EaCL 1.5: An Easy abstract Constraint optimisation Programming Language

Technical report ***CSM-324***
16th February 1999

Patrick Mills (millph@essex.ac.uk)
Edward Tsang (edward@essex.ac.uk)
Richard Williams (rjw@essex.ac.uk)
John Ford (fordj@essex.ac.uk)
James Borrett (james@gol.co.uk)
Department of Computer Science,
University of Essex,
Wivenhoe Park,
Colchester, Essex, CO4 3SQ.
United Kingdom.

Abstract

This technical report describes a new language for describing constraint satisfaction problems, called EaCL. Traditional Constraint Programming Languages have been built on top of host languages such as Prolog, Lisp, C++. This means that the user must have reasonable knowledge of the syntax and semantics of the host language before being able to use the constraint technology effectively. On top of this, the user may also be required to specify the heuristics and, or algorithm to solve the constraint problem. This leads to a bottleneck in the amount of people who have the necessary expertise in both constraint programming and the host language to implement practical systems, which use constraint satisfaction techniques. Our language is designed to abstract out as many of these details as possible, to produce a high level language, where the problem specification is the focus. With this in mind, our language is designed to be simple, high level, intuitive and declarative (the order in which constraints are specified has no significance). This technical report subsumes previous technical reports on EaCL.

Introduction

A constraint satisfaction problem is a problem where one is given a finite set of variables, each of which associated with a (normally finite) domain. Constraints restrict the values to be taken by the variables simultaneously. The problem is to assign one value to each variable satisfying all the constraints [8],[1],[2].

Constraint programming systems have had remarkable achievement in many applications. Many more applications could have benefited from it had there been more experts in the field to exploit the technology. Successful though they are, previous approaches to building constraint-programming systems have been based on taking some host language, e.g. C++ (e.g. ILOG solver [5]), Lisp (e.g. PECOS [6]) or Prolog (e.g. ECLiPSE [3], CHIP [7]), augmented in some way with constraint technology. This means that the user of these constraint programming systems needs to have two basic skills before they can make use of the traditional constraint programming systems:

- Be able to formulate the problem as a constraint satisfaction problem,
- Be able to program in the host language.

Some recent global optimisation modelling languages, e.g. HELIOS, ILOG Numerica [4],[9] allow users to define their problems, almost as they would in technical papers. Our aim is to minimise the amount of knowledge required by the end user to be able to specify their problem in our language. The approach we have taken is to design our language to be simple and intuitive to use, similar in some ways to [4],[9], but targeted at Constraint Satisfaction Problems, whilst keeping it expressive enough to define real world problems.

1. Language Definition

In this section we describe the EaCL language in detail. First of all we give the general structure of a problem, described in EaCL, and then we go on to describe the constraints and functions available and additional language features.

1.1 Overall structure of an EaCL problem file

An problem defined in EaCL is split into 4 sections (see Figure 1)

```
Problem:ProblemName
{
        Data
        {
                //Data that defines a problem instance
        }
        Domains
        {
                //Domain declarations
        }
        Variables
        {
                //Variable declarations
        }
        Constraints
        {
                //Constraint declarations
        }
        Optimisation
        {
                //Minimise or maximise some function
        }
}
```

Figure 1: The skeleton of a problem in EaCL

The data section is for declaring data, which may define a particular problem instance. The next section, the domains section, is for defining the domains of variables in the problem. The variables section is for declaring variables and arrays of variables, with a given domain. The constraint section is for defining the constraints in the problem. The optimisation section, is used to define criteria which we wish to minimise or maximise.

By structuring the problem file, in this way, we hope the user will be encouraged to think of the problem formulation process as a four-step process:

- What data defines each problem instance?
- What are the variables and domains in the problem?
- What are the constraints on solutions to the problem?
- What function do we wish to minimise or maximise?

1.2 Comments in EaCL

EaCL comments are C or C++ style comments. One line comments begin with "//" and multi-line comments can be made using the C-style "/*" to begin the comment, and "*/" to end the comment.

1.3 Domains and Variables

Three types of variables are available in EaCL: integer variables, boolean variables and set variables (variables whose value must be some set of integers or the empty set). When a variable or array of integer or set variables is declared, some domain must be specified for that variable or group of variables. The domains of integer variables are defined by the set of integers from which a value can be picked for the variable. This may be specified as either a set of integer values, or a range of integer values. Similarly, the domain of a set variable is defined by the set of integer values, a subset of which must be the value of the variable. The keywords `IntDom` and `SetDom` are used to specify what sort of domain is being declared. Boolean variables have a default domain containing only false and true, and so no domain is specified for Boolean variables. Several variables may be declared with the same domain, by listings the group of variables, separated by commas. Figure 2 shows some examples.

```
Problem:DomVarEGs
{
      Domains
      {
            IntDom Dr = [0,10];
            IntDom Ds = {1,2,3,4};
            SetDom Dss = {1,2,3,4,5,6};
      }

      Variables
      {
            SetVar x,y,z::Dss;
            IntVar t,r::Dr;
            IntVar a,b::Ds;
            IntVar iArray[100]::Dr;
            BoolVar C,D,E;
      }
}
```

Figure 2 : Examples of variable and domain declarations

In EaCL, arrays and lists are interchangeable. From now on, we shall refer to arrays and lists as just lists. Arrays of variables may be declared in the variable section of an EaCL problem file (see Section 1.3). Named lists of values, sets of values and values may be defined in the data section, whilst named lists of any type of expression can be declared in the constraints section.

1.4 Integer constraints

Below we list the set of constraints, which can be applied to integer variables and expressions. Integer expressions are expressions built from the functions and operators defined in Section 1.7 or integer variables or integer values. Constraints are also considered to be integer expressions, which return 0 if they are violated or 1 if they are satisfied. For each constraint, we list the syntax for the particular constraint, together with the valid parameters for each constraint.

=
- Syntax: *IntegerExpr1 = IntegerExpr2*
- Type: Constraint
- *IntegerExpr1* and *IntegerExpr2* are integer expressions.
- Constrains *IntegerExpr1* to be equal to *IntegerExpr2*.

<>
- Syntax: *IntegerExpr1* <> *IntegerExpr2*
- Type: Constraint
- *IntegerExpr1* and *IntegerExpr2* are integer expressions.
- Constrains *IntegerExpr1* to be not equal to *IntegerExpr2*.

>=
- Syntax: *IntegerExpr1* >= *IntegerExpr2*
- Type: Constraint
- *IntegerExpr1* and *IntegerExpr2* are integer expressions.
- Constrains *IntegerExpr1* to be greater than or equal to *IntegerExpr2*.

=<
- Syntax: *IntegerExpr1* =< *IntegerExpr2*
- Type: Constraint
- *IntegerExpr1* and *IntegerExpr2* are integer expressions.
- Constrains *IntegerExpr1* to be less than or equal to *IntegerExpr2*.

>
- Syntax: *IntegerExpr1* > *IntegerExpr2*
- Type: Constraint
- *IntegerExpr1* and *IntegerExpr2* are integer expressions.
- Constrains *IntegerExpr1* to be greater than *IntegerExpr2*.

<
- Syntax: *IntegerExpr1* < *IntegerExpr2*
- Type: Constraint
- *IntegerExpr1* and *IntegerExpr2* are integer expressions.
- Constrains *IntegerExpr1* to be less than *IntegerExpr2*.

AllDifferent
- Syntax: `AllDifferent(`*ListOfIntegerVars*`)`
- Type: Constraint
- *ListOfIntegerVars* is a list of integer variables.
- Constrains all members of *ListOfIntegerVars* to have different values.

Sequence
- Syntax: `Sequence(`*ListOfIntegerVars*`,` *BagOfIntegerVals*`)`
- Type: Constraint
- *ListsOfIntegerVars* is a list of integer variables.
- *BagOfIntegerVals* is a list of integer values, e.g. [1,2,2,4]).
- Constrains all members of *ListOfIntegerVars* to take a value from *BagOfIntegerVals*, but no one item in *BagOfIntegerVals* must be assigned to more

than one variable in *ListOfIntegerVars* (unless the same items occurs at least that many times as in the *BagOfIntegerVals*).

Member
- Syntax: `Member(`*IntegerExpr*, *SetOfIntegerVals*`)`
- Type: Constraint
- *IntegerExpr* is an integer expression.
- *SeOfIntegerValst* is a set of integer values.
- Constrains *IntegerExpr* to be a member of *SetOfIntegerVals*.

NotMember
- Syntax: `NotMember(`*IntegerExpr*, *SetOfIntegerVals*`)`
- Type: Constraint
- *IntegerExpr* is an integer expression.
- *SeOfIntegerValst* is a set of integer values.
- Constrains *IntegerExpr* not to be a member of *SetOfIntegerVals*.

1.5 Logical constraints

In EaCL constraints can be combined to make more complicated constriants, using logical constraints, which can be used to constrain the combinations of those constraints that must be satisfied. We list the set of logical constraints available in EaCL below.

AND
- Syntax: *Con1* `AND` *Con2*
- Type: Constraint
- *Con1* and *Con2* are constraints.
- The overall ANDed constraint holds if and only if both the constraints hold.

OR
- Syntax: *Con1* `OR` *Con2*
- Type: Constraint
- *Con1* and *Con2* are constraints.
- The overall ORed constraint holds if and only if one or both of the constraints hold.

XOR
- Syntax: *Con2* `Xor` *Con2*
- Type: Constraint
- *Con1* and *Con2* are constraints.
- The overall XORed constraint holds if and only if only one and only one of the constraints holds.

NOT
- Syntax: `NOT` *Con*
- Type: Constraint
- *Con* is a constraint.

- The overall NOTed constraint holds if and only if the constraint *Con* does not hold.

IFF
- Syntax: *Con1* `IFF` *Con2*
- Type: Constraint
- *Con1* and *Con2* are constraints.
- The overall constraint holds if and only if both the constraints hold or neither of them do.

IMPLIES
- Syntax: *Con1* `IMPLIES` *Con2*
- Type: Constraint
- *Con1* and *Con2* are constraints.
- The overall constraint holds if and only if *Con1* does not hold or both *Con1* and *Con2* hold.

1.6 Set constraints

Set expressions are expressions built from the functions and operators from Section 1.8, set variables or static sets containing integer values only.

Member
- Syntax: `Member`*(IntExpr*, *SetExpr*)
- Type: Constraint
- *IntExpr* is an integer expression.
- *SetExpr* is a set expression.
- Constrains *IntExpr* to be a member of *SetExpr*.

NotMember
- Syntax: `NotMember`*(IntExpr*, *SetExpr*)
- Type: Constraint
- *IntExpr* is an integer expression.
- *SetExpr* is a set expression.
- Constrains *IntExpr* not to be a member of *SetExpr*.

Subset
- Syntax: `Subset(`*SubSet*`,`*Set*`)`
- Type: Constraint
- *Subset* and *Set* are set expressions.
- Constrains *SubSet* to be a subset of *Set*.

StrictSubset
- Syntax: `StrictSubset(`*StrictSubSet*`,`*Set*`)`
- Type: Constraint
- *StrictSubset* and *Set* are set expressions.
- Constrains *SubSet* to be a strict subset of *Set*.

AllDisjoint

- Syntax: `AllDisjoint(`*ListOfSets*`)`
- Type: Constraint
- *ListOfSets* is a list of set expressions.
- Constrains the intersection of all elements of *ListOfSets* to be the empty set.

1.7 Integer functions

Arithmetic operators `+`, `−`, `*`, `/`, `%`

- Syntax: *IntegerExpr1 ArithOp IntegerExpr2* where *ArithOp* := + | − | `*` | `/` | `%`
- Type: Integer expression
- Parameters: *IntegerExpr1* and *IntegerExpr1* are integer expressions.
- Returns an integer expression, as follows:
  +, constrained to be the sum of *IntegerExpr1* and *IntegerExpr1*
  -, constrained to be *IntegerExpr1* minus *IntegerExpr1*
  *, constrained to be the product of *IntegerExpr1* and *IntegerExpr1*
  /, constrained to be *IntegerExpr1* divided by *IntegerExpr1*
  %, constrained to be the remainder of *IntegerExpr1* / *IntegerExpr1*

Abs

- Syntax: `Abs(`*IntegerExpr*`)`
- Type: Integer expression
- *IntegerExpr* is an integer expression.
- Returns an integer expression constrained to be absolute value of *IntegerExpr*.

Power

- Syntax: `Power(`*IntegerExpr*, *IntegerVal*`)`
- Type: Integer expression
- *IntegerExpr* is an integer expression.
- *IntegerVal* is an integer value.
- Returns a variable constrained to be equal to *IntegerExpr* to the power of *IntegerVal*.

Count

- Syntax: `Count(`*ListOfIntegerVariables, ListOfIntegerValues*`)`
- Type: Integer expression
- *ListOfIntegerVariables* is a list of integer variables
- *ListOfIntegerValues* is a list of integer values
- Returns an integer expression equal to the number of variables in *ListOfIntegerVariables* with a value from *ListOfIntegerValues*.

Sum

- Syntax: `Sum(`*ListOfIntegerExprs*`)`
- Type: Integer expression
- *ListOfIntegerExprs* is a list of integer expressions
- Returns an integer expression constrained to be equal to the sum of the variables in *ListOfIntegerExprs*.

ScalProd

- Syntax: `ScalProd`(*ListOfIntegerExprs*`,`*ListOfIntegerValues*`)`
- Type: Integer expression
- *ListOfIntegerExprs* is a list of integer expressions
- *ListOfIntegerValues* is a list of integer values
- Returns an integer expression constrained to be equal to the sum of the product of each element of *ListOfIntegerExprs* multiplied by the corresponding element in *ListOfIntegerValues*.

Minimum

- Syntax: Minimum(*ListOfIntegerExprs*)
- Type: Integer expression
- *ListOfIntegerExprs* is a list of integer expressions.
- Minimum is constrained to be the minimum of the *ListOfIntegerExprs*.

Maximum

- Syntax: Maximum(*ListOfIntegerExprs*)
- Type: Integer expression
- *ListOfIntegerExprs* is a list of integer expressions.
- Maximum is constrained to be the maximum of the *ListOfIntegerExprs*.

Operator [] – the indexing operator

- Syntax: *ListOfIntegerValues*[*IntegerExpr*]
- Type: Integer expression
- ListOfIntegerValues is a list integer values.
- IntegerExpr is an integer expression.
- *ListOfIntegerValues*[*IntegerExpr*] is constrained to be the *IntegerExpr*th element of *ListOfIntegerValues*.

1.8 Set functions & operators

Intersection

- Syntax: `Intersection(`*ListOfSetExprs*`)`
- Type: Set expression
- Parameters: *ListOfSetExprs* is a list of set expressions
- Returns a set expression constrained to be the intersection of all the members of *ListOfSetExprs*.

Union

- Syntax: `Union(`*ListOfSetExprs*`)`
- Type: Set expression
- *Parameters:* *ListOfSetExprs* is a list of set expressions
- Returns a set expression constrained to be the union of all the members of *ListOfSetExprs*.

#

- Syntax: `#`*SetExpr*
- Type: Integer expression

- Parameters: *SetExpr* is a set expression.
- Returns an integer expression constrained to be the size of *SetExpr*.

1.9 Lists

Lists of objects may be created in two ways, either by listing all the elements, e.g. [1,2,3,4] or using an intensional definition, where the elements of the list are specified by a predicate. The syntax for an intensional definition of a list is as follows:

`[` *expr* | *some predicate on expr*`,` *RangeDeclarations*`]`
where *RangeDeclarations* is either:
`subsets` *expr* `of` *SomeList*,
`sublists` *expr* `of` *SomeList*,
`subseqs` *expr* `of` *SomeList* or
expr `in` *SomeList* where *SomeList* is either:
a list of values,
a set of values,
a range of values, specified by `[`*minval* `..` *maxval*`]`.

This means expr can iterate over all `subsets` of some set of values, all `sublists` of some list of values, all `subseq`uence`s` of consecutive elements of some list of values, or all elements of some range of values, allowing excellent array, list and set handling, vital to produce clear and elegant specifications of many constraint programming problems.

The vertical bar "|" should be read, as such that, and commas are read as "and". An example of a list specifying the even numbers in the range of integers from 0 to 10 might be defined:

```
[ x | x in [0..10], x % 2 = 0]
```

Similarly, sets of values (this is different from set variables, as the sets must be static) may be specified in the same way, except curly brackets are used on the outside:

`{` *expr* | *some predicate on expr, RangeDeclartions*`}`
where *RangeDeclarations* is the same as defined for intensional lists

It should be noted that the predicates for defining intensional lists and sets should not contain any problem variables.

Lists and sets may be concatenated together using the ++ operator. The syntax is:

List1 ++ *List2* Or *Set1* ++ *Set2*

An example might be `[1,2,3,4] ++ [5,6,7,8]`, which would be equivalent to `[1,2,3,4,5,6,7,8]`. Similarly for sets, this would be `{1,2,3,4} ++ {5,6,7,8}`, which would be equivalent to `{1,2,3,4,5,6,7,8}`. If one wants to find the length of some list or set, we use the # operator to obtain this:

e.g. `#[1,2,3,4,5,6]` (evaluates to 6)

1.10 Forall, if, else, .. constructs

Sometimes a lot of similar constraints may need to be placed on an array or list of variables, with certain conditions on which variables are constrained and which are not. The Forall construct is for such situations, and is used to specify ranges of index values, possibly subject to some predicate, so that multiple constraints may be specified over the indices specified. The syntax of Forall is as follows:

`Forall` (*RangeDeclarations*, *optional predicate on expr*)
{
    //Constraints
}
    Where *RangeDeclarations* is the same as defined for intensional lists

An example where the Forall construct might be used is for specifying the N queens problem in EaCL, see Figure 3. The predicates must not contain any problem variables.

```
Problem:NQueens
{
     Data
     {
          //The number of queens in the problem
          N := 8;
     }

     Domains
     {
          //N possible positions per row
          IntDom Row=[0,N-1];
     }

     Variables
     {
          //One queen per row
          IntVar Q[N]::Row;
     }

     Constraints
     {
          //Make sure that no queen attacks another
          Forall (i in [0..N-1], j in [0..N-1], i < j)
          {
               Q[i] <> Q[j];
               Q[i] - Q[j] <> i - j;
               Q[i] - Q[j] <> j - i;
          }
     }
}
```

Figure 3: Example use of Forall for specifying the N queens problem

If several different conditions (these must not contain any variables) need to be placed on a range of indices, then If Else constructs may be used to enforce these. The syntax of If Else (the Else parts are optional of course) is:

```
If ( predicate on indices )
{
	//Constraints
}
Else If ( predicate2 on indices )
{
	//Constraints
}
Else
{
	//Constraints
}
```

1.11 User defined constraints and functions

Some groups of constraints and functions need to be defined several times, with slightly differences every time. User defined constraints and functions can be used to help to minimise mistakes in rewriting them and simplify the problem definition. This also makes the problem definition more modular, making changes easier to make. The syntax for a user-defined constraint is as follows:

```
Constaint NameOfConstraint(Parameters)
{
	//List of constraints whose conjunction forms this new constraint
}
```

An example is the AtLeast constraint, which can be expressed in terms of the Count function:

```
Constraint AtLeast(mn, vals, vars)
{
	Count(vars,vals) >= mn;
}
```

In addition to user-defined constraints, the user may sometimes find it useful to define their own functions. The syntax for user defined functions is:

```
Function NameOfFunction(Parameters)
{
	//Internal details
	return result;
}
```

A simple example might be a function for squaring its parameter:

```
Function Square(x)
{
	return x * x;
}
```

It should be noted that user-defined constraints and functions must be defined, before they are used in the problem file.

2. Optimisation section

The optimisation section is used to specify a criteria which we wish to minimise or maximise. The syntax for the optimisation section is:

```
Optimisation
{
      Minimise (somefunction);
}
```

Or

```
Optimisation
{
      Maximise (somefunction);
}
```

Only one Minimise or Maximise criteria may be specified for each problem, and *somefunction* must be an integer variable or integer expression. This syntax allows the minimum violation problem to be specified simply and easily using the Count function to count the number of violated constraints, aswell as more conventional integer optimisation problems.

3. An example: A Job-shop scheduling problem

A factory is asked to produce 4 products. Each product must be produced according to a deadline. Each product has to be processed by four machines, A, B, C and D in a specific sequence, each for a specific amount of time. The sequences and time requirements are listed in the following table.

	Step 1	Step 2	Step 3	Step 4
Product 1	Machine 1 4 minutes	Machine 2 5 minutes	Machine 3 4 minutes	Machine 4 3 minutes
Product 2	Machine 2 2 minutes	Machine 1 8 minutes	Machine 4 6 minutes	Machine 3 2 minutes
Product 3	Machine 4 2 minutes	Machine 3 2 minutes	Machine 2 6 minutes	Machine 1 4 minutes
Product 4	Machine 4 5 minutes	Machine 1 3 minutes	Machine 2 8 minutes	Machine 3 2 minutes

Table 1: Sequence and time requirements for a Job-Shop scheduling example problem

The problem is to find a schedule for all the jobs, which meets the sequence requirements in Table 1 and minimises the maximum time to finish all the jobs.

```
//
//Job-shop scheduling problem
//10/6/98,     Patrick Mills,   written
//2/2/99,      Patrick Mills,   modified to minimise
//                              the duration to finish
//                              all jobs.
//
Problem:JobShop
```

```
{
    Data
    {
        nprods:=        4;
        prodsmn:=       0;
        prodsmx:=       nprods-1;
        nsteps:=        4;
        stepsmn:=       0;
        stepsmx:=       nsteps-1;
        machine:=       ((1,2,3,4),
                         (2,1,4,3),
                         (4,3,2,1),
                         (4,1,2,3));
        duration:=      ((4,5,4,3),
                         (2,8,6,2),
                         (2,2,6,4),
                         (5,3,8,2));
        mxlaststart:=   23;
    }

    Domains
    {
        IntDom Time=(0,mxlaststart);
        IntDom EndTime=(0,50);
    }

    Variables
    {
        IntVar startstep(nprods)(nsteps)::Time;
        IntVar endjob(nprods)::EndTime;
    }

    Constraints
    {
        //Check sequence is time is correct and within deadline
        //(Assume product can be moved from one machine to another
        //immediately)
        Forall (pi in (prodsmn..prodsmx), si in (stepsmn..stepsmx))
        {
            //Check in sequence
            If (si < 3)
            {
                startstep(pi)(si) + duration(pi)(si) <= startstep(pi)(si+1);
            }
        }

        //JobA and JobB must not overlap
        Constraint NoOverlap(startstepA, startstepB, durationA, durationB)
        {
            (startstepA >= startstepB + durationB)
            Or
            (startstepA + durationA <= startstepB);
        }

        //Check no overlap of jobs on the same machine
        Forall (pi in (prodsmn..prodsmx), pj in (pi+1..prodsmx))
        {
            Forall (si in (stepsmn..stepsmx), sj in (stepsmn..stepsmx))
```

```
                {
                        If (machine(pi)(si) = machine(pj)(sj))
                        {
                                NoOverlap(startstep(pi)(si), startstep(pj)(sj),
                                            duration(pi)(si), duration(pj)(sj));

                        }
                }
        }

        //Calculate the end times for each job
        Forall (pi in (prodsmn..prodsmx))
        {
                endjob(pi) = startstep(pi)(3)+duration(pi)(3);
        }
    }

    //Minimise the maxium finish time of a job
    Optimisation
    {
        Minimise(Maximum((endjob(pi) | pi in (prodsmn..prodsmx))));
    }
}
```

The data from Table 1 is stored in the Data section of the problem file in the lists machine, duration and deadline. The domains of variables are the set of possible start times for each job. The variables are the start times of each step of each product, and are represented by a two dimensional array of variables. The sequence constraints are specified using the Forall construct in conjunction with the If Else construct, ensuring that each step has finished before the next starts. A user-defined constraint is used to specify that there should be no overlap between jobs, using the same machine. The auxilary variables endjob are constrained be the end time for each job. Finally, the optimisation section is used to minimise the maximum end time of any job.

4. Conclusion

We have presented EaCL, a new high level declarative language for describing constraint satisfaction and constraint optimisation problems in a structured fashion. It includes facilities for indexing arrays, generating intensional lists and sets, and building user defined constraints and functions. It is easy to use, abstracting out many of the details present in conventional constraint programming languages, yet powerful enough to specify real world problems.

5. Acknowledgements

We would like to thank Nathan Barnes for his useful comments and suggestions on the EaCL language. This project is funded by the EPSRC grant GR/L20122.

6. References

[1] Freuder, E.C. & Mackworth, A., (ed.), “Constraint-based reasoning”, MIT Press, 1994.

[2] Marriott, K. & Stuckey, P.J., “Programming with constraints, an introduction”, MIT Press, 1998.

[3] Meier, M. & Schimpf, J., "An Architecture for Prolog Extensions", Proceedings of the 3rd International Workshop on Extensions of Logic Programming, Bologna, 1992.

[4] Michel, L. & Van Hentenryck, P., "Helios: A Modeling Language for Global Optimization", Proceedings of PACT 96, pages 317-336, 24-26th April 1996, London, UK.

[5] Puget, J-F., "A C++ Implementation of CLP", Proceedings of SPICIS 94, November 1994, Singapore.

[6] Puget, J-F., "PECOS: a high level constraint programming language", Proceedings of SPICIS 92, September 1992, Singapore.

[7] Simonis, H., "The CHIP system and its applications", in Montanari, U. & Rossi, F. (ed.), Proceedings, Principles and Practice of Constraint Programming (CP'95), Lecture Notes in Computer Science, Springer Verlag, Berlin, Heidelberg & New York, 1995, 643-646.

[8] Tsang, E., "Foundations of Constraint Satisfaction", Academic Press 1993.

[9] Van Hentenryck, P., Michel, L. & Deville, Y., "Numerica: A Modeling Language for Global Optimization", The MIT Press, spring 1997.