The 7th International Symposium on Operations Research and Its Applications (ISORA'08)
Lijiang, China, October 31–Novemver 3, 2008
Copyright © 2008 ORSC & APORC, pp. 131–140

A Note on Mixed Set Programming

Jianyang Zhou

ENGINEST, 1 allée de l'Alzette – 54500 Vandoeuvre-Lès-Nancy, zhou@enginest.com

Abstract This short paper presents the "Mixed Set Programming" framework of NCL in modeling and solving constraint satisfaction problems over a mixed domain of reals, integers, Booleans, references, and sets. Modeling abstraction and the solving of a few hard combinatorial problems such as set partitioning and job-shop scheduling are illustrated.

Keywords Mixed Set Programming; Natural Modeling

1 Introduction

Combining techniques from Artificial Intelligence (AI), Operations Research (OR) and Logic Programming (LP), NCL [16,17] is an operations research language with conventional mathematical logic as syntax. NCL supports implicit typing, global semantic analysis and context-based constraint solving.

NCL's programming style is: *Mixed Set Programming*. By set programming we do not mean the simple use of set notations or set variables in a solver system, but rather rigorous and complete set theoretical formulation and reasoning in a systematic way to solve industrial problems. By "Mixed Set Programming", we mean 1) global reasoning over a mixed domain of reals, integers, Booleans, references, and sets; 2) the solver incorporates and combines a simplified form of first order logic, naïve set reasoning, numerical constraints and operations research algorithms in a cooperative way for modeling and solving constraint satisfaction problems.

2 The problem solving scheme

To facilitate the presentation, we use the following convention in variable notation: Possibly subscripted by *i, j, k, l* and numbered by *n*, we take a,b,c for Boolean variables, d for integer constant, x,y,z,w for integer variables, A,B,C,D for set variables, and f,g,h for real variables.

Throughout this paper, *inf A* denotes infimum of set A; *sup A* denotes supremum of set A, *#A* denotes the cardinal of set A; $\underline{x}$ denotes lower bound of integer variable x; $\bar{x}$ denotes upper bound of x; Δx denotes the domain of x; $\underline{A}$ denotes certainly accepted part of set variable A; ΔA denotes the fuzzy part of A; and $\bar{A}$denotes the entire domain of A; $A[n]$ denotes the n-th element of set A. "Big circle" O denotes any data input from a certain data source; NCL supports input-level typing by analyzing data type in data source directly.

We assume that all other notations are conventional.

Generally, the problem solving scheme of NCL is simple: constraint cutting and tree search – both are conventional concepts in the OR community. Briefly, NCL's constraint solving scheme is based on the following algorithm: It first uses constraints to cut the solution space actively so as to contain combinatorial explosion. Each time such *cutting* process terminates, if solution is not concluded, a *branching* scheme will be used to search for solutions. The method is to split the domain of some variable into two subsets and deal with the sub-problems corresponding to these two parts respectively. As such *cutting* and *branching* go on, the system will either reach some solution or prove *no solution* to the sub-problem, provided that initially all the variables' domains are finitely splittable. In the latter case, the system backtracks to search other branches for solutions.

On the aspect of search, NCL supports rules-based search: Programmer can specify a real expression for a variable labeling criterion to orient the search. This is done through criteria-critical rules in NCL's universal quantification logic: focusing on the quantifier index which minimizes or maximizes certain criteria. The general principles of NCL for variable and value labeling are as follows:

1. Least Slack: uncertainty of the search is as small as possible. For example, selecting the most constrained variable, or simply selecting the smallest domain variable: $\rightarrow$ (min $\#\Delta x_i$);
2. Greatest Slack: domain splitting triggers algorithmic propagation as much as possible, e.g., selecting the greatest domain variable: $\rightarrow$ (max $\#\Delta x_i$);
3. Least Regret: choice making tends to be the easiest if the difference in the search criterion is the biggest, e.g., on VRP, selecting order i on which the difference between second shortest and the shortest distances is the greatest: $\rightarrow$ (max {∀k∈ Δ next$_i$ distOrder$_{i,k}$}[2] - {∀k∈next$_i$ distOrder$_{i,k}$}[1]);
4. Ordering: respecting problem structure to keep solution space as convex as possible, e.g., selecting order i with earliest release time: $\rightarrow$ (min $\underline{\text{t1Order}_i}$);
5. Greedy Search: as greedy as possible in terms of optimization objectives, e.g., selecting order i with lowest cost: $\rightarrow$ (min costOrder$_i$).

NCL supports 2 enumeration modes with 2 directions under logical conditions:

1. Binary domain splitting ($\Rightarrow$; $\Leftarrow$): Split the domain of a variable into two halves. First try the lower part (or the upper part), and then examine the remaining part of the domain recursively;
2. Bound enumeration ($\rightarrow$; $\leftarrow$): First try the lower bound (or the upper bound), and then consider the remaining part of the domain recursively.

3 Modeling Abstraction

This section presents a part of mathematical logic models that NCL understands and that are typical for planning and scheduling problems. Modeling abstraction of NCL is based on the observation that most of combinatorial problems can be seen as a mix of basic abstract models, for example, vehicle routing is a mixed problem of set partitioning, routing and knapsack, etc.

3.1 Distinctness and Disjointness

For solving combinatorial problems, constraints related to "distinctness" are almost indispensable. Many problems such as *Queens, Knight's Tour*... all need the constraint of "distinct integers" (all-different) which states that integers $x_1 \ldots x_n$ are distinct. Such constraint with a lot of variants is basically expressed as:

$\forall i \neq j \in [1, n] \quad x_i \neq g_j$,

By analogy, in the context of sets user will need the disjoint-sets constraint:

$\forall i \neq j \in [1, n] \quad A_i \cap A_j = \emptyset$,

3.2 Set Covering and Packing

For solving packing and assignment problems, constraints related to "set covering" and "set disjointness" are frequently used. Problems such as *Set Partitioning*, *Crew Scheduling*, etc. all involve constraints as follows:

Set Covering:

$C = \cup_{i \in [1,n]} A_i$,

Packing:

1. $\forall i \neq j \in [1, n] \quad x_i \neq x_j \quad \vee \quad y_i \neq y_j$,
2. $\forall i \neq j \in [1, n] \quad x_i \neq x_j \quad \vee \quad A_i \cap A_j = \emptyset$,
3. $\forall i \neq j \in [1, n] \quad A_i \cap A_j = \emptyset \quad \vee \quad B_i \cap B_j = \emptyset$,

The above models or their variants represent a certain generality in expressing many combinatorial problems such as *Square Packing, Ship Loading, Production Scheduling,* etc.

3.3 Indexing and Sorting

Linear Integer Sorting: Integers $(z_1 \ldots z_n)$ are an ascending sorting of $(y_1 \ldots y_n)$, with order variables $(x_1 \ldots x_n)$ to permute $(y_1 \ldots y_n)$ to $(z_1 \ldots z_n)$.

$\forall i \in [1, n] \ (x_i \in [1, n] \ , y_i = z_{x_i}$,

$\forall i < j \in [1, n] \ (x_i \neq x_j, z_i \leq z_j)$,

Linear Set Sorting: Sets $(B_1 \ldots B_n)$ are an ascending sorting of $(A_1 \ldots A_n)$, with order variables $(x_1 \ldots x_n)$ to permute $(A_1 \ldots A_n)$ to $(B_1 \ldots B_n)$.

$\forall i \in [1, n] \ (x_i \in [1, n] \ , A_i = B_{x_i})$,

$\forall i < j \in [1, n] \ (x_i \neq x_j, B_i \prec B_j)$,

See the solving of the *Job-shop Scheduling* problem for an explanatory note.

Recursive Integer Sorting: Integers $(z_1 \ldots z_n)$ are sorted recursively in an ascending order through successor indicators $(x_1 \ldots x_{n-1})$.

$\forall i \in [1, n\text{-}1] \ (x_i \in [2, n] \ , z_i \leq z_{x_i})$,

The *Recursive Integer Precedence* model is used to formulate the routing aspects in *Production Scheduling* problems.

Recursive Set Sorting: Sets $(B_1 \ldots B_n)$ are sorted recursively in an ascending order through successor indicators $(x_1 \ldots x_{n-1})$.

$\forall i \in [1,n\text{-}1] \ (x_i \in [2, n] \ , B_i \prec B_{x_i})$,

The *Recursive Set Sorting* model is used to formulate the routing aspects in *Vehicle routing* and *Production Scheduling* problems.

3.4 Sum and Cumulation

Sum of reals: $f = \sum_{i \in A} g_i$,

Sum of integers: $y = \sum_{i \in A} x_i$,

Sum of Booleans: $y = \sum_{i \in A} a_i$,

The *Sum* models can be used to formulate problems such as *Knapsack*.

Cumulation:

$\forall i \in D \quad y_i = \sum_{j \in A}(x_j = i)$,

$\forall i \in D \quad y_i = \sum_{j \in A}((i \in C_j) \times x_j)$,

The *Cumulation* models can be used to formulate problems such as *Timetabling and Personnel Planning.*

A Concluding Note

A small part of the mathematical models that NCL understands are presented. To practically support descriptive intelligence and solving capability, several tens of such models are tackled in NCL's model abstraction. NCL [16] stems from Constraint Programming, but the basic differences of NCL from CP languages such as Prolog III, CLP(R), CHIP, OZ [4,10,6,14] (just to name a few) are two-fold:

- At the parser level: NCL's "*Semantic Parser"* understands natural problem formulation in mathematical logic and submits abstract models to the solver.

- At the solver level: *Mixed Set Programming* solves problems over a mixed domain of reals, integers, Booleans, references, and sets. The solver incorporates a simplified form of first order logic, naïve set reasoning, numerical constraints and operations research algorithms to solve problems in a **cooperative** manner.

In particular, compared to CP languages [4,10,6,14], NCL's contribution is the design of the Semantic Parser (AI techniques), and higher-level mathematical logic abstraction of problem formulation (LP techniques). A common point is the design and implementation of embedded algorithms (OR techniques). However, in NCL no special predicates such as *Sort*, *Alldifferent*, *Alldisjoint, Cumulative*, *Cycle/Path*, *Global Cardinality/Among/Distribute*, etc. are introduced, but NCL includes its unique algorithms for solving these related problems and many others.

For comparison, in *N. Beldiceanu and E. Contejean* [1], OR algorithms are embedded in predicates called "Global Constraints". User uses Global Constraints to formulate problems. Unfortunately, Global Constraints do not fit in NCL. This is because NCL's kernel deals with several hundreds of elementary and big algorithms. If global constraints (or specialized constraints) are explicitly defined, too many specific names and their variants plus peripheral concepts will be introduced:

- There are several tens of global algorithms such as "sort" in NCL;

- "Sort" alone has at least 4 variants for reals, integers and sets each, and may be linked to peripheral concepts such as "UnaryResource", "AscendingSorting", "Regret-BasedSearch", etc.;

- Sophisticated reasoning links between algorithms are omnipresent in NCL for context-based constraint solving.

In the Global Constraints case, NCL's user would be drowned by hundreds of specific concepts. In such a situation it would be "clumsy" for users to program in NCL. To avoid this inconvenience, NCL adopts **"Semantic Parser"** that recognizes problems' natural formulation and automatically guides the NCL engine to intelligently solve the problems. Semantic Parser liberates programmers from specific modeling using pre-defined predicates such as "sort".

Another observation is that the famous *simplex* mathematical model [5] can also be viewed as a Global Constraint: *Simplex* is for solving linear equality/inequality model - one among many constraint models. NCL is different from modeling languages such as AMPL [8] which is mainly based on linear model.

To explain the programming style of NCL, let's take the famous square-packing problem as an example. The square-packing problem consists in placing a collection of squares into one big square compactly [2]. NCL describes this problem using simple mathematical logic expressions as below.

```
d = 112,
n = 21,
$\forall$ i $\in$ [1,n] (
    $s_i$   $=$   O,
    $X_i$   $\subset$   [1, d],
    $Y_i$   $\subset$   [1, d],
    $X_i$   =   [ $x_i$, $x_i$+ ($s_i$–1) ],
    $Y_i$   =   [ $y_i$, $y_i$+ ($s_i$–1) ],
),
$\forall$ i $\neq$ g $\in$ [1,n]
    $X_i \cap X_j = \emptyset$   $\vee$   $Y_i \cap Y_j = \emptyset$,
$\forall$ i $\in$ [1,n] $\rightarrow$ ( min $\underline{x}_i$) $X_i$ = ?,     % ordering search: lowest-abscissa first
$\forall$ i $\in$ [1,n] $\rightarrow$ ( min $\underline{y}_i$) $Y_i$ = ?.     % ordering search: lowest-ordinate first
```

The sides of the small squares are given in variable array s_i (from input we get 50, 42, 37, 35, 33, 29, 27, 25, 24, 19, 18, 17, 16, 15, 11, 9, 8, 7, 6, 4, 2). Because all squares do not overlap, two squares must have their abscissas or ordinates disjoint ($X_i \cap X_j = \emptyset \quad \vee \quad Y_i \cap Y_j = \emptyset$). We search for the abscissas of the squares by placing the biggest squares with lowest-abscissa first and then we search for the ordinates of theses squares. Instantly, all 8 solutions are found after a complete exploration of the solution space.

4 Modeling and Solving Hard Problems

In this paper, all given computation results are obtained on a PC with a CPU of 1.83GHz. The models given here are the most basic ones, but the presented NCL programs are all complete solutions to the corresponding problems. Due to scalability consideration, methods such as decomposition, iteration, rules, and/or meta-heuristics are often employed to tackle effectively real problems. In practice, NCL is a language for users to program the solution method (decomposition, modeling, iteration, business rules, meta-heuristics, etc.) to deal with complex problems.

4.1 Set Partitioning

The set partitioning problem requires selecting from a set of subsets to make a partition for a given set. The test bed for this problem is cited from [9,11]. Its NCL program is:

```
nbTask = O,                          % number of tasks
nbShift = O,                         % number of shifts
TASK = [1, nbTask],
SHIFT = [1, nbShift],
∀ i ∈ SHIFT (
   $costShift_i = O$,                % cost of shift i
   $TaskShift_i = O$,                % set of tasks of shift i
),
Partition ⊂ SHIFT,
$\cup_{i \in Partition}$ $TaskShift_i$ = TASK,     % set partitioning constraint
∀ i ≠ j ∈ Partition
   $TaskShift_i \cap TaskShift_j = \emptyset$,
∀ i ∈ SHIFT → (
   min     inf $TaskShift_{i,}$      % ordering search
   min     $costShift_{i,}$          % greedy search
   max     #$TaskShift_{i,}$
)
i ∈ Partition ?,
min     $\sum_{i \in Partition} costShift_i$.
```

To each shift i is associated its set of tasks *TaskShift*$_i$ and its cost *costShift*$_i$NCL needs to compute *Partition* which is a subset of *SHIFT* such that the union of tasks of all shifts indexed by elements of *Partition* equals to *TASK*. The solution is to decide whether a shift should be selected and the search rule is: (lowest-task, minimum-cost, biggest-shift) first.

On this problem, NCL's computation results are interesting: Without any problem-specific preprocessing or symmetry breaking, all the 3 instances (nw19, nw09, nw06) not solved in [11] plus instances nw07, nw11 not tackled in [11] can be solved completely (with proof of optimality) within 20 minutes for nw19, nw09, nw06, nw07 and 46 minutes for nw11. Solutions are:

nw19: $\sum_{i \in Partition}$ costShift$_i$ = 10898,
 Partition = {62, 134, 484, 942, 1543, 2126, 2699},

nw09: $\sum_{i \in Partition}$ costShift$_i$ = 67760,
 Partition = {3, 10, 18, 27, 42, 108, 210, 424, 810, 1217, 1306, 1929,
 2170, 2581, 2651, 2904},

nw06: $\sum_{i \in Partition}$ costShift$_i$ = 7810,
 Partition = {4, 136, 200, 421, 874, 1538, 2901, 3845},

nw07: $\sum_{i \in Partition}$ costShift$_i$ = 5476,
 Partition = {23, 88, 591, 1943, 3017, 5150},

nw11: $\sum_{i \in Partition}$ costShift$_i$ = 116256,
 Partition ={9, 52, 135, 221, 333..334, 368, 2414, 2960, 3638, 4077,
 6286, 6940, 7380, 7511, 8438, 8696, 8753, 8819}

What merits attention is that data format of [9] is tailored for linear solvers. In practice, data modeling can be refit to NCL for better solving the problem.

4.2 Job-shop Scheduling

Given *n* jobs each consisting of *m* tasks that have to be processed on *m* machines, the job-shop problem involves scheduling the jobs on the machines so as to minimize the makespan subject to precedence, duration and disjunctive constraints. The problem is taken from [12]. The problem instance presented here is the famous mt10 instance. The NCL program is:

```
nbJob = O,                                % number of jobs
nbMachine = O,                            % number of machines
TIME = O,                                 % time horizon
JOB = [1, nbJob],
MACHINE = [1, nbMachine],
$\forall$ i $\in$ MACHINE (
  $\forall$ j $\in$ JOB (
    $Task_{i,j}$ = [$release_{i,j}$, $due_{i,j}$],   % time interval of job j on machine i
    $Sorted_{i,j}$ = [$t1_{i,j}$, $t2_{i,j}$],       % task of the job sorted j-th on machine i
    $Task_{i,j}$ $\subset$ TIME,
    $Sorted_{i,j}$ $\subset$ TIME,
    $order_{i,j}$ $\in$ JOB,                      % job execution order on a machine
    $Task_{i,j}$ = $Sorted_{i,orderi,j}$,           % $Task_{i,j}$ permuted to $Sorted_{i,\_}$ by $order_{i,j}$
  )
  $\forall$ j $\in$ JOB (
           $o_{i,j} = O$                        % order of job j on machines is known
    $\#Task_{o_{i,j},j} = Os$                    % task duration is known
  ),
  $\forall$ j < k $\in$ JOB (
    $order_{i,j}$ $\neq$ $order_{i,k}$,            % permutation constraint over order
    $Sorted_{i,j}$ $\prec$ $Sorted_{i,k}$,          % precedence constraint over sorted tasks
  )
),
$\forall$ j $\in$ JOB
$\forall$ i < k $\in$ MACHINE
  $Task_{o_{i,j},j}$ $\prec$ $Task_{o_{k,j},j}$      % ordering of job j on the machines
$\forall$ i $\in$ MACHINE $\rightarrow$ (
  min $\sum_{k \in JOB}$ # $\Delta Sorted_{i,k}$,    %select the most saturated machine
)
$\forall$ j $\in$ JOB (
  min $\sum_{k \in \Delta order_{i,j}} \frac{\#\Delta Sorted_{i,k}}{\# \quad \Delta order_{i,j}}$   % select the job with least slack
  max # $\Delta Task_{i,j}$                      % select the task with least slack
)
$order_{i,j}$ = ? ($\Rightarrow$),               % query on execution order
```

$\forall$ i $\in$ MACHINE
$\forall$ j $\in$ JOB
release$_{i,j}$ = ?, % query on release
min max$_{i \in MACHINE}$ t2$_{i,n}$ % to minimize the maximum of due dates

Here $Task_{i,j}$ represents the time interval of job j on machine i; though we do not know which one, $Sorted_{i,j}$ represents the time interval of the job scheduled j-th on machine i. Based on the set sorting model, the permutation function $order_{i,j}$ "sorts" *Task* to *Sorted*. Using this model, without any special techniques as done in [15], mt10 is solved completely in less than 2 minutes.

4.3 Minimizing the Cost of a Heat Exchanger

The Heat Exchanger problem [7,13] is a non-linear constrained optimization problem over the reals. It involves minimizing the cost function of a heat exchanger. Most of the constraints are linear, but the presence of some non-linear ones plus the very non-linear objective function make the problem difficult. NCL description is:

Ti1 $\in$ [150.0, 240.0],
To1 $\in$ [250.0, 490.0],
Ti2 $\in$ [150.0, 190.0],
To2 $\in$ [210.0, 340.0],
FE1 $\in$ [2.941, 10.0],
FE2 $\in$ [3.158, 10.0],
Fi1 $\geq$ 0.0,
Fi2 $\geq$ 0.0,
FB12 $\geq$ 0.0,
FB21 $\geq$ 0.0,
Fo1 $\geq$ 0.0,
Fo2 $\geq$ 0.0,
T11 = 500.0 - To1,
T12 = 250.0 - Ti1,
T21 = 350.0 - To2,
T22 = 200.0 - Ti2,
Fi1 + Fi2 = 10.0,
Fo2 + FB12 = FE2,
Fo1 + FB21 = FE1,
Fi1 + FB12 = FE1,
Fi2 + FB21 = FE2, % linear constraints above
% non-linear constraints below
FE2 $\times$ (To2 - Ti2) = 600.0,
FE1 $\times$ (To1 - Ti1) = 1000.0,
150.0 $\times$ Fi1 + To2 $\times$ FB12 - Ti1 $\times$ FE1 = 0.0,
150.0 $\times$ Fi2 + To1 $\times$ FB21 - Ti2 $\times$ FE2 = 0.0,
min 1300 $\times$ exp(0.6 $\times$ log(20000 $\times$ 6 / (4 $\times$ $\sqrt{T11 \times T12}$+ (T11+T12)))) +

1300 $\times$ exp(0.6 $\times$ log(12000 $\times$ 6 / (4 $\times$ $\sqrt{T21 \times T22}$ + (T21+T22)))) ,
$\forall$ 'i' $\in$ { 'To1', 'To2', 'Ti1', 'Ti2', 'Fi1', 'Fi2', 'Fo1', 'Fo2' } $\rightarrow$ (
max #i ,
)
i = ?.

The solution strategy for this problem is to instantiate among *To1, To2, Ti1, Ti2, Fi1, Fi2, Fo1, Fo2* a real variable whose domain range #*i* is the greatest (search rule through quantification on references to real variables). NCL solves the problem in roughly 2 minutes (with proof of optimality). The optimal solution <56825.76171875..56825.76953125> is contained in the following intervals:

To1 = <309.9998779296875..309.9999084472656>

To2 = <210.0000000000000..210.0000152587891>

Ti1 = <209.9998779296875..209.9998931884766>

Ti2 = <150.0000000000000..150.0000152587891>

Assuming *To1, To2, Ti1, Ti2* to be integers, the problem becomes much easier. NCL solves it in a few seconds: *To1 = 310, To2 = 210, Ti1 = 210, Ti2 = 150* and the optimal solution is proved to be within <56825.79296875..56825.859375>.

5 Conclusion

From AI, pattern recognition and semantic analysis techniques are incorporated in NCL for understanding mathematical logic expressions. From OR, cut and search techniques are incorporated for solving constraint satisfaction problems. From LP, logical representation and reasoning methods are incorporated for abstract modeling at a higher level. NCL's techniques on implicit typing, context-based parsing and reasoning, cooperative cutting and search algorithms over mixed domain of reals, integers, Booleans, references and sets are NCL's unique features. In short, NCL's Mixed Set Programming and Natural Modeling are novel.

References

[1] N. Beldiceanu & E. Contejean. Introducing global constraints in CHIP. Mathl. Comput. Modeling, 20(12): 97-123, 1994.

[2] C.J. Bouwkamp & A.J.W. Duijvestijn. Catalogue of simple perfect squared squares of orders 21 through 25. Eindhoven University of Technology, Technical Report 92 WSK 03, The Netherlands, November 1992.

[3] Frédéric Benhamou, William J. Older: Applying Interval Arithmetic to Real, Integer, and Boolean Constraints. J. Log. Program. 32(1): 1-24, 1997.

[4] A. Colmerauer. An introduction to Prolog III, Communications of the ACM,33(7): 69-90, 1990.

[5] G.B. Dantzig, Origins of the simplex method, in S G Nash (ed.), A history of scientific computing (Reading, MA, 1990), 141-151.

[6] M. Dincbas, P.Van Hentenryck, H. Simonis, A. Aggoun, T. Graf & F. Berthier. The constraint logic language CHIP. in: Proc. of the 3rd Annual ACM Symposium on Theory of Computing,151-158, 1988.

[7] C.A. Floudas & P.M. Pardalos. A collection of test problems for constrained global optimization problems. Springer-Verlag.1973.

[8] R. Fourer,D.M.Gay,B.W. Kernighan, AMPL: A Modeling Language for Mathematical Programming},The Scientific Press, San Fransisco, CA, 1993.

[9] K.L. Hoffman & M. Padberg, Solving airline crew scheduling problems by branch-and-cut, Management Science, vol. 39, no. 6, pp657-682, June 1993.

[10] J.Jaffar & J.-L.Lassez. Constraint Logic Programming, in: Proc. of the 14th ACM POPL Conference, 111-119, Munich, 1987.

[11] T. Müller. Solving set partitioning problems with constraint programming. In PAPPACT98, pages 313-332. The Practical Application Company Ltd, 1998.

[12] J.F. Muth & G.L. Thompson. Industrial Scheduling. Prentice Hall, 1963.

[13] W.J. Older. Using interval arithmetic for non-linear constrained optimization. Bell-Northern Research Technical Report. 1993.

[14] G. Smolka. The Oz programming model, in: Jan van Leeuwen (ed.), Computer Science Today, 324-343, Berlin, 1995.

[15] J. Zhou. A permutation-based approach for solving the Job-shop problem. Constraints 2(2): 185-213, (1997).

[16] J. Zhou. Introduction to the constraint language NCL. JLP 45(1-3): 71-103 (2000).

[17] J. Zhou. The Manual of POEM – A Development Environment for The Natural Constraint Language. Version 2.8. ENGINEST, May 2008.