtracking and by reducing the search space in an "a priori" manner. When combined with a delay mechanism, this lead to a "data-driven" computation which applies constraints actively as soon as possible. It has been shown how more sophisticated mechanisms can be built from the primitives and that such extensions can be implemented efficiently.

References

1. Clark, K.L., Mc Cabe, F. The control facilities of IC-Prolog. In *Expert systems in the micro-electronic age.*, ED Mitchie D. Edinburgh university press., 1979.

2. Cohn, A.G. Improving the Expressiveness of Many Sorted Logic. AAAI-83, Washington DC, 1983.

3. Colmerauer, A., Kanoui, H., Van Caneghem, M. "Prolog, bases theoriques et developpements actuels.". *T.S.I. (techniques et sciences informatiques). 2*, 4 (83), 271-311.

4. Dincbas, M. , Lepape, J.P. Metacontrol of logic program in METALOG. Proceedings of FGCS'84., Tokyo, Japan, November, 84, pp. 361-370.

5. Freuder E. C. "Synthesizing constraint expressions". *Comm ACM 21* (November 1978), 958-966.

6. Gallaire, H. Logic programming: further developments. IEEE symposium on logic programming, Boston, july, 85, pp. 88-99. Invited paper.

7. Haralick R.M., Elliot G.L. "Increasing tree search efficiency for constraint satisfaction problems.". *Artificial intelligence 14* (80), 263-313.

8. Kowalski R. Directions of logic programming. Proceedings of the IEEE international symposium on logic programming, Boston (USA), 85. invited paper.

9. Lauriere J.L. "A language and a program for stating and solving combinatorial problems". *Artificial intelligence 10* (1978), 29-127.

10. Mackworth, A.K. . "Consistency in network of relations". *Artificial intelligence 8*, 1 (1977), 99-118.

11. Mycroft, A. , O'Keefe R.A. "A Polymorphic type system for Prolog". *Artificial intelligence 23*, 3 (1984), 295-307.

12. Naish L. "Automating control for logic programs". *Journal of logic programming 2*, 3 (october 1985), 167-184.

13. Shapiro E. Methodology of logic programming. Proceeding of logic programming workshop. Proceedings of logic programming workshop, Praia da falencia, Portugal, 26-june 1-july, 1983, pp. 84-93.

14. Sussman, G.J. , Steele, G.L. "CONSTRAINTS: a language for expressing almost-hierarchical descriptions". *Artificial intelligence 14*, 1 (1980), 1-39.

15. Van Hentenryck, P. ,Dincbas M. Associating domain to variables in order to solve C.S.P. in logic programming. lp-10, E.C.R.C (European computer-industry research center), February, 86.

16. Walther, C. A mechanical Solution of Shubert's Steamroller by Many-sorted resolution. 4th National Conference on Artificial Intelligence (AAAI-84), Austin, 1984.