# A New Multi-Resource *cumulatives* Constraint with Negative Heights

Nicolas Beldiceanu and Mats Carlsson

SICS, Lägerhyddsvägen 18, SE-75237 Uppsala, Sweden
{nicolas,matsc}@sics.se

May 16 2001
SICS Technical Report T2001:11
ISSN 1100-3154
ISRN: SICS-T--2001:11-SE

**Abstract.** This paper presents a new *cumulatives* constraint which generalizes the original *cumulative* constraint in different ways. The two most important aspects consist in permitting multiple cumulative resources as well as negative heights for the resource consumption of the tasks. This allows modeling in an easy way new scheduling and planning problems. The introduction of negative heights has forced us to come up with new propagation algorithms and to revisit existing ones. The first propagation algorithm is derived from an idea called *sweep* which is extensively used in computational geometry; the second algorithm is based on a combination of *sweep* and *constructive disjunction*, while the last is a generalization of *task intervals* to this new context. A real-life timetabling problem originally motivated this constraint which was implemented within the SICStus finite domain solver and evaluated against different problem patterns.

**Keywords:** Resource Constraint Project Scheduling, Cumulative Constraint, Sweep, Compulsory Part.

# A New Multi-Resource *cumulatives* Constraint with Negative Heights

Nicolas Beldiceanu and Mats Carlsson

SICS, Lägerhyddsvägen 18, SE-75237 Uppsala, Sweden
{nicolas,matsc}@sics.se

**Abstract.** This paper presents a new *cumulatives* constraint which generalizes the original *cumulative* constraint in different ways. The two most important aspects consist in permitting multiple cumulative resources as well as negative heights for the resource consumption of the tasks. This allows modeling in an easy way new scheduling and planning problems. The introduction of negative heights has forced us to come up with new propagation algorithms and to revisit existing ones. The first propagation algorithm is derived from an idea called *sweep* which is extensively used in computational geometry; the second algorithm is based on a combination of *sweep* and *constructive disjunction*, while the last is a generalization of *task intervals* to this new context. A real-life timetabling problem originally motivated this constraint which was implemented within the SICStus finite domain solver and evaluated against different problem patterns.

## 1 Introduction

Within the constraint community, the *cumulative* constraint was originally introduced in [1] in order to model scheduling problems where one has to deal with a single resource of limited capacity. It has the following definition:

$$\text{cumulative}([Origin_1,..,Origin_n],[Duration_1,..,Duration_n],[Height_1,..,Height_n],Limit)\,, \qquad \textbf{(1)}$$

where $[Origin_1,..,Origin_n]$, $[Duration_1,..,Duration_n]$ and $[Height_1,..,Height_n]$ are non-empty lists of non-negative domain variables[1], and $Limit$ is a non-negative integer. The *cumulative* constraint holds if the following condition is true:

$$\forall i \in \mathbb{N} \quad \sum_{j|Origin_j \le i < Origin_j + Duration_j} Height_j \le Limit\,. \qquad \textbf{(2)}$$

From an interpretation point of view, the *cumulative* constraint matches the single resource-scheduling problem [14], where $Origin_1,..,Origin_n$ correspond to the start of each task, $Duration_1,..,Duration_n$ to the duration of each task, and $Height_1,..,Height_n$ to

[1] A domain variable is a variable that ranges over a finite set of integers. The statement *Var*`::min..max`, where `min` and `max` are two integers such that `min` is less than or equal to `max`, creates a domain variable *Var* for which the initial domain is made up from all values between `min` and `max` inclusive.

the amount of resource used by each task. The *cumulative* constraint specifies that, at any instant $i$, the summation of the amount of resource of the tasks that overlap $i$, does not exceed *Limit*.

Over the past years the *cumulative* constraint was progressively integrated within most of the current constraints systems [10], [11], [20], and extended by introducing elastic [11] or continuous [16] resource consumption. It was also used with success as an essential component of a large number of real-life applications involving resource constraints. However, feedback resulting from handling industrial problems has pointed out several serious modeling limitations. Perhaps the most serious limitation concerns the fact that quite often we have more that one cumulative resource [2]. Planning problems [3], [5] also require dealing with tasks for which we don't know the resource in advance. A second major restriction concerns the fact that the amount of resource used by each task is non-negative. By allowing both negative and positive values for the amount of resource used by a task we open the *cumulatives* constraint to producer-consumer problems where a given set of tasks has to cover another set of tasks (i.e. a demand profile). A major modeling advantage comes from the fact that the profile to cover does not necessarily need to be completely fixed in advance which is actually the case for current approaches.

For all the previously mentioned reasons, we present a new constraint called, *cumulatives*, which generalizes the original *cumulative* constraint in 4 different ways:

- First, it allows expressing the fact that we have several cumulative resources and that each task has to be assigned to one of these resources.
- Second, the amount of resource used by each task is a domain variable which can take positive or negative values.
- Third, one can either enforce the cumulated consumption to be less than or equal, or greater than or equal to a given level.
- Finally, on a given resource, the previous constraint on the cumulated resource consumption holds only for those time-points that are overlapped by at least 1 task.

The next section introduces the *cumulatives* constraint as well as several typical utilizations. Sect. 3 provides a detailed description of the main propagation algorithm. It also gives the flavor of an algorithm which combines *sweep* and *constructive disjunction* in order to derive additional pruning. Finally the last section presents the first experimental results on a large selection of typical patterns of the *cumulatives* constraint.

## 2 The *cumulatives* Constraint

The *cumulatives* constraint has the form $\text{cumulatives}(Tasks, Resources, Constraint)$, where:

- *Tasks* is a collection of tasks where each task has a *Machine*, an *Origin*, a *Duration*, a *Height* and an *End* attribute; *Duration* is a non-strictly negative domain variable, while *Machine*, *Origin*, *Height* and *End* are domain variables which may be negative, positive or zero.

– *Resources* is a collection of resources where each resource has an *Identifier* and a given *Limit*; the *Identifier* and *Limit* attributes are fixed integers which may be negative, positive or zero; Moreover the *Identifier* is a unique value for each resource; the *Machine* attribute of a task is a domain variable for which the possible values correspond to values of the *Identifier* attribute.
– *Constraint* is the less or equal (i.e. ≤) or the greater or equal (i.e. ≥) constraint.

In the rest of this paper we denote by $|C|$ the number of items of a collection $C$ and $a[i]$ the value of the attribute $a$ of the $i^{th}$ item of collection $C$. The *cumulatives* constraint holds if the two following conditions are both true:

$$\forall t \in [1,|Tasks|]:\ Origin[t] + Duration[t] = End[t]\,. \qquad (\mathbf{3})$$

$\forall t \in [1,|Tasks|],\ \forall i \in [Origin[t], End[t]-1]$:

Let $m$ be the unique value such that: $Machine[t] = Identifier[m]$,

$$\sum_{j \,\Big|\, \begin{cases} Origin[j] \le i < End[j] \\ Machine[j] = Machine[t] \end{cases}} (Height[j])\ \ Constraint\ \ Limit[m]\,. \qquad (\mathbf{4})$$

Condition (**3**) imposes for each task $t$ the fact that its end is equal to the sum of its origin and its duration. When *Constraint* is equal to ≤ (respectively ≥), Condition (**4**) enforces that, for each instant $i$ that is overlapped by at least one task $t$, such that $t$ is assigned to resource $m$, the sum of the *Height* attribute of all tasks $j$, that both are assigned on resource $m$ and overlap instant $i$, is less or equal (respectively greater or equal) than the *Limit* [2] attribute of resource $m$.

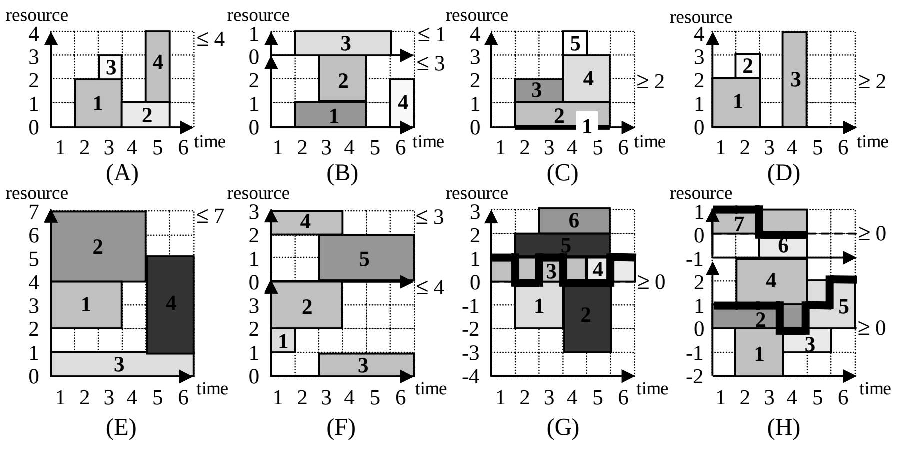


**Fig. 1.** Typical uses of the *cumulatives* constraint

[2] When *Constraint* is equal to ≥ (respectively ≤), the *Limit* attribute should be interpreted as the *minimum level* to reach on a given resource (respectively the *maximum capacity* of a given resource).

Fig. 1 gives 8 ground instances[3] of *cumulatives* , all related to a typical utilization of the constraint. Each resource is represented as a drawing where the horizontal and the vertical axes respectively correspond to the time and to the amount of used resource. A task is represented as a rectangle for which the length and the height respectively match the duration and the absolute value of the amount of resource used by the task. The position of the leftmost part of a rectangle on the time axis is the start of the task. Tasks with positive or negative heights are respectively drawn on top or below the time axis, so that we can look both to the cumulated resource consumption of the tasks with a positive height and to the cumulated consumption of the tasks with a negative height. Finally the *Constraint* parameter and the *Limit* attribute are mentioned to the right of each resource.

– Part (A) is the classical original *cumulative* constraint described in [1]. In the *Resources* parameter we have introduced one single resource, which has value 1 as identifier and 4 as its maximum capacity.
– Part (B) is an extension of the original *cumulative* constraint where we have more than one single resource.
– Part (C) is the *at least* variant of the *cumulative* constraint available in CHIP. This variant enforces to reach a minimum level between the first[4] and the last[5] utilization of the resource. In order to express the previous condition we create a dummy task of height 0 (represented by the thick line between instants 2 and 5) for which the start and the end respectively correspond to the earliest start and to the latest end of the different tasks to schedule. For this purpose we respectively use a *minimum* and *maximum*[6] constraints [6].
– Part (D) is a new variant of the previous case where the "*at least*" constraint applies only for the instants that are overlapped by at least one task.
– Part (E) is a producer-consumer problem [18] where tasks 1,2 represent producers, while tasks 3,4 are consumers. On one side, a producer task starts at the earliest start and produces a quantity equal to its height at a date that corresponds to its end. On the other side, a consumer task ends at the latest end and consumes a quantity equal to its height at a date that matches its start. The resource can be interpreted as a tank in which one adds or removes at specific points in time various quantities. The *cumulatives* constraint enforces that, at each instant, one does not consume more than what is currently available in the tank.
– Part (F) is a generalization of the previous producer-consumer problem where we have two tanks. As for the previous example the *cumulatives* constraint enforces no negative stocks on both tanks.
– Part (G) describes a covering problem where one has to cover a given workload by a set of tasks. The workload can be interpreted as the number of persons required

---

[3] Ground instances correspond to solutions of the *cumulatives* constraint. However one should keep in mind that all the attributes of a task may not be fixed when the constraint is posted.

[4] The minimum of the earliest start of the tasks.

[5] The maximum minus one of the latest end of the tasks.

[6] The minimum($M$,$\{X_1,..,X_n\}$) (respectively maximum($M$,$\{X_1,..,X_n\}$) constraint holds if $M$ is the minimum (respectively maximum) value of variables $X_1,..,X_n$.

during specific time intervals, while a task can be interpreted as the work performed by a group of persons. The height of the initially fixed tasks (i.e. tasks 1 and 2) that represent the workload is modelled with negative numbers, while the height of the tasks related to the persons (i.e. tasks 3,4,5,6) is positive. The covering constraint is imposed by the fact that, at each point in time, the *cumulatives* constraint enforces the cumulated height, of the tasks that overlap this point, to be greater than or equal to 0: at each point in time the number of available persons should be greater than or equal to the required demand expressed by the work-load to cover. A thick line indicates the cumulated profile resulting from the negative and positive heights.

– Finally, part (H) generalizes (G) by introducing 2 distinct workloads to cover.

## 3 Propagation Algorithms

The purpose of this section is to introduce two algorithms used for implementing the *cumulatives* constraint. The first algorithm, based on the idea of *sweep*, is required for checking[7] the constraint and for doing some basic pruning, while the second algorithm performs additional pruning by using *constructive disjunction* [12], [19]. We have also generalized *task intervals* [4], [11], [13] to the case where negative resource consumption (i.e. production of resource) is also allowed. Since the "at least" and "at most" sides of the *cumulatives* constraint are symmetric, we only focus on the "at least" side[8] where the constraint enforces for each resource to reach a given minimum level.

Before going further into the presentation of the algorithms, let us first introduce some notions which will be used in the different algorithms. A task $t$ of origin $Origin[t]$ and end $End[t]$ has a *compulsory part*[9] [15] if its latest start $\max(Origin[t])$ is strictly less than it earliest end $\min(End[t])$. For such a task, we call *support of the compulsory part* the interval $[\max(Origin[t]), \min(End[t])-1]$.

Within the different algorithms, this set of functions access domain variables:

– $\min(var)$ and $\max(var)$ respectively return the minimum and maximum value of a given domain variable $var$.
– ISINT( $var$ ) returns 1 if the variable $var$ is fixed, and 0 otherwise.
– FIXVAR( $var, val$ ) fixes variable $var$ to value $val$.
– REMOVEVALUEVAR( $var, val$ ) removes value $val$ from the domain variable $var$.

---

7 It is based on a necessary condition that is also sufficient when all the attributes of the different tasks are fixed.

8 To get the algorithm for the "at most" side one has to replace in the forthcoming algorithms "max(*Height*[*t*])" by "min(*Height*[*t*])", "max(*Height*[*t*])<" by "min(*Height*[*t*])>", "<*Limit*[*r*]" by ">*Limit*[*r*]", "≥*Limit*[*r*]" by "≤*Limit*[*r*]", "ADJUSTMINVAR(*Height*[*t*]" by "ADJUSTMAXVAR(*Height*[*t*]" and "max(0,*Limit*[*r*])" by "min(0,*Limit*[*r*])".

9 When *Constraint* is equal to ≥ (respectively ≤), the *compulsory part* of a task *t* is the maximum (respectively minimum) resource consumption of that task together with the interval [max(Origin[*t*]),min(End[*t*])–1].

- ADJUSTMINVAR( $var, val$ ) and ADJUSTMAXVAR( $var, val$ ) respectively adjust the minimum and maximum value of a given domain variable $var$ to value $val$.
- PRUNEINTERVALVAR( $var, low, up$ ) removes the interval of values $[low,up]$ from a given domain variable $var$.

The last five functions return fail if a contradiction was found (i.e. the domain of the pruned variable $var$ becomes empty), or return delay otherwise.

## 3.1 The *Sweep* Algorithm

The *sweep* algorithm is based on an idea that is widely used in computational geometry and that is called sweep [8, page 22], [17, pages 10-11]. Within constraint programming, sweep has also been used in [7] for implementing different variants of the non-overlapping constraint.

In dimension 2, a plane *sweep* algorithm solves a problem by moving a vertical line from left to right. The algorithm uses two data structures:

- A data structure called the *sweep-line status*, which contains some information related to the current position $\Delta$ of the vertical line,
- A data structure named the *event point series*, which holds the events to process, ordered in increasing order according to the abscissa.

The algorithm initializes the sweep-line status for the starting position of the vertical line. Then the line "jumps" from event to event; each event is handled and inserted or removed from the sweep-line status. In our context, the sweep-line scans the time axis on a given resource $r$ in order to build an optimistic[10] cumulated profile (i.e. the sweep-line status) for that resource $r$ and to perform check and pruning according to this profile and to the limit attribute of resource $r$. This process is repeated for each resource present in the second argument of the *cumulatives* constraint. Before going further into any detail, let us first give the intuition of the *sweep* algorithm on the simple case where all tasks are fixed.

Consider the illustrative example given in Fig. 2, which is associated to instance (D) of Fig. 1. Since all the tasks of the previous constraint are fixed, we want to check that, for each time point *i* where there is at least one task, the cumulated height of the tasks that overlap *i* is greater than or equal to 2. The sweep-line status records the following counters $sum\_height$ and $nb\_task$, which are initially set to 0; $sum\_height$ is the sum of the height of the tasks that overlap the current position $\Delta$ of the sweep-line, while $nb\_task$ is the number of such tasks.

Since we don't want to check every time-point, the event points correspond only to the start and to the end of each task:

- For the start of each task *t* we generate a start-event, which will respectively increment by $Height[t]$ and 1 the two previous counters $sum\_height$ and $nb\_task$.
- For the end of each task *t* we generate an end-event, which will respectively decrement by $Height[t]$ and 1 $sum\_height$ and $nb\_task$.

[10] Since the constraint enforces to reach a given minimum level, we assume that hopefully, each task will take its maximum height.

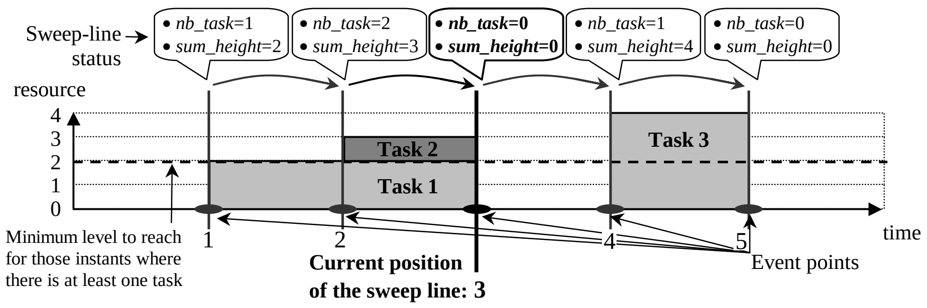


**Fig. 2.** Illustrative example of the *sweep* algorithm when all tasks are fixed

We initially generate all these events, sort them in increasing order and finally handle them as explained before. Each time we finish to handle all the events associated to a given date, and only when $nb\_task$ is strictly greater than 0, we check that $sum\_height$ is greater than or equal to the required minimum level.

The next paragraph introduces the sweep-line status and the event points we consider when the tasks are not yet fixed, while Sect. 3.1.2 explains how to prune the attributes of the tasks according to the sweep-line status.

### 3.1.1 Checking for Failure

Given that we want to catch situations where, for sure there is no solution to the *cumulatives* constraint, the *sweep* algorithm assumes that each task will take its maximum height[11] in order to facilitate to reach the required minimum level. For a given resource $r$, let us first introduce the following sets that will be used later on:

– $CHECK_r$ is the set of tasks, that simultaneously have a compulsory part, are assigned to $r$, and have a maximum height that is strictly less than $\max(0, Limit[r])$.
– $BAD_r$ is the set of tasks, that have a compulsory part, are assigned to resource $r$ and have a strictly negative maximum height.
– $GOOD_r$ is the set of tasks, that may be, or are actually assigned to resource $r$ and have a maximum height that is strictly greater than 0.

**Time-Points to Check** Since the *cumulatives* constraint considers only those instants where there is at least one task, the *sweep* algorithm has to perform a check only for those instants, that correspond to the support of the compulsory part of a task assigned to resource $r$. Moreover no check is required for those instants where we only have compulsory part of tasks for which the maximum height is greater than or equal to $\max(0, Limit[r])$. This is because the cumulated maximum height of such tasks will always be greater than or equal to the minimum required level $Limit[r]$. A first counter called $nb\_task_r$ is associated to the sweep-line status of resource $r$. It

[11] Remember that we present the "at least" side where, for each resource, the *cumulatives* constraint enforces to reach a given minimum level.

gives the number of tasks of the set $CHECK_r$ for which the compulsory part intersects the current position $\Delta$ of the sweep-line.

**Building the Optimistic Profile** For a given resource $r$, the optimistic cumulated profile is obtained by considering two kinds of contributions:

– The contributions of the tasks that belong to the set $BAD_r$; since their maximum height is strictly negative, the contribution of these tasks in the cumulated profile is bad from the point of view of the minimum level to reach. This is why we count such contribution only for those instants where it occurs for sure, that is for the support of the compulsory part of such tasks.

– The contributions of the tasks that belong to $GOOD_r$; since it is strictly positive, the contributions in height of these tasks in the cumulated profile can help to reach the required minimum level. This is why, it is counted for all instants where such a task can be placed, that is between its earliest start and its latest end.

The sum of the previous contributions is recorded in a second counter, denoted $sum\_height_r$, which is also associated to the status of the sweep-line corresponding to resource $r$. It gives for the current position of the sweep-line, the sum of the maximum height of the tasks *t* that satisfy one of the following conditions:

– Task *t* belongs to $BAD_r$ and $\max(Origin[t]) \le \Delta < \min(End[t])$,

– Task *t* belongs to $GOOD_r$ and $\min(Origin[t]) \le \Delta < \max(End[t])$.

**Recording the Tasks to Prune** In order to prepare the pruning phase we store the tasks that can intersect in time the current position $\Delta$ of the sweep line. For this purpose the sweep-line status contains an additional stack $stack\_prune_r[1..top\_prune_r]$ [12] that records these tasks.

**Table 1.** Summary of the different types of events

| Conditions for Generating an Event | Generated Events |
|---|---|
| max(*Origin*[*t*]) < min(*End*[*t*]) **and** max(*Height*[*t*]) < max(0,*Limit*[*r*]) **and** min(*Machine*[*t*]) = max(*Machine*[*t*]) = *r* | ⟨**check**, *t*, max(*Origin*[*t*]), 1⟩ ⟨**check**, *t*, min(*End*[*t*]), -1⟩ |
| max(*Origin*[*t*]) < min(*End*[*t*]) **and** min(*Machine*[*t*]) = max(*Machine*[*t*]) = *r* **and** max(*Height*[*t*]) < 0 | ⟨**profile**, *t*, max(*Origin*[*t*]), max(*Height*[*t*])⟩ ⟨**profile**, *t*, min(*End*[*t*]), –max(*Height*[*t*])⟩ |
| r∈*Machine*[*t*] **and** max(*Height*[*t*]) > 0 | ⟨**profile**, *t*, min(*Origin*[*t*]), max(*Height*[*t*])⟩ ⟨**profile**, *t*, max(*End*[*t*]), –max(*Height*[*t*])⟩ |
| r∈*Machine*[*t*] **and** ¬(ISINT(*Origin*[*t*]) **and** ISINT(*End*[*t*]) **and** ISINT(*Machine*[*t*]) **and** ISINT(*Height*[*t*])) | ⟨**pruning**, *t*, min(*Origin*[*t*]), 0⟩[13] |

**Updating the Sweep-Line Status** Table 1 depicts the events that update the different constituents of the sweep-line status introduced so far. We respectively call

[12] $top\_prune_r$ indicates how many tasks are recorded within the array $stack\_prune_r[\,]$.

[13] There is no event for removing the tasks that can't be pruned any more (i.e. the tasks that can't intersect the current position of the sweep-line); this is achieved by the pruning procedure itself by compacting the stack $stack\_prune_r$.

*check*, *profile* or *pruning* event, an event modifying $nb\_task_r$, $sum\_height_r$ or $stack\_prune_r[1..top\_prune_r]$. We choose to encode these events by using the following fields $\langle type, task, date, increment \rangle$, where:

– *type* tells whether we have a *check*, a *profile* or a *pruning* event; in case of a *pruning* event the content of the slot *increment* is irrelevant,
– *task* indicates the task which generate the event,
– *date* specifies the location in time of the event,
– *increment* gives the quantity to add to $nb\_task_r$ or to $sum\_height_r$.

**The *Sweep* Algorithm** The previous events are initially generated and sorted in increasing order of their respective date. For each date, Algorithm 1 (lines 9, 10) process all the corresponding events; in addition when $nb\_task_r$ is different from 0 it checks that the height $sum\_height_r$ of the optimistic cumulated profile is greater than or equal to the minimum level of resource $r$ (lines 7, 12).

```
1   Set $nb\_task_r$, $sum\_height_r$ and $top\_prune_r$ to 0.
2   Extract the next event &$\langle type,task,date,increment \rangle$.[14]
3   Set $d$ to $date$.
4   while $\langle type,task,date,increment \rangle \neq$ NULL do
5       if $type \neq pruning$ then
6           if $d \neq date$ then
7               if $nb\_task_r > 0$ and $sum\_height_r < Limit[r]$ then return fail.
8               Set $d$ to $date$.
9           if $type = check$ then Add $increment$ to $nb\_task_r$ else Add $increment$ to $sum\_height_r$
10      else Set $top\_prune_r$ to $top\_prune_r+1$ and set $stack\_prune_r[top\_prune_r]$ to $task$.
11      Extract the next event &$\langle type,task,date,increment \rangle$.
12  if $nb\_task_r > 0$ and $sum\_height_r < Limit[r]$ then return fail.
```

**Alg. 1.** Main loop of the *sweep* algorithm on a resource *r*

### 3.1.2 Pruning the Attributes of a Task

```
1   if $nb\_task_r = 0$ or $sum\_height_r - contribution[t]$[15] $\geq Limit[r]$ then return delay.
2   if  FIXVAR($Machine[t]$, $r$)=fail
3   or ADJUSTMINVAR($Origin[t]$, $up - \max(Duration[t]) + 1$)=fail
4   or ADJUSTMAXVAR($Origin[t]$, $low$)=fail
5   or ADJUSTMAXVAR($End[t]$, $low + \max(Duration[t])$)=fail
6   or ADJUSTMINVAR($End[t]$, $up+1$)=fail
7   or ADJUSTMINVAR($Duration[t]$, $\min(up - \max(Origin[t]) + 1, \min(End[t]) - low)$)=fail then return fail.
```

**Alg. 2.** Pruning the attributes of task *t* in order to enforce to cover an interval [*low*,*up*]

At each position of the sweep-line related to resource $r$ we perform the three following kind of pruning[16] of the attributes of the tasks that both can be assigned to re-

[14] We assume that we get a pointer to an event.
[15] *contribution*[*t*] stands for the saved maximum height of task *t* that was added to $sum\_height_r$.
[16] These three types of pruning are inserted just after line 7 and line 12 of Algorithm 1.

source $r$ and can overlap the interval [*low*,*up*] where *low* and *up*+1 are respectively the current and the next position of the sweep-line:

– A first pruning is tried out for those tasks *t* with a maximum height greater than 0 (Algorithm 2). It consist in pruning the *Machine*, *Origin*, *End* and *Duration* attributes of those tasks which are absolutely required (line 1) in order to reach a given minimum level $Limit[r]$ on interval [*low*,*up*]. More precisely we fix the *Machine* attribute to resource $r$ (line 2), and prune the minimum and maximum values of the *Origin* and *End* attributes (lines 3-6) in order to remove values for which there is no way to cover all instants of interval [*low*,*up*]. We also adjust the minimum duration (line 7) for a similar reason.
– A second pruning is undertaken for those tasks with a maximum height strictly less than $\max(0, Limit[r])$ (Algorithm 3). It consists in pruning the *Machine*, *Origin*, *End* and *Duration* attributes of those tasks *t* which would prevent to reach the minimum required level (line 1) on interval [*low*,*up*] if they would simultaneously overlap the previous interval and utilize resource $r$. For such tasks we first forbid to assign them on resource $r$ if they overlap for sure interval [*low*,*up*] (lines 2-3). Conversely, if they are assigned to $r$ we prune the *Origin*, *End* and *Duration* attributes in order to prevent any overlapping with interval [*low*,*up*] (lines 4-9).

```
1  if $sum\_height_r - contribution[t] + \max(Height[t]) < Limit[r]$ then
2      if $\min(End[t]) > low$ and $\max(Origin[t]) \leq up$ and $\min(Duration[t]) > 0$ then
3          if REMOVEVALUEVAR($Machine[t]$, $r$)=fail then return fail.
4      else if ISINT($Machine[t]$)[17] then
5          if $\min(Duration[t]) > 0$ then
6              if PRUNEINTERVALVAR($Origin[t]$, $low - \min(Duration[t]) + 1$, $up$)=fail
7              or PRUNEINTERVALVAR($End[t]$, $low+1$, $up + \min(Duration[t])$)=fail then return fail.
8          Set $maxd$ to $\max(low - \min(Origin[t]), \max(End[t]) - up - 1, 0)$.
9          if ADJUSTMAXVAR($Duration[t]$), $maxd$)=fail then return fail.
```

**Alg. 3.** Pruning the attributes of task *t* in order to forbid to intersect an interval [*low*,*up*]

– Finally a third pruning (Algorithm 4) is performed for all tasks *t* that are assigned to resource $r$ (line 1) and that overlap for sure interval [*low*,*up*] (line 2). It consists in removing from the *Height* attribute those values which would prevent to reach the minimum required level on interval [*low*,*up*] (line 3).

```
1  if  ISINT($Machine[t]$)
2  and $\min(End[t]) > low$ and $\max(Origin[t]) < up$ and $\min(Duration[t]) > 0$ then
3      if ADJUSTMINVAR($Height[t]$, $Limit[r] - (sum\_height_r - contribution[t])$)=fail then return fail.
```

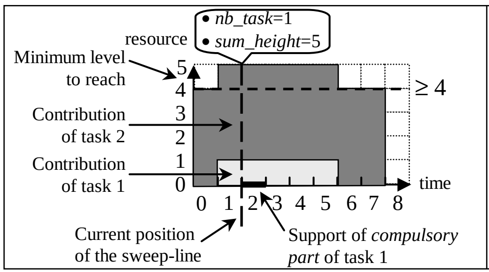
**Alg. 4.** Pruning the minimum resource consumption of task *t* according to interval [*low*,*up*]

**A Complete Illustrative Example of Pruning** Let us illustrate the previous pruning rules on the instance given in Example 1, where all the 3 types of pruning occur simultaneously. Lines 1 and 2 declare the minimum and maximum value for each attribute of the tasks. Lines 3 to 6 state a *cumulatives* constraint where we have two tasks $t_1$ and $t_2$ (see lines 3,4) and two resources (see lines 5,6). The third argument indicates that we have to reach a given minimum level on those 2 resources.

[17] Since by hypothesis we only try to prune tasks that can be assigned to resource *r*, the test ISINT(*Machine*[*t*]) means that task *t* is actually assigned to resource *r*.

```
M1::1..1   O1::1..2   D1::2..4   E1::3..6   H1::-1..1
M2::1..2   O2::0..6   D2::0..2   E2::0..8   H2::-3..4
cumulatives({machine-M1 origin-O1 duration-D1 end-E1 height-H1,
             machine-M2 origin-O2 duration-D2 end-E2 height-H2},
            {identifier-1 limit-4,
             identifier-2 limit-3}, ≥)
```

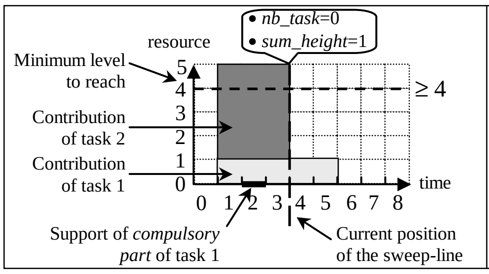
**Example 1.** An instance that triggers the three different types of pruning



(A) *Optimistic profile* during first sweep on resource 1

| | |
|---|---|
| $M_2 = 1$ | (line 2 of Algorithm 2) |
| $\min(O_2) \geq 1$ | (line 3 of Algorithm 2) |
| $\max(O_2) \leq 2$ | (line 4 of Algorithm 2) |
| $\max(E_2) \leq 4$ | (line 5 of Algorithm 2) |
| $\min(E_2) \geq 3$ | (line 6 of Algorithm 2) |
| $\min(D_2) \geq 1$ | (line 7 of Algorithm 2) |
| $\min(H_2) \geq 3$ | (line 3 of Algorithm 4) |
| $\min(H_1) \geq 0$ | (line 3 of Algorithm 4) |

(B) Pruning according to interval [2,2] on resource 1



(C) *Optimistic profile* during second sweep on resource 1

| | |
|---|---|
| $[5,7] \notin E_1$ | (line 7 of Algorithm 3) |
| $\max(D_1) \leq 3$ | (line 9 of Algorithm 3) |

(D) Pruning according to interval [4,5] on resource 1

**Fig. 3.** Pruning according to the optimistic profile

When inspecting interval $[2,2]$ while making the first sweep on resource 1 (see part (A) of Fig. 3), we have $CHECK_1 = \{t_1\}$, $BAD_1 = \varnothing$, $GOOD_1 = \{t_1, t_2\}$ and the sweep-line status is as follows: $nb\_task_1 = 1$, $sum\_height_1 = 5$. The contributions of tasks $t_1$ and $t_2$ in the optimistic profile are respectively equal to their maximum height, namely 1 and 4. Since task $t_1$ is assigned to resource 1 and has a compulsory part on interval $[2,2]$, we have to reach the required minimum level 4 on interval $[2,2]$. Because $t_2$ is the only candidate that can help to reach this minimum level 4, Algorithm 2 performs the pruning mentioned in the first six lines of part (B) in order to force $t_2$ both to be assigned to resource 1 and to overlap interval $[2,2]$. Since now tasks $t_1$ and $t_2$ are both assigned to resource 1, and overlap for sure interval $[2,2]$, Algorithm 4 adjusts the minimum height of $t_1$ and $t_2$ (see the last 2 lines of part (B)) in order to reach the minimum required level 4.

A second sweep on resource 1 (see part (C) of Fig. 3) is performed in order to saturate and part (D) describes the corresponding pruning.

**Complexity of the *Sweep* Algorithm** Let $m$ be the number of resources, $n$ the total number of tasks and $p$ the number of tasks for which at least one attribute is not fixed. First note that updating the sweep-line status and performing the pruning actions (Algorithms 2, 3 and 4) [18] is done in $O(1)$. Second, given that a task can generate at most 7 events (see Table 1), the total number of events is proportional to the number of tasks $n$. Since we first sort all these events and scan through them in order to update the sweep-line status the complexity of the check procedure on a resource is $O(n \cdot \log n)$. As the total number of calls to the pruning algorithms is less than or equal to the total number of events and since during a pruning step we consider at most $p$ tasks, the complexity of the pruning on a resource is $O(n \cdot p)$. As we have $m$ resources, the overall complexity of the *sweep* algorithm is $O(m \cdot n \cdot \log n + m \cdot n \cdot p)$.

### 3.2 Additional Pruning Algorithms

A second algorithm based on *constructive disjunction* [12], [19] and on some of the propagation rules (lines 5-9 of Algorithm 3 and lines 2-3 of Algorithm 4) of the previous pruning algorithms was implemented. For a not yet assigned task, it consists first in making the hypothesis that it is assigned on its possible resources and in performing the mentioned pruning rules. In a second phase we remove those values that were rejected in all the previous hypotheses. Finally we mention that we have generalized *task intervals* [11] to the facts that we both have negative heights and more than one machine.

## 4 Experimental Results

**Benchmark Description** As it was already shown in Sect. 2, the *cumulatives* constraint can model a large number of situations. Rather than focusing on a specific problem type, we tried to get some insight of the practical behavior of the sweep algorithm on a large spectrum of problem patterns. For this purpose we generated 576 different problems patterns by combining the possible ways to generate a problem instance as shown by Table 2.

For each problem pattern we generated 20 random instances with a fixed density for the resource consumption and computed the median time for running a limited discrepancy search of order 1 for 50, 100, 150 and 200 tasks. Benchmarks were run on a 266 Mhz Pentium II processor under Linux with a version of SICStus Prolog compiled with `gcc` version `2.95.2` (`-O2`). Our experiments were performed by fixing the tasks according to their initial ordering; within a task, we successively fixed the resource consumption, the machine, the origin and the duration attributes.

**Analysis** Parts (A) and (B) of Fig. 4 report on the best, median and worst patterns when most of the tasks are not fixed as well as when most of the tasks are fixed. A

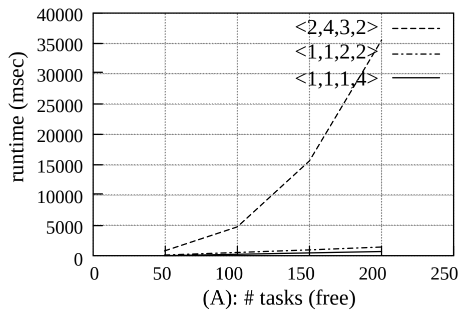
[18] We assume that all operations on domain variables are performed in $O(1)$.

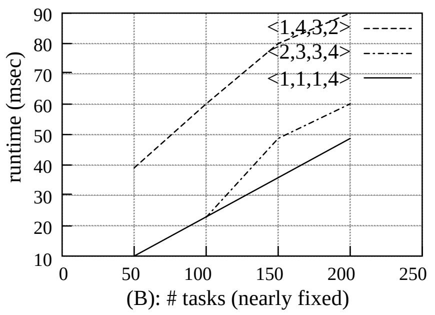
problem pattern is denoted by a tuple $\langle Origin, Duration, Resource, Machine \rangle$ where each position indicates the way to generate the corresponding characteristic. For instance part (A) tells us that, when all tasks are nearly free, the combination "random earliest start", "variable small or large duration", "positive or negative resource consumption" and "full set of machines" is the most time-consuming one.

Finally parts (C) to (F) show for each way to generate a task attribute the total median time over all problem patterns that effectively use this given method divided by the number of problem patterns. For example, (D) gives the following ordering (in increasing time) for the different way to generate the duration attribute: "fixed small duration", "variable small duration", "fixed small or large duration", "variable small or large duration".

**Table 2.** Generation of the characteristics of a problem pattern (assuming ≤ constraint)

| Characteristic | Ways to Generate the Different Characteristic of a Problem Pattern | |
|---|---|---|
| Origin | 1 (**full origin**): | min=1, max[19]=horizon. |
| | 2 (**random earliest start**): | min=random(1,0.9xhorizon)[20] max=horizon. |
| | 3 (**fixed origin**): | min=max=random(1,0.9xhorizon). |
| | $horizon = h_1 \cdot h_2 \cdot (\text{average minimum duration})$ | |
| | $h_1 = 50$ if one machine, 5 otherwise | $h_2 = 0.5$ if exist tasks with negative height, 1 otherwise |
| Duration | 1 (**fixed small duration**): | min=max=random(0,10). |
| | 2 (**fixed small or large duration**): | min=max=random(0,200). |
| | 3 (**variable small duration**): | min=random(0,10), max=min+random(0,5) |
| | 4 (**variable small or large duration**): | min=random(0,100), max=min+random(0,100) |
| Resource consumption | 1 (**fixed consumption**): | min=max=random(0,10) |
| | 2 (**variable consumption**): | min=random(0,7), max=min+random(0,5) |
| | 3 (**variable positive or negative consumption**): | min=random(-10,0), max=min+random(0,10) |
| Machine | 1 (**single machine**): | min=max=1 |
| | 2 (**full set of machines**): | min=1, max=10 |
| | 3 (**subset of machines**): | min=random(1,10), max=min+random(0,10) |
| | 4 (**fixed machine with several machines**): | min=max=random(1,10) |
| Task | 1 (**nearly fixed**): | at most 5 tasks are not completely fixed |
| | 2 (**nearly free**): | nearly all tasks are not fixed |





[19] `min` and `max` stand for the minimum and maximum values of the domain variable for which we generate the initial domain.

[20] `random`(*low*,*up*) stands for a random number *r* such that *low*≤*r*≤*up*.

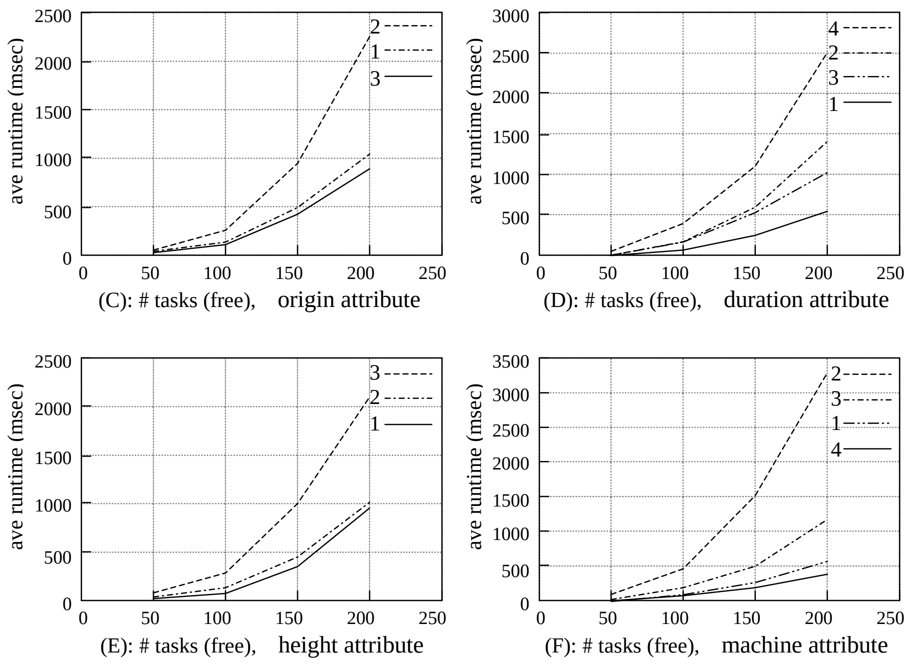


**Fig. 4.** Performance evaluation of the sweep algorithm (each legend is ordered by decreasing runtime)

## 5 Conclusion

A first contribution of this paper is to come up with a definition of the *cumulatives* constraint which for the first time completely unifies the “at most” and “at least” sides of the constraint. Surprisingly enough different variants of the “at least” side of the *cumulative* constraint were introduced in different constraint systems but, to our best knowledge, nothing was published on this topic, and one may assume that distinct algorithms were used for the “at least” and the “at most” sides. In contrast our approach allows coming up with the same algorithm for both sides. Moreover this algorithm assumes neither the duration nor the resource consumption to be fixed, and provides pruning for all the different types of attributes of a task. A second major contribution from the modeling point of view, which was never considered before, neither in the constraint nor in the operation research community [9], [14], is the introduction of negative heights for the quantity of resource used by the tasks. This opens the *cumulatives* constraint to new producer consumer problems [18] or to new problems where a set of tasks has to cover several given profiles, which may not be completely fixed initially.

## Acknowledgements

Thanks to Per Mildner for useful comments on an earlier draft of this paper.

## References

1. Aggoun, A., Beldiceanu, N.: Extending CHIP to solve Complex Scheduling and Packing Problems. Mathl. Comput. Modelling, 17(7), pages 57-73, (1993).
2. Artigues, C., Roubellat, F.: A polynomial activity insertion algorithm in a multi-resource schedule with cumulative constraints and multiple modes. In *European Journal of Operational Research (EJOR)*, 127, pages 297-316, (2000).
3. Barták, R.: Dynamic Constraint Models for Planning and Scheduling Problems. In *New Trends in Constraints* (Papers from the Joint ERCIM/Compulog-Net Workshop, Cyprus, October 25-27, 1999), LNAI 1865, Springer Verlag, (2000).
4. Baptiste, P., Le Pape, C., Nuijten, W.: Satisfiability Tests and Time-Bound Adjustments for Cumulative Scheduling Problems. In *Annals of Op.Research*, 92, pages 305-333, (1999).
5. Beck, J. C., Fox, M. S.: Constraint-directed techniques for scheduling alternative activities. In *Artificial Intelligence 121*, pages 211-250, (2000).
6. Beldiceanu, N.: Pruning for the *minimum* Constraint Family and for the *number of distinct values* Constraint Family. In *Third International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CP-AI-OR'2001*, (2001).
7. Beldiceanu, N.: Sweep as a generic pruning technique. In *TRICS: Techniques foR Implementing Constraint programming*, CP2000, Singapore (2000).
8. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: *Computational Geometry – Algorithms and Applications*. Springer, (1997).
9. Brucker, P., Drexl, A., Möhring, R., Neumann, K., Pesch, E.: Resource-constrained project scheduling: Notation, classification, models and methods, in *EJOR 112*, pages 3-41, (1999).
10. Carlsson, M., Ottosson G., Carlson, B.: An Open-Ended Finite Domain Constraint Solver. *Proc. Programming Languages: Implementations, Logics, and Programs*, vol. 1292 of Lecture Notes in Computer Science, pages 191-206, Springer-Verlag, (1997).
11. Caseau, Y., Laburthe, F.: Cumulative Scheduling with Task Intervals. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, MIT Press, (1996).
12. De Backer, B., Beringer, A.: A CLP Language Handling Disjunctions of Linear Constraints. In *Proc. 10th International Conference on Logic Programming*, pages 550-563, (1993).
13. Erschler, J., Lopez, P.: Energy-based approach for task scheduling under time and resources constraints. In *2nd International Workshop on Project Management and Scheduling*, pages 115-121, Compiègne (France), June 20-22, (1990).
14. Herroelen, W., Demeulemeester, E., De Reyck, B.: A Classification Scheme for Project Scheduling Problems. in: Weglarz J. (Ed.), *Handbook on Recent advances in Project Scheduling*, Kluwer Academic Publishers, (1998).
15. Lahrichi, A.: *Scheduling: the Notions of Hump, Compulsory Parts and their Use in Cumulative Problems.* in: *C. R. Acad. Sc.* Paris, t. 294, pages 209-211, (1982).
16. Poder, E., Beldiceanu, N., Sanlaville, E.: Computing the Compulsory Part of a Task with Varying Duration and Varying Resource Consumption. Submitted to *European Journal of Operational Research (EJOR)*, (February 2001).
17. Preparata, F. P., Shamos, M. I.: *Computational geometry. An introduction.* Springer-Verlag, (1985).

18. Simonis, H., Cornelissens, T.: Modelling Producer/Consumer Constraints. In *CP'95, First International Conference,* CP'95, Cassis, France, September 19-22, 1995, Proceedings. Lecture Notes in Computer Science, Vol. 976, Springer, pages 449-462, (1995).
19. Van Hentenryck, P., Saraswat, V., Deville, Y.: Design, Implementation and Evaluation of the Constraint Language cc(*FD*). In *A. Podelski, ed., Constraints: Basics and Trends*, vol. 910 of Lecture Notes in Computer Science, Springer-Verlag, (1995).
20. Würtz, J.: Oz Scheduler: A Workbench for Scheduling Problems. In *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence*, Nov16--19 1996, IEEE Computer Society Press, (1996).