Implementing Finite-domain Constraint Logic Programming on Top of a PROLOG-System with Delay-mechanism

Danny De Schreye*, Dirk Pollet, Johan Ronsyn, Maurice Bruynooghe**
Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, 3030 Heverlee, Belgium.

Abstract. In the past few years, an extensive amount of empirical evidence has proved the practical value of finite-domain constraint logic programming (CLP). Using special CLP-systems, many constraint satisfaction applications have been programmed very quickly and the resulting programs have a good performance. In this paper, we show how to implement a finite-domain CLP on top of a PROLOG-system equipped with a delay mechanism. The advantages are that the language features are easy to implement, the overhead caused both to the underlying PROLOG-system and to the CLP-environment itself are small and that the system is relatively portable.

1. Introduction

The pioneering work of A.Colmerauer on PROLOG II [Colmerauer 82] and PROLOG III [Colmerauer 87], the theory developed by J.Jaffar and J.L. Lassez [Jaffar & Lassez 87] and the development of systems such as CHIP [Dincbas et al 88b] and CLP(R) [Jaffar & Michaylov 87] have established the importance of constraint logic programming. The CHIP-system, using finite domain, is an elegant and powerful tool for dealing with a large class of scheduling problems. The many applications, (e.g. [Dincbas et al 88a], [Graf et al 89], [Van Hentenryck 89]), provide empirical evidence for the fact that with finite-domain CLP, a programmer can quickly solve complex constraint problems and obtain a high run-time efficiency at the same time. The underlying paradigm is that a programmer declaratively formulates the constraints of the problem at hand in a generate-and-test-like manner, and that the execution mechanism of the CLP-environment applies various efficient constraint satisfaction techniques (e.g. forward checking, looking ahead, first-fail principle) to solve this set of constraints. Therefore, an efficient run-time behaviour is obtained in combination with a low development time.

CLP differs from classical constraint problem solving (CPS) techniques in that it does not rely on the construction and manipulation of a dependency graph. However, by enforcing an advanced control regime on the coroutined activation of the given generators and constraints, it can simulate the effect of various relaxation algorithms which are used in CPS to simplify and solve the dependency graph. In this sense, CLP can be considered as a CPS-shell. We refer to the comments of P.Van Hentenreyck at [Dechter 89] for a more elaborated position on this point.

The main problems with applying finite-domain CLP to solve concrete scheduling (or CPS) problems are of a pragmatic nature. The CHIP-system is still not available on the software market. Also, as far as we know, none of the commercial PROLOG-manufacturers have developed an extension to their product to include CLP.

On the other hand, scheduling problems continue to form a source of expenses, in

* supported by the Belgian I.W.O.N.L.-I.R.S.I.A. under contract number 5203.
** supported by the Belgian National Fund for Scientific Research.

the form of management-and production-bottle-necks, in many companies. Thus, the demand for a good (elegant and efficient) CPS-shell is very high. This is why we are currently faced with an urgent need for alternative implementations of (finite-domain) CLP.

There are several approaches for building such an implementation. One is to take an existing PROLOG and to extend its execution model, as it was done in CHIP. This is a good approach, but it is also a very difficult one, requiring a deep understanding of the WAM-architecture.

In [De Schreye & Bruynooghe 89] we present a different approach. The starting point is the observation that the main differences between a simple generate-and-test program and a logically equivalent (computing the same answer substitutions for the top-level query) forward checking program are that:

1. The forward checking program generates by selecting values from predefined (finite) domains.
2. In the naive program, constraints act as passive tests, while in the forward checking program they actively reduce the size of the search space (by eliminating values from the domains).
3. In the naive program, all calls to generators precede the constraints. In the forward checking program, the activation of the generators and the constraints is interleaved using a special control rule.

Based on these observations, the implementation proposed in [De Schreye & Bruynooghe 89] consists of two components: a logic component and a control component. The logic component is encoded in a set of library predicates, written in PROLOG. They support the specification of new (finite) domains, they assign copies of these domains to the domain-variables and they redefine the built-in constraints, so that they can act actively on the domains. The control component delays the generators and activates the constraints as soon as all but one of their domain-variables have obtained a value.

In [De Schreye & Bruynooghe 89] we used the program transformation technique *Compiling Control* (see [Bruynooghe et al 89]) to compile the control component. In this way, we obtain a standard PROLOG program, imitating the forward checking execution, by starting from a generate-and-test program and a set of library predicates. For simple constraint problems, such as the N-queens problem and the five-houses puzzle, our results were comparable to those obtained with the CHIP-system. However, for real-world, large size constraint problems, the transformation phase becomes computationally very expensive. The reason is that the control compilation requires a complete (abstract) analysis of the behaviour of the given program under the new control rule. Also, the framework was not very flexible, in the sense that extensions, such as looking ahead, the first-fail principle and optimization techniques, caused various problems and therefore where not included.

In this paper we describe how a delay mechanism can be used to replace the control compilation. We show how the framework of [De Schreye & Bruynooghe 89] can be reformulated using the delay-declarations (WHEN-declarations) of NU-Prolog [Thom & Zobel 88]. We also extend the framework to include support for first-order looking-ahead and two versions of the first-fail principle (in addition to the forward checking). A general optimization procedure was also included in the original version of this paper, but it is omitted because of space restrictions (details are available from the authors). We believe that on the basis of our treatment, the reader should be able to program his own

(finite-domain) CLP-environment on top of PROLOG (provided that the PROLOG-system includes a delay-mechanism, which is the case for several systems, such as NU-Prolog, Sicstus, Prolog II).

2. The integration of domains

In this section we introduce some basic library predicates that support the generation of candidate solutions to a given problem, by selecting values from predefined domains. The important thing about them, is that the programmer is not confronted with the task of explicitly manipulating these domains. The predicates in this section are simplified versions of the ones we will propose (later in the paper) for the final environment. Some of the material in this section was already introduced in [De Schreye & Bruynooghe 89]. We recall it here, to make the paper self-contained.

In addition to a redefinition of all the built-in-constraints (an example is given at the end of the section), we introduce two library predicates: create_domain/3, which is used to construct a prototype of a finite domain, and select_from_domain/2, which is used to assign a value (from a domain) to a domain-variable. In Box 1, we give an example of how these library predicates are used in an application. The example is the N-queens problem. We follow the syntax of NU-Prolog (see [Thom & Zobel 88]) throughout the paper.

```
?- forward(test/3).

queens(N,Q):-  create_domain(queens,gendom(D,N),D),
                  gen(N,Q), safe(Q).
gendom([],0).
gendom([N|D], N):-  N>0, M is N-1, gendom(D,M).
gen(0,[]).
gen(N,[H|T]):-  N>0, M is N-1, select_from_domain(queens,H), gen(M,T).
safe([]).
safe([X]).
safe([X,Y| Z]):-  n-attack(X,1,[Y|Z]), safe([Y|Z]).
n_attack(X,T,[]).
n_attack(X,T,[Y|Z]):-  test(X,T,Y), S is T+1, n_attack(X,S,Z).
test(X,T,Y):-  Y \= X, Y \= X+T, Y \= X-T.
```

Box 1: The N-queens program.

Notice how this program resembles the usual generate-and-test formulation for the N-queens problem. The differences are: it includes a call to the library predicate create-domain/3 to create the finite domain [1,2,...,N], the usual call to permute, generating a permutation of the list [1,2,...,N], is replaced by a call to the predicate gen/2, which on its turn calls the library predicate select_from_domain/2 to instantiate each domain-variable, and there is a declaration "?- forward(test/3)" specifying that all the built-in constraints in the definition of test/3 should be executed with a forward checking algorithm. All other predicates are identical to the usual formulation. The way in which the program manipulates the domain is hidden in the definition of the two library predicates, given in Box 2.

The first two arguments of the predicate create_domain/3 are input. The first is an atom, which serves as a unique identifier for the defined domain. The second argument is a procedure call, which either computes a list D (the third argument) of all objects in the

domain, or it is equal to true, in which case the list of domain-values, D, is given as input to the third argument. The reason for these alternatives is that the domain may or may not be dependent on certain of the problems input parameters (e.g. the number of queens, N).

```
create_domain(Id,Proc,D):-  call(Proc), real_create(D,D1),
                            retractall( dom(Id,_)), assert( dom(Id,D1)).
real_create([],[]).
real_create([H|T],[(H,_)|TD]):-  real_create(T,TD).

select_from_domain(Id, (&,Id,X,Xdom)):-  dom(Id,Xdom), member( (X,1), Xdom).
member(X,[X|_).
member(X,[H|T]):-  member(X,T).
```

Box 2: Initial form of the library predicates[1].

The procedure create_domain/3 first calls the procedure Proc. After the successful completion of Proc, the third argument of create_domain/3 contains the domain-list, D. Then, the auxiliary procedure, real_create/2 changes the representation of the domain, and finally, this new representation, D1, is asserted as dom(Id,D1).

The new domain representation is a list of pairs, each consisting of a domain-value and a fresh variable (e.g. the domain [1,2,3] is represented as [(1,_), (2,_), (3,_)]). The reason why we use this partially instantiated representation is that we can now eliminate a value from the domain by merely instantiating the free variable associated with it to some reserved symbol (say 0). Thus, the list [(1,_), (2,0), (3,_)] represents the domain [1,3].

The procedure select_from_domain/2 picks up a copy of the domain identified by its first (input) argument, Id. Then, it does not associate a simple domain-value to its second (output) argument (which corresponds to the domain-variable in the call), but instead, it instantiates the domain-variable to a 4-tuple. This 4-tuple consists of a tag, &, identifying the variable as a domain-variable from that point on, the identification-atom of the domain, a fresh variable and a copy of the domain. Finally, the non-deterministic member/2-call picks a value from the domain (the next value which does not have a 0 associated to it) and assigns it to the fresh variable.

With these two predicates - and a proper redefinition of the \=-constraint -, obtaining a forward checking program has turned into a control problem. Indeed, if we partially execute a call to the select_from_domain/2-predicate, but we delay the corresponding call to member/2, then we have assigned a copy of the domain to the domain-variable, but we have postponed the actual assignment of a value. Using an appropriate redefinition of \=/2 (see below), we can now actively remove values from the domain of that variable. After this elimination of domain-elements, the call to member/2 is reactivated and the variable obtains its value from the reduced domain.

Clearly, the programmer must specify which predicates should be solved using a forward checking algorithm. He expresses this information in terms of declarations (e.g. the ?- forward(test/3) declaration for N-queens). The effect of such a declaration is that, in a preprocessing phase, the users program is transformed into a program in which every

1) The compound structures (&,Id,X,Xdom) and (X,1) in these definitions make use of a special PROLOG-notation. In PROLOG, (a,b,c) is short for ','(a, ','(b,c)) , where the single quotes distinguish between the two uses for the symbol ",".

call to a built-in-constraint in the definition of test/3 (or every call to the predicate itself, in the case of a declaration concerning a built-in-constraint, e.g. ?- forward(\=/2)) is replaced by the corresponding call to the forwardly redefined built-in. In our example, we get the new definition for test/3 of Box 3.

The definition for the forward version of \=/2, namely for\=/2, deals with a large number of cases. Some of the clauses (relevant for our example) are also shown in Box 3. The will refine this definition in the next sections.

```
test(X,T,Y):-  for\=(X,Y), for\=(Y,X+T), for\=(Y,X-T).
for\= ( (&,_,X,_), (&,_,Y,_) ):-
            ground(X), ground(Y), !, X <>Y.
for\= ( (&,_,X,_), (&,_,Y,DY) ):-
            ground(X), !, real_for\= (DY, X).
for\= ( (&,_,X,DX), (&,_,Y,_) ):-
            ground(Y), !, real_for\= (DX, Y).
for\=( (&,_,X,_), (&,_,Y,_) + T):-
            T <> (&,_,_,_),
            ground(X), ground(Y), !, X <> Y+T.
for\=( (&,_,X,_), (&,_,Y,DY) + T):-
            T <> (&,_,_,_),
             ground(X), !, S is X-T, real_for\=(DY,S).
...
real_for\= ([], _).
real_for\= ([(X,Y)|T],S):-  ( X=S, !, Y=0  ;  real_for\= (T,S) ) .
```

Box 3: The forward redefinition of \=/2.

We conclude with some comments on the special form of the values that are assigned to the domain-variables by the select_from_domain/2-predicate. The tag-field, &, identifies a variable as a domain-variable. Such an identification is important, because a number of built-in predicates will have an undesired behaviour if we can not capture (and redefine) the case of a domain-variable occurring as one of the arguments in the call. As an example, it is clear that the effect of calls such as write(X) or X=3 will have an unexpected behaviour if X is a domain-variable. These predicates need to be redefined. For write/1 we could define:

```
new_write(X):-  X = (&,_,Y,_), !, write(Y).
new_write(X):-  write(X).
```

and replace every call to write/1 in the users program by a corresponding call to new_write/1 in the preprocessing phase. Similarly, we can deal with the =/2 predicate.

Clearly, redefining the built-in predicates does not take care of problems caused by user-defined predicates which act directly on the domain-variables. For instance, if the domain consists of pairs (month(Nr1), year(Nr2)), then the user should not define a predicate

```
get_year( ( month(_), year(X) ), X) .
```

and make a call to it with a domain-variable as its argument. This will fail, because of the new structure (the 4-tuple) assigned to the variable. Again, this can be solved in the preprocessing phase. All implicit unifications in the heads of clauses can be transformed to equivalent explicit unifications in the body of the clauses (normalization). In the example, we get the new clause:

```
get_year( X, Y):-  X = ( month(_), year(Z) ), Z = Y.
```

Then, the appropriate redefinition of =/2 will deal with the unification of a domain-variable and a non-domain-variable (or constant).

A more simple solution is that the programmer should at least be aware of the type of representation used for the domain-variables, and that - in those cases where he needs access to a substructure of an actual domain-value - he should write his program accordingly.

The second component of the 4-tuple, the identifier of the domain, will not be mentioned any further in the remainder of the paper. It is included to make the forward redefinition of certain built-in constraints more efficient. For instance, =/2 renamed as for=/2 includes the clause:

for= ((&, Id, X, DX), (&, Id, Y, DY)):- !, X=Y, DX = DY.

If the Id's for X and Y are not equal, then much more complicated actions need to be taken.

3. The integration of the control

In [De Schreye & Bruynooghe 89], the logic described in boxes 1,2 and 3 (in the version obtained after the preprocessing phase) is used as input to a program transformation system (Compiling Control, see [Bruynooghe et al 89]). The second input to the transformation system is a specification of the new computation rule described above. The transformation synthesizes a new PROLOG-program that behaves, under the standard computation rule of PROLOG, as the old program behaves under the new rule.

We now discuss the problem of enforcing the appropriate control on the given (extended) generate-and-test program, by using delay-predicates. We use the WHEN-declarations of NU-Prolog (see [Thom & Zobel 88]).

Essentially, the control that we want to obtain is the following. Calls to member/2 must be delayed for as long as there are constraints available which can be forwardly executed. A constraint C(X1,...,Xn) is forwardly executable if all but one of its domain-variable-arguments are instantiated. Every constraint is delayed until it is forwardly executable.

We are now faced with a technical problem. As far as we know, it is impossible to formulate a WHEN-declaration on the member/2-predicate, which initially delays every call to the predicate and reactivates them as soon as all the constraints have been generated, but none of them is forwardly executable.

However, a simple way to achieve this control, is to place the member/2-calls behind all the constraints in the initial program and to formulate WHEN-declarations on the constraints (but not on the member/2-predicate). This corresponds to the methodology for using WAIT-declarations proposed in [Naish 86]: tests must be placed before generators and tests are delayed until they are sufficiently instantiated.

To achieve this in our framework, it is necessary to adapt the library-predicate select_from_domain/2. We replace it by two new predicates: one to assign domains to domain-variables: assign_domain/2, and one to instantiate the variables (essentially the member/2-calls): instantiate/1.

For the N-queens problem, this leads to the new program-structure of Box 4 (which forms the final basis of the framework we propose). The definition of the library predicates is given in Box 5.

The predicate assign_domain/2 is identical to select_from_domain/2, except for the call to member/2, which was omitted. Instantiate/1 consumes a list of domain-variables

```
?- forward(test/3).

queens(N,Q):-   create_domain(queens, gendom(D,N), D),
                  generators([ gen(N,Q)], V),
                  safe( Q),
                  instantiate( V).
gen(0,[]).
gen(N,[H|T]):-  N>0, M is N-1, assign_domain(queens, H), gen(M,T).
gendom/2, safe/1, n_attack/3 and test/3 : as in Box 1.
```

Box 4: Final version for the N-queens program.

```
create_domain/3 : as in Box 2.

assign_domain(Id, (&, Id, X, D)):-   dom(Id, D).
instantiate([]).
instantiate([(&,_,X,D)|T]):-   member( (X,1), D), instantiate(T).
generators([], []).
generators([Proc|T],V):-   solve_accumulate(Proc, [], V1),
                             generators(T,V2), append(V1,V2,V).
solve_accumulate(true, V, V).
solve_accumulate( (A, B), Acc, V):-   solve_accumulate(A, Acc, V1),
                                        solve_accumulate(B, V1, V).
solve_accumulate(assign_domain(Id,X), Acc, V):-   !, call( assign_domain(Id,X)),
                                                    append(Acc, [X], V).
solve_accumulate( A, V, V):-   builtin(A), !, call(A).
solve_accumulate( A, Acc, V):-   clause(A, B), solve_accumulate(B, Acc, V).
```

Box 5: The library predicates.

and assigns a value to each of them. The list of domain-variables is produced by the predicate generators/2. This predicate has a double function: it calls a list of user-defined generator-predicates and it accumulates a list of all the domain-variables created by these generators. To do this, it makes use of the solve_accumulate/2-predicate, which defines a PROLOG-meta-interpreter. Only, this meta-interpreter accumulates in his second argument the list of all domain-variables generated so far.

To make the code more readable, we included calls to append/3 in the various procedures. These can be eliminated and replaced by accumulating parameters. Also, we defined the solve_accumulate/3-predicate as a variant of the vanilla-interpreter. In order to deal with full PROLOG, it should be redefined as a variant of a complete PROLOG-meta-interpreter.

Finally, there is the control component itself. This is formulated at the level of the (forward redefinitions of the) constraints. The forward definition of each built-in constraint, forC(X1,...,Xn), is accompanied by a WHEN-declaration. The declarations delay each call to the constraint, if less than n-1 of its domain-variable-arguments are instantiated.

This raises a problem, since it is not clear at compile-time, which arguments of a given built-in-constraint will correspond to domain-variables in the users program and which will not. Therefore, on the highest level in the forward definition of each constraint, we make a case study. It distinguishes between different patterns in the call, on the basis of the presence of domain-variables. Some examples in the case of \=/2 are:

```
for\= (X,Y+Z):-   X= (&,_,_,_), Y = (&,_,_,_), Z = (&,_,_,_), !,
                  for\=ddplusd(X,Y,Z).
for\= (X,Y+Z):-   X= (&,_,_,_), Y = (&,_,_,_), !,
                  for\=ddplusa(X,Y,Z).
...
```

Then, accompanying each definition of the more specialized constraints, for\=ddplusd , for\=ddplusa, ..., we have a WHEN-declaration. As an example, we have:

```
for\=ddplusd( (_,_,X,_), (_,_,Y,_), (_,_,Z,_) )
        WHEN  (ground(X) and ground(Y))  or
              (ground(X) and ground(Z))  or
              (ground(Y) and ground(Z))
```

4. Extending the framework

In this section we briefly discuss the following extensions: first-order looking ahead and two versions of the first-fail principle.

4.1 First-order looking-ahead

It is generally believed (see [Dechter 89]) that for most applications, higher order looking-ahead does not pay off. Forward checking seems an ideal combination in view of the trade-off between actively reducing the size of the search-space and avoiding too much overhead. However, for some problems, first-order looking-ahead may still be a sensible alternative.

Assume that we have a program including a call X < Y+Z, where X,Y and Z are domain-variables, and that the programmer wants the constraint to be dealt with by a first-order looking-ahead algorithm. To achieve this, he adds the declaration:

?- look-ahead(</2).

As a result, the call is transformed to look<(X,Y+Z) by the preprocessor. One of the clauses (the one dealing with the case where X,Y and Z are domain-variables) defining this predicate in the library is given in Box 6.

```
look<ddplusd(X,Y,Z):-  min_value(X,Xmin),
                       max_value(Y,Ymax),
                       max_value(Z,Zmax),
                       Xbound is Ymax+Zmax,
                       Ybound is Xmin-Zmax,
                       Zbound is Xmin-Ymax,
                       restrict_domain(X, </2, Xbound),
                       restrict_domain(Y, >/2, Ybound),
                       restrict_domain(Z, >/2, Zbound),
                       for<ddplusd(X,Y,Z).
```

Box 6: Part of the first-order looking-ahead definition for </2.

Here, min_value(X,Xmin) succeeds if Xmin is the minimal value of the domain of X. The predicate-call restrict_domain(A, Constraint/2, Value) restricts the domain of the domain-variable A to all values X, such that Constraint(X, Value) holds. Finally, for<ddplusd/3 is the forward checking predicate for X < Y+Z, given that X,Y and Z are domain-variables.

4.2 The first-fail principle

According to the first-fail principle, it is more efficient to have a failing constraint very

early in the search. In finite-domain CLP, this principle translates to: if we are going to instantiate a domain-variable, then we should first select the variable which is most strongly constrained. This on its turn can be reformulated using two concrete conditions:

1. Select the variable with the smallest domain,
2. Select the variable that occurs in the largest number of constraints.

Implementing these two rules in our environment is particularly easy, because each generate-and-test block in the users program ends with a call to instantiate(V), where V is the list of all the domain-variables.

Now assume that instead of instantiating the domain-variables with 4-tuples, we use 5-tuples of the form (&, Id, Length, X, Domain) - to deal with the smallest domain principle - or even 6-tuples, (&, Id, Length, Constraints, X, Domain), - to deal with both the first-fail principles.

The new variables, Length and Constraints, are open ended lists. The last ground member of the list Length is an integer representing the current length of the domain. Similarly, the last ground member of the list Constraints is an integer representing the number of constraints containing the considered domain-variable.

With this information available, the instantiate/1-predicate can be redefined to select (and instantiate) the domain-variable with the smallest domain, as in Box 7. Here, select_on_Length(V, X, V1) succeeds if X is a member of V having a minimal value for its Length-variable and V1 is the list obtained by removing X from V.

Similarly, we can perform a selection on the basis of the number of constraints which contain the domain-variable, or, we could combine the two first-fail principles into new selection/instantiation-predicate instantiate_L_C/1. In this case, we would first compute the sublist of all domain-variables with a minimal domain-length and then select one of them occurring in a minimal number of constraints.

```
create_domain_L(Id, Proc, D):-  call(Proc), real_create_L(D,D1,0,Length),
                                retractall(dom(_,_,_)),assert(dom(Id,D1,Lenght)).
real_create_L([], [], L, L).
real_create_L([H|T], [(H,_)| TD], N, L):-   M is N+1, real_create_L(T, TD, M, L).
assign_domain_L(Id, (&,Id, L, X, D) ):-  dom(Id, D, L).
instantiate_L([]):-  !.
instantiate_L(V):-  select_on_Length(V, X, V1), X= (_,_,_,Y,D),
                    member( (Y,1), D), instantiate_L( V1).
generators/2 : as in Box 5.
```

Box 7: Version of the library predicates supporting first-fail on domain-size.

For each domain-variable, the values of its Length-(and Constraints-)variable are initialized in a new version of the real_create/2-predicate, and updated in (new) forward versions of the built-in-constraints. While the real_create_L/2-predicate of Box 7 creates the new representation of the domain, it also counts the initial length of the domain and asserts it, along with the new representation. Clearly, it the case of the second first-fail principle, the initial value for Constraints is always 0. Later on, whenever a forward version of a constraint eliminates a value from the domain of some domain-variable (see the definition of real_for\=/2 in Box 3 for an example), it must now update the Length-variable at the same time.

Some additional optimizations can be carried through here. For instance, if the constraint computes a new length 0 for a domain-variable, then the constraint should fail. Also, if it computes a length equal to 1, then it can instantiate the domain-variable itself

to its only remaining value. This may lead to a quicker reactivation of some of the other delayed constraints.

Updating the Constraints-variables is also performed in the redefined constraints. As an example, assume that we have a constraint X = Y, where X and Y are domain-variables, that there is a user-defined predicate cons/n including the call X = Y in its definition and that there is a declaration ?- forward(cons/n). The definition of for=dd/2 (without first-fail) is:

```
?- for=dd((_,_,X,_), (_,_,Y,_)) WHEN ground(X) or ground(Y)
for=dd( (&,Id,X,DX), (&,Id,Y,DY)):-   !, X=Y, DX = DY.
for=dd( (_,_,X1,_), Y):- ground(X1), !, Y = (_,_,Y1,_),
                                 Y1 = X1, restrict_domain( Y, =/2, X1).
for=dd( X, (_,_,Y1,_)):- ground(Y1), !, X = (_,_,X1,_),
                                 X1 = Y1, restrict_domain( X, =/2, Y1).
```

where the predicate restrict_domain/3 is the one mentioned in Box 6. The corresponding forward definition of =/2, named for=dd_C/2, which also updates the Constraints-variable of both the domain-variables occurring in the arguments of the call is given in Box 8.

```
for=dd_C(X,Y):-  constraints_plus_one([X,Y]),
                 real_for=dd_C(X,Y).
?- real_for=dd_C( (_,_,_,X,_), (_,_,_,Y,_) ) WHEN ground(X) or ground(Y)

real_for=dd(X,Y):-  for=dd_clauses(X,Y),
                    constraints_minus_one([X,Y]).
for=dd_clauses( (&,Id,_,X,DX), (&,Id,_,Y,DY)):-   !, X=Y, DX = DY.
for=dd_clauses( (_,_,_,X1,_), Y):- ground(X1), !, Y = (_,_,_,Y1,_),
                                 Y1 = X1, restrict_domain( Y, =/2, X1).
for=dd_clauses( X, (_,_,_,Y1,_)):- ground(Y1), !, X = (_,_,_,X1,_),
                                 X1 = Y1, restrict_domain( X, =/2, Y1).
```

Box 8: Forward version of =/2 updating the Constraints-variables.

Here, the predicates constraints_plus_one/1 and constraints_minus_one/1 update the value of the Constraints-variable of each domain-variable in their list-argument in the obvious way. Also, for=dd_clauses/2 is identical to for=dd/2, except that the WHEN-declaration is omitted.

It should be clear that the use of the various first-fail mechanisms produces a lot of overhead. The programmer must decide whether the first-fail principle should be included or not. To express his choice, he can use various declarations. Namely:

?- first_fail_L. , or:

?- first_fail_L_C.

in addition to a declaration for each constraint cons/n occurring in his program:

?- forward(cons/n). , or:

?- look_ahead(cons/n).

This implies that there will be three different definitions for the library predicates real_create/2, assign_domain/2 and instantiate/1. One acts on 4-tuples, one on 5- and one on 6-tuples. Depending on the declarations, the appropriate definitions are loaded by the preprocessor. For each constraint, there are 9 different definitions: one for every combination of one of the options: no first-fail (default), first_fail_L and first_fail_L_C, with one of the options: ordinary execution (default - these will however take special care in handling domain-variables), forward execution and looking-ahead.

Discussion

We have described a PROLOG-implementation of finite-domain CLP. It makes use of a delay-mechanism, the WHEN-declarations of NU-Prolog. Based on our description, we believe that it should be easy for an experienced PROLOG-programmer to implement his own CLP-environment. On the other hand, the described implementation does not alter the underlying PROLOG-system in any way. Therefore, it does not cause overhead for programs which make no use of the CLP-environment. Also, the implementation is based on library predicates, not on a meta-interpreter (although the generators are executed under meta-interpretation). Thus, in general, we avoid the overhead of an extra layer of interpretation. Finally, it is usually not hard to define one delay-mechanism in terms of another. Since most PROLOG-systems will probably include a delay-facility in the near future, this makes our system relatively portable.

We developed various applications within the environment. Both typical test-examples, such as the N-queens problem and the five-houses puzzle, and real scheduling problems were successfully implemented. Below, we give a table with a comparison of different execution-times for computing the first solution to the N-queens problem. The comparison includes four programs: the simple generate-and-test program, a coroutining version of this program, the forward checking program of Box 4 and the forward checking program using first-fail on the length of the domains. The experiment was performed with NU-Prolog (and our environment) on a SUN3/280.

N	G & T	Corout.	For. Check.	For.Check.+F.F.
4	0.02	0.01	0.14	0.40
10	156.00	0.83	3.03	4.52
16	-	171.38	4.85	5.08

Table 1: Execution times for the first N-queens solution.

One of our more realistic experiments was the implementation of a program for the computation of a schedule for the nursing staff of a division in the university hospital of Leuven. The division includes 23 nurses. The schedule is made for a period of one month. Some of the nurses requests can not always be fulfilled. They are encoded into functions which must be optimized. Also, night-shifts and weekend-work can usually not be totally equally distributed within the given period of one month. Therefore, information of schedules for the previous months is kept in a database. Using this information, the distribution of the night-shifts and weekends are also performed by optimizing certain functions.

Computing an optimal schedule for one month takes the program approximately (depending on the actual constraints for that month) 2 minutes and 30 seconds. It takes a head-nurse more than one week to make a (less optimal) schedule by hand.

Clearly, there are many improvements that can be made to the implementation described in this paper. It was not our aim to present the ultimate efficient techniques, but to give a conceptual overview of a working design. A lot of search included in the implementation of the first-fail principle (scanning lists) can certainly be reduced by using more clever search-strategies. Several other refinements are possible as well.

Finally, on the level of the concepts, several improvements are desirable. The most important one is the use of hierarchies of constraints, as proposed in [Borning et al 89].

This provides a more elegant approach to the problem of optimizing a solution. Our further work will therefore focus on both types of improvements, in addition to some further experiments with large size applications.

References

[Borning et al 89] A.Borning, M.Maher, A.Martindale, M.Wilson, Constraint Hierarchies and Logic Programming, in Proc. of the Sixth International Conference on Logic Programming, eds. G.Levi and M.Martelli, 1989, pp.:149-164.

[Bruynooghe et al 89] M.Bruynooghe, D.De Schreye, B.Krekels, Compiling Control, J.Logic Programming, 1989 (6), pp.: 135-162.

[Colmerauer 82] A.Colmerauer, PROLOG II: Manuel de reference et modele theorique, Technical Report, GIA - Faculte de Science de Luminy, 1982.

[Colmerauer 87] A.Colmerauer, Opening the Prolog III Universe, BYTE Magazine, 12(9), 1987, pp.: 177-182.

[Dechter 89] R.Dechter, ed., Proc. Workshop on Constraint Problem Solving, IJCAI89, Detroit, 1989.

[De Schreye & Bruynooghe 89] D.De Schreye, M.Bruynooghe, The Compilation of Forward Checking Regimes through Meta-interpretation and Transformation, Proc. Workshop on Meta-programming in Logic Programming, MIT-Press, 1989.

[Dincbas et al 88a] M.Dincbas, H.Simonis, P.Van Hentenryck, Solving a Cutting-Stock Problem in Constraint Logic Programming, in Proc. of the 5th International Conference on Logic Programming, 1988, pp.: 42-58.

[Dincbas et al 88b] M.Dincbas, P.Van Hentenryck, H.Simonis, A.Aggoun, T.Graf, F.Bertheir, The Constraint Logic Programming Language CHIP, in Proc of FGCS88, 1988, pp.: 693-702.

[Graf et al 89] T.Graf, P.Van Hentenryck, C.Pradelles, L.Zimmer, Simulation of Hybrid Circuits in Constraint Logic Programming, in Proc. of IJCAI89, 1989, pp.:72-77.

[Jaffar & Lassez 87] J.Jaffar, J-L.Lassez, Constraint Logic Programming, in Proc. 14th ACM Principles of Programming Languages Conference, Munich, 1987.

[Jaffar & Michaylov 87] J.Jaffar, S.Michaylov, Methodology and implementation of a CLP System, in Proc. of the 4th International Conference on Logic Programming, 1987, pp.: 196-218.

[Naish 86] L.Naish, Negation and Control in Prolog, LNCS 238, Springer-Verlag, 1986.

[Thom & Zobel 88] J.Thom, J.Zobel, NU-Prolog Reference Manual, Version 1.3, Technical Report 86/10, Machine Intelligence Project, Computer Science Department, University of Melbourne, 1988.

[Van Hentenryck 89] P.Van Hentenryck, Constraint Satisfaction in Logic Programming, Logic Programming Series, MIT-Press, Cambridge, 1989.