*Joint NCC & IIS Bull.*, Comp. Science, 12 (1999), 52–57


# New facilities of constraint logic programming*

T.M. Yakhno, E.S. Petrov

In this paper we focus on new facilities of a logic programming system ECL$^i$PS$^e$ and propose a way of using these facilities for implementation of a constraint programming method called Subdefinite Computations. We briefly describe the Interval Domain library which employs the proposed implementaion technique for solving non-linear constraints. The capabilities of the Interval Domain library are illustrated by the Euclidean Steiner Tree Problem.

## 1. Introduction

Constraint Logic Programming (CLP) is a powerful paradigm of implementation of complex applications. It combines automated reasoning and various constraint satisfaction techniques borrowed from mathematics and artificial intelligence. Usually, constraints express some specific knowledge, logical statements describe some desired behaviour, and together they make a constraint logic program.

During recent twenty years Constraint Programming has developed a true variety of methods (Arc Consistency algorithm [11], Subdefinite (SD) Computations [12], Interval Newton method [8], Tolerance Propagation [10]) and tools (UniCalc [1], ILOG-solver [13], Numerica [9]).

Employing constraint solving techniques in Logic Programming has brought into existence CHIP, PROLOG-III, CLP(BNR), ECL$^i$PS$^e$ which are classics of CLP [2, 4, 6, 5, 7].

SD computations are a constraint satisfaction technique numerically solving non-linear systems of equations and inequalities which may contain discontinuous functions, imprecise coefficients, real and integer variables. Given such a system, it produces a multi-dimensional box that contains all the solutions to the system.

ECL$^i$PS$^e$ is a CLP system. It allows a user to program constraint satisfaction techniques directly at the language level. The paper discusses these facilities (Section 2), SD computations (Section 2.1), and a technique for implementation of SD computations in ECL$^i$PS$^e$ (Section 2.2). Section 3 describes Interval Domain library for resolution of non-linear constraints in ECL$^i$PS$^e$ on the basis of SD computations. The capabilities of the library are illustrated by a geometric application.

## 2. ECL$^i$PS$^e$

ECL$^i$PS$^e$ is an abbreviation for *ECRC Common Logic Programming System*. It is a prolog-based system aimed at serving as a platform for creating various extensions of logic programming. ECL$^i$PS$^e$ offers two data types, *meta-term* and *delayed goal*, which significantly simplify this process.

A meta-term consists of two or more terms. The first term, called *prolog value* of the meta-term, is accessible to any predicate. The other (one or more), called *meta-attributes* of the meta-term, are accessible only to few tools which convert meta-terms to standard prolog data and vice versa. A meta-term is written like `T{name1:T1, ... }`, where `T` is its prolog value, `T1` is its meta-attribute `namei`, etc.

Formally, a delayed goal is a term unifiable only with itself and free variables. Delayed goals represent actions which should be done in the future. There are three major actions on delayed goals: creation, scheduling for execution, and execution of all scheduled goals. A delayed goal is written like `'GOAL'(G)`; the label `'GOAL'` indicates that execution of the goal `G` is delayed.

*We extend our thanks to Institut franco-russe A.M. Liapunov d'informatique et de mathématiques appliquées (grant 98–06).

```
q(1, X, Y):-
  make_suspension(q(1, X, Y), 3, F),                    % [1]
  extract(Y, Y_dom, Y_goals),                           % [2]
  assign(Y, Y_dom, [F|Y_goals]),                        % [3]
  extract(X, X_dom, X_goals),                           % [4]
  compute_the_domain_of_X(1, X_dom, Y_dom, Changed),    % [5]
  ( var(Changed) -> true ;                              % [6]
    assign(X, X_dom, []),                               % [7]
    schedule_woken(X_goals),                            % [8]
    wake                                                % [9]
  ).
```

**Figure 1.** A simplified code of a calculation function

Using meta-attributes and delayed goals, an application can organize its information and operation flows independently of prolog standards.

## 2.1. SD Computations

If $Q$ is a relation and $x$, $y$, ... are variables, then the formula $Q(x\,y\ldots)$ is a *constraint*. Each variable is associated with a domain; we write $x$ instead of *domain of* $x$. Given a set of constraints, SD computations remove some values which violate the constraints from the domains.

SD computations pay much attention to domain representation because it is a question of efficiency. Storing and processing a greater amount of information requires a greater amount of time. Formally, a domain representation is a function $(\cdot)^\star$ which widens an arbitrary domain up to the closest representable one.

A constraint $Q(x\,y\ldots)$ defines the following transformations of $x$, $y$, ...

$$x := \mathrm{pr}_1(Q \cap x \times y \times \ldots)^\star,$$
$$y := \mathrm{pr}_2(Q \cap x \times y \times \ldots)^\star, \ldots.$$

which are called *calculation functions* ($\mathrm{pr}_i$ is a projection on the $i$-th coordinate, $\times$ is the Cartesian product).

For each calculation function, its left hand side (LHS) is the variable to the left of :=, and its right hand side (RHS) is the set of variables really needed to recompute the LHS. Usually, we have LHS $\notin$ RHS.

A set with two operations "add an element" and "remove an element" is called a *flow*. Given a set of constraints, SD computations form a flow of calculation functions defined by these constraints and, until the flow is exhausted, proceed as follows: (1) remove a function from the flow and execute it, (2) if that has changed the function's LHS, then add to the flow all the functions containing this changed LHS in their RHS's. Informally, the calculation functions are "bridges" which carry changes from one domain to another.

## 2.2. Implementation in ECL$^i$PS$^e$

This section briefly describes an implementation of SD computations in ECL$^i$PS$^e$.

First of all, each variable $x$ occurring in constraints is turned into a meta-term

```
X{sd:var(Min, Max, Fs)}
```

whose meta-attribute `sd` stores the domain of $x$ (the interval `[Min, Max]`) and the calculation functions having $x$ on their RHS's (the list `Fs`).

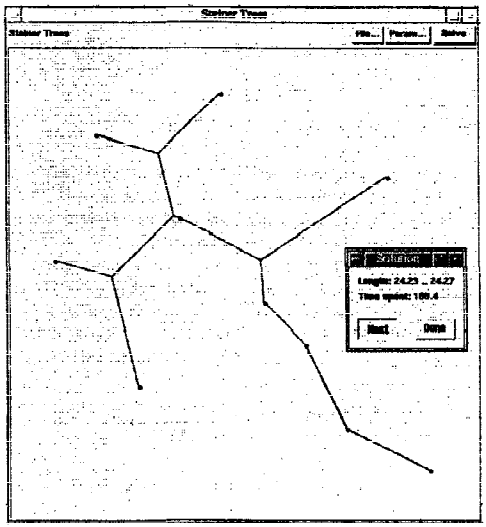

**Figure 2.** A Euclidean Steiner tree

Each $n$-ary relation $Q$ is associated with an $(n+1)$-ary predicate q. This predicate processes meta-attributes of its arguments as follows. Suppose there is a constraint $Q(x\,y)$. Then a call `q(1, X, Y)` is equivalent to execution of the calculation function $x := \mathrm{pr}_1(Q \cap x \times y)^\star$. Likewise, `q(2, X, Y)` is equivalent to $y := \mathrm{pr}_2(Q \cap x \times y)^\star$. Thus, a suitable representation for calculation functions in ECL$^i$PS$^e$ is delayed goals. ECL$^i$PS$^e$ maintains a list of all delayed goals which plays the role of the flow of calculation functions. Addition of a new element to the flow is equivalent to creation of a delayed goal, and removal of an element is equivalent to execution of a delayed goal.

Figure 1 shows an implementation of a calculation function in ECL$^i$PS$^e$. Since execution of a delayed goal destroys it, the predicate creates (`[1]`) and adds (`[2]`–`[4]`) a copy of the delayed goal being executed to the calculation functions having $y$ on their LHS's (it is assumed that $x$ is not on the LHS of the calculation function being executed). Next, $x$'s domain is recomputed (`[5]`). If the domain has got tighter, then all functions with $x$ on their RHS's are scheduled for execution and executed (`[6]`, `[9]`).

## 3. Interval Domain Library

The technique discussed above is employed in a new ECL$^i$PS$^e$ library "Interval Domain" (ID). The ID library accepts equations and inequalities written in the standard way and, for each variable occurring there, computes an interval the variable ranges within. The constraints may contain arbitrary combinations of arithmetic operations, trigonometric functions, functions ln, exp, $n$-ary min and max, and relations $=$ and $\leq$. Besides that, the library supports the user-defined functions and includes predicates for operating on domains, changing accuracy, and locating solutions. We illustrate some capabities of The ID library by the Euclidean Steiner tree problem.

The Steiner tree problem is stated as follows. Given a finite set $R$ of required points, find a set $S$ of Steiner points such that, for any set $S'$ of points, the tree spanning $R \cup S$ (the Steiner tree) is no longer than the tree spanning $R \cup S'$. Figure 2 shows an example of such a tree (junctions and circles are respectively the Steiner and required points).

Steiner trees are well-studied objects [3].

1. Each Steiner point is incident to exactly three edges which meet at the angle of $2\pi/3$.
2. The number of Steiner points is at most the number of required points minus two, that is $\|S\| \leq \|R\| - 2$.

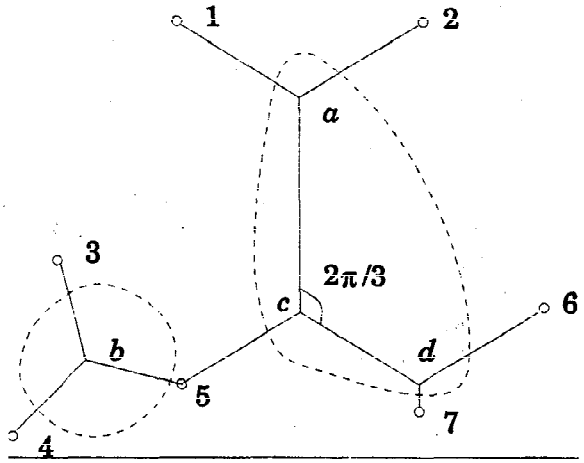


**Figure 3.** A Steiner tree and two its subtrees

3. A Steiner tree can be only $2/\sqrt{3} = 1.1547\ldots$ times shorter than the tree spanning its required points.

The ID library has been applied to computing positions of points in Steiner trees of a prespecified *topology*. Let $T$ be a Steiner tree spanning the required and Steiner points $R$ and $S$. Let some function $n$ map $R \cup S$ to consecutive natural numbers. The topology of $T$ is a tree $(V, E)$ such that (1) $v \in V$ iff $v = n(p)$ for some point $p \in R \cup S$, and (2) $(v, w) \in E$ iff $v = n(p)$, $w = n(q)$ for some edge $(p, q)$ in $T$. Intuitively, what remains of $T$, if we ignore its geometry, is its topology.

If all the edges, incident to one or more required point, are removed from $T$, then it falls into disjoint subtrees, each containing only Steiner points. Because of this disjointness, different subtrees can be processed independently of each other. Therefore, without loss of generality, we assume that there is only one such a subtree, i.e. $\|S\| + 2 = \|R\|$.

Note that there must be a point $s^\star \in S$ such that two of its neighbours w.r.t. T are in $R$ (otherwise $\|S\| + 2 > \|R\|$). We add directions to edges of $T$ as if $s^\star$ were its root. With each point $p \neq s^\star$, we associate (1) $\rho_p > 0$, the length of the edge directed towards $p$, (2) $\alpha_p$, the angle between that edge and the positive part of $x$-axis, (3) $x_p$, $y_p$, the coordinates of $p$. If $p$ is in $R$, then its coordinates $x_p$ and $y_p$ are already known.

If $s \neq s^\star$ and $(s, p)$, $(s, q)$ are directed edges, then there should hold

$$x_p = x_s + \rho_p \cos\alpha_p, \quad y_p = y_s + \rho_p \sin\alpha_p,$$

and $|\alpha_s - \alpha_p| = |\alpha_s - \alpha_q| = \pi/3$, $|\alpha_p - \alpha_q| = 2\pi/3$. If $p, q \in R$, $s \in S$ are neighbours of $s^\star$, then their parameters should meet

$$\begin{aligned}
&\alpha_p = \alpha_s + 2\pi/3, \quad x_p = x_{s^\star} + \rho_p \cos\alpha_p, \quad y_p = y_{s^\star} + \rho_p \sin\alpha_p,\\
&\alpha_q = \alpha_s - 2\pi/3, \quad x_q = x_{s^\star} + \rho_q \cos\alpha_q, \quad y_q = y_{s^\star} + \rho_q \sin\alpha_q,\\
&\qquad\qquad x_s = x_{s^\star} + \rho_s \cos\alpha_s, \quad y_s = y_{s^\star} + \rho_s \sin\alpha_s.
\end{aligned}$$

Let us describe the tree shown in Fig. 3 by constraining `Xp`, `Yp`, `Rp` and `Ap` (that is $x_p$, $y_p$, $\rho_p$ and $\alpha_p$). The required and Steiner points are respectively mapped to integers and letters. Thus, $V$ ={1, 2, 3, 4, 5, 6, 7, $a$, $b$, $c$, $d$}, $E$ ={(1, $a$), (2, $a$), (3, $b$), (4, $b$), (5, $b$), (5, $a$), (6, $d$), (7, $d$), ($a$, $c$), ($c$, $d$)}, and the topology is $(V, E)$.

This tree falls into two subtrees, each generating a separate subsystem of constraints. The first subtree contains only $b$ (its root), the second contains $a$ (its root), $c$, $d$. Figure 3 shows the query for

```
/*-----------------------The 1st subtree-----------------------*/
A4*==A3+2*pi/3,      X3*==Xb+R3*cos(A3),   Y3*==Yb+R3*sin(A3),
A_5*==A3-2*pi/3,     X4*==Xb+R4*cos(A4),   Y4*==Yb+R4*sin(A4),
                     X5*==Xb+R_5*cos(A_5), Y5*==Yb+R_5*sin(A_5),
/*-----------------------The 2nd subtree-----------------------*/
abs(A5-Ac)*==pi/3,   X5*==Xc+R5*cos(A5),   Y5*==Yc+R5*sin(A5),
abs(Ad-Ac)*==pi/3,   Xd*==Xc+Rd*cos(Ad),   Yd*==Yc+Rd*sin(Ad),
abs(A6-Ad)*==pi/3,   X6*==Xd+R6*cos(A6),   Y6*==Yd+R6*sin(A6),
abs(A7-Ad)*==pi/3,   X7*==Xd+R7*cos(A7),   Y7*==Yd+R7*sin(A7),
abs(A5-Ad)*==2*pi/3, abs(A6-A7)*==2*pi/3,
A1*==Ac-2*pi/3,      X1*==Xa+R1*cos(A1),   Y1*==Ya+R1*sin(A1),
A2*==Ac+2*pi/3,      X2*==Xa+R2*cos(A2),   Y2*==Ya+R2*sin(A2),
                     Xc*==Xa+Rc*cos(Ac),   Yc*==Ya+Rc*sin(Ac),
/*---------------------The required points---------------------*/
[X1, X2, X3, X4, X5, X6, X7] = [4, 11, 1, 0, 4.5, 14, 11],
[Y1, Y2, Y3, Y4, Y5, Y6, Y7] = [11, 11, 5, 0, 1.5, 3.5, 1],
[R1, R2, R2, R3, R4, R5, R_5, R6, R7, Rc, Rd] ** 0..infty,
locate([Xa,Xb,Xc,Xd,Ya,Yb,Yc,Yd]).
```

**Figure 4.** The Steiner tree query

cooridnates of $a$, $b$, $c$, $d$. The answer is `Xa=Xa{[7.652, 7.654]}`, `Ya=Ya{[8.981, 8.983]}`, `Xb=Xb{[1.913, 1.914]}`, `Yb=Yb{[2.078, 2.081]}`, `Xc=Xc{[7.757, 7.759]}`, `Yc=Yc{[3.464, 3.470]}`, `Xd=Xd{[10.986, 10.988]}`, `Yd=Yd{[1.682, 1.687]}`, which means that $a \in [7.652, 7.654] \times [8.981, 8.983]$, $b \in [1.913, 1.914] \times [2.078, 2.081]$, and so on.

## 4. Conclusion

This paper is focused on a technique for integration of SD computations into a CLP system ECL$^i$PS$^e$. The integration has become possible due to reasons of different nature: (1) a declarative approach to problems of constraint and logic programming, and (2) similarity of control mechanisms in SD computations and the ECL$^i$PS$^e$. These reasons have made integration similar to diffusion.

This technique has been applied to implementation of the ID library. The benefit of the library is that it brings the power of constraint solving methods into a conventional programming language. Another point is that, having ECL$^i$PS$^e$ as a base system, users can combine ID with other techniques available under ECL$^i$PS$^e$.

In the future, our efforts will be applied to development of applications on the basis of the ID library, and combining within ECL$^i$PS$^e$ SD computations and techniques from the interval analysis.

We extend our thanks to the A.P. Ershov Institute of Informatics Systems SB RAS, the Russian Research Institute of Artificial Intelligence and Institut Franco-Russe A. M. Liapunov d'informatique et de mathématiques appliquées (grant 98–06).

## References

[1] A. Babichev, O. Kadyrova, T. Kashevarova, A. Semenov, *Unicalc — as a tool for solving problems with inaccurate and sub-definite data*, Interval Computations, **3**, No 5, 1992, 13–16.

[2] F. Benhamou and W. J. Older, *Programming in CLP(BNR)*, Proc. PPCP'94, Newport, USA, 1994.

[3] N. Christofides. *Graph theory: an Algorithmic Approach*, Management Science, Academic Press, Imperial College, London, 1975.

[4] J. Cohen, *Constraint logic programming languages*, Communications of the ACM, **33**, No 7, July 1990, 52–68.

[5] A. Colmerauer, *An introduction to Prolog III*, Communications of the ACM, **33**, No 7, July 1990, 69–90.

[6] M. Dincbas, H. Simonis, P. Hentenryck, A. Aggoun, T. Graf, and E. Berthier, *The constraint logic programming language CHIP*, FGCS'88, Tokyo, Japan, Nov. 1988. ICOT.

[7] ECRC. *ECL$^i$PS$^e$ 3.5: ECRC Common Logic Programming System. User's Guide.*, 1995.

[8] E. R. Hansen, R. I. Greenberg, *An interval Newton method*, Applied Mathematics and Computing, 12, 1983, 89–98.

[9] P. Hentenryck, L. Michel, Y. Deville, *Numerica: a Modelling Language for Global Optimization*, The MIT Press, Cambridge, MA, 1997.

[10] E. Hyvőnen, *Constraint reasoning based on interval arithmetic: the tolerance propagation approach*, Artificial Intelligence, **58**, 1992, 71–112.

[11] A. K. Mackworth, *Consistency in networks of relations*, Artificial Intelligence, **8**, No 1, 1977, 99–118.

[12] A. S. Narinyani, *Subdefiniteness and basic means of knowledge representation*, Computers and Artificial Intelligence, **2**, No 5, 1983, 443–452.

[13] J.-F. Puget, *A C++ implementation of CLP*, Proc.SPICIS, Singapore, 1994.