Introducing Global Constraints in CHIP*

Nicolas Beldiceanu
COSYTEC,
Parc Club Orsay-Université,
4, rue Jean-Rostand,
91893 ORSAY CEDEX
FRANCE

Evelyne Contejean
LRI, CNRS URA 410,
Université Paris Sud,
Bâtiment 490,
91405 ORSAY CEDEX
FRANCE

Abstract

The purpose of this paper is to show how the introduction of new primitive constraints (e.g. *among, diffn, cycle*) over finite domains in the constraint logic programming system CHIP result in finding very rapidly good solutions for a large class of difficult sequencing, scheduling, geometrical placement and vehicle routing problems. The *among* constraint allows to specify sequencing constraints in a very concise way. For the first time, the *diffn* constraint allows to express and to solve directly multidimensional placement problems where one has to consider non overlapping constraints between n-dimensional objects (e.g. rectangles, parallelepipeds). The *cycle* constraint makes possible to specify a wide range of graph partitioning problems that could not yet be expressed by using current constraint logic programming languages. One of the main advantage of all these new primitives is to take into account more globally a set of elementary constraints. Finally, we point out that all the previous primitive constraints enhance the power of the CHIP system significantly, allowing to solve real life problems that were not within reach of constraint technology before.

1 Introduction

CHIP (Constraint Handling in Prolog) [9, 1] is a constraint logic programming language designed to tackle real world "constrained search" problems with a short development time and a good efficiency. Constraint logic programming [16, 17] combines logic, used to specify a set of possibilities explored using a very simple built-in search method, with constraints, used to minimise the size of the search by eliminating impossible alternatives in advance. Unlike conventional programming languages where one uses specific algorithms and data structures adapted to the problem, all the current constraint logic programming languages offer a very restricted set of basic primitive constraints and control structures for expressing new constraints. However, our practical experience has shown that this can sometimes lead to inefficiencies because:

- very often there is a large gap between the constraints of the original problem and the constraints available in the language,
- in many problems most of the constraints are conditional. This means that they can not be stated initially, but they depend of some previous choices that occur during the enumeration procedure,
- some constraints of the application express a kind of global conditions which can hardly be expressed with elementary constraints or control structures of the language,

*This work is supported by the French Ministère de la Recherche et de l'Espace (projet COLINE décision d'aide n° 92 S 0600), the Esprit CCL WG, and ANVAR.

- sometimes, basic constraint propagation schemes like arc or path consistency [6] are not powerful enough to take into consideration the interaction between elementary constraints. This means that, in terms of deduction and execution time, general propagation techniques can be very inefficient when compared to specific algorithms that use appropriate data structures.

In order to partially overcome the previous lack of expression and deductive power of conventional constraint logic programming languages, we use the following processes. Our purpose is to identify suitable abstractions that enable, at the same time, a declarative statement of the problem and an operational behaviour matching the best available pruning techniques. This paper describes these new abstractions: the *among, diffn* and *cycle* global constraints. It points out that these global constraints are major abstraction common to a large class of sequencing, scheduling [3], geometrical placement [12, 11] and vehicle routing problems [7].

The paper is structured as follows: in section 2 we give a brief overview of the CHIP system. In section 3 we present the *among* constraint and its declarative semantics. In section 4 we show how to use the *among* constraint in order to solve the car sequencing problem [10]. In section 5 we present the *diffn* constraint, its declarative semantics and its extensions. In section 6 we describe a difficult three-dimensional packing problem given in [5] page 50, where one has to pack 17 parallelepipeds in a cube in such a way that no hole occurs and that none of them overlaps each other. In section 7 we describe how to use the *diffn* constraint to solve an assignment and scheduling problem where a specific constraint related to a pipelining process occurs (see [18], page 260). In section 8 we present the *cycle* constraint and its extensions. Finally, in the last section, we describe a vehicle routing problem for which we use the *cumulative* [2] and *cycle* constraints together.

2 Brief overview of CHIP

CHIP is a constraint logic programming language combining the declarative aspect of Prolog with the efficiency of constraint solving techniques. It extends conventional Prolog-like logic languages by introducing three new computation domains: finite domains, booleans and rationals. For each of them CHIP uses specialised constraint solving techniques: consistency checking techniques for finite domains, equation solving in Boolean algebra for booleans and a symbolic simplex-like algorithm for rationals. CHIP has been successfully applied to a large number of industrial problems especially in the area of planning, manufacturing, logistics, circuit design and financial planning [8]. Originally designed at ECRC, CHIP is further developed and marketed by COSYTEC.

Constraint logic programming is based on a combination of Artificial Intelligence, Operations Research and Mathematical methods. One of the basic extensions of CHIP is the introduction of finite domains. A constraint in finite domains is a relation between a set of domain variables. A domain variable is a variable that ranges over a finite set of natural numbers. Among constraints over finite domains, one can find usual arithmetic constraints, symbolic constraints and higher-order optimisation predicate that implements a kind of branch and bound search.

In order to tackle more efficiently scheduling and placement problems, a new symbolic constraint was recently introduced in CHIP: the *cumulative* constraint [2]. Because this constraint will be used in the rest of this paper in conjunction with the *diffn* and *cycle* constraint, let us recall shortly its definition

$$cumulative([O_1, \ldots, O_m], [D_1, \ldots, D_m], [R_1, \ldots, R_m], L),$$

where $[O_1, \ldots, O_m], [D_1, \ldots, D_m]$ and $[R_1, \ldots, R_m]$ are non-empty lists of domain variables that have the same length m, and where L is a natural number. The constraint *cumulative* holds if the following condition is true:

$$\forall i \in I\!N \sum_{j | O_j \leq i \leq O_j + D_j - 1} R_j \leq L$$

From an interpretation point of view the *cumulative* constraint matches directly the single resource scheduling problem, where $O_1, \ldots, O_m$ correspond to the start of the tasks, $D_1, \ldots, D_m$ to the duration of the tasks, and $R_1, \ldots, R_m$ to the amount of resource used by each task. The natural number L is the total amount of available resource that must be shared at any instant by the different tasks. The *cumulative* constraint states that, at any instant i of the schedule, the summation of the amount of resource of the tasks that overlap i, does not exceed the upper limit L.

As an introductory example to the CHIP language, we present how a very small scheduling problem can be expressed in CHIP. We consider seven tasks where each task is characterised by a duration and an amount of used resource (see Table 1). The aim is to find a schedule that minimises the general end while not exceeding the capacity 13 of the resource.

task	t1	t2	t3	t4	t5	t6	t7
duration	16	6	13	7	5	18	4
resource	2	9	3	7	10	1	11

Table 1: Data for the scheduling problem

The following program outlines a CHIP program over finite domains solving the previous example.

```
top(LO,End) :-                                          % line  1
    LO = [O1,O2,O3,O4,O5,O6,O7],                        % line  2
    LD = [16, 6,13, 7, 5,18, 4],                        % line  3
    LR = [ 2, 9, 3, 7,10, 1,11],                        % line  4
    LE = [E1,E2,E3,E4,E5,E6,E7],                        % line  5
    End :: 1..30,                                       % line  6
    LO :: 1..30,                                        % line  7
    LE :: 1..30,                                        % line  8
    O1 + 16 #= E1,                                      % line  9
    O2 +  6 #= E2,                                      % line 10
    O3 + 13 #= E3,                                      % line 11
    O4 +  7 #= E4,                                      % line 12
    O5 +  5 #= E5,                                      % line 13
    O6 + 18 #= E6,                                      % line 14
    O7 +  4 #= E7,                                      % line 15
    maximum(End,LE),                                    % line 16
    cumulative(LO,LD,LR,13),                            % line 17
    min_max(label(LO),End).                             % line 18

label([]).                                              % line 19
label([X|Y]) :-                                         % line 20
    indomain(X),                                        % line 21
    label(Y).                                           % line 22
```

The predicate `top/2` (see line 1) corresponds to the main predicate to compute the schedule. The arguments of `top/2` are a list of variables that represents the starting date of each task and a domain variable that corresponds to the end of the schedule. Lines 2 to 5 make explicit the origin, duration, amount of resource and end of the different tasks of the problem. As described by the domain declarations (see lines 6-8), the domain of the general end ranges from 1 to 30 and the domain of the origin and end of the tasks ranges from 1 to 30. The link between the origin and the end of a task i (see lines 9-15) is expressed as

$$O_i + D_i \texttt{\#=} E_i,$$

where `#=` is the equality constraint symbol over finite domain, where O_i, D_i and E_i are respectively the origin, the duration and the end of task i. The *maximum* constraint (see line 16)

We can see all the intermediate results of respective cost 28, 27 and the optimal solution `[1,17,10,10,5,5,1]` of cost 23 (see Figure 1). After this short introduction of the CHIP language, we will now present in the next sections the *among, diffn* and *cycle* constraints.

3 Among constraint

The *among* constraint was introduced in CHIP in order to specify the way values can be assigned to variables. The *among* constraint can be seen as an extension of the *atleast* and *atmost* constraints (at least, at most N variables take value V). One of the most interesting feature of the *among* constraint is that it allows to express directly a set of "overlapping" *atleast*, *atmost* constraints. This constraint occurs in many time table problems where one *atleast*, *atmost* constraint has to be verified for each period of n consecutive time units. There exists five different variants of the *among* constraint. We now give the declarative semantics of the first variant

$$among(N, [X_1, \ldots, X_s], [C_1, \ldots, C_s], [V_1, \ldots, V_m]),$$

where N is a domain variable, $[X_1, \ldots, X_s]$ is a list of domain variables, $[C_1, \ldots, C_s]$ and $[V_1, \ldots, V_m]$ are lists of natural numbers. The constraint holds if the following conditions are both true:

(1) $\forall i \in [1, m-1] : V_i < V_{i+1}$,

(2) exactly N terms among $X_1+C_1, \ldots, X_s+C_s$ take their value in the list of values $[V_1, \ldots, V_m]$.

One of the main advantages of the first variant of the *among* constraint is the fact that the first parameter is a domain variable. It can be used when it is required to know the exact number of times that a set of values is taken by a set of variables or when this parameter has to be used

in some other constraints. The second variant of the *among* constraint is used when it is required to specify a lower and upper bound for the number of times that a set of values is taken by a set of variables. More precisely, we give the declarative semantics of the second variant

$$among([Low, Up], [X_1, \ldots, X_s], [C_1, \ldots, C_s], [V_1, \ldots, V_m]),$$

where Low and Up are natural numbers, $[X_1, \ldots, X_s]$ is a list of domain variables, $[C_1, \ldots, C_s]$ and $[V_1, \ldots, V_m]$ are lists of natural numbers. The constraint holds if the following conditions are both true:

(1) $\forall i \in [1, m-1] : V_i < V_{i+1}$,

(2) at least Low and at most Up terms among $X_1 + C_1, \ldots, X_s + C_s$ take their value in the list of values $[V_1, \ldots, V_m]$.

The last three variants allow to state directly a set of "overlapping" *among* constraints. From a semantic point of view, these variants can be expressed directly by using the second variant of the *among* constraint. However from an efficiency point of view in terms of memory utilisation and pruning, these last three variants are much more powerful. We now give the declarative semantics of the third variant

$$among([Low, Up, Seq], [X_1, \ldots, X_s], [C_1, \ldots, C_s], [V_1, \ldots, V_m]),$$

where Low, Up and Seq are natural numbers, $[X_1, \ldots, X_s]$ is a list of domain variables, $[C_1, \ldots, C_s]$ and $[V_1, \ldots, V_m]$ are lists of natural numbers. The constraint holds if the following conditions are all true:

(1) $\forall i \in [1, m-1] : V_i < V_{i+1}$,

(2) $0 < Seq \leq s$,

(3) $\forall i \in [1, s - Seq + 1]$, let $j = i + Seq - 1$.
at least Low terms and at most Up terms among the list of Seq consecutive terms $X_i + C_i, \ldots, X_j + C_j$ take their value in the list of values $[V_1, \ldots, V_m]$.

The next variant combines the second and the third variant in order to improve the propagation

$$among([Low, Up, Seq, Least, Most], [X_1, \ldots, X_s], [C_1, \ldots, C_s], [V_1, \ldots, V_m]),$$

where Low, Up, Seq, $Least$, $Most$ are natural numbers, $[X_1, \ldots, X_s]$ is a list of domain variables, $[C_1, \ldots, C_s]$ and $[V_1, \ldots, V_m]$ are lists of natural numbers. The constraint holds if the following conditions are all true:

(1) $\forall i \in [1, m-1] : V_i < V_{i+1}$,

(2) $0 < Seq \leq s$,

(3) $\forall i \in [1, s - Seq + 1]$, let $j = i + Seq - 1$
at least Low terms and at most Up terms among the list of Seq consecutive terms $X_i + C_i, \ldots, X_j + C_j$ take their value in the list of values $[V_1, \ldots, V_m]$,

(4) at least $Least$ and at most $Most$ terms among $X_1 + C_1, \ldots, X_s + C_s$ take their value in the list of values $[V_1, \ldots, V_m]$.

Finally the last variant allows to state directly a set of "included" *among* constraints. We now give the declarative semantics of the last variant

$$among([Low, Up, Seq, LowInc, UpInc, SeqInc], [X_1, \ldots, X_s], [C_1, \ldots, C_s], [V_1, \ldots, V_m]),$$

where Low, Up, Seq, $LowInc$, $UpInc$ and $SeqInc$ are natural numbers, $[X_1, \ldots, X_s]$ is a list of domain variables, $[C_1, \ldots, C_s]$ and $[V_1, \ldots, V_m]$ are lists of natural numbers. The constraint holds if the following conditions are all true:

(1) $\forall i \in [1, m-1] : V_i < V_{i+1}$,

(2) $0 < Seq \leq s$,

(3) $0 < SeqInc$, $s - Seq \equiv 0[SeqInc]$,

(4) $\forall i \in [1, ((s - Seq)/SeqInc) + 1]$: let $j = Seq + (i-1) * SeqInc$
at least $Low + (i-1) * LowInc$ and at most $Up + (i-1) * UpInc$ terms among $X_1 + C_1, \ldots, X_j + C_j$ take their value in the list of values $[V_1, \ldots, V_m]$.

In the next section, we show how to combine the different variants of the *among* constraint in order to solve the car sequencing problem [10].

4 The car sequencing problem revisited

Problem Purpose

The purpose of this example is to show how to use the *among* constraint in order to model, in a very efficient way in terms of the number of variables and constraints, the car sequencing problem presented in [10].

Problem Statement

The car sequencing problem occurs in assembly line scheduling within car manufacturing. The problem consists of sequencing a set of cars that require a set of options on an assembly line. For each possible option, the line has a capacity constraint which dictates how frequently it can occur on the line. We will consider the example given in [10] with 10 cars and 5 options. Table 2 gives, for each option the capacity of the assembly line, and the fact that a given car uses the option or not. The capacity of an option corresponds to an integer ratio n/m which tells that at most n cars among m consecutive cars on the assembly line could take this option.

	capacity	car 1	car 2	car 3	car 4	car 5	car 6	car 7	car 8	car 9	car 10
option 1	1/2	1	0	0	0	0	0	1	1	1	1
option 2	2/3	0	0	1	1	1	1	0	0	1	1
option 3	1/3	1	0	0	0	0	0	1	1	0	0
option 4	2/5	1	1	0	0	1	1	0	0	0	0
option 5	1/5	0	0	1	1	0	0	0	0	0	0

Table 2: Data for the car sequencing problem

Problem Solution

We describe how to express the basic constraints of the problem, and how to improve the behaviour of the program by adding redundant constraints which allow to detect inconsistency earlier.

Problem Representation

The cars are clustered in 6 classes $\{1\}$, $\{2\}$, $\{3,4\}$, $\{5,6\}$, $\{7,8\}$, $\{9,10\}$, each class containing all the cars requiring the same set of options. For each position of the assembly line we create a domain variable that corresponds to the class of cars handled at this position. All the previous variables are grouped in the list of variables `LV`.

Constraint Statement

We express the fact that for each class we have to produce a fixed number of cars by giving for each value (*i.e.* class) the number of times it should occur in the variables (*i.e.* slot of the assembly line). This is directly expressed as one *among* constraint for each class. For example the constraint `among(2,LV,L0,[6])` states that the value 6 (*i.e.* car of class 6) should occur exactly 2 times in the list of variables `LV`. `L0` corresponds to a list of 0's of same length as list `LV`. We look now how to express the capacity constraints associated to each option. We find out for each option which classes effectively use this option and how many cars require this option. For example the first option is required by 5 cars of class 1, 5 and 6. The capacity constraint "1 car out of 2" is directly expressed by `among([0,1,2,5,5],LV,L0,[1,5,6])` which enforce the following conditions:

- at least 0 and at most 1 out of 2 consecutive variables of the list of variables `LV` take their value in the set $\{1, 5, 6\}$,
- at least 5 and at most 5 of the list of variables `LV` take their value in the set $\{1, 5, 6\}$.

Finally, it is possible to increase the performance of the program by adding redundant constraints derived from the total number of cars that use a given option and from the capacity constraint associated with this option. More precisely, if we have to sequence N cars, M of them requiring a given option O for which we have the capacity constraint A/B, then we know that the slots from 1 to C must contain at least $M - A.((N-C)divB) - ((N-C)modB)$ cars having option O. This redundant constraint is expressed by one *among* constraint for each class. We now give the corresponding CHIP program that states all the previous *among* constraints associated to the problem.

```
top(L) :-                                                % line  1
    L0 = [ 0, 0, 0, 0, 0, 0, 0, 0, 0,  0],               % line  2
    LV = [S1,S2,S3,S4,S5,S6,S7,S8,S9,S10],               % line  3
    LV :: 1..6,                                          % line  4
    among(1,LV,L0,[1]),                                  % line  5
    among(1,LV,L0,[2]),                                  % line  6
    among(2,LV,L0,[3]),                                  % line  7
    among(2,LV,L0,[4]),                                  % line  8
    among(2,LV,L0,[5]),                                  % line  9
    among(2,LV,L0,[6]),                                  % line 10
    among([0,1,2,5,5],LV,L0,[1,5,6]),                    % line 11
    among([0,2,3,6,6],LV,L0,[3,4,6]),                    % line 12
    among([0,1,3,3,3],LV,L0,[1,5  ]),                    % line 13
    among([0,2,5,4,4],LV,L0,[1,2,4]),                    % line 14
    among([0,1,5,2,2],LV,L0,[3    ]),                    % line 15
    among([1,2,2,1,2,2],LV,L0,[1,5,6]),                  % line 16
    among([2,4,4,2,3,3],LV,L0,[3,4,6]),                  % line 17
    among([1,4,4,1,3,3],LV,L0,[1,5  ]),                  % line 18
    among([2,5,5,2,5,5],LV,L0,[1,2,4]),                  % line 19
    among([1,5,5,1,5,5],LV,L0,[3    ]),                  % line 20
    labelling(LV).                                       % line 21
```

Lines 3 to 4 are used to create the slot variables, lines 5 to 10 specify the number of cars to produce, lines 11 to 15 state the capacity constraint for each option and lines 16 to 20 express the redundant constraints. Finally line 21 calls an enumeration procedure that tries to assign values to the list of slot variables. In Figure 2 we give explicitly the correspondence between the *among* constraint, used at each line of the previous CHIP program, and elementary *among* constraint. Each arrow, ranging from variable S_i to variable S_j with constants Low, Up, $Values$, corresponds to the constraint $among([Low, Up], [S_i, \ldots, S_j], [0, \ldots, 0], Values)$. For example the constraint `among([2,4,4,2,3,3],LV,L0,[3,4,6])` used at line 17 corresponds to the conjunction of the three following elementary *among* constraints:

Figure 2: Elementary *among* constraints associated to the car sequencing problem.

Computation results

In order to test the efficiency of this approach we have performed the following experiment. As in the original paper [10], we assume that the assembly line supports five different options with capacity constraints 1 out of 2, 2 out of 3, 1 out of 3, 2 out of 5 and 1 out of 5. We ask for an overall percentage of utilisation of the resource of 90% and test for different number of cars. The Table 3 gives the time needed for finding a first solution on a SUN/SPARC station IPC (24MB).

Number of cars	100	200
CPU-Time in milli-seconds	590	1100

Table 3: Results for the car sequencing problem

5 Diffn constraint

The *diffn* constraint was introduced in CHIP in order to handle multidimensional placement problems [12] that occur in scheduling, cutting or geometrical placement problems. The intuitive idea is to extend the *alldifferent* constraint which works on a set of domain variables to a non overlapping constraint between a set of objects defined in an n-dimensional space. The basic *diffn* constraint takes as arguments a list of n-dimensional rectangles that are defined in the following

way. We call an n-dimensional rectangle a tuple of domain variables $(O_1, \ldots, O_n, L_1, \ldots, L_n)$; O_i and L_i are respectively called the origin and the length of the previous n-dimensional rectangle in i^{th} dimension. Other parameters of the *diffn* constraint will be introduced later. We now give the declarative semantics of the basic *diffn/1* constraint

$$\mathit{diffn}([[O_{11}, \ldots, O_{1n}, L_{11}, \ldots, L_{1n}], \ldots, [O_{m1}, \ldots, O_{mn}, L_{m1}, \ldots, L_{mn}]])$$

The constraint *diffn/1* holds if we have an empty list or if the following conditions are all true:

(1) $\forall i \in [1, m], \forall j \in [1, n] : O_{ij}$ is a domain variable or a natural number,

(2) $\forall i \in [1, m], \forall j \in [1, n] : L_{ij}$ is a domain variable or a natural number,

(3) $\forall i \in [1, m], \forall j \in [1, n] : L_{ij} \neq 0$,

(4) $\forall i \in [1, m], \forall j \in [1, m]\ j \neq i, \exists k \in [1, n] \mid O_{ik} \geq O_{jk} + L_{jk} \vee O_{jk} \geq O_{ik} + L_{ik}$

From an interpretation point of view, the last condition corresponds to the fact that for each pair $i, j (i \neq j)$ of n-dimensional rectangles, there exists at least one dimension k where i is after j or j is after i. In Figure 3, we sketch five different cases of the *diffn* constraint. The first case (A) corresponds to a non overlapping constraint among three segments. The second and third cases (B, C) correspond to a non overlapping constraint between rectangles [21] where (B) is a special case where the lengths of all the rectangles in the second dimension are equal to 1; it can be interpreted as a machine assignment problem where each rectangle corresponds to a task that has to be placed in time and assigned to a specific machine [4]. The forth case (D) corresponds to a non overlapping constraint between parallelepipeds [19]. The fifth case can be interpreted as a non overlapping constraint between parallelepipeds that are assigned to the same box [14]; the first dimension corresponds to the number of the box, while the three next dimensions give the position of a parallelepiped inside the box.

Other constraints occurring in geometrical placement problems concern the "volume" of the objects that are involved in a *diffn* constraint. In this paragraph we extend the previous *diffn/1* in order to deal with such kinds of constraint. For this purpose we introduce two additional parameters

$$\begin{aligned}&\mathit{diffn}([[O_{11}, \ldots, O_{1n}, L_{11}, \ldots, L_{1n}], \ldots, [O_{m1}, \ldots, O_{mn}, L_{m1}, \ldots, L_{mn}]],\\ &\quad [Min_1, \ldots, Min_m], [Max_1, \ldots, Max_m]),\end{aligned}$$

where $[Min_1, \ldots, Min_m]$ and $[Max_1, \ldots, Max_m]$ are non-empty list of natural numbers which correspond respectively to the minimum and maximum volume attached to each object. The constraint *diffn/3* holds if the previous conditions hold and if the following condition is also true:

(5) $\forall i \in [1, m]\ :\ Min_i \leq L_{i1} \times \ldots \times L_{in} \leq Max_i$

These parameters can also be used in conjunction with the *among/4* constraint in order to specify that we have an object for which the orientation is not yet fixed. For example, if we have to place two parallelepipeds of respective size (5,6,17) and (7,7,10) in such a way that they do not overlap, we would use the following CHIP program.

```
top :-
     P1 = [0x1,0y1,0z1,Lx1,Ly1,Lz1],            % line  1
     P2 = [0x2,0y2,0z2,Lx2,Ly2,Lz2],            % line  2
     P1 :: 0..100,                              % line  3
     P2 :: 0..100,                              % line  4
     among(1,[Lx1,Ly1,Lz1],[0,0,0],[ 5]),       % line  5
     among(1,[Lx1,Ly1,Lz1],[0,0,0],[ 6]),       % line  6
     among(1,[Lx1,Ly1,Lz1],[0,0,0],[17]),       % line  7
     among(2,[Lx2,Ly2,Lz2],[0,0,0],[ 7]),       % line  8
     among(1,[Lx2,Ly2,Lz2],[0,0,0],[10]),       % line  9
     diffn([P1,P2],[510,490],[510,490]).        % line 10
```

diffn([[1,1], [3,2], [5,3]])

diffn([[1,2,1,1], [3,1,2,1], [4,3,3,1]])

diffn([[1,2,2,2], [3,1,2,1], [4,2,3,3]])

diffn([[1,1,1,1,2,4], [2,1,1,2,2,3], [4,2,1,2,4,1]])

diffn([[1,1,1,2,1,1,1,1], [2,1,1,1,1,1,1,1], [2,2,2,1,1,1,1,1]])

Figure 3: Five examples of use of the diffn constraint

diffn([[1,3,2,2], [3,1,2,1], [6,1,3,3]], [6,2,9], [6,2,9],[9,5])

Figure 4: Example of utilisation of the end variables

The end parameter can be used for placement problems in order to give explicitly the limits of the placement space. This is especially useful when the lengths of the objects are not initially fixed: in this case, considering only the origin and length of an object, would give an over estimation of these limits. The end parameter can also be used in scheduling problems where it is required to assign tasks in time and on machines, while minimising the general end of the schedule or the number of machines used.

The next parameter of the *diffn* constraint is used in order to state distance constraints between two given objects

$$\begin{aligned} &\mathit{diffn}([[O_{11},\ldots,O_{1n},L_{11},\ldots,L_{1n}],\ldots,[O_{m1},\ldots,O_{mn},L_{m1},\ldots,L_{mn}]],\\ &\quad [Min_1,\ldots,Min_m],[Max_1,\ldots,Max_m],\\ &\quad [End_1,\ldots,End_n],\\ &\quad [[I_{11},I_{21},D_1],\ldots,[I_{1d},I_{2d},D_d]]), \end{aligned}$$

where $[[I_{11},I_{21},D_1],\ldots,[I_{1d},I_{2d},D_d]]$ is a list of distance constraints. The constraint *diffn/5* holds if the previous conditions hold and if the list of distance constraints is empty or if the following additional conditions are true.

(7) $\forall i \in [1,d] : 1 \leq I_{1d} \leq m,$

(8) $\forall i \in [1,d] : 1 \leq I_{2d} \leq m,$

(9) $\forall i \in [1,d] :$ D_i is a domain variable or a natural number,

(10) $\forall i \in [1,d] :$ let $a = I_{1i}, b = I_{2i}$ then we have

$$D_i = \sum_{j=1}^{n} \mathit{maximum}(0, O_{aj} - O_{bj} - L_{bj}, O_{bj} - O_{aj} - L_{aj})$$

diffn([[1,3,2,2], [3,1,2,1], [6,1,3,1]], [4,2,3], [4,2,3], [9,5], [[1,2,1],[1,3,4],[2,3,1]])

Figure 5: Example of utilisation of the distance constraint

Finally, the last parameter of the *diffn* constraint is used in order to restrict the utilisation of a region by the objects

$$\begin{aligned}
&\mathit{diffn}([[O_{11},\ldots,O_{1n},L_{11},\ldots,L_{1n}],\ldots,[O_{m1},\ldots,O_{mn},L_{m1},\ldots,L_{mn}]],\\
&\quad [Min_1,\ldots,Min_m],[Max_1,\ldots,Max_m],\\
&\quad [End_1,\ldots,End_n],\\
&\quad [[I_{11},I_{21},D_1],\ldots,[I_{1d},I_{2d},D_d]],\\
&\quad [[[R_{11},\ldots,R_{1n},S_{11},\ldots,S_{1n}],T_1,U_1],\ldots,[[R_{p1},\ldots,R_{pn},S_{p1},\ldots,S_{pn}],T_p,U_p]]),
\end{aligned}$$

where $[[[R_{11},\ldots,R_{1n},S_{11},\ldots,S_{1n}],T_1,U_1],\ldots,[[R_{p1},\ldots,R_{pn},S_{p1},\ldots,S_{pn}],T_p,U_p]]$ is a list of region constraints. The constraint *diffn/6* holds if the list of region constraints is empty or if the following additional conditions are all true.

(11) $\forall i \in [1,p], \forall j \in [1,n]$: R_{ij} is a natural number,

(12) $\forall i \in [1,p], \forall j \in [1,n]$: S_{ij} is a natural number different from 0,

(13) $\forall i \in [1,p]$: $1 \leq T_i \leq n$,

(14) $\forall i \in [1,p]$: let E_i be the set of integers e such that :
$\forall j \in [1,n] : O_{ej} + L_{ej} > R_{ij} \quad \wedge \quad R_{ij} + S_{ij} > O_{ej}$
if E_i is empty then $U_i = 0$
else let $soon_i = minimum_{e \in E_i}(maximum(R_{i,T_i}, O_{e,T_i}))$
let $late_i = maximum_{e \in E_i}(minimum(R_{i,T_i} + S_{i,T_i} - 1, O_{e,T_i} + L_{e,T_i} - 1))$
then $U_i = late_i - soon_i + 1$

Each region i is described by it origins $R_{i1},\ldots,R_{in}$ and it sizes $S_{i1},\ldots,S_{in}$ in each dimension. The parameter T_i gives the dimension in which we want to get the utilisation U_i of the region i. U_i corresponds to the difference between the last and the first use of the region in a given dimension. The Figure 6 gives an example of utilisation of region constraints.

In the next two sections we will show how to use the *diffn* constraint in order to solve a three-dimensional packing problem and an assignment scheduling problem.

6 Solving a three-dimensional packing problem

Problem Purpose

The purpose of this example is to show how to use the *diffn* constraint in order to solve a three-dimensional packing problem. This problem also illustrates how the *cumulative* constraint can be used to express redundant constraints and how to build a constructive placement enumeration procedure that takes into account the fact that all placement space has to be completely filled.

```
diffn(   [ [1,1,2,2], [4,3,2,1], [6,5,3,1] ], [4,2,3], [4,2,3], [9,6],
         [[1,2,1]],[ [[2,2,5,3],1,4], [[2,2,5,3],2,2], [[8,1,1,3],1,0]])
```

Figure 6: Example of utilisation of the region constraint

Problem Statement

The problem was presented by J. H. Conway [5], a mathematician from the university of Cambridge. It consists to find out how to pack 17 parallelepipeds of given sizes into a $5 \times 5 \times 5$ cube, in such a way that none of them overlaps each other. The fact that the summation of the volumes of the different parallelepipeds is equal to the volume of the cube makes the problem quite hard. Table 4 gives the size of the different parallelepipeds of the problem.

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17
1	1	1	1	1	1	2	2	2	2	2	2	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1
4	4	4	4	4	4	3	3	3	3	3	3	1	1	1	1	1

Table 4: Size of the parallelepipeds

Problem Representation and Constraint Statement

Let OX_i, OY_i, OZ_i $(i = 1 \ldots 17)$ be the coordinate of the origin of parallelepiped i on the x, y and z-axes, let DX_i, DY_i, DZ_i $(i = 1 \ldots 17)$ be the size of the parallelepiped i on the x, y and z-axes, and let SX_i, SY_i, SZ_i $(i = 1 \ldots 17)$ the surfaces of the projection of parallelepiped i on the planes yz, xz and xy. From the packing constraint, we can derive the following three necessary conditions corresponding to *cumulative* conditions on the planes yz, xz and xy :

$$\forall i \in [1,5]: \quad \sum_{j|OX_j \leq i \leq OX_j + DX_j - 1} SX_j \leq 25 \qquad \sum_{j|OY_j \leq i \leq OY_j + DY_j - 1} SY_j \leq 25 \qquad \sum_{j|OZ_j \leq i \leq OZ_j + DZ_j - 1} SZ_j \leq 25$$

We now give the corresponding CHIP program. It creates the origin, duration and complementary surface variables associated to the parallelepipeds in the different dimensions and sets up one *cumulative* constraint for each dimension and one *diffn* constraint.

```
top(LP) :-

  % ORIGIN IN THE <> DIMENSIONS          % COMPLEMENTARY SURFACE
  OX = [OX01,..,OX17],                   SX = [SX01,..,SX12,1,1,1,1,1],
  OY = [OY01,..,OY17],                   SY = [SY01,..,SY12,1,1,1,1,1],
```

```
  OZ = [OZ01,..,OZ17],
  OX :: 1..5,
  OY :: 1..5,
  OZ :: 1..5,

  % DURATION IN THE <> DIMENSIONS
  DX = [DX01,..,DX12,1,1,1,1,1,1],
  DY = [DY01,..,DY12,1,1,1,1,1,1],
  DZ = [DZ01,..,DZ12,1,1,1,1,1,1],
  [DX01,..,DX06] :: 1..4,
  [DX07,..,DX12] :: 2..3,
  [DY01,..,DY06] :: 1..4,
  [DY07,..,DY12] :: 2..3,
  [DZ01,..,DZ06] :: 1..4,
  [DZ07,..,DZ12] :: 2..3,

  % PARALLELEPIPEDS
  P01 = [OX01,OY01,OZ01,DX01,DY01,DZ01],
  ..................................
  P12 = [OX12,OY12,OZ12,DX12,DY12,DZ12],
  P13 = [OX13,OY13,OZ13,   1,   1,   1],
  ..................................
  P17 = [OX17,OY17,OZ17,   1,   1,   1],
  LP = [P01,..,P17],

  % VOLUMES OF THE PARALLELEPIPEDS
  LV = [ 8, 8, 8, 8, 8, 8,
        12,12,12,12,12,12,
         1, 1, 1, 1, 1],

  SZ = [SZ01,..,SZ12,1,1,1,1,1,1],
  [SX01,..,SX06] :: 2..8,
  [SX07,..,SX12] :: 4..6,
  [SY01,..,SY06] :: 2..8,
  [SY07,..,SY12] :: 4..6,
  [SZ01,..,SZ06] :: 2..8,
  [SZ07,..,SZ12] :: 4..6,

  % LINK BETWEEN COMPLEMENTARY
  % SURFACE AND DURATION
  SX01*DX01= 8,..,SX06*DX06= 8,
  SX07*DX07=12,..,SX12*DX12=12,
  SY01*DY01= 8,..,SY06*DY06= 8,
  SY07*DY07=12,..,SY12*DY12=12,
  SZ01*DZ01= 8,..,SZ06*DZ06= 8,
  SZ07*DZ07=12,..,SZ12*DZ12=12,

  % CUMULATIVE AND DIFFN CONSTRAINT
  cumulative(OX,DX,SX,25),
  cumulative(OY,DY,SY,25),
  cumulative(OZ,DZ,SZ,25),
  diffn(LP,LV,LV,[6,6,6]),

  % ENUMERATION PROCEDURE
  Enum = [P01,..,P17],
  place_parallelepipeds(Enum,1,[]).
```

Enumeration procedure

Because all the placement space has to be occupied, the idea of the enumeration procedure is to try to fill systematically the space. Thus, the enumeration procedure is based on the following idea; at each choice, we compute the deepest valley and select a parallelepiped for which we fix its x-origin at the bottom of the deepest valley and we fix its x-size. When the level of the deepest valley increase, we place the parallelepipeds already fixed at the bottom of the previous deepest valley: this corresponds to a rectangle placement problem (*i.e.* we have to fix the y and z coordinates of the parallelepipeds). The enumeration procedure is as follows:

```
place_parallelepipeds([],PreviousValley,PreviousSelect) :-
  check_if_deepest_valley_change(PreviousValley,999999,PreviousSelect,_).
place_parallelepipeds([P|Rest],PreviousValley,PreviousSelect) :-
  find_deepest_valley([P|Rest],NewValley,999999),
  check_if_deepest_valley_change(PreviousValley,NewValley,PreviousSelect,NewSelect),
  fix_to_bottom_of_deepest_valley([P|Rest],NewValley,Chosen,NewRest),
  place_parallelepipeds(NewRest,NewValley,[Chosen|NewSelect]).

find_deepest_valley([],Valley,Valley).
find_deepest_valley([[X|_]|Rest],Result,Valley) :-
  domain_info(X,Min,_,_,_,_),
  Min < Valley,
  !,
  find_valley(Rest,Result,Min).
find_deepest_valley([_|Rest],Result,Valley) :-
```

```
  find_valley(Rest,Result,Valley).

check_if_deepest_valley_change(Valley,Valley,Select,Select).
check_if_deepest_valley_change(PreviousValley,NewValley,PreviousSelect,Select) :-
  PreviousValley < NewValley,
  place_rectangles(PreviousSelect),
  remove_previous_valley(PreviousSelect,Select,NewValley).

remove_previous_valley([],[],_).
remove_previous_valley([P|R],[P|S],F) :-
  P = [X,_,_,DX,_,_],
  E is X + DX,
  E > F,
  !,
  remove_previous_valley(R,S,F).
remove_previous_valley([_|R],S,F) :-
  remove_previous_valley(R,S,F).

fix_to_bottom_of_deepest_valley([[Valley,Y,Z,DX|R]|Rest],Valley,
                                [Valley,Y,Z,DX|R],Rest) :-
  indomain(DX).
fix_to_bottom_of_deepest_valley([[X|R]|Rest],Valley,Chosen,[[X|R]|NewRest]) :-
  X #> Valley,
  fix_to_bottom_of_deepest_valley(Rest,Valley,Chosen,NewRest).
```

The predicate `find_deepest_valley` computes the level of the deepest valley in the first dimension; it corresponds to the earliest start in the first dimension of the not yet fixed parallelepipeds. The predicate `check_if_deepest_valley_change` checks if the earliest start in the first dimension changed: if so, it first calls the predicate `place_rectangles` which fixes all the parallelepipeds that where fixed at the bottom of the previous valley, and finally calls the predicate `remove_previous_valley` that filters out all the parallelepipeds that are above the current earliest start in the first dimension. The predicate `fix_to_bottom_of_deepest_valley` selects a parallelepiped and fixes its origin and size in the first dimension; the origin is fixed at the earliest start in the first dimension.

Computation result

The program develops 830 nodes in order to find a first solution after 17 seconds on a SUN/SPARC station IPC (24MB). Figure 7 gives an example of a placement obtained by the previous CHIP program.

7 Solving an assignment and scheduling problem

Problem Purpose

The purpose of this example is to show how to use the *diffn* constraint to solve an assignment and scheduling problem that is situated in the context of a silicon compiler. The compiler takes a mathematical formula as entry and produces the corresponding integrated circuit in the three following steps; it first generates an operation graph, then assigns a set of components to the previous operations, and finally generates a sequencer that controls the circuit. We will focus on the main part of the compiler which corresponds to a non standard assignment and scheduling problem where it is required to take into account very specific constraints coming from the electronic part.

Figure 7: A solution for the Conway pack problem

Problem Statement

The input of our problem is a directed acyclic graph of elementary operations. Each vertex of the graph corresponds to an operation, while each edge represents a precedence constraint between two given operations. Each operation of the graph has to be assigned to a component. Each component is characterised by its surface, the list of operations it can implement and their corresponding duration. Assignment constraints correspond to the fact that an operation can only be associated to a component which can handle this kind of operation. Scheduling constraints correspond to the fact that two operations that are assigned to the same component can not overlap in time. Electronic functioning constraints deal with the pipelining of the circuit and the use of synchronisation points. Because of the introduction of the pipelining [20], each component can not be used more than the associated latency time of the pipeline. The latency of a pipeline is a measure of how long it takes a single data to pass through the pipeline. The latency constraint corresponds to the fact that the difference between the last and the first use of each component should not exceed the latency time of the circuit. To make the design of the sequencer of the circuit easier, synchronisation points are set at regular periods. The corresponding constraint states that operations should not intersect synchronisation points. Finally, the optimisation criterion is a linear term $a \times L + b \times S$ where variables L and S correspond respectively to the latency time of the circuit and to the summation of the surface of the components effectively used for implementing the circuit; a and b are non negative integers that one can choose in order to favour the speed or the surface of the circuit: using more components would decrease the latency time but increase the surface of the circuit.

Constraint Statement

In this paragraph, we explain how all the constraints of the problem can be stated in a straightforward way. Let n be the number of operations to schedule, let m be the number of available components for implementing the circuit. To each operation i ($i = 1, \ldots, n$) we associate three

domain variables corresponding respectively to the start of the operation O_i, to the duration of the operation D_i and to the component assigned to the operation C_i.

In order to state the link between the component C_i effectively assigned to an operation i and the duration D_i of that operation we use the following *element* constraint

$$element(C_i, [d_1, \ldots, d_m], D_i)$$

The meaning of this constraint is that the C_i^{th} element of the list $[d_1, \ldots, d_m]$ is D_i. The integer d_c $(c = 1, \ldots, m)$ corresponds to the duration of operation i when it is implemented by component c. For each vertex between operations i and j of the operation graph we create the following precedence constraint

$$O_j \geq O_i + D_i$$

In order to state the fact that an operation i can not overlap the synchronisation point of each cycle, we introduce a domain variable K_i which corresponds to the cycle where the operation occurs. Using this variable K_i, the synchronisation constraint is directly expressed as

$$K_i \times Cycle \leq O_i$$
$$O_i + D_i \leq (K_i + 1) \times Cycle$$

For the scheduling constraint (*i.e.* two operations that are assigned to the same component can not overlap in time) we use a *diffn* constraint in which we put together all the operations to schedule. The first and second dimension of this constraint corresponds respectively to the time and to the components.

$$diffn([[O_1, C_1, D_1, 1], \ldots, [O_n, C_n, D_n, 1]])$$

We express the latency constraint (*i.e.* the difference between the last and the first utilisation of a component should not exceed the latency time of the circuit) with the region constraint parameter associated to the *diffn* constraint. To each component c $(c = 1, \ldots, m)$, we associate a domain variable U_c, which corresponds to the utilisation of the component (*i.e.* the difference between the last and the first use of that component), and we create a fixed region of origins $0, c$ sizes $Up, 1$ where Up corresponds to an upper bound of the completion time. Thus, we complete the previous *diffn* constraint by adding the following list of region constraints

$$diffn([[O_1, C_1, D_1, 1], \ldots, [O_n, C_n, D_n, 1]],$$
$$[[[0, 1, Up, 1], 1, U_1], \ldots, [[0, m, Up, 1], 1, U_m]])$$

In order to link the latency time L of the circuit to the utilisation of the different components we use the following *maximum* constraint

$$maximum(L, [U_1, \ldots, U_m])$$

Finally, we have to link the total surface S of the circuit with the components effectively chosen for implementing the circuit. For this purpose we associate a 0-1 domain variable B_c $(c = 1, \ldots, m)$ to each component c. We link variables B_c and U_c $(c = 1, \ldots, m)$ with the following *minimum* constraint

$$minimum(B_c, [1, U_c])$$

Using the previous 0-1 variables and the surface $surf_c$ of each component c, we can now state the total surface S of the circuit as the following equality constraint

$$S = surf_1 \times B_1 + \ldots + surf_m \times B_m$$

Figure 8 illustrates for a very simple graph of operations (A), the corresponding representation (B), and the different associated constraints (C).

Figure 8: Representation for the assignment scheduling problem

Computation result

We test the previous approach on a well-known benchmark of circuit synthesis: the elliptic filter [18] of order five (see Figure 9). Part (A) gives the graph of operations, where the available components are adders and multipliers that have a respective execution time of 50 and 70 ns. With a cycle of 100 and a maximum latency time of 750 we find an optimal solution that minimises the number of used components. This solution uses only three adders and one multiplier. Each component is represented by a line where we put all the operations handled by the component (B).

8 Cycle constraint

The *cycle* constraint was introduced in CHIP to tackle complex vehicle routing problems [7] that could not yet be expressed with current constraint logic programming systems. Also, experiments in solving complex decision making problems have shown the possibility to use together the *cycle* and the *cumulative* constraint in order to solve scheduling problems. The *cycle* constraint has a set of parameters. For clarity, we start describing the basic *cycle/2* constraint. The other parameters will be introduced later in the next paragraphs. We now give the declarative semantics and the interpretation of the basic *cycle/2* constraint

$$cycle(N, [S_1, \ldots, S_m]),$$

where N is a domain variable, and $[S_1, \ldots, S_m]$ is a non-empty list of domain variables. The constraint *cycle/2* holds if the following conditions are true:

(1) $\forall i \in [1, m] : 1 \leq S_i \leq m$

(2) $\forall i \in [1, m], \forall j \neq i \in [1, m] : S_i \neq S_j$

(3) $\forall\ i \in [1, m]$:
let C_i be the set of integers defined in the following way:
$i \in C_i$, if $j \in C_i$ then $S_j \in C_i$;
then the previous scheme defines exactly N distinct sets.

Figure 10: Representation of a directed graph and two examples of use of the *cycle* constraint

In a second interpretation, the *cycle/2* constraint can be considered as the number N of cycles of a permutation $< S_1, \ldots, S_m >$[1]. For example, the constraint $cycle(3, [1,3,4,2,6,5])$ is verified since the permutation $< 1,3,4,2,6,5 >$ contains three distinct cycles. Following this interpretation, one can note that, if the domain of the variables $S_1, \ldots, S_m$ range from 1 to m, then the total number of solutions of $cycle(N, [S_1, \ldots, S_m])$ corresponds to the number of way to arrange m objects into N cycles. These numbers are called the Stirling number of first kind (see [15], page 243). Let us show a CHIP program that does the previous counting.

```
stirling(N,M) :-
    setval(nsol,0),          % initialise a global variable for counting number of solutions
    length(Ls,N),            % generate a list of N free variables
    Ls ::  1..N,             % declare the previous list as a list of domain variables
    cycle(M,Ls),             % set up the fact that want a permutation that contains M cycles
    labelling(Ls),           % generate the search space
    incval(nsol,_),          % update number of solutions
    fail.                    % fail in order to enforce backtracking for finding next solution
stirling(N,M) :-
    getval(nsol,Nsol),       % get number of solutions and print it out
    writeln(stirling(N,M,Nsol)).
labelling([]).               % succeed if no more variables
labelling([S|Rs]) :-         % if at least one variable S
    indomain(S),             % then try to assign the different possible value of S
    labelling(Rs).           % and continue with the remaining variables
```

The query $stirling(8,3)$ prints out $stirling(8,3,13132)$.

From a procedural point of view, even the simplest case, where the first parameter N is equal to one, corresponds to an NP-hard [13] problem: namely, the existence of a Hamiltonian circuit in a given directed graph. This is why a partial lookahead procedure is used to reduce the domain of variables $S_1, \ldots, S_m$.

Weighted cycle

One of the first constraints that appears when looking at vehicle routing problems concerns the number or "amount" of nodes that can be put together in the same *cycle*. In this paragraph we extend the previous *cycle/2* in order to deal with this constraint. For this purpose we introduce three additional parameters

$$cycle(N, [S_1, \ldots, S_m], [W_1, \ldots, W_m], Min, Max),$$

where $[W_1, \ldots, W_m]$ is a non-empty list of domain variables which correspond respectively to the weight attached to each node, and where Min and Max are natural numbers. The constraint *cycle/5* holds if the previous conditions hold and if the following additional condition is true:

(4) $\forall\, i \in [1,m]$:
let C_i be the set of integers defined in the following way:
$i \in C_i$, if $j \in C_i$ then $S_j \in C_i$;
let $e_{i_1}, \ldots, e_{i_{k_i}}$ be the integers occuring in C_i,
then we have: $Min \leq W_{e_{i_1}} + \cdots + W_{e_{i_{k_i}}} \leq Max$

From an interpretation point of view the *cycle/5* constraint can be viewed as the problem of finding N distinct cycles of a given minimum and maximum weight Min and Max, where the weight of a given cycle corresponds to the summation of the weights of the nodes occurring in the cycle. In Figure 11, we give an example of weighted cycles. For each node (A) we give the index of the node and its weight. We show a solution (B) where we have three cycles of respective weight 8, 71+9=80 and 9+12=21; all the previous weights belong to the interval [5,

[1] If we consider the first interpretation where we deal with directed graphs then it seems more natural to call the constraint "circuit"; however when we consider permutation then "cycle" is a more appropriate name.

Figure 11: Example of weighted cycles

Incompatible nodes

In this paragraph we extend the previous *cycle/5* constraint in order to express the fact that some nodes have to be in distinct cycles. For this purpose we introduce one additional parameter

$$cycle(N, [S_1, \ldots, S_m], [W_1, \ldots, W_m], Min, Max, [D_1, \ldots, D_p]),$$

where $[D_1, \ldots, D_p]$ is a list of natural numbers. The constraint *cycle/6* holds if the previous conditions hold and if the following conditions are both true:

(5) $\forall d \in [1, p] : 1 \leq D_d \leq m$

(6) $\forall\ i \in [1, m]$:
let C_i be the set of integers defined in the following way:
$i \in C_i$, if $j \in C_i$ then $S_j \in C_i$;
then $\forall k \in [D_1, \ldots, D_p], \forall l \neq k \in [D_1, \ldots, D_p] : C_k \cap C_l = \emptyset$

From an interpretation point of view one can partition the nodes into two distinct sets:

- nodes that occur in the list $[D_1, \ldots, D_p]$; according to the original problem, these nodes can be seen as a pool of resources, *i.e.* peoples, vehicles, or machines.
- nodes that do not occur in the list $[D_1, \ldots, D_p]$; these nodes can be seen as tasks that have to be performed by one of the previous resources.

In Figure 12, we give an example where nodes 1 and 2 are incompatible nodes; each node occurs in one distinct *cycle* of weight 4.

The previous interpretation is extremely useful in many practical applications. As we will see later in the next paragraphs, incompatible nodes are the key point for further extensions that make it possible to express constraints on specific cycles. This specificity allows to handle problems where the resources have different characteristics (skills, capacities) that have to be considered while building each *cycle*. Following this idea, the next section will introduce the notion of weight associated to the *cycle* of an incompatible node.

Weight of a specific cycle

In this paragraph we extend the previous *cycle/6* constraint in order to express constraints about the weight of the cycle associated to a given incompatible node. For this purpose we introduce one additional parameter

$$cycle(N, [S_1, \ldots, S_m], [W1, \ldots, W_m], Min, Max, [D_1, \ldots, D_p], [L_1, \ldots, L_p]),$$

Figure 12: Example of cycles with incompatible nodes

where $[L_1, \ldots, L_p]$ is a list of domain variables. The constraint *cycle/7* holds if the previous conditions hold and if the following additional condition is true:

(7) $\forall\ i \in [1, p]$:
let $d = D_i$,
let C_d be the set of integers defined in the following way:
$d \in C_d$, if $j \in C_d$ then $S_j \in C_d$;
Let $e_{d_1}, \ldots, e_{d_{k_d}}$ be the integers occurring in C_d,
then we have: $L_i = W_{e_{d_1}} + \cdots + W_{e_{d_{k_d}}}$

This parameter is useful in many practical applications: according to the type of resources that we are dealing with, it can be used to specify constraints on the minimum and maximum amount of work of a specific machine, capacity of a specific lorry or number of towns that should be visited. Let us take a simple example where we wish to generate three cycles that hold respectively 3, 4 and 2 nodes. Here is a simple CHIP program that produces such kind of configuration.

```
cycles(LS) :-
     LW = [1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ],
     LS = [S1,S2,S3,S4,S5,S6,S7,S8,S9],
     LS :: 1..9,
     cycle(3,LS,LW,1,9,[1,2,3],[3,4,2])
     labelling(LS).
```

The query $cycles(L)$ returns [4,6,9,5,1,7,8,2,3] as a first answer. In the next paragraph we introduce a new parameter which allows to name specific cycles.

Name associated to a given node

In this paragraph we extend the previous *cycle/7* constraint in order to be able to name the cycle which is associated to a given node. For this purpose we introduce one additional parameter

$$cycle(N, [S_1, \ldots, S_m], [W1, \ldots, W_m], Min, Max, [D_1, \ldots, D_p], [L_1, \ldots, L_p], [M_1, \ldots, M_m]),$$

where $[M_1, \ldots, M_m]$ is a non empty list of domain variables. The constraint *cycle/8* holds if the previous conditions hold and if the following additional conditions are both true:

Figure 13: Example of cycles with name variables

This parameter can be used for expressing a wide range of compatibility/incompatibility constraints such as:

- compatibility constraint between tasks and resource nodes: initially when the *cycle* constraint is set, the domain value of the name variable M_i associated to a given task node i is set to the resources that can effectively handle this task.
- enforce two tasks nodes to be done by the same resource: this is simply done by unifying the two names variables associated to the two tasks.
- conditional incompatibility among resources: if a given task is to be performed by a given resource then other tasks should not be performed by other given resources.
- minimum "skill" of the resources associated to a given set of tasks: let us consider a set of 3 elementary tasks that have to be done by 3 distinct resources. Suppose each resource has a known level, namely 1 or 2. We can use the name variables to express the fact that we want at least 2 resources of level 2 for handling the 3 given tasks.

In the next paragraph we will present the last extension which consists of associating an origin variable to the task nodes in order to express the fact that they should be done within a given period.

Figure 14: Example of cycles with origin variables

In the next paragraph we show an example of use of the cycle constraint for solving a vehicle tour planning problem.

9 Solving a vehicle tour planning problem

Problem Purpose

The purpose of this example is to show how to use the *cycle* constraint in order to solve a vehicle tour planning problem. We also show how to use the *cumulative* constraint as a redundant constraint to enhance the propagation.

Problem Statement

The problem is to plan the tours of a fleet of vehicles in order to deliver specific quantity of goods to a set of locations. Each vehicle has a maximum capacity and can only go to specific locations. The goal is to balance all the locations for the different vehicles, while minimising the total travelling cost associated to the fleet of vehicles. We define the cost of a vehicle to be the sum of the travel costs between all the locations successively visited by this vehicle. We will consider an example where we have 20 locations to supply with 3 vehicles of respective capacity 10, 20 and 20; each vehicle should visit between 6 and 7 locations. The tables 5 and 6 give the travel cost matrix between the different locations and the locations that can not be visited by specific vehicle.

99	36	12	34	23	99	45	12	65	45	78	02	43	71	08	30	81	38	41	31
56	99	87	98	27	19	34	09	82	34	52	86	32	58	21	45	63	26	89	96
65	34	99	98	78	54	53	21	78	67	53	45	90	21	34	52	67	89	52	39
03	34	05	99	45	13	25	26	82	09	76	65	03	04	01	93	45	34	23	63
92	08	23	56	99	54	23	74	80	40	32	51	48	92	98	71	73	45	69	43
84	56	12	35	89	99	32	16	09	05	83	48	14	16	73	46	89	73	25	94
65	48	73	24	26	54	99	65	93	65	41	14	37	76	90	94	63	58	32	35
83	81	94	67	95	45	34	99	80	81	84	60	57	53	24	53	43	52	67	62
65	67	23	47	54	38	98	76	99	34	65	45	80	98	76	37	82	61	13	64
45	63	25	89	76	54	23	89	37	99	16	89	01	23	56	85	24	59	83	26
34	76	12	32	45	86	59	91	42	50	99	34	72	52	12	43	71	06	70	23
45	64	23	89	07	05	63	25	31	47	63	99	10	19	81	25	72	54	35	45
46	86	82	42	60	25	82	63	34	71	70	80	99	41	45	68	53	27	90	72
45	73	25	17	09	06	54	31	12	63	75	69	24	99	32	79	42	34	52	76
76	43	21	31	43	26	68	54	30	20	89	21	32	56	99	23	15	26	73	23
43	25	76	90	80	73	41	23	65	79	87	54	32	14	50	99	12	41	69	75
76	44	26	78	92	54	34	11	36	94	26	80	95	78	63	24	99	63	26	87
65	24	53	17	99	76	56	34	37	41	20	74	07	54	23	76	83	99	77	34
87	45	23	53	21	80	98	11	45	76	09	78	54	13	16	68	53	20	99	73
62	14	89	09	02	56	87	43	23	56	43	66	54	19	94	74	24	43	18	99

Table 5: Travel cost between the different locations

1	2	3
1	3	8
4	17	9
5	18	12
16		13
20		19

Table 6: Forbidden locations associated to each vehicle

Problem Solution

The problem is modelled in terms of a graph covering problem. The directed graph is defined in the following way: for each location and for each vehicle we create a node; for each pair of nodes (except when both nodes correspond to a vehicle), we create a link. We label the location nodes from 1 to 20 and the vehicle nodes from 21 to 23. The basic problem is now to find three distinct circuits in such a way that each node of the graph is visited exactly once. For each node i of the graph, we create three domain variables S_i, C_i, and A_i that correspond respectively to the successor of node i, to the travel cost between node i and node S_i and to the vehicle which visits node i. We express the fact that we want to have three circuits that go through six or seven locations by using the following *cycle* constraint

$$cycle(3, [S_1, \ldots, S_{23}], [1, \ldots, 1, 0, 0, 0], 6, 7, [21, 22, 23], \_, [A_1, \ldots, A_{23}])$$

The third argument of the previous *cycle* constraint corresponds to the weight of the nodes. Because we want to constrain the number of location nodes in a given circuit to be between six and seven, we associate a weight of one to each location node and a weight of zero to each vehicle node. The sixth argument of the *cycle* constraint indicates the set of vehicle nodes and specifies that they should not belong to the same circuit. Finally we use also the name variables $A_1, \ldots, A_{23}$ to specify the compatibility constraint between vehicles and locations. For doing this we just have to remove from the name variables the values that are forbidden using the following

inequality constraints

$$\begin{array}{lll} A_{01} \#\backslash= 21 & A_{03} \#\backslash= 22 & A_{08} \#\backslash= 23 \\ A_{04} \#\backslash= 21 & A_{17} \#\backslash= 22 & A_{09} \#\backslash= 23 \\ A_{05} \#\backslash= 21 & A_{18} \#\backslash= 22 & A_{12} \#\backslash= 23 \\ A_{16} \#\backslash= 21 & & A_{13} \#\backslash= 23 \\ A_{20} \#\backslash= 21 & & A_{19} \#\backslash= 23 \end{array}$$

For example, the first column specifies that locations 1, 4, 5, 16 and 20 can not be visited by vehicle number one. In order to express the maximum capacity constraint of each vehicle we use another *cycle* constraint

$$\begin{array}{l} cycle(3, [S_1, \ldots, S_{23}], [5,1,3,2,1,6,2,2,1,4,1,1,5,1,2,1,3,1,1,2,0,0,0], 0, 100, [21,22,23], \\ \quad [K_1, K_2, K_3], [A_1, \ldots, A_{23}]) \end{array}$$

The third argument of the previous *cycle* constraint corresponds to the quantity to bring to each location, while the seventh argument is a list of domain variables which correspond to the total quantity conveyed by each vehicle. In order to express that the maximum capacities of the vehicles are respectively 10, 20 and 20, we declare the domain of the variables K_1, K_2, K_3 from 0 to 10, 0 to 20 and 0 to 20. For each location node i of the graph, we link the successor and cost variables S_i and C_i by the following *element* constraint

$$element(S_i, [Line_i|0,0,0], C_i)$$

where $Line_i$ corresponds to the i^{th} line of the travel cost matrix (see Table 5). The meaning of this constraint is that the S_i^{th} element of the list $[Line_i|0,0,0]$ is C_i. The fact that the last three values of the second argument are equal to zero means that there is no cost between location and vehicle nodes. Using the following linear equality constraint, we link the cost variables C_i to the total travelling cost C associated to the fleet of vehicles

$$C \#= C_1 + C_2 + \cdots + C_{20}$$

We now state two *cumulative* constraints that correspond respectively to the maximum number of location that can be visited by a vehicle and to the maximum capacity of each vehicle. These conditions were already expressed by the two previous *cycle* constraints. However, these conditions correspond to bin-packing problems and one can use the *cumulative* constraint to improve the propagation.

$$\begin{array}{l} cumulative([A_1, \ldots, A_{20}], [1, \ldots, 1], [1, \ldots, 1], 7) \\ cumulative([A_1, \ldots, A_{20}|21], [1, \ldots, 1|1], [5, \ldots, 2|10], 20) \end{array}$$

in the second *cumulative* constraint we introduced a fixed task of origin, duration and high 21, 1 and 10. This is done in order to express that the maximum capacity of the first vehicle is equal to the limit of the *cumulative* constraint 20 minus the high 10 of the dummy task we introduced.

Computation result

The program finds a first solution of cost 349 and a solution of cost 237 where the first, second and third vehicle visits respectively 7, 7 and 6 locations with a load of 9, 20 and 16.

10 Conclusion

In this paper, we have introduced the *among*, *diffn* and *cycle* constraints which have been implemented in CHIP in order to improve the efficiency of constraint logic programming languages for solving difficult sequencing, scheduling, placement and vehicles routing problems. We have provided a set of parameters that allow to express directly a wide range of constraints. The main originality of the *among* constraint is to allow to express directly a set of "overlapping" constraints

on sequence of domain variables in a very concise way. The originality of the *diffn* constraint is to extend the disequality constraint between domain variables to a disequality constraint between rectangles or parallelepipeds without introducing any new type of variables as in [11]. This is specially important when it is required to combine the *diffn* constraint with the other constraints of CHIP. Finally the *cycle* constraint allows to express directly graph covering constraints that could not easily be expressed with current constraint logic programming languages. One of the main advantages of these new constraints is to take into account more globally a set of elementary constraints. This makes it possible to reduce the efficiency gap between highly specialised algorithms, tailored just for one problem, and constraint logic programming, while preserving the declarative aspect of constraint logic programming. Another main advantage is the fact that all these new constraints can be combined to solve complex problems where placement, scheduling and routing constraints occurs simultaneously.

Acknowledgements:

We would like to thank all the COSYTEC team, especially Abderrahmane Aggoun and Helmut Simonis, for many fruitful discussions. We gratefully thank Mehmet Dincbas for his continuous support. We also thank Phillipe Charlier who was an intensive user and debugger of the new constraints introduced in this paper. Finally we thank Philip Kay for taking time to review a draft of this paper.

References

[1] A. Aggoun and N. Beldiceanu. Overview of the CHIP Compiler System. In *8th International Conference on Logic Programming*, pages 775–789, Paris, France, June 1991.

[2] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Journal of Mathematical and Computer Modelling*, 17(7):57–73, 1993.

[3] K.R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, 1974.

[4] A. Billionnet, M. C. Costa, and A. Sutter. Les problèmes de placement dans les systèmes distribués. *Technique et Science Informatiques*, 8(4):307–337, 1989.

[5] S.T. Coffin. The puzzling world of polyhedral dissections. Oxford University Press, 1990.

[6] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[7] P. Dejax, M. Desrochers, and M. Haouari. Les problèmes de tournées avec contraintes de fenêtres de temps, l'état de l'art. *RAIRO Operations Research*, 24(3):217–244, 1990.

[8] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, and T. Graf. Applications of CHIP to industrial and engineering problems. In *First Int. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Tullahoma, Tennesee, USA, June 1988.

[9] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F.Berthier. The Constraint Logic Programming Language CHIP. In *Proc. Int. Conf. on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Tokyo, 1988.

[10] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the car-sequencing problem in Constraint Logic Programming. In *European Conference on Artificial Intelligence (ECAI-88)*, pages 290–295, Munich, Germany, August 1988.

[11] F. du Verdier and J.P. Tsang. Un raisonnement spatial par propagation de contraintes. In *11ièmes Journées Internationales: Les Systèmes Experts & leurs Applications*, pages 297–314, Avignon, 1991.

[12] H. Dyckhoff. A typology of cutting and packing problems. *European Journal of Operational Research*, 44:145–159, 1990.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. Freeman and Co, 1979.

[14] H. Gehring, K. Menschner, and M. Meyer. A computer-based heuristic for packing pooled shipment containers. *European Journal of Operational Research*, 44:277–288, 1990.

[15] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete mathematics, a Foundation for Computer Science*. Addison-Wesley, 1989.

[16] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. 14th ACM Symp. on Principles of Programming Languages, Munich*, Germany, 1987.

[17] J. Jaffar and M.J. Maher. Constraint Logic Programming: a Survey. *Journal of Logic Programming*, 19/20:503–581, May/July 1994.

[18] S.Y. Kung, H.J. Whitehouse, and T. Kailath. *VLSI and Modern Signal Processing*. Prentice-Hall, 1985.

[19] K. Li and K. Cheng. On three-dimensional packing. *SIAM J. Comput.*, 19(5):847–867, October 1990.

[20] I.E. Sutherland. Micropipelines. *Communications of ACM*, 32(6), June 1989.

[21] T. Tokuyama, T. Asano, and S. Tsukiyama. A Dynamic Algorithm for Placing Rectangles without Overlapping. *Journal of Information Processing*, 14(1):30–35, 1991.