An Extension of the WAM for Hybrid Interval Solvers

Frédéric Goualard and Frédéric Benhamou
IRIN, Université de Nantes
B.P. 92208, 44322 Nantes Cedex 3 — France
Freric.[Goualard,Benhamou]@irin.univ-nantes.fr

Laurent Granvilliers
LIFO, Université d’Orléans
B.P. 6759, 45067 Orléans Cedex 2 — France
Laurent.Granvilliers@lifo.univ-orleans.fr

Abstract

This paper discusses the implementation of DecLIC, an extension of the clp(fd) language to handle interval constraints. The main solving capabilities of DecLIC are based on the use of two different levels of local consistency, namely hull-consistency and box-ϕ-consistency, a slight yet computationally relevant generalization of box-consistency. Experimental evidence of the qualities and drawbacks of constraint solving based on these two notions is provided and benchmark comparisons, carried out with a prototype of DecLIC, show that combination permits, on certain cases, relevant speed-ups in the resolution process. The compiler of clp(fd) translates CLP programs to C code via the Warren Abstract Machine (WAM). The extension of the WAM instruction set necessary to support the new features of DecLIC is described, along with the new internal structures.
1 Introduction

Constraint Programming is now commonly used to solve a wide range of industrial combinatorial problems (finite domains constraints), and also emerges as a promising alternative to classical numerical methods to tackle continuous problems (interval constraints). Due to a number of correctness issues in numerical computations [15] (round-off errors, cancellations), constraints involving real-valued variables are particularly difficult to reliably solve. Originating from the pioneering work of Cleary [12], many constraint programming languages such as BNR-Prolog [27], clp(BNR) [9], Prolog IV [10], ILOG Solver [28], and Numerica [33], address this problem by replacing floating-point arithmetic with interval arithmetic [3, 24], which guarantees the correctness of computations.

The above mentioned solvers handle constraint systems by combining local consistency techniques and filtering [35, 23, 21]: a domain of possible values (interval) is associated to each variable; a constraint states a relation between some variables; solving a particular constraint then relies in discarding from some domains certain values for which the relation does not hold (inconsistency). This is done by applying contracting operators called Constraint Narrowing Operators [4].

When dealing with real-valued variables, it is not always possible to exactly narrow down the domains since—for example— some solutions may be not representable with floating-point numbers. In practice, the enforced consistencies, among which one may cite hull-consistency [4] and box-consistency [8], are approximations of the true consistency notion. The constraints of a system form the nodes of a network (called a constraint store) where two constraints are linked whenever they share at least one variable. The constraint system is then solved by propagating the domain modifications induced by the applications of constraint narrowing operators to all the constraints involving the modified variables through the links of the constraint network until reaching a stable state.

Several CLP languages such as Prolog IV, clp(BNR), and Unicalc [29], are based on hull-consistency: they handle constraints by decomposing them into conjunctions of so-called primitive constraints (e.g. ‘\(x + y = z\)’, ‘\(x \times y = z\)’, ... ) as proposed in [12], then enforcing hull-consistency over all the resulting constraints through domain propagation. The drawbacks of such a method are by now well known: introduction of new variables induced by the decomposition leads to unnecessary approximation. Moreover, the constraint
network thus obtained is far more complicated than the one which would have been obtained by globally considering complex constraints. In addition to efficiency issues, this last point is a major problem when one wants to inspect the constraint store for debugging or profiling purposes [6].

On the other hand, other constraint languages, such as Helios or Numerica [33] implement box-consistency, a notion introduced by Benhamou et al. in [8]. Box-consistency permits to efficiently process complex constraints without decomposition. It was primarily implemented in a CLP language called Newton [32] which demonstrated that box-consistency often outperforms 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; there exist problems where it is less effective than hull-consistency. We have then decided to design a CLP language, DecLIC (Declarative Language with Interval Constraints), where several solvers interact. In particular, selected constraints are handled globally by box-consistency, whereas others are decomposed into primitive constraints. Hull-consistency is enforced on this last constraint type. The main objective is to benefit from the effectiveness of both methods. DecLIC has been implemented by extending clp(fd) [14], a well known CLP language over finite domains developed by Codognet and Diaz whose source code is widely available\(^1\). clp(fd) extends the Warren Abstract Machine (WAM) [36] to compile into C code Prolog-like programs describing problems over finite domains [13, 14]. In DecLIC, the WAM instruction set was modified and extended to take into account the processing of real constraints with hybrid solvers. Interval constraints providing a generic framework, the resulting system is a compiler/interpreter with constraint solving capabilities over integers, booleans and reals.

The extension of the WAM to handle constraints over the reals has been previously investigated by Lee and Lee in [19]. Our work differs in at least two points from the one described in the above mentioned paper: first, we have kept the so-called RISC approach [34, 14] (internally, all constraints are expressed in terms of one primitive constraint of the form $X \in r$) and second, our extension has been designed with interaction of solvers in mind, allowing us to introduce different consistencies in the same constraint programming system.

The main contributions of this paper are: the description of a minimal

\(^1\)ftp://ftp.inria.fr/INRIA/Projects/loco/clp_fd/
extension of the WAM allowing the interaction of several solvers; the introduction of a slight yet computationally relevant generalization of box-consistency (called box ϕ -consistency) which is shown to outperform it on all the constraint systems we have tested; the demonstration that none of the two consistencies commonly used in CLP solvers is the best for all the constraint systems; and the experimental evidence that combination of several consistencies, as is done in our system, permits to drastically speed-up the solving of some problems.

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

2 Preliminary Notions

This section presents notions used in the rest of the paper. The reader is referred to [3, 24] for a comprehensive presentation of interval analysis and to [4, 5] for a detailed description of interval constraints.

2.1 Basics

Let I denote the set of (closed/opened) floating-point intervals (simply referred in the following as intervals), i.e. all the compact subsets of R whose glb. and lub. are floating-point numbers. Let Σ₁ = ( R, F₁, R₁ ) be a structure where R is the set of the reals, F₁ a set of function symbols and R₁ a set of relation symbols. Let V₁ = { x₁, x₂, ... } be a countable set of variables taking their value over R. We then define a real constraint as being an atomic formula built from Σ₁ and V₁. In the same way, we define interval constraints over the structure Σ₂ = ( I, F₂, R₂ ) and the countable set of variables V₂ = { X₁, X₂, ... }. Given a real constraint c(x₁, ..., xₙ) (resp. an interval constraint C(X₁, ..., Xₙ)), ρ₁ (resp. ρ₂) denotes its associated relation. A Cartesian product of n intervals I₁ × ··· × Iₙ will be called a box. Let Hull(ρ) be the smallest box containing the relation ρ, and var(c) the set of variables occurring in c. In the following, we use the notation [a, b] as
a shorthand for the set: \( \{ x \in \mathbb{R} : a \leq x \leq b \} \). In the rest of this section, the arity of a function or a relation is considered to be \( n \).

Given a function \( f \) defined over the reals, an interval extension of \( f \) is a function \( F \) defined over intervals as follows:

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

For example, one may define the interval extensions \( \oplus, \otimes \) of the basic operations \(+, \times\) as described below [24]:

\[
\begin{align*}
[a, b] \oplus [c, d] &= [\downarrow (a + c), \uparrow (b + d)] \\
[a, b] \otimes [c, d] &= [\min(\downarrow (ac), \downarrow (bc), \downarrow (bd)), \\
&\quad \max(\uparrow (ac), \uparrow (ad), \uparrow (bd))]
\end{align*}
\]

where \( \downarrow (a) \) is the greatest float smaller or equal to \( a \), and \( \uparrow (a) \) is the smallest float greater or equal to \( a \).

In this paper, we use a particular form of interval extension, called natural interval extension in [25], defined as follows: given \( f : \mathbb{R}^n \rightarrow \mathbb{R} \), its natural interval extension \( F : \mathbb{I}^n \rightarrow \mathbb{I} \) is obtained from \( f \) by replacing in the expression of \( f \) each constant \( a \) by \( \text{Hull}(\{a\}) \), each basic operation by its corresponding interval operation and each variable by an interval variable.

An interval constraint \( C(X_1, \ldots, X_n) \) is defined as an interval extension of a real constraint \( c(x_1, \ldots, x_n) \) in a very similar way.

The projection of an interval constraint \( C(X_1, \ldots, X_n) \) wrt. \( k \in \{1, \ldots, n\} \) and a box \( I \), denoted \( C^{(k, I)} \), is defined as the univariate interval constraint obtained by replacing all the \( X_i \)'s but \( X_k \) by the intervals \( I_i \).

### 2.2 Constraint Narrowing operators

Operationally, given a real constraint \( c(x_1, \ldots, x_n) \), a constraint solver aims at reducing the domains associated to \( x_1, \ldots, x_n \). This reduction process for \( c \) is abstracted by the key notion of Constraint Narrowing Operators [5], which are contracting monotone functions taking as input a box and returning a box from which have been discarded some of the elements which do not belong to \( p_c \). These operators, thereafter abbreviated CNO, are formally defined as follows:

**Definition 1 (Constraint Narrowing Operator over \( \mathbb{I} \) [5]).** Given \( \rho \) a \( n \)-ary relation over \( \mathbb{R} \), the function \( N : \mathbb{I}^n \rightarrow \mathbb{I}^n \) is a constraint narrowing
operator for $\rho$ iff for every $u, v \in \mathbb{I}^n$, the three following properties hold:

- $N(u) \subseteq u$ (contractance)
- $u \cap \rho \subseteq N(u)$ (correctness)
- $u \subseteq v \Rightarrow N(u) \subseteq N(v)$ (monotonicity)

In the following, a constraint system is defined as a pair $(S, I)$ 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 $\rho_{c_i}$, and $I$ 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 CNOs of the system included in $I$ by using the Narrowing Algorithm [5] described by Algorithm 1. The key instruction is the application of the CNO $N_i$ on $I$, which enforces some consistency for the associated constraint $c_i$. As said previously, the two consistencies used in $\text{DecLIC}$ are hull-consistency and box-consistency, whose definitions are given below.

**Algorithm 1: The Narrowing algorithm.**

2.3 Local consistency notions

In this section, we provide the definitions for the two main consistency notions used in $\text{DecLIC}$ as well as some comments on their advantages and drawbacks.
Definition 2 (Hull-consistency). The real constraint \( c(x_1, \ldots, x_n) \) is said hull-consistent wrt. a box \( I \) iff \( \forall k \in \{1, \ldots, n\} : I_k = \text{Hull}(I_k \cap \{ a_k \in \mathbb{R} \mid \forall j \in \{1, \ldots, n\} \setminus \{k\} : \exists a_j \in I_j \text{ s.t. } (a_1, \ldots, a_n) \in \rho_c \}) \)

Intuitively, a constraint \( c \) is hull-consistent wrt. a box \( I \) if, for every strict sub-box \( I' \) of \( I \), there exists an element of \( \rho_c \) which is in \( I \) and not in \( I' \).

Hull-consistency is enforced over constraints by decomposition into conjunctions of primitive constraints (usually \( x + y = z, x \ast y = z, \exp(x) = y, \sin(x) = y \), and so on). The CNOs for those primitives are implemented using Relational Interval Arithmetic [12] (see an example below); 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 supported by the constraint programming system. Therefore, CNOs based on hull-consistency are particularly efficient when constraints to be processed are “quasi-primitives”. The drawbacks are that the introduction of new variables due to the decomposition process drastically reduces the capability to tighten the domains of the variables the user is interested in. As pointed out in [8], this is particularly true when the same variables appear more than once in the constraints since each occurrence of a variable \( v \) is considered as a new variable \( v' \) with the same domain as \( v \).

Example 1 (A CNO for \( “x + y = z” \)). Enforcing hull-consistency for the constraint \( x + y = z \) and the domains \( I_x, I_y, \) and \( I_z \), is done by decomposing it into three statements:

\[
\begin{align*}
p_1 & : I_x \leftarrow I_x \cap (I_z \ominus I_y) \\
p_2 & : I_y \leftarrow I_y \cap (I_z \ominus I_x) \\
p_3 & : I_z \leftarrow I_z \cap (I_x \oplus I_y)
\end{align*}
\]

Consistency is obtained by the computation of each \( p_i \).

More generally, enforcing hull-consistency over a constraint \( c \) in terms of the variables \( \{x_1, \ldots, x_k\} \) is done by expressing each variable in terms of the others (this cannot be done in general with complex constraints involving the same variable more than once, which explains the decomposition into primitives).

Definition 3 (Box-consistency). Let \( c \) be a real constraint, \( C \) an interval extension of \( c \), \( k \) an integer in \( \{1, \ldots, n\} \), and \( I \) a box. The constraint \( c \) is said box-consistent wrt. \( k, C \) and \( I \) iff \( I_k = \text{Hull}(I_k \cap \{ a_k \in \mathbb{R} \mid \text{Hull}(\{a_k\}) \in \rho_{C^{(k,n)}} \}) \).
Intuitively, a constraint $c$ is box-consistent wrt. an integer $k$, an interval extension $C$ and a box $I$ if for every strict sub-box $I'_k$ of $I_k$, there exists an element of $\rho_{C(k,I)}$ which is in $I_k$ and not in $I'_k$.

Box-consistency is enforced over a constraint $c(x_1, \ldots, x_n)$ as follows: $n$ CNOs $N_1, \ldots, N_n$ (implementing typically an interval Newton method [31]) are associated to the $n$ univariate interval constraints $C^{(1,I)}, \ldots, C^{(n,I)}$. Each $N_k$ reduces the domain of $x_k$ by computing the leftmost and rightmost canonical intervals\(^2\) such that $C^{(k,I)}$ holds (leftmost and rightmost quasi-zeros). All the $C^{(k,I)}$ along with their associated CNOs and an initial box $I$ form a constraint system which is handled in the same fashion as in the hull-consistency case.

Figure 1 shows the intervals computed for box-consistency and hull-consistency in the univariate case. Note that the interval obtained here by enforcing the box-consistency is wider than the one obtained with hull-consistency since the function $f$ possesses one quasi-zero which cannot be distinguished from a true zero when using the interval extension of $f$.

As said in [8], box-consistency is far more effective than hull-consistency when dealing with complex constraints involving the same variables many times, since the global processing of these constraints avoids losing some useful information. Nevertheless, finding the leftmost and rightmost quasi-zeros is computationally expensive (the interval Newton method requires

\[^2\text{That is, intervals such that there does not exist any floating-point number between the bounds.} \]
many function evaluations). Moreover, this operation must be done for every variable involved in the constraint. Therefore, box-consistency is generally not the solution of choice when a complex constraint involves many different variables.

Among the new notions introduced in DecLIC is the introduction of box$_\varphi$-consistency, a weak instance of box-consistency where the searching for leftmost and rightmost quasi-zeros stops when the computed intervals enclosing them have width less than $\varphi$ (vs. canonical intervals). This weakening is shown in Section 5 to drastically speed-up the computation of solutions.

### 2.4 Box$_\varphi$-consistency

In this section, we show that operators based on box$_\varphi$-consistency are Constraint Narrowing Operators despite the introduction of parameter $\varphi$.

**Definition 4 (Box$_\varphi$-consistency).** Let $c$ be a $n$-ary constraint, $C$ an interval extension of $c$, $k$ an integer in $\{1, \ldots, n\}$, $I = I_1 \times \cdots \times I_n$ a box, and $\varphi$ a positive floating-point number. The constraint $c$ is said box$_\varphi$-consistent wrt. $k$, $C$ and $I$ iff:

$$I_k = \text{Hull}(I_k \cap \{a_k \in \mathbb{R} \mid C(I_1, \ldots, I_{k-1}, \text{Hull}(\{a_k - \varphi, a_k + \varphi\}, I_{k+1}, \ldots, I_n))\})$$

**Definition 5 (Box$_\varphi$-consistency operator).** Given a $n$-ary constraint $c$, a positive floating-point number $\varphi$, and a box $I$, let $N^c_\varphi : \mathbb{I}^n \rightarrow \mathbb{I}^n$ be an operator defined as follows: $N^c_\varphi(I) = I'$ where $I'$ is the largest box included in $I$ s.t.:

$$\forall k \in \{1, \ldots, n\} : c \text{ is box}_\varphi\text{-consistent wrt. } k, C \text{ and } I'$$

**Proposition 1.** Given a $n$-ary constraint $c$ and a positive floating-point number $\varphi$, the box$_\varphi$-consistency operator $N^c_\varphi$ is a constraint narrowing operator for $\rho_c$.

**Proof.** We have to prove the three following properties:

$$\forall I, J \in \mathbb{I}^n :$$

$$\begin{align*}
N^c_\varphi(I) & \subseteq I \quad (1) \\
I \cap \rho_c & \subseteq N^c_\varphi(I) \quad (2) \\
I & \subseteq J \Rightarrow N^c_\varphi(I) \subseteq N^c_\varphi(J) \quad (3)
\end{align*}$$
First note that the interval extensions of the relations associated to the constraints we support are monotone interval extensions \([8]\). In the following, let \(I' = N_\varphi^c(I)\) and \(J' = N_\varphi^c(J)\). Let \(S^k_I\) be the set defined by:

\[
S^k_I = \{ a_k \in \mathbb{R} \mid C(I_1, \ldots, I_{k-1}, \text{Hull}(\{a_k - \varphi, a_k + \varphi\}), I_{k+1}, \ldots, I_n) \}
\]

We recall the fundamental theorem of interval arithmetic by Moore \([24]\):

**Theorem 1 (Fundamental theorem of interval arithmetic).** Let \(C\) be a monotone interval extension of a real constraint \(c\). The following property does hold:

\[
\forall a_1 \in I_1, \ldots, a_n \in I_n : c(a_1, \ldots, a_n) \implies C(I_1, \ldots, I_n)
\]

**Contractance** (1). By Definition 5.

**Correctness** (2). The proof is done by contradiction. Let \(b\) be an element of \(I \cap \rho_c\), and \(b \not\in I'\). Consider the box \(J \subseteq I\) defined by

\[
J = \text{Hull}(I' \cup \{b\})
\]

By monotonicity of \(C\) and Moore’s theorem, we know that

\[
C(J_1, \ldots, J_{k-1}, \text{Hull}(\{b_k - \varphi, b_k + \varphi\}), J_{k+1}, \ldots, J_n)
\]

does hold, and that \(S^k_{I'} \subseteq S^k_J\). As a consequence,

\[
\forall k \in \{1, \ldots, n\} : J_k = \text{Hull}(J_k \cap \{a_k \in \mathbb{R} \mid C(J_1, \ldots, J_{k-1}, \\
\text{Hull}(\{a_k - \varphi, a_k + \varphi\}), J_{k+1}, \ldots, J_n) \})
\]

(otherwise, there would exist a box smaller than \(J\) containing all the elements of \(I'\) and \(\{b\}\), which contradicts the fact that \(J\) is the smallest box with that property (Equation 4)).

Hence, for all \(k \in \{1, \ldots, n\}\), \(c\) is box_\(\varphi\)-consistent wrt. \(k\), \(C\) and \(J\). Since \(J \supseteq I'\), this contradicts the fact that \(I'\) is the largest box included in \(I\) for which \(c\) is box_\(\varphi\)-consistent, which ends the demonstration.

**Monotonicity** (3). Let \(I' = N_\varphi^c(I)\) and \(J' = N_\varphi^c(J)\). By contractance of \(N_\varphi^c\), we have \(I' \subseteq J\). Proving \(I' \subseteq J'\) follows from the fact that \(J' \subset I'\) would lead to \(J'\) not being the largest box included in \(J\) for which \(c\) is box_\(\varphi\)-consistent.

\[\blacksquare\]
A brief overview of DecLIC

The motivations which led us to the design of DecLIC were threefold: to provide an academic platform for further developments of CLP(Intervals) languages and solving techniques over the reals; to supply the necessary language facilities and compiler architecture for the introduction of several constraint solvers; and to experiment with the implementation idea provided in clp(fd) and due to Pascal Van Hentenryck et al. [34] of reducing the primitive set of constraints to a unique constraint: the $X$ in $r$ constraint — where $X$ is a variable and $r$ a domain — which states that $X$ must belong to the domain $r$.

DecLIC reuses all the Prolog features and an important part of the code of clp(fd) [14] (interpreter and compiler from Prolog to C via the WAM [36]) and has been implemented with compatibility in mind, such that all clp(fd) programs can be compiled by DecLIC with few or no modification at all. Nevertheless, DecLIC is at present less efficient than clp(fd) on problems involving only finite domain variables since all domains are represented by floating-point intervals in DecLIC, whereas clp(fd) represents compact as well as sparse domains with integers. A future improvement pointed out in the conclusion is to reuse the work done on the interaction of solvers to add a specialized solver over integer-valued variables into DecLIC, which would use the techniques implemented in clp(fd).

In DecLIC, one can 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 into two sets: $fd$-variables are constrained to represent only integer values while $i$-variables take real values. Note that, unlike clp(BNR), the set to which a $c$-variable belongs is not statically determined by means of a type statement but dynamically by constraining the variable with the is_integer/1 constraint.

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 of choice to solve “small” constraints and complex constraints involving many different variables. However, the choice of the consistency to apply is at present left to the user.

As far as semantics is concerned, DecLIC is basically a language instantiating the general framework of CLP($\mathcal{X}$) languages [18] over reals. The main solvers are incomplete and the semantics of the approximations of the solution space performed by these solvers is precisely described in [5]. Its
syntax is the same as Prolog with some new relation symbols used to state constraints, namely: $\Diamond$ for non-linear constraints to be decomposed into primitives (use of hull-consistency) and $$\Diamond$$ for non-linear constraints to be processed globally (use of box-consistency), where $\Diamond \in \{=, \leq, \leq, \geq, <, >\}$.

In order to improve correctness, we have chosen to handle open intervals as well as closed ones, as is done in Prolog IV. Bracket handling allows to detect inconsistencies which could not be detected otherwise.

3.1 A constraint system handled by hull-consistency

We first provide an example of a program using hull-consistency to compute the coordinates of a regular pentagon [20], knowing that:

- all vertices are at the same distance from the center,
- the distance between two vertices is the same for all of them.

We fix one vertex and the order between the vertices, such that the pentagon be a convex figure (not a star).

The DecLIC code for the regular pentagon is:

```prolog
pentagon(p(X1,Y1),p(X2,Y2),p(X3,Y3),p(X4,Y4),p(X5,Y5),D):-
  norm(p(X1,Y1),1.0), norm(p(X2,Y2),1.0),
  norm(p(X3,Y3),1.0), norm(p(X4,Y4),1.0),
  norm(p(X5,Y5),1.0), distance(p(X1,Y1),p(X2,Y2),D),
  distance(p(X2,Y2),p(X3,Y3),D), distance(p(X3,Y3),p(X4,Y4),D),
  distance(p(X4,Y4),p(X5,Y5),D), distance(p(X5,Y5),p(X1,Y1),D),
  Y1 in {0.0}, X1 in {1.0}, % X1 in the range [1.0..1.0]
  Y2 < Y1, X3 < X2, Y4 > Y3, X5 > X4, Y5 > Y1,
  D in o(0.3)..o(10000.0). % D in the range (0.3..10000.0)

norm(p(X,Y),N):- X**2+Y**2 = N**2.
distance(p(X1,Y1),p(X2,Y2),D):- (X1-X2)**2+(Y1-Y2)**2 = D**2.
:- pentagon(p(X1,Y1),p(X2,Y2),p(X3,Y3),p(X4,Y4),p(X5,Y5),D),
  solve([[X2,X3,X4,X5,Y2,Y3,Y4,Y5,D]]). % Solving at 1.0e-8 (default)
```

Hull-consistency is here the consistency of choice since the constraints are not very complex and involve variables occurring only once. Since hull-consistency is too weak to sufficiently narrow down the domains, we use the predicate `solve` which recursively splits the domain of all its arguments in turn, and re-invokes the narrowing algorithm on each sub-part through backtracking until all the domains the user is interested in have a sufficiently tight width.
3.2 A constraint system handled by box-consistency

A classical benchmark in the interval community is finding the zeros of the functions [31]:

\[ f_i(x_1, \ldots, x_n) = x_i(2 + 5x_i^2) + 1 - \sum_{j \in J_i} x_j(1 + x_j) \quad (1 \leq i \leq n) \]

with \( J_i = \{ j \mid j \neq i \land \max(1, i - 5) \leq j \leq \min(n, i + 1) \} \). These are the Broyden-banded functions. What is important to note here is that, given \( i \), the \( f_i \)'s form a “regular” polynomial system for which the use of Newton methods is especially appropriate.

As it was confirmed by experimental evidence, it is more efficient to consider this system of equations as a whole, instead of handling it incrementally, achieving fixed-point after each constraint is added to the store. We then give the possibility to declare a system of equations (or inequations . . . ) as a single piece by linking all of them with the and connector. All the constraints linked with and are put together in the store and relaxation of the constraint network occurs only once.

An example of code to solve the problem for \( i = 4 \) would be:

```prolog
:- domain([X1,X2,X3,X4],-1,1), % X1=[-1..1], X2=[-1..1], ...
X1*(2+5*X1**2) + 1 - (X1*(1+X1) + X2*(1+X2)) $$= 0 and
X2*(2+5*X2**2) + 1 - (X1*(1+X1) + X3*(1+X3)) $$= 0 and
X3*(2+5*X3**2) + 1 - (X1*(1+X1) + X2*(1+X2) + X4*(1+X4)) $$= 0 and
X4*(2+5*X4**2) + 1 - (X1*(1+X1) + X2*(1+X2) + X3*(1+X3)) $$= 0,
solve([X1,X2,X3,X4]).
```

3.3 Mixing domains and methods

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) may be mixed in the same DecLIC program. This allows the programmer to use the best suited method for each constraint. The domain a variable ranges over 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 \( X \land Y = Z \) constrains its three arguments to be booleans —integer variables ranging in \([0,1])\).
3.3.1 Integer problem over real constraints

Thanks to the use of the constraints `is_integer/1` and `is_fd/1`, it is possible to discriminate integer solutions of a system made up of real constraints. This is illustrated by the Yoshigahara problem [30]: given $A, \ldots, I$, use all digits 1 through 9 only once such that \( \frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1 \), where $BC$, $EF$ and $HI$ stand for the number obtained by concatenating the digit values of the variables.

The `DecLIC` program for the Yoshigahara problem is:

```prolog
:- L=[A,B,C,D,E,F,G,H,I],
   is_fd(L), % solutions are bound to be positive integer ones
   alldifferent(L), % a digit can be used only once
   domain(L,1,9), % every variable must be bound to a digit
   A/(10*B+C)+D/(10*E+F)+G/(10*H+I) $= 1,
   labeling(L).
```

The Yoshigahara problem has only one integer solution (up to permutations of the quotients):

\[
\frac{5}{32} + \frac{7}{68} + \frac{9}{72} = 1
\]

3.3.2 Combination of hull- and box-consistency

Newton methods are at their best when the problem is finding the roots of some polynomials, but are quite inefficient when dealing with a lot of small constraints, such as it may arise when creating constraints recursively. On the contrary, the decomposition of large polynomials, to be treated by hull-consistency, introduces many new variables, which slows down the computation. For example, if we try to solve the system $P_1$ described below:

\[
\begin{align*}
(1) & \quad x^{10} - 2x^9 + 6x^8 - 10x^7 + 17x^6 - 24x^5 + 52x^4 - 80x^3 + 72x^2 - 64x + 32 = 0 \\
(2) & \quad x \prod_{i=1}^{10} y_i = 0 \\
(3) & \quad \sum_{i=1}^{10} y_i^2 = 0
\end{align*}
\]

it is best to solve (1) with Newton methods ($\approx$), and (2) and (3) by enforcing hull-consistency ($\approx$). The code would then be:

```prolog
product([],1).
product([X|L],P):- product(L,Q), P $= X*Q.
sumSquare([],0).
```

14
sumSquare([X|L],S) :- sumSquare(L,T), S $= X**2+T.
:- length(L,10), % L= Y1, Y2, Y3, ..., Y10
    X in -1.0e8..1.0e8, L2=[X|L],
    X**10 + 17*X**6 + 6*X**8 + 52*X**4 + 72*X**2 - 2*X**9 - 24*X**5 -
    10*X**7 - 80*X**3 - 64*X + 32 $$= 0,
    product(L2,0), % X*Y1*Y2*Y3*...*Y10 = 0
    sumSquare(L,0), % Y1**2 + Y2**2 + ... + Y10**2 = 0
    solve(L2).

See Section 5 for a comparison of the computation times obtained by the different consistencies on the examples of this section.

### 3.4 Automatic selection of local consistencies

In the DecLIC language, the choice of the consistency to apply to a particular constraint is deferred to the user. To let the system decide automatically which consistency to choose would certainly be preferable since it would enhance declarativity (only one kind of operator for all constraints) and would not require extra knowledge on constraint solving. However, automatically achieving such a choice appears difficult. “Rules” given previously for choosing between box-consistency and hull-consistency are:

**Case 1.** If the constraint contains few variables occurring many times, then use box-consistency;

**Case 2.** If the constraint contains many variables occurring few times, then use hull-consistency.

A study of the precise meaning of “few” and “many” is difficult if we restrict ourselves to the view “one consistency per user constraint” since constraints may contain some variables occurring many times and other ones occurring few times. Another solution could be the introduction of a new hybrid consistency merging the advantages of both, as investigated in [7]. Such a hybrid consistency would consider constraint projections (that is, unary constraints obtained from the user constraints by replacing all the variables but one by their domains) rather than user constraints. As a consequence, choosing the consistency to apply to a projection would be straightforward:

**Case 1.** If the variable occurs more than once, then use box-consistency;

**Case 2.** If the variable occurs only once, then use hull-consistency.
The study of this hybrid consistency is currently carried out and is the topic of another paper by the authors.

4 Implementation

The details of DecLic implementation are described hereafter. First, a variation of the NAR propagation algorithm (see page 6) is presented; description of the internal data structures follows; finally, compilation of DecLic programs in terms of WAM instructions is exposed.

4.1 The RISC approach

As said previously, all primitive constraints are implemented in clp(fd) in terms of conjunctions of $X$ in $r$ constraints. This approach leads to a very specialized (thus efficient) hard-coded propagation scheme. One of the drawbacks is that it is not trivial to introduce constraints which are not translated into primitives such as the constraints globally handled by box-consistency. However, constraints handled by box-consistency are translated into unary constraints (projections) which are solved using the $X$ in $r$ primitive. Thence, the $X$ in $r$ constraint is an efficient choice for hull-consistency and permits a smooth integration of box-consistency in the same model.

Since we are seeking for a minimal modification of the clp(fd) computational model, we have chosen to introduce a module to handle global constraints propagation only, which interacts as little as possible with the clp(fd) propagation algorithm. The main benefits of this approach are: we can choose easily the way the two propagation algorithms interact; and we keep the efficient propagation scheme of clp(fd) when global constraints are not used. Another problem was that the $X$ in $r$ constraint was not primarily intended to be used with real domains. In particular, one can compute the $r$ part by reasoning on the bounds in clp(fd) (e.g. $X$ in $4*Y..123-(12*Q+T)$). This is not possible when dealing with floats since the necessary rounding cannot be guaranteed to be in the right direction whenever more than one floating-point operation is involved [15]. As a consequence, the syntax for $r$ is restricted to the two forms below when dealing with real-valued variables:

- $X$ in $<\text{float}>..<\text{float}>
- X$ in $f(Y,Z,T,\ldots)$, where $f$ is a function from $\mathbb{R}^n$ to $\mathbb{R}$
4.2 Operational model

A comprehensive presentation of the instructions and structures of the Warren Abstract Machine as well as the clp(fd) internals is beyond the scope of this paper. We refer the reader to [2, 36, 13, 14] for this purpose. In the following, we will call \( D\)-constraints the constraints to be handled by hull-consistency and then decomposed into primitives, and \( N\)-constraints those handled globally by box-consistency.

Solving constraint systems in DecLIC is done using IncNar (see Algorithm 2 page 18), a variation of the Narrowing Algorithm Nar. IncNar works incrementally and is called whenever a constraint is added to the store. 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. The store may be viewed as three sub-systems containing constraints linked between each other by variable dependencies:

1. a “set” \( S_1 \) gathering all \texttt{is_integer} and \texttt{is_fd} constraints;
2. a “set” \( S_2 \) gathering constraints handled by box-consistency;
3. a “set” \( S_3 \) gathering constraints handled by hull-consistency.

For example, a store for Program \( P \):

\[
: - \texttt{is_integer}(X), \texttt{is_fd}(Y), \\
\quad X+Y**2 \enspace \texttt{is} \enspace Z, \enspace Z \enspace \texttt{is} \enspace 6, \\
\quad 3*X**3-4*X**2+2*X-7 \enspace \texttt{is} \enspace Y.
\]

would be the one given in Figure 2. Note that when dealing with constraints handled by hull-consistency, IncNar takes as input the primitive constraints obtained by decomposing the user constraints.

The addition of a new constraint \( c \) to the store is done as described by Algorithm 2. The strategy adopted is to re-invoke \texttt{is_integer} (or \texttt{is_fd}) 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 heuristic is supported by the results of numerous experimental tests.
IncNar(in \((c, N)\) \% constraint to be added
inout \(S_1, S_2, S_3\) \% the 3 boxes of the store
inout \(I = I_1 \times \cdots \times I_n\) \% domains of all the variables

begin

foreach \(i\) in \(\{1, 2, 3\}\) do

\(CS_i := \emptyset\) \% set of constraints of \(S_i\) to be reinvoked

endfor

\(\alpha := \text{typeof}(c)\) \% \(\alpha = 1, 2\) or 3 depending on the “type”

\% of \(c\): is\_integer(), f\$.. or f$$..,

\(CS_\alpha := \{(c, N)\}\) \% \(c\) is to be invoked in store \(S_\alpha\)

\(S_\alpha := S_\alpha \cup \{(c, N)\}\) \% \(c\) is added to the store corresponding to its type

while \((\exists i \in \{1, 2, 3\} | CS_i \neq \emptyset)\) and \((\forall i' \in \{1, \ldots, n\} : I_i' \neq \emptyset)\) do

\% loop until no longer any constraint to be reconsidered in any of the 3 stores

\(j := \min\{\{i \in \{1, 2, 3\} | CS_i \neq \emptyset\}\}\)

Choose one \((c_l, N_l)\) in \(CS_j\)

\% we choose a constraint to be re-invoked in the set corresponding to the store with lowest index (highest priority)

\(I' := N_l(I)\) \% we narrow \(I\) using the CNO \(N_l\)

if \(I' \neq I\) then

foreach \(k\) in \(\{1, 2, 3\}\) do

\(CS_k := CS_k \cup \{(c_m, N_m) \in S_k | \exists x_0 \in \text{var}(c_m) \land I_o' \neq I_o\}\)

\% 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

\(I := I'\)

endif

\(CS_j := CS_j \setminus \{(c_l, N_l)\}\)

endwhile

end.

Algorithm 2: Incremental Narrowing Algorithm.
4.3 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) [14]: 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. Figure 4, page 26, illustrates the use of all the structures for the program of page 24.

4.3.1 Propagation lists

In Algorithm 2, INCNAR uses three sets, $CS_1$, $CS_2$, and $CS_3$, to gather the constraints to be re-invoked. In DecLIC, as in most existing solvers, these sets are managed as FIFO lists called propagation lists. DecLIC uses two propagation lists (one for clp(fd)), the set $CS_1$ having no counterpart:\footnote{is_integer() and is_fd() constraints are re-invoked immediately whenever the domain of a fd-variable is modified.}

The **Global Propagation List (GPL)** is a linked list of all the frames of the variables whose domains have been modified;

The **Newton Propagation List (NPL)** is a linked list of all the constraint frames of the constraints handled by box-consistency which are to be reconsidered.
The heads and tails of the two lists are kept in registers, as described in Table 1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>AF</td>
<td>pointer to the current environment frame</td>
</tr>
<tr>
<td>BP</td>
<td>pointer to the head of the GPL (Global Propagation List)</td>
</tr>
<tr>
<td>TP</td>
<td>pointer to the tail of the GPL</td>
</tr>
<tr>
<td>CF</td>
<td>pointer to the current constraint frame</td>
</tr>
<tr>
<td>BNP</td>
<td>pointer to the head of the NPL (Newton Propagation List)</td>
</tr>
<tr>
<td>ENP</td>
<td>pointer to the tail of the NPL</td>
</tr>
</tbody>
</table>

Table 1: Main registers associated to the data structures

4.3.2 Argument frame

An argument frame (see Figure 3, page 21) is created for each clause in which at least one constraint appears. It contains a pointer to each c-variable and p-variable (see page 11) 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.

4.3.3 Constraint frames

To each constraint in the program corresponds one constraint frame gathering the information necessary for its (re-)invocation. DecLIC manipulates two types of constraint frames: \(D\)-constraint frames created for “\(X\) in \(r\)” constraints, and \(N\)-constraint frames for global constraints. The main difference between the two constraint frames lies in the fact that each \(X\) in \(r\) constraint is invoked by a call to a specific C function associated to it, while constraints handled by box-consistency are all processed by the same C function hard-coded in DecLIC (which implements the operator using interval Newton methods to find leftmost and rightmost quasi-zeros of an expression); \(N\)-constraint frames contain all the arguments needed by that function. Figure 3 page 21, describes the two structures in details.
Figure 3: DecLIC structures.
4.3.4 C-variable frame

A c-variable\(^4\) frame is created at runtime on the heap \([36]\) for every constrained variable (see Figure 3 page 21 for its structure). As in \(\text{clp(fd)}\), a variable frame is divided into three parts:

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

2. a part containing range information: the left and right bounds (coded as IEEE doubles \([17]\)) of the variable domain, and an information word coding 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. A stamp \((\text{Range\_Stamp})\) is used to prevent the trailing of the range each time it is modified. The whole frame coding for the range is trailed at most once per choice-point (time stamps technique described in \([1]\)): a global stamp is incremented at the creation of every choice-point, and decremented when a choice-point is deleted; trailing occurs only when \(\text{Range\_Stamp}\) (current global stamp at the last trailing) is different from the global stamp;

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. Stamp \(\text{Chains\_Stamp}\) is used in the same way as \(\text{Range\_Stamp}\) (see previous point). Note that both stamps are necessary since chains and range are not modified at the same moments.

4.4 \(X \text{ in } r\) compilation

As said above, every primitive constraint supported by \(\text{DecLIC}\) is defined in terms of the constraint \(X \text{ in } r\), and the use of \textit{indexical}\(^5\) \([14]\) is mainly restricted to \texttt{dom()} —which relates here to compact domains only. This is due to the fact that, when dealing with floats, it is unsafe to compute domains by reasoning over bounds; the only safe way is to compute domains

---

\(^4\)See page 11 for the definition of what a c-variable is.

\(^5\)That is, functors allowing to select some parts of a domain: \texttt{dom()} (the whole domain), \texttt{min()} (the left bound), \texttt{max()} (the right bound).
with intervals. The “r” part is compiled into a function which is reinvoked whenever one of the variable domains involved is modified.

**Example 2.**

\[ X \in \text{dom}(Y) + (0(3.4)\ldots7\times3). \ % X \in \text{dom}(Y) \cup (3.4,21] \]

is compiled into

\[
\text{Begin\_Fd\_Constraint(1)} \\
\text{fd\_ind\_dom(0,1) \ % R(0) <- dom(1st variable in argument frame) (Y)} \\
\text{fd\_rounding\_down \ % rounds toward -infinity} \\
\text{fd\_float(2,3.4) \ % T(2) <- 3.4} \\
\text{fd\_rounding\_up \ % rounds toward +infinity} \\
\text{fd\_integer(1,7) \ % T(1) <- 7} \\
\text{fd\_integer(3,3) \ % T(3) <- 3} \\
\text{fd\_term\_mul\_term(1,3) \ % T(1) <- T(1)*T(3)} \\
\text{fd\_interval\_range(1,2,16,8) \ % R(1) <- o(T2)..T(1)} \\
\text{fd\_range\_add\_range(0,1) \ %^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^
4.5.2 N-constraint processing.

At compile-time, DecLIC translates every constraint of the form $f \diamond 0$ (with $\diamond \in \{=, \leq, \ldots \}$) into a character string representing $f$ in a coded prefixed form. For example, the constraint in $p(X,Y) :- X^2 + Y^2 = 1$ is translated into:

"-(+(^(#(0),i(2)),^(#(1),i(2))),i(1))"

where $^\star$ stands for exponentiation operator, $i(n)$ refers to the integer $n$ and $#(m)$ represents the $m$th variable in the argument frame of the clause.

An installation function (see details in Section 4.6) is generated for each 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 permits speeding-up the narrowing process (experimental evidence has shown a speed-up by a factor of 10 when solving systems as a whole vs. handling each constraint of the systems in turn).

**Example 4.**

\[- X^2 + Y^2 = 1 \text{ and } X^2 - Y = 0.\]

*is compiled into*

```verbatim
Begin_Private_Pred
fd_set(2,0) % Creation of an argument frame for X and Y
init_newton_system % Newton propag list <- empty
fd_install_newton_constraint(1,0) % calls Fd_install_Newton(1) for
% 0th variable (i.e. X)
```
Figure 4, page 26, gives an example of the constraint network created at runtime to solve the system of Example 4.

4.6 Constraint handling at runtime

At runtime, the installation of the N-constraints by \texttt{fd_install_newton} is done as follows. Let

\[
\begin{align*}
  c_1 &: f_1(x_{11}, \ldots, x_{m_1}) = 0 \\
  \vdots \\
  c_n &: f_n(x_{1n}, \ldots, x_{mn}) = 0
\end{align*}
\]

be \(n\) N-constraints occurring in an installation function. Given the N-constraint \(c_i : f_i(x_{1i}, \ldots, x_{mi}) = 0\):

For each projection of \(c_i\) wrt. the variable \(x_{ji}\),

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

- the partial derivative \(\frac{\partial f_i}{\partial x_j}\) is computed symbolically and inserted in \(NF_{i,j}\),

- since the computation of the projection associated to \(NF_{i,j}\) depends on the domains of all the variables \(x_{1i}, \ldots, x_{mi}\), this N-constraint frame is added in the \texttt{Chain\_Newton} list of the variable frame of all those variables,

- \(NF_{i,j}\) is added to the \texttt{Newton Propagation List} whose head and tail addresses are kept respectively into the \texttt{BNP} and \texttt{ENP} registers.

After installation, the NPL is relaxed as described in Algorithm 3 by the instruction \texttt{fd\_call\_newton\_constraints}.
Figure 4: a constraint network.
**Algorithm 3: Newton relaxation.**

### 4.7 Interaction of solvers

The interaction of the two different solvers is achieved through the variable domains. As said previously, relaxation is done first with the N-constraints before considering D-constraints.

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

- its variable frame is added to the end of the *general propagation list* 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 indicates which dependency chain of the variable frame needs to be considered for re-invocation of the constraints it links). It is worth noticing that if $c$ is a N-constraint, `Chain_Newton` is *not* marked to be reconsidered, since the relaxation over all N-constraints needing to be reinvoked will be done immediately; moreover, if $x$ was already in the GPL, re-invocation of constraints in `Chain_Newton` is forbidden by modifying its `Queue_Propag_Mask`;

```plaintext
NEWTONNAR(in: NPL, a linked chain of N-constraint frames to be reinvoked;
            out: failure or success)
begin
  cstr ← first frame of NPL
  repeat
    Apply Newton narrowing on $\text{Tell}_{Fdv_{Adr}}$ of $cstr$
    if failure then
      Delete NPL
      Return failure
    else
      if Range of $\text{Tell}_{Fdv_{Adr}}$ modified then
        Push $\text{Tell}_{Fdv_{Adr}}$ frame in GPL
        Push all N-constraint frames depending on $\text{Tell}_{Fdv_{Adr}}$ in NPL
      endif
    endif
    cstr ← next frame of NPL
  until fixed-point is reached \(\text{i.e. NPL is empty}\)
  return success
end.
```
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 Algorithm 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 the whole process repeats from step 1, until a stable state (i.e. emptiness of GPL) is reached.

4.8 WAM extension

We will only describe here new instructions added to the WAM instructions set, leaving apart the modifications of already existing instructions. An in-depth presentation of all the other WAM instructions can be found in [13, 14].

\texttt{fd_rounding}\{ \texttt{up} \texttt{down} \}. Used in \( X \) in \( \mathbf{r} \) constraints to achieve outward rounding when computing bounds of the \( \mathbf{r} \) part. Recall that it is unsafe to use more than one primitive operation (\(+, -, \ldots\)) in the expression of a bound as the error introduced when rounding the result may change sign (i.e. the result becomes rounded in the wrong direction).

\texttt{fd_float}(T, n). Executes \( T \leftarrow \text{(double)}(n) \) (used when computing \( \mathbf{r} \) part of \( X \) in \( \mathbf{r} \) constraints).

\texttt{fd_infinity}(T). Executes \( T \leftarrow +\infty \). Used on request when the user specifies a constraint of the form \( X \) in \( 3.5 .. \infty \), for example.

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

\texttt{fd_install_newton_constraint}(nb, X). Initialises the argument frame pointer to \( X \), then invokes \( nb^{th} \) constraint installation function.

\texttt{fd_install_newton}(str). Installs N-constraint represented by string \( str \): parses \( str \) to create the trees representing the constraint and its derivatives; creates a N-constraint frame per projection and adds them to the
Benchmarks | Hull | Box | Box_ϕ | Box_ϕ/Hull
--- | --- | --- | --- | ---
Bifurcation | 5624 | 14195 | 2478 | 0.44
Pentagon | 1058 | 5652 | 2426 | 2.29
Chemical Eq. | 5556 | 14621 | 4993 | 0.90
i₁ | 413 | 224 | 117 | 0.28
i₂ | 467 | 228 | 115 | 0.25
i₄ | ? | ? | 130500 | ?
Cosnard 20 | ? | 62701 | 28910 | ?
Broyden-Banded 10 | ? | 3416 | 917 | ?
Wilkinson | 3049 | 539 | 437 | 0.14
P₁ | 2510 | ? | 10100 | 4.02
P₂ | ? | 181871 | 28955 | ?

Times in milliseconds.

Table 3: Experimental results.

**Chain Newton** entry of the variable pointed by the AF register as well as to the NPL.

**fd_call_newton_constraints.** Executes the relaxation of the NPL, followed by general relaxation of the GPL on success. Used when adding a N-constraint.

## 5 Experimental Results

DecLIC has been tested on various examples from numerical analysis and CLP(Interval) benchmarks. Two versions were tested: one with a solver enforcing “perfect” box-consistency and another one enforcing box_ϕ-consistency.
5.1 Benchmarks analysis

Table 3 presents the computational results of DecLIC. They were obtained on a Sun UltraSparc 1 with a precision of $10^{-8}$ (width of the computed domains). DecLIC uses for the most part an improvement factor of 10\(^6\), and the $\varphi$ parameter of box\(_\varphi\)-consistency is set to 0.1. Hull is the time obtained when using only hull-consistency, Box the time when using “perfect” box-consistency, Box\(_\varphi\) the time for box\(_\varphi\)-consistency, and Hull+Box\(_\varphi\) the time when using a combination of hull-consistency and box\(_\varphi\)-consistency. Times of computation exceeding 10 min. are replaced by a question mark (?).

Bifurcation [11] (test from numerical bifurcation) relates to the solving of a non-linear constraint system involving three variables; Pentagon (coordinates of a regular pentagon), extracted from [20], is the benchmark described in page 12; Chemical Eq. [22] describes a chemical problem: given the stoichiometric equation of the propane combustion in the atmosphere

$$C_3H_8 + \frac{R}{2}(O_2 + 4N_2) \rightarrow (CO_2, H_2O, N_2, CO, H_2, H, OH, O, NO, O_2)$$

find the number of moles of all the products formed per mole of propane consumed (time given in last column of the second array is the one obtained by using hull-consistency on quasi-primitive constraints and box\(_\varphi\)-consistency on complex constraints. The increase in time is due to the fact that those complex constraints involve many variables occurring only once); \(i_1\) (10 variables, 10 equations), \(i_2\) (10 variables, 10 equations), and \(i_4\) (20 variables, 20 equations), are non-linear constraint systems [16, 25]; Cosnard 20 [26] is a system of 20 equations and 20 variables obtained by discretizing the nonlinear integral equation:

$$\int_0^1 H(s, t)(u(s) + s + 1)^3 \, ds = 0$$

where

$$H(s, t) = \begin{cases} 
    s(1 - t), & s \leq t, \\
    t(1 - s), & s > t 
\end{cases}$$

\(^6\)When propagating, a domain is considered to have been modified by the application of a CNO only if its size decreased by more than 10%.
Broyden-Banded 10 (cf. page 13) is the Broyden-Banded problem with 10 variables; Wilkinson [37] relates to finding the roots of the polynomial:

\[ P(x) = \prod_{i=1}^{20} (x + i) + \varepsilon x^{19}, \quad \text{with} \quad \varepsilon = 2^{-23} \]

\( P_1 \) is the system described in page 14; \( P_2 \) is the system described below:

\[
\begin{cases}
    x_1^5 - 28x_1^4 + 288x_1^3 - 1358x_1^2 + 2927x_1 - 2310 & = 0 \\
    x_2^4 - 10x_2^3 + 35x_2^2 - 50x_2 + 24 & = 0 \\
    x_3^4 - 4x_3^3 - 53x_3^2 + 60x_3 + 108 & = 0 \\
    x_4^5 - 23x_4^4 + 155x_4^3 - 241x_4^2 - 420x_4 & = 0 \\
    x_5^5 - 12.5x_5^4 - 28.5x_5^3 + 341x_5^2 + 848x_5 + 336 & = 0 \\
    x_6^7 - 28x_6^6 + 322x_6^5 - 1960x_6^4 + 6769x_6^3 - 13132x_6^2 + 13068x_6 - 5040 & = 0 \\
    \sum_{i=1}^{6} x_i & = 9 \\
    \sum_{i=1}^{6} ix_i & = 46 \\
    \prod_{i=1}^{6} x_i & = 0 
\end{cases}
\]

Table 3 demonstrates that no consistency is the good choice for every problem: for example, hull-consistency is twice fast as box-consistency on Pentagon, whereas box-consistency outperforms hull-consistency by a factor of 4 on a problem such as \( i_2 \). Moreover, one may easily see that box-\( \psi \)-consistency is far more efficient than box-consistency on all the benchmarks given here. Benchmarks such as Cosnard 20 or Pentagon do not benefit from the possibility to use both hull-consistency and box-consistency as the constraints they involve are of the same type (complex constraints involving variables occurring many times, or constraints with variables having few occurrences). On the other hand, problems such as \( P_1 \) and \( P_2 \) which involve constraints of both types show that a significant speed-up may be obtained by judiciously choosing the consistency to apply on each constraint.

6 Conclusion

In this paper, we have shown that it is possible to implement an efficient solver over continuous domains by extending the WAM used in clp(fd) with a few new instructions, and by modifying some of the internal structures. The speed-up obtained with box-\( \psi \)-consistency demonstrates that for some
problems, enforcing perfect box-consistency does not achieve the best trade-off between interval contraction and computation time. Hull-consistency can then be a better choice. As said previously, deferring the choice of the consistency to the user spoils the declarativity of the language. A better solution could be the introduction of a new hybrid consistency merging the advantages of both, as investigated in [7]. There would then be only one kind of operator for all constraints.

We also plan to add dynamic constraint generation in order to be able to support conditioning and computer algebra methods such as Gröbner bases, and to modify the treatment of constraints over integer-valued variables in order to obtain performances comparable to those of clp(fd).

Acknowledgement

We are indebted to Daniel Diaz and Philippe Codognet for having released the source code of clp(fd) and for their help and comments on the development of DecLIC. This work was partially supported by the DiSciPi ESPRIT project № 22532 and by the INRIA LOCO project. We also thank the editors of this special issue and the referees for their numerous comments and their helpful suggestions to improve this paper.

References


