An Extension of the WAM for Cooperative Interval Solvers

Frédéric Goualard and Laurent Granvilliers
LIFO, Université d'Orléans,
B.P. 6759, 45067 ORLEANS Cedex 2, France
Tel.: +33 2 38 41 72 95
{Frederic.Goualard,Laurent.Granvilliers}@lifo.univ-orleans.fr

Abstract: We present in this paper the implementation of `DecLIC`, a new CLP language achieving the cooperation of two different local consistencies, *viz.* hull-consistency [1] and box-consistency [3]. `DecLIC` is based on `clp(fd)` [8], a solver over Finite Domains compiling CLP programs into C via the *Warren Abstract Machine* (WAM) [15]. The extension of the WAM instruction set necessary to support `DecLIC` new features is described along with the new internal structures. A prototype has been implemented and is shown to be efficient on benchmarks from various domains.

Keywords: Constraint Logic Programming, CLP(Intervals) language, local consistency, cooperative solvers, WAM extension.

1. Introduction

Since its introduction in 1987 by Cleary [6], Relational Interval Arithmetic has been used in many CLP languages such as `Prolog IV` [4], `clp(BNR)` [11], `Unicalc` [12]. These solvers handle complex constraints by decomposing each of them into conjunctions of "primitives" (e.g. '$x + y = z$', '$x \times y = z$',...) as exposed in [6], and then enforcing some consistency over all the resulting constraints through domain propagation.

The drawbacks of such a method are by now well known: introduction of new fresh variables induced by the decomposition leads to a loss of reduction of the variable domains the user is interested in. Moreover, the constraint network thus obtained is far more complicated than the one which would have been obtained by considering complex constraints as a whole. This last point is a major problem when one wants to inspect the constraint store for debugging or profiling purposes.

Benhamou *et al.* have introduced in [3] a new consistency notion, called *box-consistency*, which allows —along with Newton-related methods— to process efficiently complex constraints without decomposing them. They have implemented their method in a CLP language called `Newton`, showing thus that box-consistency outperforms most of the other existing techniques on some standard benchmarks.

Nevertheless, as we will see below, box-consistency is not the solution of choice for all constraint systems to be solved; actually there exists many problems where it is less effective than decomposing constraints into primitive ones, then enforcing hull-consistency.

We have then decided to implement a new CLP language, `DecLIC` (Declarative Language with Interval Constraints), where several solvers cooperate. In particular, one of them handles constraints as a whole by enforcing box-consistency on them, and an other one decomposes constraints into "primitives", and then enforces hull-consistency. We thus benefit from the effectiveness of both methods, each one handling efficiently the constraints the other does not.

`DecLIC` has been implemented with `clp(fd)` [8] as a starting-point. One of the main achievement of `clp(fd)` was the extension of the Warren Abstract Machine (WAM) [15] in order to design a compiler translating Prolog-like programs describing problems to be solved over Finite Domains into C code. We have modified and extended the WAM instruction set described in [7, 8] to take into account the new possibilities offered by `DecLIC`. The resulting system is a compiler/interpreter able to solve efficiently constraint systems over integers, booleans and reals.

The remaining of the paper is organized as follows: section 2 recalls some definitions about domain propagation, hull-consistency and box-consistency, pointing out the advantages and drawbacks of both consistency notions; section 3 presents briefly an overview of `DecLIC`; section 4 exposes the implementation details, the new WAM instructions along with the new internal structures; section 5 gives experimental results on various standard benchmarks; and we conclude in section 6, discussing future developments.

2. Preliminary Notions

Due to lack of space, we will restrict ourselves in this section to the presentation of the notions we specifically need. Refer to [1, 2] for an introduction to the general framework.

Let $\mathbb{I}$ denote the set of (closed/opened) floating-point intervals, *viz.* all the compact sets included in $\mathbb{R}$ whose lowest and greatest elements are floating-point numbers (referred in the following simply as *intervals*). Let $\Sigma_1 = \langle\ \mathbb{R}, \mathcal{F}_1, \mathcal{R}_1\ \rangle$ be a structure where $\mathbb{R}$ is the set of reals, $\mathcal{F}_1$ a set of function symbols and $\mathcal{R}_1$ a set of relation symbols. Let $V = \{x_1, x_2, \ldots\}$ be a set of variables taking their value over $\mathbb{R}$.

We then define *a real constraint* as being an atomic formulae built from Σ_1 and V. In the same way, we define *interval constraints* over the structure $\Sigma_2 = \langle \mathbb{I}, \mathcal{F}_2, \mathcal{R}_2 \rangle$ and the set of variables $\{y_1, y_2, \ldots\}$. Given a real constraint $c(x_1, \ldots, x_n)$ (resp. an interval constraint $C(y_1, \ldots, y_n)$), ρ_c (resp. ρ_C) denotes the corresponding relation. $I = I_1 \times \cdots \times I_n$ is a Cartesian product of n intervals, called a *box*, standing for the domains of the variables appearing in the constraints. In other words, all the variable domains involved are intervals with machine numbers as bounds.

Informally, given a real constraint $c(x_1, \ldots, x_n)$, a constraint solver aims at reducing the domains of $x_1, \ldots, x_n$. This reduction process for c is described by *Constraint Narrowing Operators* [1] (CNO for short) which are contracting monotonic functions taking as input a box and returning a box from which have been discarded (some of) the elements which do not belong to ρ_c. As this "narrowing" cannot always be achieved perfectly when dealing with relations over reals, all the work is done over an *approximate domain* [2] of $\mathbb{R}$ (that is a subset of the power set of $\mathbb{R}$ with some properties —see *op. cit.*). For `DecLIC`, the approximate domain is $\mathbb{I}$. A *Constraint System* is defined as being a pair (S,X) where $S = \{(c_1, N_1), \ldots, (c_m, N_m)\}$ is a set of pairs made of a real constraint c_i and a CNO N_i for ρ_{c_i}, and X is a n-ary Cartesian product of intervals representing the domains of the variables appearing in the constraints.

Solving a constraint system relies on the computation of the greatest common fixed-point of all the CNO of the system by using the *Narrowing Algorithm* described in [3]. The key instruction is the application of any CNO N_i on X, which enforces some *consistency* for the associated constraint c_i. The main consistencies used in `DecLIC` are *hull-consistency* [1] and *box-consistency* [3], whose definitions are given below.

Definition 1 (Hull-consistency) *The real constraint $c(x_1, \ldots, x_n)$ is said* hull-consistent wrt a box I *iff* $\forall i \in \{1, \ldots, n\}$*:*

$$I_i = \mathsf{Hull}(I_i \cap \{a_i \in \mathbb{R} \mid \forall j \in \{1, \ldots, i-1, i+1 \ldots, n\} : \exists a_j \in I_j \ s.t.\ (a_1, \ldots, a_n) \in \rho_c\})$$

where $\mathsf{Hull}(\rho)$ *is the smallest interval containing the relation* ρ*.*

Hull-consistency is enforced over complex constraints by decomposing them into conjunctions of "primitives". The solver then uses Relational Interval Arithmetic [6] to enforce hull-consistency over each primitive (the overall consistency is obtained by domain propagation). The main advantage of such an approach is that computation of hull-consistency can be implemented very efficiently for the set of primitives the solver supports. Therefore, the solver is particularly efficient when constraints to be processed are "quasi-primitives". The drawbacks are that the introduction of new variables due to the decomposition process reduces drastically the ability of tightening the domains of the variables the user is interested in. As pointed out in [3], this is particularly true when the same variables appear more than once in the constraints.

The definition of box-consistency is based on the notion of *interval extension*.

Definition 2 (Interval extension) *An interval function* $F : \mathbb{I}^n \to \mathbb{I}$ *is an* interval extension *of* $f : \mathbb{R}^n \to \mathbb{R}$ *iff*

$$\forall I_1, \ldots, I_n \in \mathbb{I} \ : \ a_1 \in I_1, \ldots, a_n \in I_n \Rightarrow f(a_1, \ldots, a_n) \in F(I_1, \ldots, I_n)$$

In this paper, we use a particular kind of interval extension, called natural interval extension *in [14], defined as follows: given* $f : \mathbb{R}^n \to \mathbb{R}$*, its natural interval extension* F *is obtained from* f *by replacing in the expression of* f *each constant* a *by* $\mathsf{Hull}(a)$*, each operation by its corresponding interval operation and each variable by an interval variable.*

An interval constraint $C(y_1, \ldots, y_n)$ *is an* interval extension *of a real constraint* $c(x_1, \ldots, x_n)$ *iff*

$$\forall I_1, \ldots, I_n \in \mathbb{I} \ : \ \exists a_1 \in I_1, \ldots, \exists a_n \in I_n, (a_1, \ldots, a_n) \in \rho_c \Rightarrow (I_1, \ldots, I_n) \in \rho_C$$

For example, given $c : f(x_1, \ldots, x_n) = g(x_1, \ldots, x_m)$, F (resp. G) the natural interval extension of f (resp. g), its interval extension is $F(y_1, \ldots, y_n) = G(y_1, \ldots, y_m)$ where the equality over intervals is interpreted as $I_1 = I_2 \Leftrightarrow I_1 \cap I_2 \neq \emptyset$.

Let $C(y_1, \ldots, y_n)$ be an interval constraint, k an integer in $\{1, \ldots, n\}$ and I a box. The k-th *projection* of C wrt. I, denoted $C^{(k,I)}$ is the univariate interval constraint $C(I_1, \ldots, I_{k-1}, y_k, I_{k+1}, \ldots, I_n)$.

Definition 3 (Box-consistency) *Let* c *be a real constraint,* C *its interval extension,* k *an integer in* $\{1, \ldots, n\}$*, and* I *a box. The constraint* c *is said* box-consistent wrt. k and I *iff* $I_i = \mathsf{Hull}(I_i \cap \{a_i \in \mathbb{R} \mid \mathsf{Hull}(\{a_i\}) \in \rho_{C^{(k,I)}}\})$.

Informally, box-consistency is enforced over the projections of the interval extensions of the initial real constraints. Given a constraint $c(x_1, \ldots, x_n)$, an integer k and a box I representing the current domains, a CNO (implementing typically a modified version of the Newton method) reduces the domain of x_k by computing the leftmost and rightmost canonical intervals[1] such that $C^{(k,I)}$ holds. For an initial set of constraints, a set of CNO computing box-consistency for each projection of each constraint is generated, which defines a constraint system. The resulting constraint system is handled by the Narrowing Algorithm.

As said in [3], box-consistency is far more accurate than hull-consistency when dealing with complex constraints involving the same variables many times, as it does not decompose them, avoiding thus to lose some useful information. Nevertheless,

[1]that is, intervals such that there does not exist any floating-point number between the bounds.

finding the leftmost and rightmost elements described above costs a lot (the function associated to the constraint must be evaluated many times along with its derivative). Moreover, note that this must be done for every variable appearing in the constraint. It then appears that box-consistency is not the solution of choice when a complex constraint involves many different variables.

3. An overview of DecLIC

`DecLIC` reuses all the Prolog features of `clp(fd)` [8] (interpreter and compiler from Prolog to C *via* the WAM) and has been implemented with compatibility in mind, such that all `clp(fd)` programs may be compiled by `DecLIC` with few or no modification at all.

`DecLIC` is able to solve problems merging boolean, integer and real constraints. Variables appearing in constraints (referred as *c-variables* —purely Prolog variables are called *p-variables*) are divided in two sets: *fd-variables* are constrained to represent only integer values while *i-variables* may take real values. Note that, unlike `clp(BNR)`, the set to which a c-variable belongs is not determined statically by means of a type statement but dynamically by constraining the variable with the `is_integer/1` constraint.

Complex constraints are built from relation symbols in $\{=, <=, >=, <, >, \backslash=\}$ prefixed by a "#" for linear constraints[2], "$" for non-linear constraints to be decomposed into primitives (use of hull-consistency), and "$$" for non-linear constraints to be processed as a whole (use of box-consistency).

3.1. Bracket handling

In order to improve correctness, we have chosen to handle opened intervals as well as closed ones. Initialization is made with functors `o()` —opened— and `c()` —closed. Bracket handling allows to detect inconsistencies which could not be detected otherwise. During computation, the brackets of a variable domain may change from "closed" to "opened". This is clearly the case whenever bounds cannot be represented by floats.

Example:

```
:- X**2$=Y, Y in -3.5..o(0).   % Y=[-3.5..0)
no % A square cannot be strictly negative.
:- X in -10..10, % X=[-10..10]
   X**2$=2.
```

[2]at present, linear constraints are processed the same way as non-linear ones. Nevertheless, we provide a particular form to state them as their decomposition into primitives may easily be optimized.

```
X = (-1.414213562373096 ..  +1.414213562373096)      % hull-consistency narrows down the
                              % input domains such that all solutions are in the domains given as output.
```

3.2. Mixing domains and methods

It is possible to mix in the same `DecLIC` program some constraints involving variables from different domains (integer, boolean or real) as well as constraints over the same domain handled by different methods (hull-consistency *vs.* box-consistency). The domain a variable ranges in is determined explicitly by the user (use of `is_integer/1`) or implicitly, depending on the constraints the variable appears in (e.g. the constraint `and(`X`,`Y`,`Z`)` stating that $X \wedge Y = Z$ constrains its three arguments to be booleans —integer variables ranging in [0..1]).

When dealing with constraints over reals, the choice of the method to be used is dictated by their structure. Nevertheless, this choice is at present up to the user. As a rule of thumb, box-consistency must be preferred when dealing with complex constraints involving few variables, especially when they appear more than once, whereas hull-consistency is for choice for "small" constraints and complex constraints involving many different variables.

4. Implementation

In the following, we will call *D-constraints* the constraints decomposed into primitives, and *N-constraints* those handled as a whole.

4.1. Operational model

Solving systems of constraints in DecLIC is done using (a variation of) the Narrowing Algorithm NAR [3]. Our version of NAR works incrementally: it is called whenever a constraint is added to the *store* (that is, the set of all constraints stated so far). It applies the CNO of the new constraint to the set of variables involved, re-invoking in sequence all constraints whose variable domains have been modified. Our store may be viewed as three boxes containing constraints linked between each other by variable dependencies:

1. a "box" S_1 gathering all `is_integer` constraints;

2. a "box" S_2 gathering constraints handled by box-consistency;

3. a "box" S_3 gathering constraints handled by hull-consistency.

For example, a store for the program P:

```
:- is_integer(X), is_integer(Y), X+Y**2$=Z, Z$<6,
   3*X**3-4*X**2+2*X-7$$=Y.
```

would be the one given in figure 1.

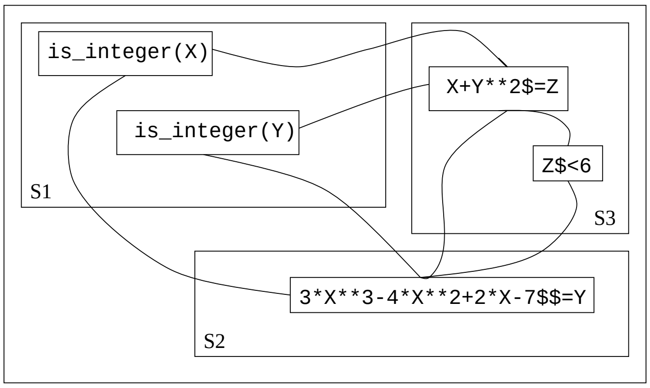

Figure 1: Store for the program P.

The addition of a new constraint c to the store is done as described by the algorithm in table 1. Note that the strategy adopted is to re-invoke `is_integer` constraints as soon as possible, and secondly, to reach the fixed-point over constraints handled by box-consistency before considering constraints handled by hull-consistency. This is a heuristic supported by experimental evidences.

4.2. Data structures

We describe in this section the internal structures used by `DecLIC` to handle constraints. They are mainly the same as in `clp(fd)` [8]: a frame for each variable appearing in a constraint, a frame for each constraint, and a frame collecting all the arguments appearing in constraints for some clause.

4.2.1. Argument frame

An argument frame (see figure 2) is created for each clause where at least one constraint appears in. It contains a pointer to each c-variable and p-variable involved in one of the constraints of the clause. The argument frame corresponds to the environment in which the constraints of the clause are processed (see figure 3 for an example of use).

```
IncNar( in (c, N) % constraint to be added
        inout S1, S2, S3 % the 3 boxes of the store
        inout X = X1 × ··· × Xn ) % domains of all the variables
begin
    foreach i in {1,2,3} do
        CSi := ∅ % set of constraints of Si to be reinvoked
    endfor
    α := typeof(c) % α = 1, 2 or 3 depending on the "type" of c: is_integer(), f$.. or f$$..
    CSα := {(c,N)} % Cα is the set of constraints to be re-invoked in the store α
    Sα := Sα ∪ {(c,N)} % c is added to the store corresponding to its type
    while (∃i ∈ {1,2,3} | CSi ≠ ∅) and (X ≠ ∅) do
    % we loop until there is no longer any constraint to be reconsidered in any of the three stores
        j := min({i ∈ {1,2,3} | CSi ≠ ∅})
        Choose one (cl, Nl) in CSj
            % we choose a constraint to be re-invoked in the set corresponding
            % to the store with lowest index (highest priority)
        X' := Nl(X) we narrow X using the CNO Nl
        if X' ≠ X then
            foreach k in {1,2,3} do
                CSk := CSk ∪ {(cm, Nm) ∈ Sk | ∃vo ∈ var(cm) ∧ X'o ≠ Xo}
                % all the constraints using a variable whose domain
                % has been narrowed down are put in the set of constraints
                % to be re-invoked corresponding to their type.
            endfor
            X := X'
        endif
        CSj := CSj \ {(cl, Nl)}
    endwhile
end.
```

Table 1: Incremental Narrowing Algorithm.

4.2.2. Constraint frames

To each constraint in the program corresponds one constraint frame gathering the informations necessary for its (re-)invocation. `DecLIC` manipulates two types of constraint frames: *D-constraint frames* created for "X in r" constraints, and *N-constraints frames* for complex constraints treated as a whole. The "X in r" constraint is —as in `clp(fd)`— the most basic constraint of `DecLIC`. All primitive constraints are implemented by a conjunction of X in r constraints. For example, we have:

```
'x*y=z'(X,Y,Z):-
     X in z_div_y(Z,Y),
     Y in z_div_x(Z,X),
     Z in x_mul_y(X,Y).
```

where `z_div_y` (resp. `z_div_x` and `x_mul_y`) is a C function computing the interval resulting from the quotient of the domain of `Z` by the domain of `Y` (resp. `Z/X` and `X*Y`). Refer to [6] for an example of algorithm computing the CNO of `x*y=z`.

The main difference between the two constraint frames relies in that each X in r constraint is invoked by a call to a C function associated to it, while constraints handled by box-consistency are all processed by the same C function hard-coded in

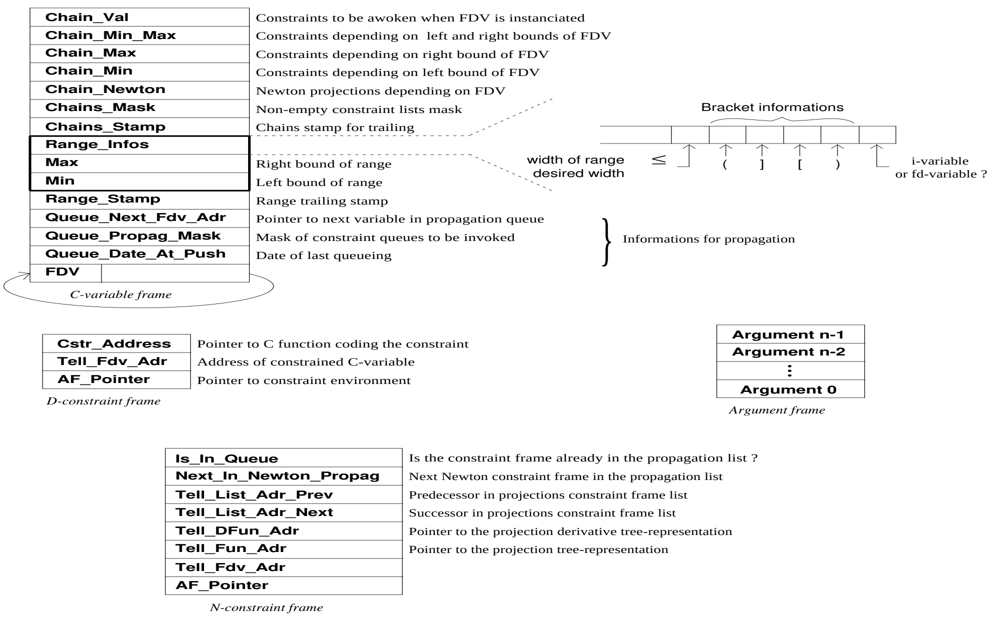

Figure 2: `DecLIC` structures.

`DecLIC`, such that N-constraint frames contain all the arguments for that function. Figure 2 describes in details the two structures.

4.2.3. C-variable frame

A c-variable frame is created at runtime on the heap for every constrained variable (see figure 2 for its structure). As in `clp(fd)`, a variable frame is divided into three parts:

1. a part devoted to propagation (done by queuing range-modified variables);

2. a part containing range informations: the left and right bounds (coded as IEEE [9] doubles) of the variable domain, and an information word coding for the form of the left and right brackets along with the type of the variable (i.e. i-variable or fd-variable). This information is consulted whenever the variable domain is modified: if the variable is a FD one, its domain is inward-rounded on the nearest integer values before propagation occurs;

3. a part containing *dependency chains* where addresses of the constraint frames of the constraints to be reinvoked each time the variable domain is modified are chained.

4.3. X in r compilation

As said above, every primitive constraint supported by `DecLIC` is defined in terms of the constraint X in r. Unlike `clp(fd)`, the use of *indexicals*[3] [8] is mainly restricted to `dom()` —which relates to compact domains only. This is due to the fact that, when dealing with floats, it is not safe to compute domains by reasoning over bounds; the only safe way is to compute domains with intervals. The "r" part is compiled into a function which is reinvoked whenever one of the variable domains used is modified.
Example:

```
X in dom(Y)+(o(3.4)..7*3).
```

is compiled into

```
Begin_Fd_Constraint(1)
      fd_ind_dom(0,1)   % R(0) <- dom(1st variable in argument frame) (Y)
      fd_rounding_down  % rounds toward -infinity
      fd_float(2,3.4)   % T(2) <- 3.4
      fd_rounding_up    % rounds toward +infinity
      fd_integer(1,7)   % T(1) <- 7
      fd_integer(3,3)   % T(3) <- 3
      fd_term_mul_term(1,3) % T(1) <- T(1)*T(3)
      fd_interval_range(1,2,1,16,8) % R(1) <- o(T2)..T(1)
      fd_range_add_range(0,1) %^-^-- 16=left opened, 8=right closed
      fd_tell_range(0) % X <- dom(X) intersected with R(0)
      fd_proceed
End_Fd_Constraint
```

4.4. Constraint handling at compile-time

4.4.1. D-constraint processing.

At compile-time, each D-constraint encountered is decomposed into calls to primitive constraints available in the `DecLIC` library:
Example:

```
:- X**2 + Y**2 $= 1,
   X**2 - Y    $= 0.
```

is compiled (decomposed) into the WAM instructions sequence given in table 2. Note that common sub-expressions (here `X**2`) are not shared.

4.4.2. N-constraint processing.

At compile-time, `DecLIC` translates every constraint of the form `f$$··0` into a character string representing `f` in a coded prefixed form. For example, "`:- X**2+Y**2$$=1.`" is translated into `"-(+(^(#(0),i(2)),^(#(1),i(2))),i(1))"` where `^` stands for exponentiation operator, `i(x)` refers to the *integer* `x` and `#(y)` represents the `y`$^{\text{th}}$

[3] That is, functors allowing to select some parts of a domain: `dom()` (the whole domain), `min()` (the left bound), `max()` (the right bound).

```
put_y_variable(4,0) % A(0) <- X
put_y_variable(8,1)
call("xx=y",2)      % Y(8) $= X**2
put_y_variable(3,0) % A(1) <- Y
put_y_variable(7,1)
call("xx=y",2)      % Y(7) $= Y**2
put_y_value(8,0)    % A(0) <- X**2
put_y_value(7,1)    % A(1) <- Y**2
put_integer(1,2)    % A(2) <- 1
```

```
call("x+y=z",3)    % X**2 + Y**2 = 1
put_y_value(4,0)   % A(0) <- X
put_y_variable(2,1)
call("xx=y",2)     % Y(2) $= X**2
put_y_value(2,0)   % A(0) <- X**2
put_integer(0,1)   % A(1) <- 0
put_y_value(3,2)   % A(2) <- Y
call("x+y=z",3)    % X**2 + 0 $= Y
```

Table 2: Decomposition of constraints.

variable in the argument frame of the clause. An installation function is generated for every N-constraint encountered, except when several N-constraints are linked into a system with the `and` operator, in which case only one installation function is created for all linked N-constraints. A consequence of such a sharing is that all projections of all constraints in the system are known simultaneously when applying relaxation, which allows speeding-up the narrowing process (experimental evidences have shown a factor of 10 speed-up when solving systems as a whole *vs.* solving each constraint of the systems in turn).

Example:

```
:- X**2 + Y**2 $$= 1 and
   X**2 - Y    $$= 0.
```

is compiled into

```
Begin_Private_Pred
 fd_set_x_AF(2,0)     % Creation of an argument frame for X and Y
 init_newton_system   % Newton propag list <- empty
 fd_install_newton_constraint_with_x_AF(1,0) % calls Fd_install_Newton(1)
 fd_call_newton_constraints % Newton relaxation + general relaxation
 ...
End_Pred

Begin_Fd_Install_Newton(1)
 fd_install_newton("-(+(^(#(0),i(2)),^(#(1),i(2))),i(1))")
 fd_install_newton("-(^(#(0),i(2)),#(1))")
 fd_proceed
End_Fd_Install_Newton
```

Figure 3 gives an example of the constraint network created at runtime to solve the system of the previous example.

4.5. Constraint handling at runtime

At runtime, the installation of the N-constraints by `fd_install_newton` is done as follows: Let $c_1 : f_1(x_{1_1}, \ldots, x_{m_1}) = 0$, $\ldots$, $c_n : f_n(x_{1_n}, \ldots, x_{m_n}) = 0$, n N-constraints occurring in an installation function. Given the N-constraint $c_i : f_i(x_{1_i}, \ldots, x_{m_i}) = 0$, for each projection of c_i wrt. the variable x_{j_i},

- a N-constraint frame $NF_{i,j}$ is created (it will be used to tighten the domain of x_{j_i} with the constraint c_i),

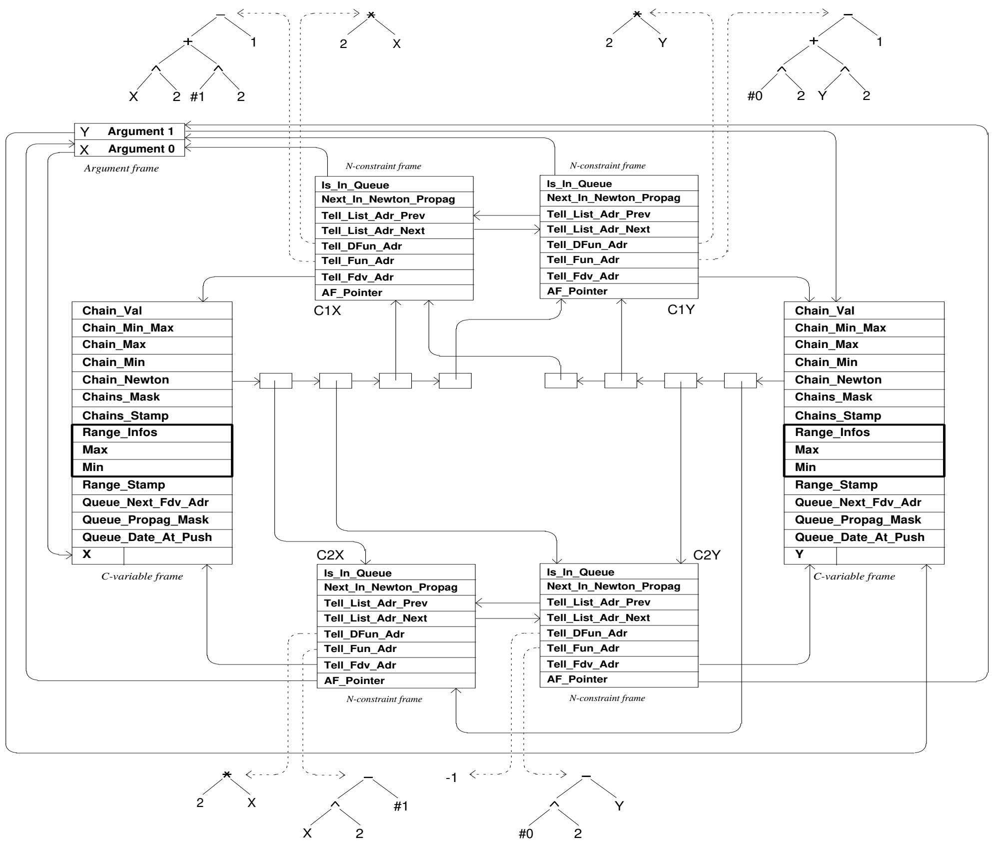

Figure 3: A constraint network.

- the partial derivative $\frac{\partial f_i}{\partial x_j}$ is computed symbolically and inserted in $NF_{i,j}$,
- as the computation of the projection associated to $NF_{i,j}$ depends on the domains of all the variables $x_{1_i}, \ldots, x_{m_i}$, this N-constraint frame is added in the `Chain_Newton` list of the variable frame of all those variables,
- $NF_{i,j}$ is added to a *Newton Propagation List* (NPL) whose head and tail addresses are kept respectively into the `BNP` and `ENP` registers.

  After installation, the NPL is relaxed as described in table 3.

4.6. Cooperation of the solvers

When the domain of a variable x is modified due to invocation of a constraint c,

Input: NPL, a linked chain of N-constraint frames to be reinvoked.
Output: failure or success.

```
BEGIN
    cstr ← first frame of NPL
    REPEAT
        Apply Newton narrowing on Tell_Fdv_Adr of cstr
        IF failure THEN
            Delete NPL
            exit returning failure
        ELSE
            IF Range of Tell_Fdv_Adr modified THEN
                Push Tell_Fdv_Adr frame in GPL
                Push all N-constraint frames depending on Tell_Fdv_Adr in NPL
            ENDIF
        ENDIF
        cstr ← next frame of NPL
    UNTIL fixed-point is reached % i.e. NPL is empty
    return success
END
```

Table 3: Newton relaxation.

- its frame is added at the end of a *general propagation list* (GPL) linking all variables whose dependency chains are to be reconsidered,
- The `Queue_Propag_Mask` of x is set according to which part of its domain has changed. It is worth noticing that if c is a N-constraint, `Chain_Newton` is *not* marked to be reconsidered, as the relaxation over all N-constraints will be done immediatly; moreover, if x was already in the GPL, re-invocation of constraints in `Chain_Newton` is forbidden by modifying its `Queue_Propag_Mask`,
- the GPL is relaxed in the following way: variable-frame *vf* at the head of GPL is popped, then

    1. all constraint frames of its `Chain_Newton` list are linked in the NPL and Newton relaxation is invoked (see table 3),
    2. all constraint frames of the other dependency chains of *vf* are reconsidered by reinvoking the constraint they represent (this may add new variables to the GPL),
    3. the next variable in GPL is popped and all the process repeats from step 1, until fixed-point (i.e. emptyness of GPL) is reached.

4.7. WAM extension

We will only consider here new instructions added to the WAM instructions set, leaving apart all modifications of already existing instructions.

Benchmarks	*Var*	*Sol*	$\texttt{DecLIC}_1$ (s.)	$\texttt{DecLIC}_2$ (s.)
Yoshigahara	9	6	8.05	7.56
Broyden320	320	1	166.15	46.51
Cosnard40	40	1	317.50	102.03
Combustion	10	2	119.23	89.77
Pentagon	5	1	1.10	0.93
Bifurcation	3	12	12.82	2.46
Kinematics	8	16	5.50	2.18

Table 4: Experimental results.

`init_newton_system`. Sets `ENP` to point to the beginning of the Newton propagation list materialized by a dummy N-constraint frame.

`fd_install_newton_constraint(nb,X)`. Initialises the Argument Frame Pointer to X, then invokes `nb`$^{\text{th}}$ constraint installation function.

`fd_rounding_`$\left\{\begin{array}{c}\texttt{up}\\ \texttt{down}\end{array}\right\}$. Used in `X in r` constraints to achieve outward rounding of `r`. Note that outward rounding cannot be guaranteed if more than one primitive operation $(+, -, \ldots)$ is used to compute a bound.

`fd_float(T,n)`. Executes `T` $\leftarrow$ `(double)(n)` (used when computing `r` part of `X in r` constraints).

`fd_install_newton(str)`. Installs N-constraint represented by string `str`.

5. Experimental Results

We tested DecLIC on various benchmarks from the interval community. Those chosen here are such that most of the features of DecLIC are used, including the cooperation of solvers. Moreover, we worked with two versions: one with a solver enforcing "perfect" box-consistency and an other one enforcing "box(w)-consistency", that is box-consistency with a tolerance on the bounds of width w. This weakening of the consistency is shown to speed-up drastically the computation in some cases.

Table 4 presents the computational results of `DecLIC`. They were computed on a Sun Sparc 10 with precision of 10^{-8} (width of the computed domains). `DecLIC` uses for the most part an *improvement factor* of 10%[4] and the w parameter of box(w)-consistency is set to 0.1. The labels of columns mean: *Var* for the number of variables, *Sol* for the number of real solutions, $\texttt{DecLIC}_1$ for the computation time of `DecLIC` using box-consistency, $\texttt{DecLIC}_2$ for the computation time of `DecLIC` using

[4]when propagating, a domain is said modified by the application of a CNO if its size decreases by more than 10%.

box(w)-consistency. Yoshigahara is a puzzle extracted from [13] with integer as well as real solutions, where only integer ones are desired. Pentagon (coordinates of a regular pentagon), extracted from [10], involves non-linear constraints approximated by box(w)-consistency and inequality constraints approximated by hull-consistency, and points out the efficiency of both methods in this case. The other problems are handled by interval Newton methods, enforcing box(w)-consistency on each constraint. The results show that box(w)-consistency (column $\mathtt{DecLIC}_2$) significantly improves box-consistency (column $\mathtt{DecLIC}_1$), which explains the speed-up on Cosnard40 (Moré-Cosnard function with 40 variables) and Broyden320 (Broyden-banded function with 320 variables). However `DecLIC` does not implement any conditioning operator — which is pointed out as a possible improvement in the conclusion — then it does not solve very efficiently a problem like Combustion (combustion problem for a temperature of 3000 degrees) [14]. The last problems are Bifurcation (test from numerical bifurcation) and Kinematics (robot kinematics problem), extracted from [5].

6. Conclusion

In this paper, we have shown that it was possible to implement an efficient solver over several domains by extending the WAM used in clp(fd) —a solver dealing only with integers— with few new instructions, and by modifying some of the internal structures. The speed-up obtained with box(w)-consistency demonstrates that it is too demanding for many problems to enforce perfect box-consistency and that hull-consistency is for them a better choice. Actually, a precise theoretical study and comparison of the two consistencies is still to be done. In particular, it would be interesting to find a partition in two classes of all the constraint systems, allowing to decide which consistency is the best to apply for a particular problem. This will be our main direction for research in the near future.

We plan also to add constraint dynamic generation in order to be able to support conditioning and computer algebra methods such as Gröbner bases.

Acknowledgement

Discussions with Pr. Frédéric Benhamou on preliminary versions of this paper are gracefully acknowledged. We thank also the reviewers for their helpful comments.

References

[1] F. Benhamou. Interval Constraint Logic Programming. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 10, pages 1–21. Springer Verlag, 1995.

[2] F. Benhamou. Heterogeneous Constraint Solving. In *Proceedings of the fifth International Conference on Algebraic and Logic Programming (ALP'96), LNCS 1139*, pages 62–76, Aachen, Germany, 1996. Springer-Verlag.

[3] F. Benhamou, D. McAllester, and P. van Hentenryck. CLP(Intervals) Revisited. In *Proceedings of International Logic Programming Symposium (ILPS'94)*, Ithaca, NY, USA, 1994.

[4] Frédéric Benhamou and Touraïvane. Prolog IV: langage et algorithmes. In *JFPL'95: IVèmes Journées Francophones de Programmation en Logique*, pages 51–65, Dijon, France, 1995. Teknea.

[5] D. Bini and B. Mourrain. Handbook of Polynomial Systems. February 1997.

[6] J. G. Cleary. Logical Arithmetic. *Future Generation Computing Systems*, 2(2):125–149, 1987.

[7] Ph. Codognet and D. Diaz. `wamcc`: Compiling Prolog to C. In *International Conference on Logic Programming (ICLP'95)*, pages 317–331, 1995.

[8] Ph. Codognet and D. Diaz. Compiling Constraints in `clp(fd)`. *Journal of Logic Programming*, 27(3):185–226, 1996.

[9] IEEE. IEEE Standard for Binary Floating-Point Arithmetic. Technical Report IEEE Std 754-1985, 1985. Reaffirmed 1990.

[10] O. Lhomme. *Contribution à la résolution de contraintes sur les réels par propagation d'intervalles*. PhD thesis, University of Nice-Sophia Antipolis, july 1994.

[11] W. Older and A. Vellino. Constraint Arithmetic on Real Intervals. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Papers*. The MIT Press, 1993.

[12] A. Semenov and A. Leshchenko. Interval and Symbolic Computations in the Unicalc Solver. In *Proceedings of INTERVAL'94*, pages 206–208, St-Petersburg, Russia, 1994.

[13] D. Singmaster. Quelques divertissements numériques. *La Recherche*, (278):822–823, Juillet/Août 1995.

[14] Pascal Van Hentenryck, Laurent Michel, and Frédéric Benhamou. `Newton` - Constraint Programming over non-linear Constraints. *Science of Programming*, 1997. Forthcoming.

[15] D. H. D. Warren. An abstract Prolog Instruction Set. Technical Report 309, SRI International, october 1983.