Beldiceanu, N.: Extending CHIP in order to solve complex scheduling and placement problems. Math. Comput. Model. 17(7...

Article *in* Mathematical and Computer Modelling · January 1992

DOI: 10.1016/0895-7177(93)90068-A

2 authors:

Abderrahmane Aggoun

36 PUBLICATIONS **1,211** CITATIONS

SEE PROFILE

Nicolas Beldiceanu

École des Mines de Nantes

148 PUBLICATIONS **2,101** CITATIONS

SEE PROFILE

Mathl. Comput. Modelling Vol. 17, No. 7, pp. 57–73, 1993
Printed in Great Britain

0895-7177/93 $6.00 + 0.00
Pergamon Press Ltd

EXTENDING CHIP IN ORDER TO SOLVE COMPLEX SCHEDULING AND PLACEMENT PROBLEMS

ABDERRAHMANE AGGOUN AND NICOLAS BELDICEANU
COSYTEC, Parc Club Orsay-Université
4 rue Jean-Rostand, 91893 Orsay cedex, France

(Received and accepted May 1992)

Abstract—In this paper, we show how the introduction of a new primitive constraint over finite domains in the constraint logic programming system CHIP allows us to find very good solutions for a large class of very difficult scheduling and placement problems. Examples on the cumulative scheduling problem, the 10 jobs × 10 machines problem, the perfect square problem, the strip packing problem and the incomparable rectangles packing problem are given, showing the versatility, the efficiency and the broad range of application of this new constraint. We point out that no other existing approach can address simultaneously all the problems discussed in this paper.

1. INTRODUCTION

CHIP (Constraint Handling in Prolog) [1,2] 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 [3] combines logic, used to specify a set of possibilities explored using a very simple built-in search method, with constraints, used to minimize the size of the search space by eliminating impossible alternatives in advance. The constraint search approach to scheduling has been advocated strongly in the U.S.A. by Mark Fox [4]. However, for scheduling and placement problems, existing constraint logic programming languages turn out to be not efficient enough to compete with specialized programs coming from the Operations Research area.

We report on research carried out at COSYTEC aimed at identifying 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 one of these new abstractions: the *cumulative* constraint. We show that the *cumulative* constraint is a major abstraction common to a large class of scheduling and placement problems.

The paper is structured as follows: in Section 2, we give a brief overview of the CHIP system. In Section 3, we present the *cumulative* constraint and its corresponding declarative semantics. In Section 4, we describe two typical scheduling problems: the first one presents a ship loading problem [5] where one has to deal with a limited amount of resources; the second one describes the famous 10 jobs × 10 machines problem [6] where disjunctive and precedence constraints occur. Finally, in the last section, we describe six placement problems: the first one is a very difficult packing problem where one has to pack squares of different sizes [7] in a large square in such a way that no hole occurs and that none of them overlap each other; the second one describes a strip packing problem [8]; the last four problems [9–12] are classical rectangle packing problems where the size of the rectangles is not always initially fixed.

We would like to thank all the COSYTEC team, especially Helmut Simonis, for many fruitful discussions on scheduling problems. We gratefully thank Mehmet Dincbas for his continuous support. We thank also Philip Kay for helpful comments on this paper. Finally, we also thank Alexander Herold for his support since investigation of solving scheduling problems began at ECRC (European Computer-Industry Research Centre).

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 specialized 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 areas of planning, manufacturing, logistics, circuit design and financial planning [13]. Originally designed at ECRC (European Computer-Industry Research Centre), CHIP is now marketed by COSYTEC.

Constraint logic programming, CLP, 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 which 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 predicates which implement a kind of branch and bound search. The combination of a constraint solver in a finite domain with the non-determinism of logic programming liberates the user from the tree-search programming and makes CLP especially attractive for handling combinatorial problems like scheduling and placement.

As an introductory example to the CHIP language, we present how a very small scheduling problem can be expressed in CHIP. We consider four tasks where each task is characterized by a duration, a set of successors and a set of tasks that should not be processed simultaneously (see Table 1). The aim is to find a schedule of the tasks that minimizes the end of the project.

Table 1. Data for the scheduling problem.

task	duration	successors	disjunction
t1	3	t2, t3	
t2	4	t4	t3
t3	2	t4	t2
t4	1		

A precedence constraint between task i and j implies that task j must start after the completion of task i. A disjunctive constraint between task i and j means that either task j must start after completion of task i, or task i must start after completion of task j. The following program outlines a CHIP program over finite domains solving the previous example.

```
project([T1,T2,T3,T4]) :-                          % line  1
      [T1,T2,T3,T4]  :: 0..10,                     % line  2
      T1 + 3 #<= T2,                               % line  3
      T1 + 3 #<= T3,                               % line  4
      T2 + 4 #<= T4,                               % line  5
      T3 + 2 #<= T4,                               % line  6
      min_max(label([T2,4,T3,2],T4),T4).           % line  7
label(Disj,End) :-                                 % line  8
      disjunctive(Disj),                           % line  9
      indomain(End).                               % line 10
disjunctive([T1,D1,T2,D2]) :-                      % line 11
      T1 + D1 #<= T2.                              % line 12
disjunctive([T1,D1,T2,D2]) :-                      % line 13
      T2 + D2 #<= T1.                              % line 14
```

The predicate `project/1` (see line 1) corresponds to the main predicate to compute the schedule. The argument of `project/1` is a list of variables that represents the starting date of each

task. As described by the domain declaration (see line 2), the domain of each variable ranges from 0 to 10. A precedence constraint (see lines 3–6) between task i and task j is expressed as

$$T_i + d_i \# <= T_j,$$

where `#<=` is the inequality constraint symbol over finite domains, where T_i and T_j are respectively the starting dates of task i and task j, and where d_i is the duration of task i. Finally, the built-in optimisation predicate `min_max/2` (see line 7) minimizes the starting date of task 4. `min_max/2` is a higher-order predicate which implements the branch and bound search exploiting the non-determinism of CHIP. Its first argument is a non-deterministic goal over which the search space is defined. Its second argument is a cost function. In this example, the goal is the predicate `label/2`, while the cost function is the domain variable T_4. The first and second arguments of `label/2` (see line 8) correspond respectively to a disjunction and to the starting date of the last task. Using the predicate `disjunctive/1`, the predicate `label/2` states the disjunctive constraint (see line 9) and fixes the origin of the last task (see line 10) using the built-in non-deterministic predicate `indomain/1` (see line 10), which is a generator of values for domain variables. A disjunctive constraint (see line 9) between task i and task j is expressed in a non-deterministic way by the predicate `disjunctive/1` (see lines 11–14) which tries different possible orderings of tasks i and j.

In disjunctive scheduling problems, the utilisation of non-determinism to express disjunctive constraints in separate clauses is superior from the declarative and expressiveness standpoints. However, it turns out to be inefficient for solving hard disjunctive scheduling problems. Indeed, each constraint is handled locally once a choice point is made. To achieve a better handling for this kind of constraint, we introduce the *cumulative* constraint.

3. CUMULATIVE CONSTRAINT

Originally, the *cumulative* constraint was introduced in CHIP to tackle complex scheduling problems which could not be solved efficiently with current constraint logic programming systems. Also, experiments in solving complex decision making problems have shown the possibility of extending the use of the *cumulative* constraint in order to solve placement problems. We now describe the declarative semantics and the interpretation of the *cumulative* constraint

$$\text{cumulative}([S_1, \ldots, S_n], [D_1, \ldots, D_n], [R_1, \ldots, R_n], L),$$

where $[S_1, \ldots, S_n], [D_1, \ldots, D_n]$ and $[R_1, \ldots, R_n]$ are non-empty lists of domain variables that have the same length n, and where L is a natural number. For a domain variable V, we note respectively $\min(V)$ and $\max(V)$ the smallest and the greatest value of the domain of the variable V. Let:

$$a = \text{minimum}(\min(S_1), \ldots, \min(S_n)),$$
$$b = \text{maximum}(\max(S_1) + \max(D_1), \ldots, \max(S_n) + \max(D_n)),$$

The constraint *cumulative* holds if the following condition is true:

$$\sum_{j / S_j \leq i \leq S_j + D_j - 1} R_j \leq L, \qquad \forall\, i \in [a, b].$$

Procedurally, the implementation of the *cumulative* constraint corresponds to a specialization of the lookahead [14] declaration. From an interpretation point of view, the *cumulative* constraint matches directly the single resource scheduling problem, where $S_1, \ldots, S_n$ correspond to the start of the tasks, $D_1, \ldots, D_n$ correspond to the duration of the tasks, and $R_1, \ldots, R_n$ to the amount of resources used by each task. The natural number L is the total amount of available resource which 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.

In Figure 1, we sketch three different cases of the *cumulative* constraint. The first case (A) corresponds to a general use. We show in bold the profile curve of amount of resource required by three tasks: task 1 uses one unit of the resources during four consecutive periods, tasks 2 and 3 use two units during respectively two and three periods. We point out that at any time, the total amount of resources used by the different tasks is always less than or equal to three. The second case (B), where all the task durations are equal to 1, corresponds to a bin-packing problem. The third case (C), where the last parameter of the *cumulative* constraint is equal to 1, corresponds to a disjunctive scheduling problem.

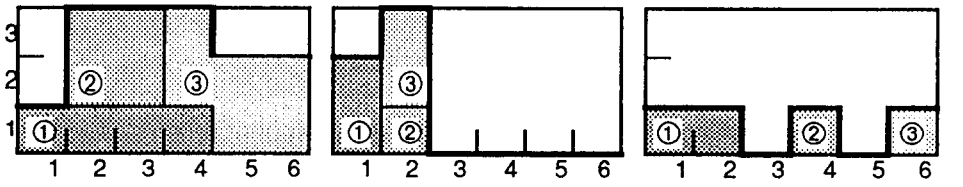

Figure 1. Three examples of use of the cumulative constraint.

4. SOLVING SCHEDULING PROBLEMS

Since the 1960s, scheduling problems have captured the interest of many researchers. Scheduling selects among alternative plans and assigns resources and times for each activity so that they obey the temporal restrictions of activities and the capacity limitations of a set of shared resources. From a computational complexity perspective, scheduling problems have been proven to be NP-Hard [15]. Many differing approaches have been tried for solving scheduling problems. Operations Research takes some restricted cases of scheduling problems (single resource scheduling, job-shop and flow-shop scheduling, due date scheduling, etc. ...) and analyzes the corresponding underlying mathematical properties in order to find out the complexity of the restricted problem, to get accurate upper and lower bounds of the optimal solution, to propose specialized heuristics, and to find algorithms that work well on average. Operations Research gives good results for many of the restricted cases, but has a major drawback when faced with real problems where many different constraints have to be taken into account simultaneously. Most methods that work well on restricted cases and that rely on specific mathematical properties cannot be used for complex problems. For this reason, more general approaches were tried in order to deal with the complexity of scheduling problems. The expert system approach was first used in order to introduce knowledge about dispatching rules used in a shop. Then simulated annealing [16] was used for developing general algorithms that give high-quality solutions. More recently, tabu search was introduced [17] in order to add more flexibility to the simulated annealing method. However, simulated annealing suffers from two drawbacks: it is difficult to handle problems where different kinds of constraints occur because all these constraints have to be integrated into one evaluation function; highly constrained problems where the number of solutions is very limited can not be solved. Recently, constraint logic programming was introduced as a new approach for solving scheduling problems [18]. Using the *cumulative* constraint presented in the previous section, we will now show how CHIP can solve two very difficult scheduling problems.

4.1. Scheduling with Cumulative and Precedence Constraints

PROBLEM PURPOSE. The purpose of this example is to show how to solve a scheduling problem where *cumulative* and precedence constraints occur [5]. It contains some discussions about how tasks with fixed surfaces are best handled.

PROBLEM STATEMENT. The problem is to find a schedule that minimizes the time to unload and to load a ship. The work contains a set of 34 elementary tasks. Each task has to be handled by a given number of people and during a given period of time (see Table 2). For each task, only the associated surface is known (i.e., the product of the task duration by the needed amount of resource). From a practical point of view, this last constraint is very important because it leads

to a good utilisation of the resource [19]. This last point will be illustrated later on a comparative example. Besides the usual precedence constraints that arise in scheduling problems, there is also a *cumulative* constraint because of the limited amount of resources (i.e., people), which must be shared by the different tasks.

Table 2. Data for the ship loading problem.

N	task surface	successors
01	12	02, 04
02	16	03
03	12	05, 07
04	24	05
05	25	06
06	10	08
07	12	08
08	12	09
09	12	10, 14
10	16	11, 12
11	12	13
12	10	13
13	4	15, 16
14	15	15
15	6	18
16	9	17
17	12	18
18	14	19, 20, 21
19	4	23
20	4	23
21	4	22
22	8	23
23	28	24
24	40	25
25	16	26, 30, 31, 32
26	3	27
27	3	28
28	12	29
29	8	
30	9	28
31	6	28
32	3	33
33	6	34
34	6	

PROBLEM SOLUTION. We describe successively the problem representation, the way constraints are stated, and finally the heuristic which is used.

PROBLEM REPRESENTATION. Assume L is the limit on the amount of available resource which can be used for the scheduling, and n is the number of tasks of the scheduling problem.

We first add a fictitious *end* task of duration 0 that is preceded by all the other tasks. To each task $i\,(i = 1, \ldots, n)$ we associate three domain variables corresponding respectively to the start of the task S_i, to the duration of the task D_i and to the amount of resource needed by the task R_i.

CONSTRAINT STATEMENT. For each task i the surface constraint is expressed as

$$D_i \times R_i = \text{Surface}_i,$$

where Surface_i corresponds to the surface of task i (see Table 2). For example, the surface of the first task is equal to 12. This means that the pair of variables (D_1, R_1) can only take one of the following pair of values: (1,12), (2,6), (3,4), (4,3), (6,2), (12,1).

A precedence constraint between task i and task j is directly expressed as

$$S_j \geq S_i + D_i.$$

Finally, the resource constraint about the total amount of available resource L is expressed as

$$\text{cumulative}\,([S_1, \ldots, S_n], [D_1, \ldots, D_n], [R_1, \ldots, R_n], L).$$

CHOOSING A GOOD GENERATION ORDERING. Because tasks may be executed simultaneously, the following labeling procedure **disjunctive/4** used for disjunctive scheduling (i.e., one task is before or after another task) is no longer valid.

```
disjunctive(S1,D1,S2,D2) :-
      S2 #>= S1 + D1.
disjunctive(S1,D1,S2,D2) :-
      S1 #>= S2 + D2.
```

An additional problem comes from the fact that we have to fix the duration or the amount of resource used by each task (because of the surface constraint, fixing one of the parameters will fix the other one). The generator procedure is based on the following idea. At each choice point, we choose a task and label the different possible starting points and the different possible durations. The generator procedure is the following program:

```
labeling([S1|S],[D1|D]) :-
      indomain(S1),
      indomain(D1),
      labeling(S,D).
labeling([ ],[ ]).
```

The first and the second argument of **labeling/2** are two lists of domain variables corresponding respectively to the origin and to the duration of the different tasks. The predicate **indomain/1**, a non-deterministic predicate, is a generator of values for domain variables.

COMPUTATION RESULTS. Table 3 gives for different resource limits (L) concerning the number of available people, the corresponding optimal finishing date (D) of the schedule, and the time (T) in seconds on a SUN/SPARC station IPC(12MB) required for finding the optimal solution and for proving its optimality.

Table 3. Results for the ship loading problem.

L	D	T	L	D	T	L	D	T	L	D	T
			11	48	1.48	21	31	0.28	31	24	0.04
2	204	0.41	12	43	0.45	22	30	0.15	32	24	0.03
3	169	2.71	13	43	0.60	23	30	0.13	33	24	0.05
4	122	1.90	14	38	0.46	24	30	0.15	34	24	0.03
5	91	1.43	15	36	0.29	25	26	0.13	35	24	0.03
6	82	1.06	16	34	0.30	26	26	0.14	36	24	0.05
7	68	2.01	17	34	0.28	27	26	0.13	37	24	0.04
8	56	0.96	18	33	0.26	28	24	0.03	38	24	0.04
9	53	0.75	19	33	0.25	29	24	0.03	39	24	0.03
10	50	1.15	20	31	0.34	30	24	0.03	40	23	0.03

DISCUSSION. In this paragraph, we will show in a comparative example on the ship loading problem, why it can be inadequate to fix the duration or the amount of resources used by the tasks. For practical applications, this is very important because it can lead to a bad resource utilization. Suppose we consider a resource limit of eight people. Figure 2 gives for the optimal solution of cost 56, the profile curve of amount of used resources and the origin and contribution of each task. In this solution, we can observe that CHIP has to "play" on the duration and on the amount of resources used by the different tasks in order to "pack" them as much as possible.

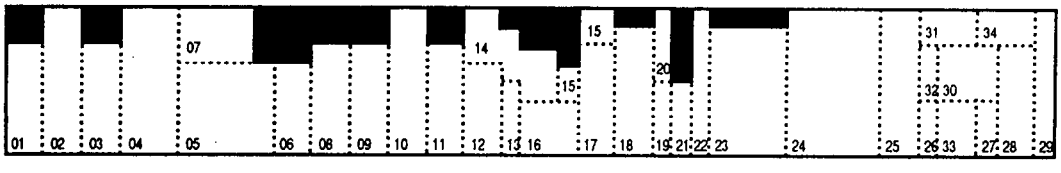

Figure 2. Optimal solution of the ship loading problem with a limit of eight people (fixed surface).

Now, if we consider that the duration D and the amount of resource R of each task N are fixed and we take the original data (see Table 4) given in [5], we get an optimal solution of cost 67 (see Figure 3) which corresponds to a bad utilization of the resources.

Table 4. Duration and amount of resource of the tasks for the ship loading problem.

N	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17
L	03	04	04	06	05	02	03	04	03	02	03	02	01	05	02	03	02
R	04	04	03	04	05	05	04	03	04	08	04	05	04	03	03	03	06
N	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
L	02	01	01	01	02	04	05	02	01	01	02	01	03	02	01	02	02
R	07	04	04	04	04	07	08	08	03	03	06	08	03	03	03	03	03

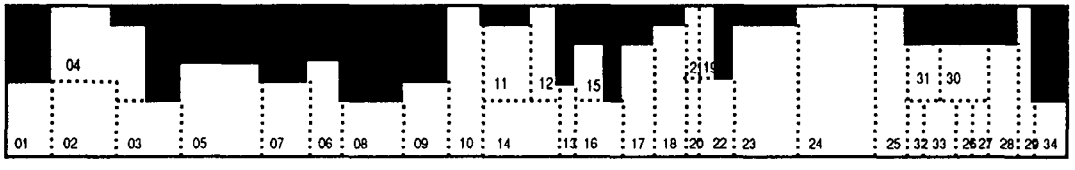

Figure 3. Optimal solution of the ship loading problem with a limit of eight people (fixed duration).

4.2. Job-Shop Scheduling

PROBLEM PURPOSE. The purpose of this example is to show how to solve a complex job-shop problem where disjunctive and precedence constraints occur. We will take the famous 10 jobs × 10 machines problem defined in 1963 in the book *Industrial Scheduling* of Muth and Thompson [6]. This problem was considered as one of the most rewarding tests for job-shop scheduling. Finding the optimal solution was an open problem during more than 20 years where a lot of approaches were tried [20–26]. The best available upper bound around 1981–1982 has been 935 and no reasonable lower bound was known at this time. In 1985 the best available bounds 907 and 930 were both obtained at the Mathematical Center (Amsterdam), using Lagrangian relaxation methods. Using a highly specialized search algorithm, the problem was completely solved by Pinson in 1987 [27]. Our aim is to show how a very simple declarative CHIP program, together with the *cumulative* constraint, can also find the optimal solution.

PROBLEM STATEMENT. The job-shop problem consists of scheduling n jobs on m machines. To each job corresponds a set of tasks that have to be carried out in a fixed order. Each task has to be executed on a given machine. The following hypotheses must also hold:

- at any time, a machine can execute only one task,
- the execution of a task cannot be interrupted,
- waiting between two consecutive tasks of a given job is allowed,
- each machine is completely available during the schedule,
- each machine is independent from the other machines,
- each job is independent from the other jobs.

Each line of Table 5 corresponds to one job and gives a list of pairs $(JXYZ, D)$ where $JXYZ$ is a task name and D the duration. In the task name X, Y and Z are hexadecimal numbers and correspond respectively to the job number, to the order of the task in the job, and to the machine where the task is proceeded; e.g., $Ja59$ denotes task number 5 in the job 10 and which is proceeded on machine number 9. The problem is to find an ordering of the tasks that minimizes the end date of the schedule.

PROBLEM SOLUTION. We describe successively how precedence and disjunctive constraints are stated, and give the results corresponding to the use of a basic labeling procedure.

Table 5. Data for the 10 jobs × 10 machines problem.

```
[J111,29],[J122,78],[J133, 9],[J144,36],[J155,49],[J166,11],[J177,62],[J188,56],[J199,44],[J1aa,21].
[J211,43],[J223,90],[J235,75],[J24a,11],[J254,69],[J262,28],[J277,46],[J286,46],[J298,72],[J2a9,30].
[J312,91],[J321,85],[J334,39],[J343,74],[J359,90],[J366,10],[J378,12],[J387,89],[J39a,45],[J3a5,33].
[J412,81],[J423,95],[J431,71],[J445,99],[J457, 9],[J469,52],[J478,85],[J484,98],[J49a,22],[J4a6,43].
[J513,14],[J521, 6],[J532,22],[J546,61],[J554,26],[J565,69],[J579,21],[J588,49],[J59a,72],[J5a7,53].
[J613,84],[J622, 2],[J636,52],[J644,95],[J659,48],[J66a,72],[J671,47],[J687,65],[J695, 6],[J6a8,25].
[J712,46],[J721,37],[J734,61],[J743,13],[J757,32],[J766,21],[J77a,32],[J789,89],[J798,30],[J7a5,55].
[J813,31],[J821,86],[J832,46],[J846,74],[J855,32],[J867,88],[J879,19],[J88a,48],[J898,36],[J8a4,79].
[J911,76],[J922,69],[J934,76],[J946,51],[J953,85],[J96a,11],[J977,40],[J988,89],[J995,26],[J9a9,74].
[Ja12,85],[Ja21,13],[Ja33,61],[Ja47, 7],[Ja59,64],[Ja6a,76],[Ja76,47],[Ja84,52],[Ja95,90],[Jaa8,45].
```

CONSTRAINT STATEMENT. For each task, we create a domain variable corresponding to the effective start of the task. We also create a domain variable E for the end of the schedule.

For each job, the predicate **gen_prec/2** generates a set of precedence constraints corresponding to the sequencing of the tasks in the given job. The last precedence constraint corresponds to a precedence constraint between the last task of the job and the schedule end. The constraints generation procedure is the following program:

```
gen_prec([J1,J2|R],[D1,D2|S]) :-
        J1 + D1 #<= J2,
        gen_prec([J2|R],[D2|S]).
gen_prec([_],[_]).
```

The two arguments of **gen_prec/2** correspond respectively to the list of task origins of a given job and to their corresponding durations.

For each machine, the predicate **gen_disj/3** generates one disjunctive constraint corresponding to the fact that two tasks that are proceeded on the same machine should not overlap. The disjunction is handled as a special case of the *cumulative* constraint where the height of each task and the height of the schedule (the maximum amount of available resource) are both equal to 1. The three arguments of **gen_disj/3** corresponds respectively to the number of jobs, to the origin of the tasks that are executed on the same machine, and to their corresponding duration. The predicate **gen_one/2** generates a list of value 1 corresponding to the respective amount of resource used by the tasks.

```
gen_disj(N,LOrigin,LDuration) :-
        gen_one(N,LOne),
        cumulative(LOrigin,LDuration,LOne,1).
gen_one(N,[1|R]) :-
        N > 0,
        M is N - 1,
        gen_one(M,R).
gen_one(0,[ ]).
```

If we consider the data of the 10 jobs × 10 machines problem (see Table 5), the generation of the corresponding constraints will look as follows:

```
top(Max):-
        E :: 1..Max,
        [J111,J122,J133,J144,J155,J166,J177,J188,J199,J1aa] :: 0..Max,
        ..........................................................
        gen_prec([J111,J122,J133,J144,J155,J166,J177,J188,J199,J1aa,E],
                 [ 29,  78,   9,  36,  49,  11,  62,  56,  44,  21,1]),
                 ..........................................................
        gen_disj(10,
                 [J111,J211,J911,J321,J521,J721,J821,Ja21,J431,J671],
                 [ 29,  43,  76,  85,   6,  37,  86,  13,  71,  47]),
                 ..........................................................
```

COMPUTATION RESULTS. Using the basic labeling procedure for disjunctive tasks described in [18], the program finds on a SUN/SPARC station IPC(12MB) a solution of cost 1088 in 1.06 seconds and the optimal solution of cost 930 in 1506 seconds. The labeling procedure consists of ordering all the tasks that use machine 1 first, then ordering those using machine 2, and so on.

5. SOLVING PLACEMENT PROBLEMS

Spatial data [28] such as lines, rectangles, surfaces and volumes are becoming increasingly important for many applications such as computer graphics, computer-aided design, geographic information systems, computational geometry, and other areas. If constraint logic programming intends to deal with these kinds of applications, then it is necessary to understand the representation of such spatial data structures and the formulation of spatial constraint propagation [29]. This section describes a step towards that goal. We will focus on the problem of allocating a set of iso-oriented rectangles without overlapping [30] which has widespread applications in all the domains mentioned before. Using the *cumulative* constraint presented in Section 3, we will show how CHIP can solve variants of the previous problem.

5.1. Perfect Square

PROBLEM PURPOSE. The purpose of this example is to show how to solve a two-dimensional packing problem where squares are involved. This problem illustrates the link between non-overlapping and *cumulative* constraints.

PROBLEM STATEMENT. The problem is to find out how to pack squares of given sizes into a large square in such a way that none of them overlap any other. All the squares have a different size and the summation of the surfaces of the different squares is equal to the surface of the space where the squares are placed. Table 6 gives the corresponding data for two instances of the problem, where (N) corresponds to the size of the large square to pack, and (S) is the size of the squares to pack. It has been shown in [7] that the smallest number of squares that can be packed in a large square with the previous conditions is 21.

Table 6. Size of the squares for the perfect square problem.

N	S
112	2 4 6 7 8 9 11 15 16 17 18 19 24 25 27 29 33 35 37 42 50
175	1 2 3 4 5 8 9 14 16 18 20 29 30 31 33 35 38 39 43 51 55 56 64 81

PROBLEM REPRESENTATION. Let N be the size of the large square in which to pack the smaller squares, let n be the number of squares to pack, let X_i $(i = 1, \ldots, n)$ be the coordinate of the origin of square i on the x-axis, let Y_i $(i = 1, \ldots, n)$ be the coordinate on the y-axis of square i, and let S_i $(i = 1, \ldots, n)$ be the size of square i (see Figure 4).

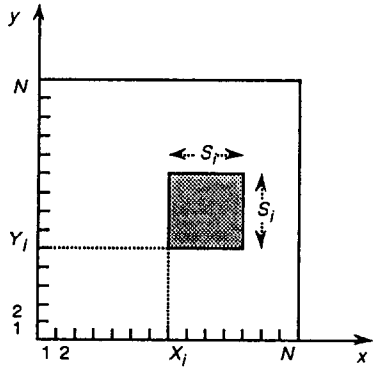

Figure 4. Representation for the square packing problem.

From the packing constraint, we can derive the following two necessary conditions corresponding respectively to *cumulative* conditions on the x and y axes:

$$\sum_{j/X_j \leq i \leq X_j+S_j-1} S_j \leq N, \qquad \forall\, i \in [1,N], \tag{1}$$

$$\sum_{j/Y_j \leq i \leq Y_j+S_j-1} S_j \leq N, \qquad \forall\, i \in [1,N]. \tag{2}$$

These conditions state that the coordinates on the x-axis and the coordinates on the y-axis have both to verify the same *cumulative* condition. The link between *cumulative* constraint and packing in two dimensions is developed in [31].

PROBLEM SOLUTION. From the previous remark, the problem solution is divided into two steps: in a first step, we search for all *cumulative* solutions on the x-axis and on the y-axis; in a second step, we try to combine these solutions in order to solve the initial packing problem. We now consider respectively the two steps.

GENERATION OF THE CUMULATIVE SOLUTIONS. Using the *cumulative* constraint, the generation of all possible *cumulative* solutions is as follows:

```
generate_cumulative_solutions(N,LSize,LStart) :-
      generate_variables(LStart,LSize,N),
      cumulative(LStart,LSize,LSize,N),
      labeling(LStart).
generate_variables([Var|R],[Size|S],N) :-
      M is N - Size + 1,
      Var :: 1..M,
      generate_variables(R,S,N).
generate_variables([ ],[ ],_).
```

According to the list of square size and to the size of the large square where to pack, the predicate `generate_variables/3` generates a list of domain variables corresponding to the origin of each square. The origin variable of each square is created in such a way that the square does not finish after N. The enumeration procedure is based on the following; at each choice point, we consider the earliest possible start associated to the not yet placed squares. Because the surface of all the squares to pack is equal to the surface of the large square where to pack, we know that no hole is allowed. Therefore, at least one square should start at this earliest start. The labeling procedure is thus:

```
labeling([X|Y]) :-
      lmindomain([X|Y],E),
      fix_min([X|Y],E,R),
      labeling(R).
labeling([ ]).
fix_min([X|R],E,R) :-
      X = E.
fix_min([X|R],E,[X|S]) :-
      X #> E,
      fix_min(R,E,S).
```

The predicate `lmindomain/2` computes the earliest start E of the not yet fixed squares, and `fix_min/3` tries to fix the origin of one square to value E. In order to restrict as much as possible the search space, we fix large squares first. This is simply done by passing to the labeling procedure the list of the origin of the squares sorted in decreasing order according to their respective size.

COMBINING THE CUMULATIVE SOLUTIONS. Once we have generated the set of all *cumulative* solutions, we try to combine two *cumulative* solutions in order to get a solution for the original

packing problem. The first *cumulative* solution will correspond to the x-coordinates of the squares and the second solution will correspond to the y-coordinates. The corresponding program is as follows:

```
combine([Sol|RSol],LSize) :-
      combines(RSol,Sol,LSize),
      combine(RSol,LSize).
combine([ ],_).
combines([Sol2|RSol],Sol1,LSize) :-
      non_overlap(Sol1,Sol2,LSize),
      !,
      writeln(solution(Sol1,Sol2)),
      combines(RSol,Sol1,LSize).
combines([Sol2|RSol],Sol1,LSize) :-
      combines(RSol,Sol1,LSize).
combines([ ],_,_).
non_overlap([X|RSol1],[Y|RSol2],[S|RSize]) :-
      no_intersect([X,Y,S],RSol1,RSol2,RSize),
      no_overlap(RSol1,RSol2,RSize).
non_overlap([ ],_,_).
no_intersect([X1,Y1,S1],[X2|RSol1],[Y2|RSol2],[S2|RSize]) :-
      no_inter([X1,Y1,S1],[X2,Y2,S2]),
      no_intersect([X1,Y1,S1],RSol1,RSol2,RSize).
no_intersect(_,[ ],[ ],[ ]).
no_inter([X1,Y1,S1],[X2,Y2,S2]) :-
      X1 + S1 =< X2,
      !.
no_inter([X1,Y1,S1],[X2,Y2,S2]) :-
      X2 + S2 =< X1,
      !.
no_inter([X1,Y1,S1],[X2,Y2,S2]) :-
      Y1 + S1 =< Y2,
      !.
no_inter([X1,Y1,S1],[X2,Y2,S2]) :-
      Y2 + S2 =< Y1.
```

The predicate `combine/2` tries to associate each possible pair of *cumulative* solutions in order to obtain a solution for the original problem. The predicate **`combines/3`** tries to associate a given *cumulative* solution to one of the remaining *cumulative* solutions. The predicate **`non_overlap/3`** checks whether all the squares associated to the two given *cumulative* solutions overlap or not. The predicate **`no_intersect/4`** checks whether a given square intersects one of the remaining squares associated to the two *cumulative* solutions. Finally, the predicate **`no_inter/2`** checks whether the two given squares intersect or not. Figure 5 gives an example of a perfect square of order 24 obtained by CHIP.

COMPUTATION RESULTS. Table 7 gives, for two instances of the problem (n), corresponding respectively to 21 and 24 squares, the computation results. We first give the total time ($T1$) needed for finding all the *cumulative* solutions and for proving that no other *cumulative* solution exists. Then we give the corresponding number of choice points ($C1$) and the time ($T2$) spent to combine the *cumulative* solutions in order to obtain all the solutions of the original packing problem. All the times are given in seconds on a SUN/SPARC station IPC(12MB).

5.2. Strip Packing

PROBLEM PURPOSE. The example shows how the *cumulative* constraint can be used in conjunction with the cardinality constraint [32] in order to solve efficiently a complex rectangle packing

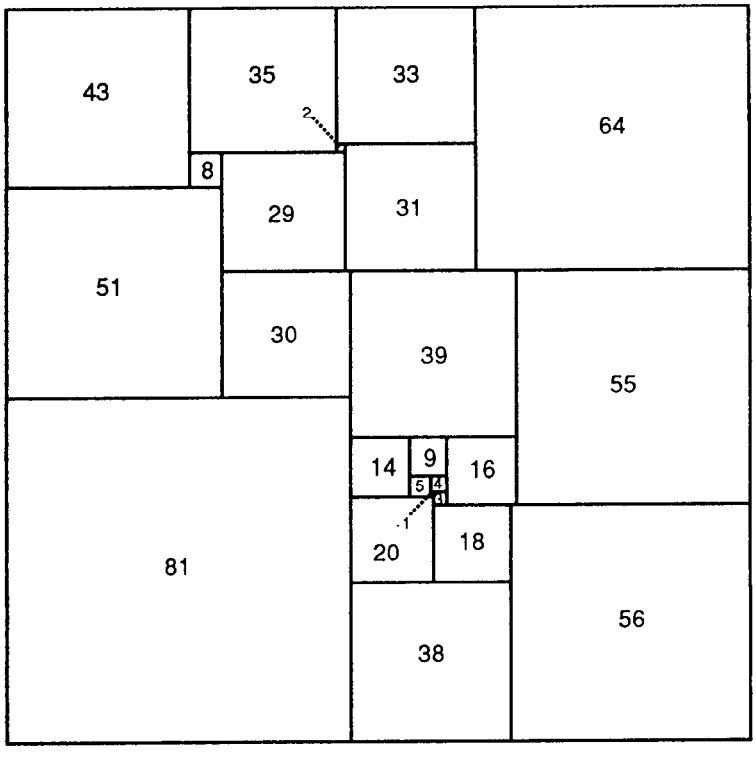

Figure 5. A solution for the perfect square of order 24.

Table 7. Results for the perfect square packing problem.

n	T1	C1	T2
21	37.98	2651	0.11
24	99.56	6962	0.63

problem. For this problem, the CHIP program outperforms more specialized algorithms from Operations Research.

PROBLEM STATEMENT. This problem, also called strip packing, is described in [33]. Let L_n be a list of n rectangles of given sizes; they must be packed into a semi-infinite strip of fixed width W. Packing of the strip must be such that (i) rectangles do not overlap each other or the boundaries of the strip; (ii) the rectangles are packed with their sides parallel to the sides of the strip; and (iii) the height of the packing (the maximum height of the top of any rectangle) is minimized. The rectangles must be packed in their given orientations, i.e., no rotation is allowed. Figure 6 shows an example of a strip packing problem with six rectangles.

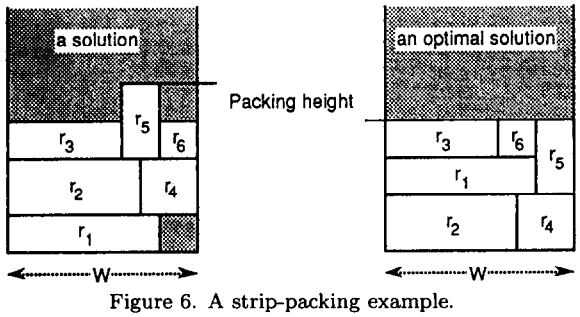

Figure 6. A strip-packing example.

For this problem, the data was taken from Komarnicki's Ph.D. thesis on heuristics for strip packing problems [8]. In this example, we have to pack a set of 48 rectangles into a strip of width 200. The width and the height of each rectangle is given in Table 8.

Table 8. Data for the strip-packing problem.

r_1 = (160,50)	r_{17} = (42,10)	r_{33} = (4,21)
r_2 = (149,32)	r_{18} = (82,34)	r_{34} = (25,20)
r_3 = (104,50)	r_{19} = (68, 8)	r_{35} = (24,20)
r_4 = (98,30)	r_{20} = (46,34)	r_{36} = (16,19)
r_5 = (82,20)	r_{21} = (53,16)	r_{37} = (16,19)
r_6 = (70,32)	r_{22} = (54,12)	r_{38} = (40,17)
r_7 = (70,26)	r_{23} = (46,16)	r_{39} = (16,17)
r_8 = (68,32)	r_{24} = (45,16)	r_{40} = (40,16)
r_9 = (64,16)	r_{25} = (20,52)	r_{41} = (32,14)
r_{10} = (63,24)	r_{26} = (14,50)	r_{42} = (12,14)
r_{11} = (53, 9)	r_{27} = (12,32)	r_{43} = (24,12)
r_{12} = (52,16)	r_{28} = (12,32)	r_{44} = (36,12)
r_{13} = (51,46)	r_{29} = (12,24)	r_{45} = (24,12)
r_{14} = (44, 8)	r_{30} = (16,22)	r_{46} = (4,12)
r_{15} = (42,12)	r_{31} = (6,22)	r_{47} = (40,10)
r_{16} = (42,30)	r_{32} = (10,21)	r_{48} = (25,20)

PROBLEM REPRESENTATION. Let W be the width of the strip, let H be the height of the packing, let n be the number of rectangles to pack, let X_i and Y_i $(i = 1, \ldots, n)$ be the coordinates of the origin of rectangle i on the x and y-axes, let W_i and H_i $(i = 1, \ldots, n)$ be the width and the height of rectangle i. As for the perfect square problem, we can derive two conditions corresponding in this case to two distinct necessary conditions of the rectangle packing problem:

$$\sum_{j/X_j \leq i \leq X_j + W_j - 1} H_j \leq H, \qquad \forall\, i \in [1, W], \tag{1}$$

$$\sum_{j/Y_j \leq i \leq Y_j + H_j - 1} W_j \leq W, \qquad \forall\, i \in [1, H]. \tag{2}$$

PROBLEM SOLUTION. From the previous remark, we use the two *cumulative* conditions as redundant constraints. These conditions are stated using the following *cumulative* constraints:

$$\text{cumulative}\,([[X_1, \ldots, X_n], [W_1, \ldots, W_n], [H_1, \ldots, H_n], H]),$$
$$\text{cumulative}\,([[Y_1, \ldots, Y_n], [H_1, \ldots, H_n], [W_1, \ldots, W_n], W]).$$

The non-overlapping constraint is not expressed by a test, as for the perfect square problem (see predicate `no_inter/2` in Subsection 5.1), we now use a constraint similar to the cardinality constraint [32] in order to achieve an active pruning. We represent the fact that two given rectangles i and j do not overlap by the following constraint:

$$\text{cardinality}\,(1, *, [X_i + W_i \leq X_j,\ X_j + W_j \leq X_i,\ Y_i + H_i \leq Y_j,\ Y_j + H_j \leq Y_i]).$$

The above cardinality constraint holds if, out of the four following constraints $X_i + W_i \leq X_j$, $X_j + W_j \leq X_i$, $Y_i + H_i \leq Y_j$, $Y_j + H_j \leq Y_i$, at least one is satisfied.

COMPUTATION RESULTS. Using a first-fit like heuristics, the CHIP program finds a solution of cost 286 after 6.9 seconds and a solution of cost 285 after 507 seconds on a SUN/SPARC station IPC(12MB). With a non-fully automatic CHIP program, where the user can interactively guide the search process, we enhance the previous result and obtain a solution of cost 280 (see Figure 7). A lower bound for the optimal solution is 274 and can be computed by dividing the total surface of the 48 rectangles by the width of the strip.

Komarnicki [8] takes a different approach, developing a highly specialized heuristics. Within this approach, the best solution has a cost of 295 which improved an old existing solution of cost 337 found by Baker [34].

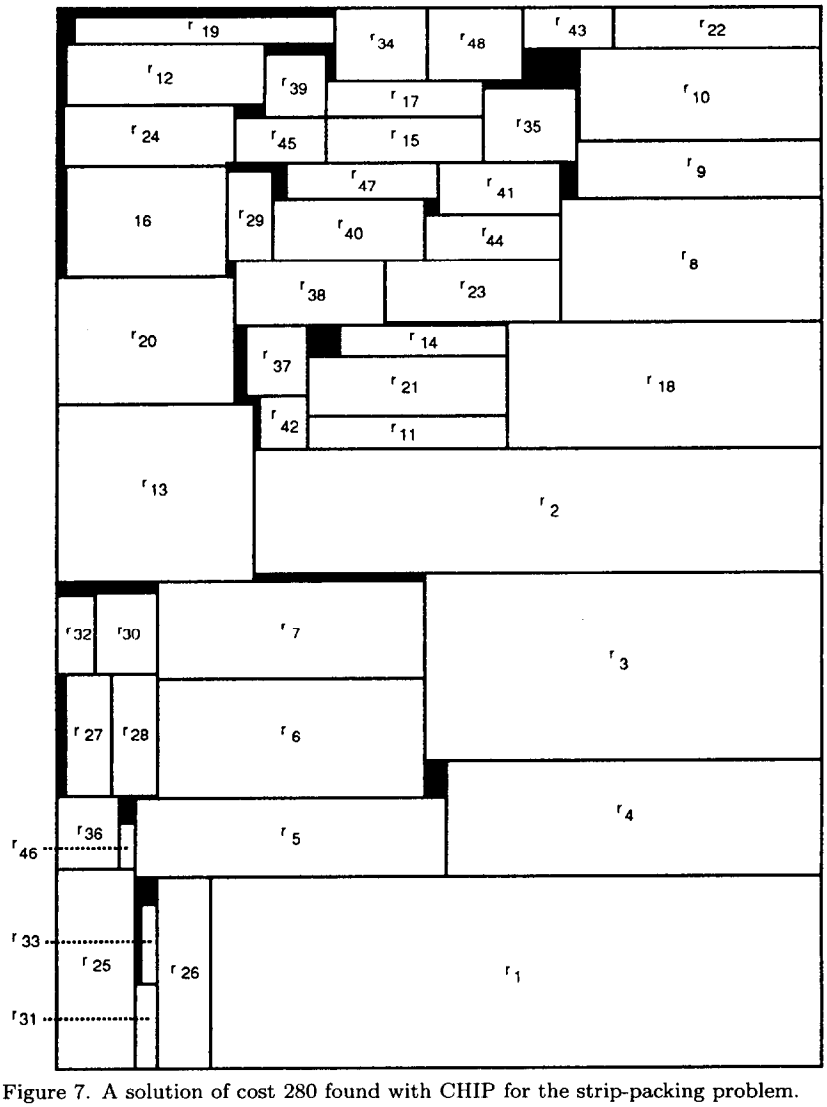

Figure 7. A solution of cost 280 found with CHIP for the strip-packing problem.

5.3. Classical Rectangle Packing Problems

PROBLEM PURPOSE. The purpose of this paragraph is to present briefly four classical rectangle packing problems where the objective is to find all the solutions. The main characteristic of these problems is the fact that the size of the rectangles may not be initially known. As the resolution of these problems is very similar to the one used for the perfect square and strip packing problems, we will only give a short description of each problem, followed by the corresponding computational results.

PROBLEM STATEMENT. We first describe the basic constraints common to the four problems. Let L_n be a list of n rectangles and let R be a rectangle of fixed width w and height h. The packing of the rectangles of L_n in the rectangle R must be such that (i) rectangles of L_n do not overlap each other or the boundaries of R; (ii) rectangles of L_n are packed with their sides parallel to the sides of R; (iii) the summation of the surfaces of the rectangles of L_n must be equal to the surface of R. The information about the size of each rectangle is given in Table 9.

We now give the details for each problem:

- The first problem [9] consists of packing six rectangles of given size, all the rectangles must be packed in their given orientation.
- The second problem [10] corresponds to the first problem, except for the fact that rotation is allowed.

- The third problem is taken from [11] and consists of packing four rectangles for which only the minimum width and height are known.
- The last problem, known as the incomparable rectangles packing problem [12], consists of packing seven rectangles of unknown width and height. In this case, an additional constraint must also hold between each pair of rectangles of the list of elementary rectangles L_n. This additional constraint between two rectangles R_1 and R_2 of respective width w_1, w_2 and height h_1, h_2 holds if and only if the four following conditions are true:

$$C_1 : (w_1 > w_2) \vee (h_1 > h_2), \qquad C_2 : (w_1 > h_2) \vee (h_1 > w_2),$$
$$C_3 : (w_2 > w_1) \vee (h_2 > h_1), \qquad C_4 : (w_2 > h_1) \vee (h_2 > w_1).$$

These conditions correspond to the fact that R_1 cannot be included in R_2 and that R_2 can also not be included in R_1. Figure 8 shows the solution of the incomparable rectangles packing problem. The CHIP program computes the size and the position of each rectangle and proves that no other solution exists.

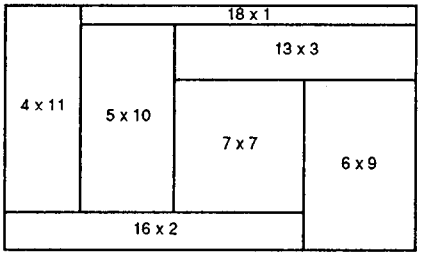

Figure 8. Solution for the incomparable rectangles packing problem.

COMPUTATION RESULTS. Table 9 summarizes the main characteristics of the four packing problems and gives the corresponding computation results.

Table 9. Results for four classical rectangle packing problems.

problem	number of rectangles	size of rectangles	rotation	size of big rectangle	number of solutions	time in seconds
Pfefferkorn	6	(6,2) (4,2) (2,3) (2,3) (2,3) (2,1)	no	(8,5)	24	0.2
Laurière	6	(6,2) (4,2) (2,3) (2,3) (2,3) (2,1)	yes	(8,5)	72	0.7
Tong	4	(4..9,4..9) (4..9,4..9) (4..9,4..9) (4..9,4..9)	yes	(9,9)	288	2.4
Reingold	7	(1..22,1..13)	yes	(22,13)	4	11.6

The size of the rectangles are given as a pair of integers (w, h), where w corresponds to the width and h to the height. If a rotation is allowed, then the size of a given rectangle to pack is either (w, h) or (h, w). In case that a rectangle size is not initially fixed, we give the corresponding minimum and maximum values. The last column shows the time needed on a SUN/SPARC station IPC(12MB) for finding all the solutions and for proving that no other solution exists.

For the first three problems, Table 10 compares CHIP with a C program especially developed for solving floor-planning problems [35]. We give the total number of choice points needed for finding all the solutions and for proving that no other solution exists.

Table 10. Comparison results for the first three problems.

problem	total number of choice points needed by CHIP	total number of choice points needed by a special purpose C program
Pfefferkorn	106	14,051
Laurière	150	96,846
Tong	1,160	660,134

6. CONCLUSION

In this paper, we have introduced the *cumulative* constraint in CHIP in order to improve the efficiency of constraint logic programming languages for solving scheduling and placement problems. Combining this new *cumulative* constraint with the other constraints of CHIP we have achieved the following results:

- For resource allocation problems, it is the first time that tasks with non-fixed duration and amount of resource can be handled in an optimal way in order to have a good overall resource utilization.
- It is the first time that a very simple declarative program was used in order to find the optimal solution for the 10 jobs × 10 machines problem; this contrasts with highly specialized codes which use the dominance criterion and reduction procedures that do not hold when additional constraints have to be considered.
- The link between non-overlapping and the *cumulative* constraint was stressed in [31]. It is the first time that this link is effectively used in order to solve a highly constrained packing problem such as the square packing problem presented in Subsection 5.1.
- For the instance of strip-packing problem presented in Subsection 5.2, the solution of cost 280 found with CHIP is the best solution currently known at this time.
- For the rectangle packing problems where the size of the rectangles is not initially fixed, it is the first time that a constraint logic programming language has outperformed more specific floor-planning programs.
- Finally, we point out that all the previous results were obtained by using only one single new abstraction: the *cumulative* constraint.

We hope that these results will enhance the credibility of constraint logic programming [36].

REFERENCES

1. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Berthier, The constraint logic programming language CHIP, In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, Tokyo, (1988), pp. 693–702.
2. A. Aggoun and N. Beldiceanu, Overview of the CHIP compiler system, In *Proceedings of the Eighth International Conference on Logic Programming*, Paris, (June 1991), (Edited by K. Furukawa), pp. 775–789, The MIT Press, Cambridge, MA.
3. J. Jaffar and J.L. Lassez, Constraint logic programming, In *Proceedings of the Fourteenth ACM Symposium on Principles of Programming Languages*, Munich, (1987).
4. M.S. Fox and K. Sycara, Overview of CORTES: A constraint based approach to production planning, scheduling and control, In *Proceedings of the Fourth International Conference on Expert Systems in Production and Operations Management*, (1990).
5. ROSEAUX, *Exercices et Problèmes Résolus de Recherche Opérationnelle*, T.3, pp. 279–282, Masson, (1985).
6. J.F. Muth and G.L. Thompson, *Industrial Scheduling*, Prentice-Hall, Englewood Cliffs, NJ, (1963).
7. A.J.W. Duijvestijn, Simple perfect squared square of lowest order, *Journal of Combinatorial Theory, Series B* **25** (2), 240–243 (1978).
8. F. Komarnicki and A. Lahrichi, Nouvelles heuristiques pour le problème de découpe à deux dimensions, Presented at the 11^{th} *IFIP Conference on System Modelling and Optimization*, Copenhagen, Denmark, (1983).
9. C.E. Pfefferkorn, A heuristic problem solving design system for equipment or furniture layouts, *Communications of ACM* **18** (5), 286–297 (May 1975).
10. J.L. Laurière, A language and a program for stating and solving combinatorial problems, *Artificial Intelligence* **10** (1), 106–107 (1978).

11. C. Tong, Toward an engineering science of knowledge-based design, *AI in Engineering* **2** (3), 133–166 (July 1987).
12. R. Reingold, A. Yao and B. Sands, Tiling with incomparable rectangles, *Journal of Recreational Mathematics* **8** (2), 112–119 (1975–1976).
13. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun and T. Graf, Applications of CHIP to industrial and engineering problems, Presented at the *First International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Tullahoma, Tennessee, (June 1988).
14. P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, p. 64, The MIT Press, Cambridge, MA, (1989).
15. M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, pp. 236–244, Freeman and Co., San Francisco, CA, (1979).
16. S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi, Optimization by simulated annealing, *Science* **220**, 671–680 (1983).
17. F. Glover, Tabu search, CAAI Report 83-3, University of Colorado, Boulder, CO, (1988).
18. M. Dincbas, H. Simonis and P. Van Hentenryck, Solving large combinatorial problems in logic programming, *Journal of Logic Programming* **8** (1), 75–93 (1990).
19. V. Giard, *Gestion de projets*, p. 50, Edition Economica, Paris, (1991).
20. E. Ignall and L. Schrage, Application of the branch and bound technique to some flow-shop scheduling problem, *Opns. Res. Quart.* **13** (3), 400–412 (1965).
21. E. Balas, Machine sequencing via disjunctive graphs: An implicit enumeration algorithm, *Opns. Res.* **17** (6), 941–957 (1969).
22. K.R. Baker, *Introduction to Sequencing and Scheduling*, Wiley, New York, NY, (1974).
23. A.H.G. Rinnooy Kan, *Machine Scheduling Problem: Classification, Complexity and Computations*, Nyhoff, The Hague, (1976).
24. R.W. Bouma, Job-shop scheduling: A comparison of three enumeration schemes in a branch and bound approach, Master's thesis, Erasmus University, Faculty of Econometrics and Operations Research, Rotterdam, (1982).
25. J. Grabowsky, A new algorithm of solving the flow-shop problem, *Operational Research in Progress* **13** (2), 57–75 (1982).
26. J.R. Barker and G.B. McMahon, Scheduling the general job-shop, *Managment Science* **31** (5), 594–598 (1985).
27. E. Pinson, Le problème de job-shop, Ph.D. Thesis, University of Paris 6, France, (1988).
28. H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA, (1989).
29. F. du Verdier and J.P. Tsang, Un raisonnement spatial par propagation de contraintes, In *11*$^{\text{ièmes}}$ *Journées Internationales: Les Systèmes Experts & leurs Applications*, Avignon 91, pp. 297–314, EC2, Nanterre, (1991).
30. T. Tokuyama, T. Asano and S. Tsukiyama, A dynamic algorithm for placing rectangles without overlapping, *Journal of Information Processing* **14** (1), 30–35 (1991).
31. M. Biró, Object-oriented interaction in resource constrained scheduling, *Information Processing Letters* **36** (2), 65–67 (1990).
32. P. Van Hentenryck and Y. Deville, The cardinality operator: A new logical connective for constraint logic programming, In *Proc. Eighth International Conference on Logic Programming*, Paris, France, (June 1991), (Edited by K. Furukawa), pp. 745–759, The MIT Press, Cambridge, MA.
33. E.G. Coffman and G.S. Lueker, Jr., *Probabilistic Analysis of Packing and Partitioning Algorithms*, pp. 155–168, Wiley, New York, (1991).
34. Baker, Brown and Katseff, A 5/4 algorithm for two-dimensional packing, *Journal of Algorithms* **2** (4), 348–368 (1981).
35. R. Maculet, ARCHIPEL: Intelligence artificielle et conception assistée par ordinateur en architecture. Représentation des connaissances spatiales (Algèbre de Manhattan) et raisonnement spatial avec contraintes, Ph.D. Thesis, p. 183, University of Paris 6, France, (1991).
36. H. Gallaire, Boosting logic programming, In *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, (1987), (Edited by J.-L. Lassez), pp. 928–962, The MIT Press, Cambridge, MA.