Implementing Cumulative Functions with Generalized Cumulative Constraints

PIERRE SCHAUS*, UCLouvain, ICTEAM, Belgium
CHARLES THOMAS, UCLouvain, ICTEAM, Belgium
ROGER KAMEUGNE, UCLouvain, ICTEAM and University of Maroua, Belgium / Cameroon

Modeling scheduling problems with conditional time intervals and cumulative functions has become a common approach when using modern commercial constraint programming solvers. This paradigm enables the modeling of a wide range of scheduling problems, including those involving producers and consumers. However, it is unavailable in existing open-source solvers and practical implementation details remain undocumented. In this work, we present an implementation of this modeling approach using a single, generic global constraint called the Generalized Cumulative. We also introduce a novel time-table filtering algorithm specifically designed to handle tasks defined on conditional time-intervals. Experimental results demonstrate that this approach, combined with the new filtering algorithm, performs competitively with existing solvers enabling the modeling of producer and consumer scheduling problems and effectively scales to large-scale problems.

1 Introduction

The success of Constraint Programming as a modeling and solving technology for hard scheduling problems is well established (Laborie, Rogerie, Shaw, et al. 2018). The modeling flexibility has been further enhanced with the introduction of conditional time-interval variables (Laborie and Rogerie 2008), which represent the execution of tasks that may or may not be executed. Such variables allow to conveniently model problems with alternative resources or optional tasks such as in (Cappart and Schaus 2017; Kinable et al. 2014; Kizilay et al. 2018; Kumar et al. 2018). Building on top of conditional time-intervals, the same authors introduced an algebraical model for cumulative resources called cumulative functions (Laborie, Rogerie, Shaw, et al. 2009). It provides an efficient and convenient way to model producer-consumer problems with optional tasks such as in (Cappart, Thomas, et al. 2018; Gedik et al. 2018; Liu et al. 2018; Thomas and Schaus 2024). To the best of our knowledge, CPOptimizer (Laborie, Rogerie, Shaw, et al. 2018) and OptalCP (Vilím and Pons 2023) are the only two (commercial) solvers that fully support conditional time-interval variables and cumulative functions and this modeling paradigm is not supported by existing open-source solvers[1].

We propose an implementation of this modeling paradigm with lower and upper-bound restrictions on cumulative functions using a generalized cumulative constraint that supports negative heights (Beldiceanu and Carlsson 2002). This more general form of the cumulative constraint allows variable and negative resource consumptions as well as optional tasks. We introduce a new filtering algorithm for this constraint that is based on time-tabling (Fahimi et al. 2018; Gay et al. 2015a; Letort et al. 2015) and does additional filtering on the height and length of tasks. The algorithm proceeds in two steps. First, it uses the *Profile* data structure recently introduced by (Gingras and C. Quimper 2016) and builds the optimistic/pessimistic profiles of resources produced/consumed by tasks. It then processes each task to prune its bounds by comparing it to the profile. The experimental results show that the newly introduced algorithm performs well compared to existing ones, particularly in large-scale scheduling problems solved with a greedy search.

*Corresponding Author.
[1]API languages such as Minizinc also support optional variables at the modeling layer (Mears et al. 2014)

Authors' Contact Information: Pierre Schaus, ORCID: 0000-0002-3153-8941, pierre.schaus@uclouvain.be, UCLouvain, ICTEAM, Louvain-la-Neuve, Belgium; Charles Thomas, ORCID: 0000-0002-7360-5372, cftmthomas@gmail.com, UCLouvain, ICTEAM, Louvain-la-Neuve, Belgium; Roger Kameugne, ORCID: 0000-0003-1809-9822, rkameugne@gmail.com, UCLouvain, ICTEAM and University of Maroua, Louvain-la-Neuve / Maroua, Belgium / Cameroon.

2 Modeling with Cumulative Functions

2.1 Conditional Time-Intervals

The domain of a conditional time-interval variable can be expressed as a subset of $\{\bot\} \cup \{[s, e) \mid s \leq e, s, e \in \mathbb{Z}\}$ (Laborie and Rogerie 2008). It is fixed if $x = \bot$ (the interval is not executed) or $x = [s, e)$ (the interval starts at s and ends at e). For convenience, the description of the filtering algorithm manipulates conditional time-interval domains through the following attributes for each task $i \in T$:

- $p_i \in \{true, false\}$ is the execution status ($false = \bot$).
- $s_i \in [\underline{s}_i, \overline{s}_i]$ and $e_i \in [\underline{e}_i, \overline{e}_i]$ are the (non negative) start/end variables.
- $d_i \in [\underline{d}_i, \overline{d}_i]$ is the (non negative) duration variable.

A bound consistent filtering on the relation $s_i + d_i = e_i$ is maintained atomically on the corresponding attributes, and the domain of a time-interval variable becomes $\bot$ whenever one attribute range (s, d or e) becomes empty which may trigger an inconsistency if $p_i = true$. The whole domain of a conditional time-interval $i \in T$ is thus encoded by a tuple of variables $x_i = \langle s_i, d_i, e_i, p_i \rangle$.

2.2 Cumulative Functions

For modeling cumulative problems, (Laborie, Rogerie, Shaw, et al. 2009) introduced the idea of cumulative function expressions. They allow the expression of step-wise integer functions of conditional time-interval variables. The contribution of a single time-interval variable $i \in T$ is represented by an elementary cumulative function receiving an interval x_i and a non-negative consumption value c_i or a non-negative height range $[\underline{c}_i, \overline{c}_i]$. The elementary functions are *pulse*, *stepAtStart* and *stepAtEnd* (Laborie, Rogerie, Shaw, et al. 2009) as represented in Figure 1.

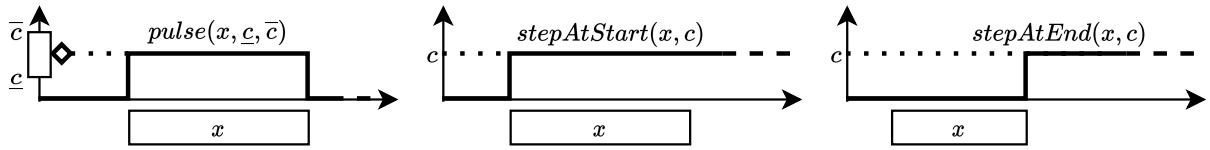

Fig. 1. Elementary cumulative functions.

Cumulative functions (elementary or not) can then be combined with the *plus/minus* operators to form a more complex function representing the addition or subtraction of two functions. A cumulative function can thus be viewed as an Abstract Syntax Tree (AST) with elementary functions at the leaf nodes.

Example 1. *Three tasks A, B and C contribute to a cumulative function $f = stepAtStart(A, 2) - (pulse(B, 1) + stepAtEnd(C, 1))$. The resulting AST and cumulative function are represented in Figure 2. On the right, the solid parts represent the time windows of the tasks while the hatched parts represent their contribution to the profile. A dotted line represents the final cumulative function f.*

A cumulative function can be flattened using Algorithm 1 to extract a set of cumulative tasks (tasks with their positive or negative heights represented by a tuple (x_i, c_i)). This algorithm traverses the AST and collects the activities at the leaf nodes, reversing the height signs for the activities flattened on the right branch of a minus node. Note that whenever encountering a $stepAtStart(x, c)$ leaf node, the cumulative task collected is defined on a fresh task interval x' related to the original one with: $x'.s = x.s$ (same start as x), $x'.p = x.p$ (same execution status), and $x'.e = Horizon$ (ending at the time horizon) to reflect the definition of a step function. Similarly, in the case of $stepAtEnd(x, c)$, a new task interval x' with $x'.s = x.e$ (same start as end of x) is collected.

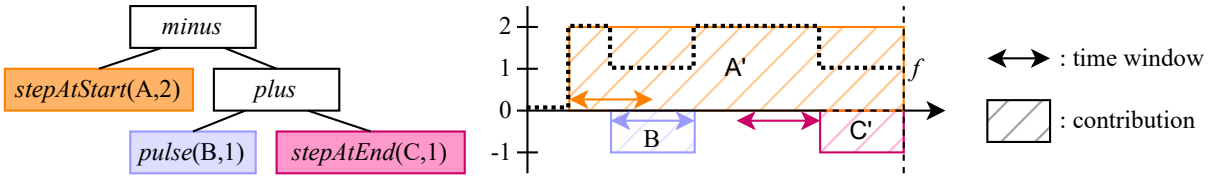

Fig. 2. Example 1: AST (left) and cumul. function (right).

Algorithm 1: flatten(n)

Input: n: A node of the AST
Output: Set of cumulative tasks

1 **if** $n = pulse(x, c)$ **then** $\{(x, c)\}$
2 **else if** $n = stepAtStart(x, c)$ **then** $\{(x' \leftarrow \langle x.s, Horizon - x.s, Horizon, x.p\rangle, c)\}$
3 **else if** $n = stepAtEnd(x, c)$ **then** $\{(x' \leftarrow \langle x.e, Horizon - x.e, Horizon, x.p\rangle, c)\}$
4 **else if** $n = plus(n_l, n_r)$ **then** $flatten(n_l) \cup flatten(n_r)$
5 **else if** $n = minus(n_l, n_r)$ **then** $flatten(n_l) \cup \{(x, -c) \mid (x, c) \in flatten(n_r)\}$
6 **else** *error* `// Unknown pattern`

Example 2. *Continuing example 1, flatten(f) collects the cumulative tasks $\{(A', 2), (B, -1), (C', -1)\}$ illustrated on the right of Figure 2.*

2.3 Constraints on Cumulative Functions

The height of a cumulative function can be restricted with a lower and upper bound at every time overlapping at least one executing time-interval: $\underline{C} \leq f \leq \overline{C}$. This constraint is called $alwaysIn(f, \underline{C}, \overline{C})$ in (Laborie, Rogerie, Shaw, et al. 2009). It can be enforced by posting a `GeneralizedCumulative` constraint requiring a resource consumption between $\underline{C}$ and $\overline{C}$ for the set of cumulative tasks T collected with the *Flatten* function. The whole domain of a conditional (cumulative) task $i \in T$ is encoded by a tuple of variables $\langle s_i, d_i, e_i, c_i, p_i\rangle$ where the integer variable c_i represents its consumption (possibly negative or hybrid) on the resource. The set of tasks T is partitioned into three subsets as given in Definition 1.

Definition 1. *The set $O = \{i \in T \mid p_i = \{true, false\}\}$ denotes the set of optional tasks, those for which it is not decided yet if they will execute or not on the resource. The set $R = \{i \in T \mid p_i = true\}$ denotes the set of required tasks, those which will surely be executed on the resource. The set $E = \{i \in T \mid p_i = false\}$ denotes the set of excluded tasks, those which will surely not be executed on the resource. The sets O, R and E are disjoint and form a partition of the set T, i.e., $T = O \cup R \cup E$.*

Definition 2. *The* `GeneralizedCumulative`$(T, \underline{C}, \overline{C})$ *constraint enforces the cumulated heights of the tasks of T to be within the fixed capacity range $[\underline{C}, \overline{C}]$ for every time at which at least one task of T is executed:*

$$\sum_{i \in R | s_i \leq \tau < e_i} c_i \in [\underline{C}, \overline{C}] \qquad \forall \tau \in \bigcup_{i \in R} [s_i, e_i) \tag{1}$$

Restricted Time Interval Scope. To enforce the capacity range at every time point across the entire horizon rather than only when at least one task executes, one can add a dummy task that spans the entire horizon, with a mandatory execution status and a height of zero. More generally, the time scope can also be limited to a specific task, by linking the status variables of the tasks to the overlap of this task.

Consistency. Enforcing global bound-consistency for the cumulative constraint is NP-Hard (Garey and Johnson 1979). However, enforcing it on the decomposition into sum constraints at every point in the horizon, as in (1), is not NP-Hard. Unfortunately, this approach is inefficient for large horizons, as it requires a sum constraint to be enforced at every time point. Therefore, we aim to achieve the same filtering in polynomial time relative to the number of tasks, rather than the size of the horizon. This filtering is generally referred to as *time-tabling* filtering. Intuitively, it works by aggregating all the fixed parts (when $\underline{s}_i < \overline{e}_i$) to create a consumption profile. It then postpones the earliest execution of a task whenever it detects that starting it at this time would overload the resource capacity.

3 Timetabling Algorithm

This section first introduces the *Profile* data structure then explains the time-tabling filtering algorithm for filtering the task interval attributes.

3.1 The Profile Data Structure

The *Profile* data structure is an aggregation of adjoined rectangles of different lengths and heights introduced in (Gay et al. 2015a; Gingras and C. Quimper 2016) to record the resource utilization of tasks. The end of a rectangle is the beginning of the next one. The starting (resp. ending) of a rectangle is represented by a tuple called *time point* with a component representing the beginning (resp. ending) time of the rectangle and other components that contain information relative to the resource consumption for the duration of the rectangle. These time points are sorted in increasing order of time and are kept in a linked list structure called the *Profile*. An original idea of (Gingras and C. Quimper 2016) is to have pointers $tp(\underline{s}_i)$, $tp(\underline{e}_i)$ and $tp(\overline{e}_i)$ linking to the time point associated with $\underline{s}_i$, $\underline{e}_i$, and $\overline{e}_i$ respectively.

3.2 Timetable Filtering

To check and adjust the attributes of tasks, the *Profile* data structure from (Gingras and C. Quimper 2016) is extended to compute both a minimum and maximum profile range at each time point. In our adaptation of the *Profile* data structure (hereafter referred to as *timeline* and noted P), a time point $tp \in P$ consists of a tuple $\langle time, \underline{P}, \overline{P}, \#fp \rangle$, where *time* corresponds to the start time of the time point, $\underline{P}$ (resp. $\overline{P}$) to the minimum (resp. maximum) resource profile of tasks over time and $\#fp$ to the number of fixed parts of tasks that overlap over *time*.

Intuitively, the minimum profile is obtained by considering the upper bound of the negative contribution and the lower bound of the positive contribution of each task. In contrast, the maximum profile considers the upper bound of the positive contribution and the lower bound of the negative contribution of each task. The upper bound of the negative (resp. positive) contribution of a task corresponds to its largest negative (resp. positive) height during the whole possible time window of the task while the lower bound of its negative (resp. positive) contribution corresponds to its smallest negative (resp. positive) height during the fixed part of the task. More formally, the minimum $\underline{P}$ (resp. maximum $\overline{P}$) resource profile at time t is defined as

$$\underline{P}_t = \sum_{i \in O \cup R | \underline{s}_i \leq t < \overline{e}_i} \min(\underline{c}_i, 0) + \sum_{i \in R | \overline{s}_i \leq t < \underline{e}_i} \max(\underline{c}_i, 0) \tag{2}$$

$$\overline{P}_t = \sum_{i \in O \cup R | \underline{s}_i \leq t < \overline{e}_i} \max(\overline{c}_i, 0) + \sum_{i \in R | \overline{s}_i \leq t < \underline{e}_i} \min(\overline{c}_i, 0) \tag{3}$$

where $i \in O \cup R \mid \underline{s}_i \leq t < \overline{e}_i$ corresponds to the set of all optional and required tasks whose time window overlaps the time t and $i \in R \mid \overline{s}_i \leq t < \underline{e}_i$ corresponds to the set of all required tasks that have a fixed part that overlaps the time t.

The proposed timetable filtering algorithm is split into two steps: First, the profile data structure P is initialized by creating and ordering the time points, linking them with the tasks, computing their minimum and maximum profile ranges, and checking their consistency by comparing the profile ranges with the maximum and minimum capacity. Second, each task is checked during its entire possible span in terms of profile and capacity to adjust its domain.

3.3 Timeline Initialization and Consistency Check

The function *initializeTimeline* (Algorithm 2) initializes the timeline, links the corresponding time points to tasks, and computes the minimum and maximum profile of the resource utilization. It receives as input the set of tasks T and the bounds of the resource capacity $[\underline{C}, \overline{C}]$. First, events are generated by iterating over the tasks that are not absent (Line 2). For each non-absent task i, two events are generated which correspond to the minimum start time ($\underline{s}_i$), and the maximum end time ($\overline{e}_i$) of the task. For tasks that are present and with a fixed part, two additional events are generated for the maximum start time ($\overline{s}_i$) and the minimum end time ($\underline{e}_i$) of the task.

Then, these events are processed by order of increasing time to initialize the timeline P (Line 10). A current time point tp is maintained and corresponds to the last time point of the timeline being initialized. New time points are created only when the time of the event differs from the current time point tp as checked at Line 12. This means that if several events share the same time, they will all contribute towards the same time point. When a new time point is created, it is appended at the end of the timeline which links it to the previous last time point. When processing a start min ($\underline{s}$) or end max ($\overline{e}$) event, the current time point is linked to the event at Lines 20 and 24 to retrieve the time points corresponding to the start and end of tasks in constant time later.

Three accumulators are used to keep track of the minimum ($\underline{P}$) and maximum ($\overline{P}$) profile as well as the number of fixed parts of tasks overlapping the current time point ($\#fp$). Depending on the nature of the event, the positive or negative contribution of its associated task is either added to (in case of $\underline{s}$ or $\overline{s}$) or subtracted (in case of $\overline{e}$ or $\underline{e}$) from the accumulators and the current time point tp is updated.

As the timeline is initialized, the consistency of the minimum and maximum profiles is checked for each time point at Line 13. Once a time point is completed (when creating the next time point), we ensure that if at least one task is guaranteed to execute during the time point ($\#fp > 0$), the minimum profile $\underline{P}$ is not above the maximum capacity of the resource $\overline{C}$ and the maximum profile $\overline{P}$ is not below the minimum capacity of the resource $\underline{C}$.

The worst-case complexity in time of the function `initializeTimeline` is $O(n \log(n))$ and is due to the sorting of the events used in the loop of Line 10. Example 3 illustrates the profile initialization for three tasks.

Algorithm 2: initializeTimeline($T, \underline{C}, \overline{C}$)

```
events ← {}                                                    // Initialize the set of events
for i ∈ T | i ∈ O ∪ R do
    events ← events ∪ {s_i, ē_i}                               // Add events for non-absent tasks
    if i ∈ R ∧ s̄_i < e_i then
        events ← events ∪ {s̄_i, e_i}                           // Add events for required tasks with fixed part
P ← 0; P̄ ← 0                                                   // Lower/Upper bound of the profile at current time point
#fp ← 0                                                        // Number of overlapping fixed parts of current time point
tp ← new tp(min(events.time))                                  // Current time point
P ← {tp}                                                       // All time points created so-far
for e ∈ events sorted by time do
    i ← e.task
    if tp.time < e.time then
        if #fp > 0 ∧ (P > C̄ ∨ P̄ < C) then
            return Fail                                        // Capacity violation detected
        tp ← new tp(e.time)                                    // New time point
        P.append(tp)
    if e = s_i then                                            // Process earliest start event
        P ← P + min(c_i, 0)                                    // Add pessimistic negative contribution
        P̄ ← P̄ + max(c̄_i, 0)                                    // Add optimistic positive contribution
        tp.P ← P; tp.P̄ ← P̄; tp.#fp ← #fp; tp(s_i) ← tp         // Update tp and link task to tp
    if e = ē_i then                                            // Process latest end event
        P ← P − min(c_i, 0)                                    // Remove pessimistic negative contribution
        P̄ ← P̄ − max(c̄_i, 0)                                    // Remove optimistic positive contribution
        tp.P ← P; tp.P̄ ← P̄; tp.#fp ← #fp; tp(ē_i) ← tp         // Update tp and link task to tp
    if p_i = true ∧ s̄_i < e_i then
        if e = s̄_i then                                        // Process start of fixed parts event
            P ← P + max(c_i, 0)                                // Add pessimistic positive contribution
            P̄ ← P̄ + min(c̄_i, 0)                                // Add optimistic negative contribution
            #fp ← #fp + 1                                      // Increment overlapping fixed part counter
            tp.P ← P; tp.P̄ ← P̄; tp.#fp ← #fp
        if e = e_i then                                        // Process end of fixed parts event
            P ← P − max(c_i, 0)                                // Remove pessimistic positive contribution
            P̄ ← P̄ − min(c̄_i, 0)                                // Remove optimistic negative contribution
            #fp ← #fp − 1                                      // Decrement overlapping fixed part counter
            tp.P ← P; tp.P̄ ← P̄; tp.#fp ← #fp
return P                                                       // Return the complete timeline
```

Example 3. *Let us consider three tasks :*

$A : \langle s = [0,1], d = [3,4], e = [3,4], c = [1,2], p = true \rangle;$
$B : \langle s = [2,4], d = [3,4], e = [5,7], c = 2, p = true \rangle;$
$C : \langle s = [3,8], d = [1,3], e = [4,9], c = [-2,1], p = \{true, false\} \rangle;$

and a resource C of capacity bounds $[0, 1]$. The time points and their attributes are shown in Table 1.

time	*event(s)*	$\underline{P}$	$\overline{P}$	*#fp*
0	$\underline{s}_a$	0	2	0
1	$\overline{s}_a$	1	2	1
2	$\underline{s}_b$	1	4	1
3	$\underline{e}_a, \underline{s}_c$	-2	5	0
4	$\overline{e}_a, \overline{s}_b$	0	3	1
5	$\underline{e}_b$	-2	3	0
7	$\overline{e}_b$	-2	1	0
9	$\overline{e}_c$	0	0	0

Table 1. Time points computed from the tasks of Example 3.

Figure 3 shows a representation of the tasks and the resulting profile ranges. Note that the events $\overline{s}_c$ and $\underline{e}_c$ are not considered as task C has no fixed part.

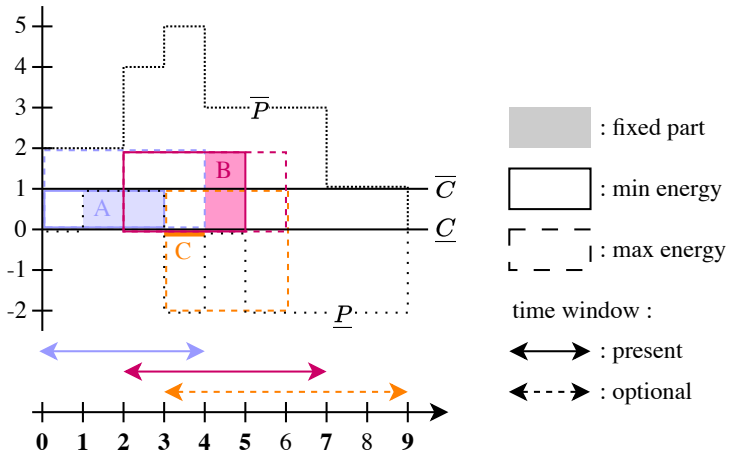

Fig. 3. Representation of the tasks of Example 3 with the resulting minimum ($\underline{P}$) and maximum ($\overline{P}$) profile.

3.4 Filtering

Once the timeline is initialized and its consistency verified, the profile range (minimum and maximum profile of tasks at any time point) is used to filter the time windows, capacity, and duration of tasks. Four different filtering rules are used:

Forbid. The first rule is used to reduce the time windows of tasks by checking for each task $i \in T$ if starting (resp. ending) the task at its minimum start time (resp. maximum end time) is possible in regards to the profile range and the capacity range. If this is not the case, the task is pushed to the earliest (resp. latest) time at which it can be placed without violating the capacity range. Formally, the rule is defined as:

$$\forall tp \in P \mid \underline{s}_i \leq tp.time < \min(\overline{s}_i, \underline{e}_i), \quad tp.\underline{P} + \max(\underline{c}_i, 0) > \overline{C} \vee tp.\overline{P} + \min(\overline{c}_i, 0) < \underline{C} \Rightarrow \underline{s}_i \geq tp.next.time \quad (4)$$

$$\forall tp \in P \mid \max(\overline{s}_i, \underline{e}_i) \leq tp.time < \overline{e}_i, \quad tp.\underline{P} + \max(\underline{c}_i, 0) > \overline{C} \vee tp.\overline{P} + \min(\overline{c}_i, 0) < \underline{C} \Rightarrow \overline{e}_i \leq tp.prev.time \quad (5)$$

Note that as tasks are modeled with interval variables, the rule $e_i = s_i + d_i$ is enforced internally in the variable. Thus, when a task i has its minimum start time $\underline{s}_i$ (resp. maximum end time $\overline{e}_i$) adjusted, its minimum end time $\underline{e}_i$ (resp. maximum start time $\overline{s}_i$) is also updated by the relation $\underline{e}_i \geq \underline{s}_i + \underline{d}_i$ (resp. $\overline{s}_i \leq \overline{e}_i - \underline{d}_i$) embedded in the interval variable.

Mandatory. The second rule detects if the execution of a task $i \in T$ at a time point tp is necessary to avoid a violation of the resource capacity range at this time point. If this is the case, the task is adjusted to make sure it is present during the time point:

(1) if the task is optional, it is set to present;
(2) its time window is adjusted such that it executes over the whole time point;
(3) its height is adjusted to avoid the profile range violation.

In formal terms, the rule is defined as:

$$\forall tp \in P \mid tp.time \in [\underline{s}_i, \overline{e}_i) \wedge tp.\#fp > 0,$$

$$tp.\underline{P} - \min(\underline{c}_i, 0) > \overline{C} \vee tp.\overline{P} - \max(\overline{c}_i, 0) < \underline{C} \Rightarrow \begin{cases} p_i \leftarrow true \\ \overline{s}_i \leq tp.time \\ \underline{e}_i \geq tp.next.time \\ \underline{c}_i \geq deficit \text{ if } deficit > 0 \\ \overline{c}_i \leq overload \text{ if } overload < 0 \end{cases} \tag{6}$$

where $deficit = \underline{C} - (\overline{P} - \max(\overline{c}_i, 0))$ and $overload = \overline{C} - (\underline{P} - \min(\underline{c}_i, 0))$ are the deficit and overload of height needed for the profile range to intersect the capacity range.

Height. This rule adjusts the height of each task $i \in T$ in regard to the resource capacity range. The maximum (resp. minimum) value of the height of a task is bounded by the difference between the maximum (resp. minimum) capacity of the resource and the minimum (resp. maximum) profile without the contribution of the task. Two cases are possible:

(1) If the task has a fixed part, its height is checked and adjusted directly at each time point of the fixed part:

$$\forall tp \in P \mid \overline{s}_i \leq tp.time < \underline{e}_i, \quad \overline{c}_i \leq \overline{C} - (tp.\underline{P} - \min(\underline{c}_i, 0)) \ \wedge \underline{c}_i \geq \underline{C} - (tp.\overline{P} - \max(\overline{c}_i, 0)) \tag{7}$$

Note that if a task is present ($i \in R$), its minimum positive ($\max(\underline{c}_i, 0)$) and negative ($\min(\overline{c}_i, 0)$) contributions have been used to compute the profile range and thus these values must be subtracted in the above equation.

(2) If the task has no fixed part, the maximum available height over the minimum overlapping interval of the task is used to adjust its height:

$$\overline{c}_i \leq \max_{tp \in P \mid \underline{e}_i - 1 < tp.end \wedge tp.time \leq \overline{s}_i} \overline{C} - (tp.\underline{P} - \min(\underline{c}_i, 0)) \tag{8}$$

$$\underline{c}_i \geq \min_{tp \in P \mid \underline{e}_i - 1 < tp.end \wedge tp.time \leq \overline{s}_i} \underline{C} - (tp.\overline{P} - \max(\overline{c}_i, 0)) \tag{9}$$

The minimum overlapping interval was introduced by (Gay et al. 2015b) and is defined for a task $i \in T$ as the interval $[\underline{e}_i - 1, \overline{s}_i]$. Intuitively, it corresponds to the smallest interval such that, no matter its start time, length or end time, the task executes during at least one time unit of this interval.

Length. The last rule is inspired by (Ouellet and C.-G. Quimper 2019). It adjusts the maximum duration $\overline{d}$ of the tasks that do not have a fixed part (for the tasks with a fixed part, the adjustment would be redundant with the *Forbid* rule). The principle is to find the longest time span during which the task can be scheduled without a resource capacity violation. Again, we rely on the internal propagation rules of the interval variables to set the task as absent if the adjustment would empty its domain and to update the $\underline{s}_i$ and $\overline{e}_i$ attributes following the relations $\underline{s}_i \geq \underline{e}_i - \overline{d}_i$ and $\overline{e}_i \leq \overline{s}_i + \overline{d}_i$. Formally, the rule is written as:

$$\overline{d}_i \leq \max_{[a,b)\in A} b - a$$
where
$$A = \left\{[a,b) \subseteq [\underline{s}_i, \overline{e}_i] \mid \forall tp \in P \text{ with } tp.time \in [a,b),\ tp.\underline{P} + \max(\underline{c}_i, 0) \leq \overline{C} \wedge tp.\overline{P} + \min(\overline{c}_i, 0) \geq \underline{C}\right\} \tag{10}$$

Intuitively, A is the set of intervals where the task can be scheduled without underloading the minimum and overloading maximum capacity.

Algorithm. The function *timetabling* of Algorithm 3 enforces these rules for each task $i \in T$. This algorithm receives as input the set of tasks T and the bounds of the resource capacity $[\underline{C}, \overline{C}]$. The algorithm iterates over all non-fixed tasks at Line 3. Each task is processed in three steps:

First (step 1), the *Forbid* rule is used to adjust the minimum start time of the task by iterating forward over the time points in the interval $[\underline{s}_i, \min(\overline{s}_i, \underline{e}_i))$ in the loop at Line 6. Second (step 2), the *Forbid* rule is used to adjust the maximum end time of the task by iterating backward over the interval $[\max(\overline{s}_i, \underline{e}_i), \overline{e}_i)$ in the loop at Line 11.

The last step iterates over the time points of the remaining part of the time window. It depends if the task has a fixed part or not. In the former case (step 3), the loop at Line 16 iterates over the time points in the fixed part of the task and adjusts its height following the 1st case of the *Height* rule. In the case where the task has no fixed part (step 3'), the loop at Line 28 is executed. The height of the task is adjusted according to the 2nd case of the *Height* rule and its maximum length is adjusted following the *Length* rule. These three steps are illustrated in Figure 4.

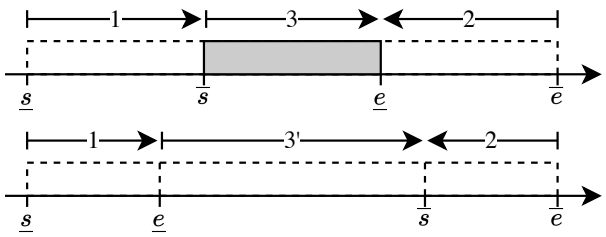

Fig. 4. Timetabling processing order for a task with a fixed part (top) and one without (bottom). Those steps (1, 2, 3, and 3') are also referenced in Algorithm 3 comments.

Algorithm 3: timetabling($T, \underline{C}, \overline{C}$)

$P \leftarrow$ initializeTimeLine(T) // Compute the profile
$O \leftarrow \{i \in T \mid p_i = \{true, false\}\}$; $R \leftarrow \{i \in T \mid p_i = true\}$ // Optional/Required tasks
for $i \in O \cup R$ **do**
  if i *is fixed* **then** continue
  $tpf \leftarrow tp(\underline{s}_i)$ // Get the time point at minimum start
  while $tpf.time < \min(\overline{s}_i, \underline{e}_i)$ **do** // Step 1 (forward): adjust minimum start time
    // Check if task can start at this time point, if not push its start
    if $tpf.\underline{P} + \max(\underline{c}_i, 0) > \overline{C} \vee tpf.\overline{P} + \min(\overline{c}_i, 0) < \underline{C}$ **then** $\underline{s}_i \leftarrow \max(tpf.next.time, \underline{s}_i)$
    else checkIfMandatory($i, tpf, \underline{C}, \overline{C}$)
    $tpf \leftarrow tpf.next$ // Move to next time point
  $tpb \leftarrow tp(\overline{e}_i).prev$ // Get the time point just before maximum end
  while $tpb.next.time > \max(\overline{s}_i, \underline{e}_i)$ **do** // Step 2 (backward): adjust maximum end time
    // Check if task can end after this time point, if not pull its end
    if $tpb.\underline{P} + \max(\underline{c}_i, 0) > \overline{C} \vee tpb.\overline{P} + \min(\overline{c}_i, 0) < \underline{C}$ **then** $\overline{e}_i \leftarrow \min(tpb.time, \overline{e}_i)$
    else checkIfMandatory($i, tpb, \underline{C}, \overline{C}$)
    $tpb \leftarrow tpb.prev$ // Move to previous time point
  if $\overline{s}_i < \underline{e}_i$ **then** // Step 3: Adjust height (task with a fixed part)
    while $tpf.time < \underline{e}_i$ **do** // Scan over fixed part $[\overline{s}_i, \underline{e}_i - 1,]$
      checkIfMandatory($i, tpf, \underline{C}, \overline{C}$)
      $\underline{c}' \leftarrow \underline{C} - (tpf.\overline{P} - \max(\overline{c}_i, 0))$; $\overline{c}' \leftarrow \overline{C} - (tpf.\underline{P} - \min(\underline{c}_i, 0))$ // Available height range
      if $i \in R$ **then**
        $\underline{c}' \leftarrow \underline{c}' + \min(\overline{c}_i, 0)$; $\overline{c}' \leftarrow \overline{c}' + \max(\underline{c}_i, 0)$ // Adjust for task's own contribution P
      $\underline{c}_i \leftarrow \max(\underline{c}', \underline{c}_i)$; $\overline{c}_i \leftarrow \min(\overline{c}', \overline{c}_i)$ // Restrict min/max task consumption
      $tpf \leftarrow tpf.next$ // Move to next time point
  else // Step 3': Adjust height and length (task without a fixed part)
    $\overline{d}^* \leftarrow 0$ // Track maximum feasible duration
    $s' \leftarrow \underline{s}_i$ // Track start of current feasible interval
    $\underline{c}^* \leftarrow \underline{C} - tpf.prev.\overline{P} + \max(\overline{c}_i, 0)$ // Min height over minimum overlapping interval
    $\overline{c}^* \leftarrow \overline{C} - tpf.prev.\underline{P} + \min(\underline{c}_i, 0)$ // Max height over minimum overlapping interval
    while $tpf.time < \overline{s}_i$ **do** // Scan over minimum overlapping interval $[\underline{e}_i - 1, \overline{s}_i]$
      $\overline{d}^* \leftarrow \max(tpf.time - s', \overline{d}^*)$
      if $tpf.\underline{P} + \max(\underline{c}_i, 0) > \overline{C} \vee tpf.\overline{P} + \min(\overline{c}_i, 0) < \underline{C}$ **then** $s' \leftarrow tpf.next.time$ // Reset start
      else checkIfMandatory($i, tpf, \underline{C}, \overline{C}$)
      $\underline{c}^* \leftarrow \min(\underline{C} - tpf.\overline{P} + \max(\overline{c}_i, 0), \underline{c}^*)$
      $\overline{c}^* \leftarrow \max(\overline{C} - tpf.\underline{P} + \min(\underline{c}_i, 0), \overline{c}^*)$
      $tpf \leftarrow tpf.next$
    $\overline{d}^* \leftarrow \max(\overline{e}_i - s', \overline{d}^*)$; $\overline{d}_i \leftarrow \min(\overline{d}^*, \overline{d}_i)$ // Apply Length rule
    $\underline{c}_i \leftarrow \max(\underline{c}^*, \underline{c}_i)$; $\overline{c}_i \leftarrow \min(\overline{c}^*, \overline{c}_i)$ // Restrict min/max task consumption

In order to compute the maximum length of the task, we maintain the variables $\overline{d}^*$ which contains the maximum length found so far and s' which corresponds to the earliest time at which the task can start and span without capacity violation until the current time point. During the loop, $\overline{d}^*$ is updated if the difference between the current time point time and s' is greater than its previous value. If the task is detected as infeasible during a time point, s' is set to the end of this time point (Line 30). Note that in order to take into account the parts of the time window that are not considered in Loop 28, s' is initialized to $\underline{s}$ and the difference between $\overline{e}$ and s' is considered at the end of the loop (Line 35).

Similarly, the minimum and maximum available heights are computed as $\underline{c}^*$ and $\overline{c}^*$ and used to update the height of the task if it has no fixed part (Line 36). Note that these variables are initialized based on the height available at the time point before the start of Loop 28 in order to consider the whole minimum overlapping interval of the task.

The *Mandatory* rule is enforced over the whole time window of each task. The function *checkIfMandatory* of Algorithm 4 is used to check this rule and apply its adjustments if needed. It is called in each loop, at Lines 8, 13, 17 and 31.

Algorithm 4: checkIfMandatory($i, tp, \underline{C}, \overline{C}$)

1 **if** $\#fp > 0 \wedge (tp.\underline{P} - \min(\underline{c}_i, 0) > \overline{C} \vee tp.\overline{P} - \max(\overline{c}_i, 0) < \underline{C})$ **then**
2 $\quad p_i \leftarrow true$
3 $\quad \overline{s}_i \leftarrow \min(tp.time, \overline{s}_i)$
4 $\quad \underline{e}_i \leftarrow \max(tp.next.time, \underline{e}_i)$
5 $\quad deficit \leftarrow \underline{C} - (tp.\overline{P} - \max(\overline{c}_i, 0))$
6 $\quad overload \leftarrow \overline{C} - (tp.\underline{P} - \min(\underline{c}_i, 0))$
7 $\quad$ **if** $deficit > 0$ **then** $\underline{c}_i \leftarrow \max(deficit, \underline{c}_i)$
8 $\quad$ **if** $overload < 0$ **then** $\overline{c}_i \leftarrow \min(overload, \overline{c}_i)$

The worst-case time complexity of Algorithm 3 is $O(n^2)$. Note that some optimizations are possible depending on the set of tasks given as input. Indeed, the *Mandatory* propagation rule is only useful if the tasks have a mix of positive and negative heights or if there is a minimum capacity $\underline{C}$ to enforce. If all the height variables are fixed to a single value, the *Height* adjustment rule is not necessary. Similarly, the *Length* adjustement rule is only needed if the tasks have variable lengths. If these three rules can be ignored, only steps 1 and 2 need to be performed and the unnecessary loops at Lines 16 and 28 can be avoided.

Another optimization is to avoid considering some tasks when performing the timetabling algorithm. Indeed, fixed tasks that occur before the start of the earliest unfixed task or after the end of the latest unfixed task are not useful for filtering. Thus, such tasks do not need to be processed which reduces the number of time points in the profile. To do so, we use the same process as the *Fruitless Fixed Tasks Removal* of (Gay et al. 2015a).

Example 4. *When executing the timetabling algorithm on the same tasks as in Example 3, the adjustments are:*

(1) *When processing task A at time point* 1*, its maximum height* $\overline{c}_A$ *is adjusted to* 1 *by the Height rule.*
(2) *When processing task B at time point* 2*, the task is detected as non-feasible at this time point by the Forbid rule. Its minimum start* $\underline{s}_B$ *is thus pushed to the next time point at time* 3 *where the adjustment stops as the task is feasible at this time. This also updates its minimum end to* $\underline{e}_B = 6$ *due to the internal relation* $\underline{e}_B \geq \underline{s}_B + \underline{d}_B$.
(3) *When processing task C at time point* 4*, the task is detected as mandatory. The following adjustments are done:*
   (a) *The task is set to present.*

(b) *Its maximum starting time is set to* $\overline{s}_C = 4$ *which also updates its maximum ending time to* $\overline{e}_C = 7$ *due to the internal relation* $\overline{e}_C \leq \overline{s}_C + \overline{d}_C$.
(c) *Its minimum ending time is set to* $\underline{e}_C = 5$.
(d) *As the value overload* $= -1$ *is negative, the maximum height of the task is set to* $\overline{c}_C = -1$.

Note that even if task B is processed before task C and its fixed part increased, task C is adjusted based on the profile range which has been computed based on the state of task B at the start of the propagation.

The state of the tasks after timetabling is:

$A : \langle s = [0,1], d = [3,4], e = [3,4], c = 1, p = true \rangle$
$B : \langle s = [3,4], d = [3,4], e = [6,7], c = 2, p = true \rangle$
$C : \langle s = [3,4], d = [1,3], e = [5,7], c = [-2,-1], p = true \rangle$

Figure 5 shows the state of the tasks as well as the adjustments done.

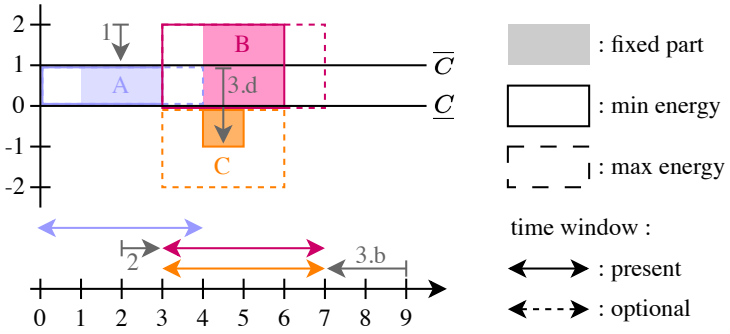

Fig. 5. Representation of the tasks of Example 3 after timetabling. Adjustments are shown as gray arrows when applicable.

4 Related Work

Some forms of the generalized cumulative constraint are available in both open-source and commercial constraint programming systems. We distinguish between those that use a multi-resource API, based on the work of (Beldiceanu and Carlsson 2002), and those that offer a conditional task interval API with cumulative functions.

4.1 Generalized multi-resource constraint

In (Beldiceanu and Carlsson 2002), a timetabling filtering algorithm for the `GeneralizedCumulative` constraint with k alternative resources was introduced. This constraint does not rely on conditional task intervals or cumulative functions (which were only introduced later in (Laborie, Rogerie, Shaw, et al. 2009)); instead, it operates directly on arrays of integer variables. The signature of the constraint is:

$$cumulatives(s[1,n], d[1,n], e[1,n], c[1,n], m[1,n], \underline{C}[1,k], \overline{C}[1,k])$$

where s, d, e, and c denote respectively the start times, durations, end times, and consumptions (which can be positive or negative). The variables m indicate, for each task, the resource on which it executes, with domain on $\{1,\ldots,k\}$. Finally, $[\underline{C}[j], \overline{C}[j]]$ specifies the capacity range of resource j.

Although this constraint does not directly support conditional task intervals, it is straightforward to connect the two modeling approaches. Indeed, a dummy resource can be used to indicate if a task is not present in the multi-resource formulation. Conversely, the conditional task approach supports multiple resources by using an additional *alternative* constraint (Laborie and Rogerie 2008) that represent the choice of alternative resources. This

requires duplicating the activity once for each alternative resource, but it has the advantage that each alternative task can be filtered independently during propagation. In contrast, the multi-resource formulation propagation can only exploit the minimum over the different possible starts across all resources, which can lead to a weaker filtering.

The filtering algorithm of (Beldiceanu and Carlsson 2002) proceeds by collecting a set of tasks at relevant time points before performing propagation. This algorithm is implemented in the open-source solver Gecode (Schulte et al. 2006) and according to its documentation, the same algorithm is also implemented in Sicstus (Carlsson et al. 2025). The algorithm presents some substantial differences with our algorithm.

First, our approach is closer to the strategy described in (Gay et al. 2015a) for the classical cumulative constraint, which constructs a profile and then tests each task against it. We rely on the *Profile* data structure (Gingras and C. Quimper 2016) to prune the tasks forward and backward, thus reducing the number of propagation calls required to reach a fixpoint. In contrast to (Gay et al. 2015a), which represents the profile as a simple list of rectangles, our algorithm can retrieve the starting rectangle in $O(1)$ thanks to the *Profile*. Consequently, although the worst-case quadratic time complexity per call remains similar to that of (Beldiceanu and Carlsson 2002) and (Gay et al. 2015a), our algorithm can be faster in practice. Second, our algorithm can possibly prune more the height and the duration attributes of the tasks. These filtering differences are detailed next.

Backward propagation. For the adjustment of the maximum end time ($\overline{e}$) of tasks due to the *Forbid* rule, when a task requires two or more consecutive adjustments, our algorithm performs all the adjustments in a single call to the timetabling algorithm compared to the version of (Beldiceanu and Carlsson 2002) where only one adjustment is done per call. Indeed, in (Beldiceanu and Carlsson 2002), detection and adjustment are performed only in a forward fashion. Thus, the filtering of the maximum end time of tasks is done only in regards to the current end of the task time window. This means that if a task has its maximum end time adjusted, whether or not the maximum end time can be further adjusted will be considered in the next call to the filtering algorithm. In practice, this may lead to situations where the filtering algorithm of (Beldiceanu and Carlsson 2002) needs several consecutive calls in order to completely adjust the end time of a task and reach a fixed point. In contrast the algorithm presented in this paper does such adjustments in a single pass as illustrated in Example 5.

Example 5. *Let us consider three tasks:*

- *Task A is fixed and present, starts at 4, ends at 5 and has a height of 1.*
- *Task B is fixed and present, starts at 7, ends at 8 and has a height of 1.*
- *Task C is present, has a minimum start of 0, a maximum end of 10, a fixed length of 3 and a height of 1.*

The maximum capacity of the resource $\overline{C}$ is 1. Figure 6 shows the three tasks before propagation. The dashed blue rectangle represents the time window $[\underline{s}, \overline{e})$ of task C.

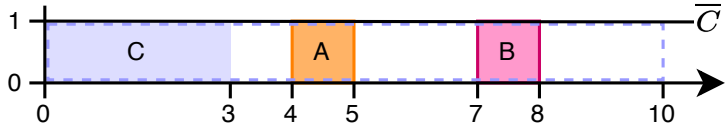

Fig. 6. Tasks of example 5

During the backward propagation of our timetabling algorithm, task C is detected as infeasible at times 8 then 5, and its maximum end time is successively updated to 7 then 4 in a single call. The propagation from (Beldiceanu and Carlsson 2002) iterates the profile in one direction and detects the infeasibility of task C only at time 8 and adjusts its maximum end at 7 during the first call. A second call to the algorithm is necessary in order to adjust the maximum end at time 4.

Height Adjustment Difference. For height adjustment, the difference occurs in the case where a task has no fixed part (second case of the *Height* filtering rule). In this case, our algorithm considers the maximum height available over the minimum overlapping interval of the task (Gay et al. 2015b).

The propagation from (Beldiceanu and Carlsson 2002) uses another height adjustment rule (see Alg. 4 in their paper). It considers the maximum available height only if the task examined has both its earliest completion time ($\underline{e}$) and its latest start time ($\overline{s}$) in the same event interval. That means that if one or more other tasks affect the available height between the $\underline{e}$ and $\overline{s}$ of a task, leading them to occur during different events, the height of the task is not adjusted. Example 6 illustrates this.

Example 6. *Let us consider three tasks:*

- *Task A is fixed and present, starts at 4, ends at 12 and has a height of 2.*
- *Task B is fixed and present, starts at 6, ends at 10 and has a height of -1.*
- *Task C has a minimum start of 0, a maximum end of 16 and a fixed length of 6. Its height is in the interval* $[1, 4]$

The maximum capacity of the resource $\overline{C}$ *is 4. Figure 7 shows the three tasks before propagation. The dashed blue rectangle represents task C maximum energy. The dotted black line shows the minimum profile* $\underline{P}$.

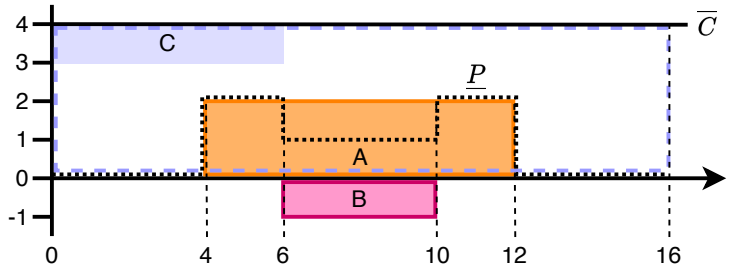

Fig. 7. Tasks of example 6

During the execution of the timetabling algorithm, the maximum height of task C is adjusted to 3 as this value corresponds to the maximum available height over its minimum overlapping interval (from time 5 to time 10). This filtering does not occur with the `GeneralizedCumulative` *constraint from (Beldiceanu and Carlsson 2002) as the* $\underline{e}_C$ *and* $\overline{s}_C$ *of task C are not in the same event due to the presence of task B. If the task B is removed, then both algorithms obtain the same filtering as the height of task C is adjusted to 2.*

Length Adjustment Difference. The difference in filtering on the maximum length adjustment of the tasks occurs if the profile prevents a task to be present in at least two different parts of its time window. In this case, the algorithm presented in this paper identifies the longest interval in the profile where the task can fit and adjusts the task's maximum length accordingly.

In contrast, the filtering of (Beldiceanu and Carlsson 2002) one single conflict point at a same time. When encountering conflicting time point in the profile, the task maximum length is updated based on the maximum length between the conflicting time point and either the minimum start or the maximum end of the task. Example 7 illustrates this.

Example 7. *Let us consider three tasks:*

- *Task A is fixed and present, starts at 3, ends at 6 and has a height of 3.*
- *Task B is fixed and present, starts at 10, ends at 14 and has a height of 3.*
- *Task C has a minimum start of 0, a maximum end of 16 and a height of 2. Its length is in the interval* $[2, 16]$

The maximum capacity of the resource $\overline{C}$ is 4. Figure 8 shows the tree tasks before propagation. The dashed blue rectangle represents task C maximum energy. The black dotted line shows the maximum capacity ($\overline{C}$). In this example, both tasks A and B prevent task C to be present at the same time.

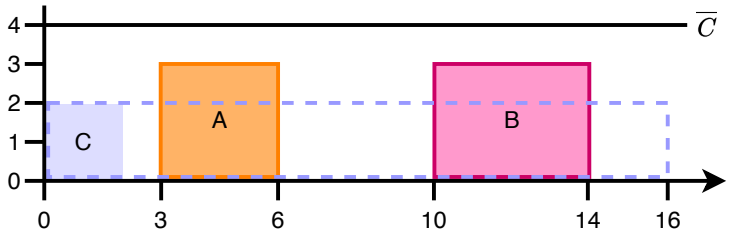

Fig. 8. Tasks of example 7

During the execution of the timetabling algorithm, the maximum length of task C is adjusted to 4 as it is the length of the longest interval where the task can be placed (from time 6 to time 10). When filtering with the propagation algorithm from (Beldiceanu and Carlsson 2002), the length is instead adjusted to 10, since this corresponds to both the length available after task A and the length available before task B.

4.2 Closed source solvers

Both CPOptimizer (Laborie, Rogerie, Shaw, et al. 2018) and OptalCP (Vilím and Pons 2023) have an implementation of the cumulative functions modeling paradigm but the details of their filtering algorithms has not been published and both solvers are closed-source. Therefore, the exact filtering used for their range constraints on cumulative functions is difficult to assert. Based on the filtering and the number of backtracks on some examples, we observed that CPOptimizer achieves a filtering similar to the Gecode implementation when a cumulative function requires a generalized cumulative constraint. For specific cases, like when a cumulative function models a non-overlap constraint, the constraint is reformulated by CPOptimizer into a standard disjunctive constraint for which strong dedicated filtering exists (such as edge-finding, not-first/not-last, etc.).

Contrary to ours, the CPOptimizer modeling API does not allow cumulative functions with a negative height. Indeed, while cumulative tasks can have a negative contribution, a cumulative function with a negative height will cause a failure in CPOptimizer. This is equivalent to having a constraint $f \geq 0$ on any cumulative function f used in the model. It is not known whether this limitation is due to a design choice or a limitation in the filtering algorithm used for the constraint.

5 Experiments

Our approach and filtering algorithm are open source and available in MaxiCP (Schaus et al. 2024), a Java solver that extends MiniCP (Michel et al. 2021). We compare it with the closed-source state-of-the-art commercial solver CPOptimizer (Laborie, Rogerie, Shaw, et al. 2018) as well as the open-source solver Gecode (Schulte et al. 2006) that implements a version of the `GeneralizedCumulative` constraint of (Beldiceanu and Carlsson 2002). As cumulative functions are not available in Gecode, we carefully modeled them in a way similar to how they would be flattened by Algorithm 1. Three cumulative problems that involve reservoirs or negative consumptions are considered. We use the same fixed static search strategy for all solvers and problems to ensure that only the filtering strength and propagation speed influence the results. The experiments were carried out on a MacBook Pro M3 with 32GB of memory. The source code of the models is available in the supplementary material.

For each problem, we report the results as a plot of the cumulative number of instances solved within a virtual time limit. We also provide a pairwise comparison of solvers on the number of backtracks for instances that were commonly solved.

Each model involves a set of conditional task intervals $i \in T$. Recall that the attributes for each interval are s_i the start time, d_i the duration, e_i the end time, p_i the status of a task (present or absent) when applicable.

5.1 The RCPSP with Consumption and Production of Resources (RCPSP-CPR)

The Resource-Constrained Project Scheduling Problem with Consumption and Production of Resources (RCPSP-CPR) (Koné et al. 2013) is an extension of the classic Resource-Constrained Project Scheduling Problem (RCPSP) (Dike 1964; Ding et al. 2023) with additional reservoir resources that must always be kept positive. Each task is present and, in addition to classical renewable resource consumption (modeled with pulse functions), tasks also consume a specified amount of reservoir resources at their start and produce a specified amount at their end (modeled with stepAtStart and stepAtEnd). The objective is to minimize the makespan.

Model. T denotes the set of tasks, Rn is the set of renewable resources, Rs is the set of reservoir resources, and P is the set of precedences between pairs of tasks. The capacity of each renewable resource $k \in Rn$ is denoted C_k and the initial level of each reservoir resource $u \in Rs$ is denoted C^r_u. Each task i consumes c_{ki} unit of the renewable resource $k \in Rn$, consumes c^-_{ui} unit of reservoir resource $u \in Rs$ at its start time and produces c^+_{ui} unit of reservoir resource $u \in Rs$ at its end time.

The model is written as:

$$minimize \ \max_{i \in T} e_i \tag{11}$$

subject to

$$e_i \leq s_j \qquad \forall (i,j) \in P \tag{12}$$

$$ren_k = \sum_{i \in T} pulse(i, c_{ki}) \qquad \forall k \in Rn \tag{13}$$

$$res_u = step(0, C^r_u) + \sum_{i \in T} stepAtStart(i, -c^-_{ui}) + \sum_{i \in T} stepAtEnd(i, c^+_{ui}) \qquad \forall u \in Rs \tag{14}$$

$$ren_k \leq C_k \qquad \forall k \in Rn \tag{15}$$

$$res_u \geq 0 \qquad \forall u \in Rs \tag{16}$$

The objective (11) is to minimize the makespan. The constraints (12) ensure the precedences between tasks. The renewable and reservoir resources are constrained in (13) and (14) respectively. The resource constraints are posted at (15) and (16) respectively.

The search consists in a static binary branching that selects variables in the order in which tasks are declared in the instance file. The first un-assigned start variables is selected and assigned to its minimum value in the left branch. The right branch removes the minimum value from the domain.

Results. We experiment on 55 instances with 15 tasks adapted from (Koné et al. 2013). A time limit of 600s is set per instance for finding and proving optimality. As shown in Figure 9, MaxiCP solves more instances than Gecode and roughly the same set of instances as CPOptimizer.

Figure 10 reports for each pair of solvers, for each instance solved commonly by both solvers, an x, y- plot of the number of backtracks. The comparison between CPOptimizer and MaxiCP reveals that the number of backtracks is nearly identical, suggesting that CPOptimizer is simply faster on this problem, as both solvers achieve similar levels of filtering on this problem, but MaxiCP requires less backtracks than Gecode form some of the instances.

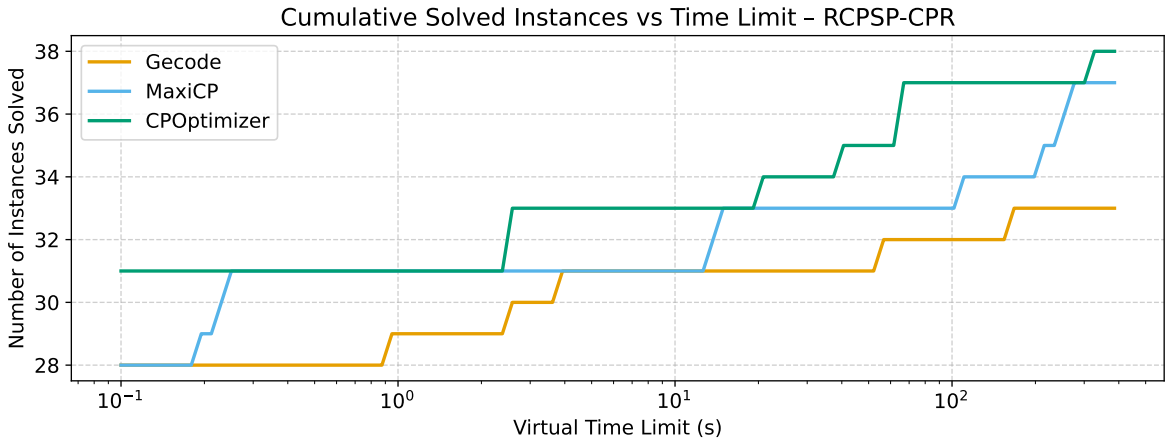

Fig. 9. Number of solved instances in function of time for the RCPSP-CPR Problem.

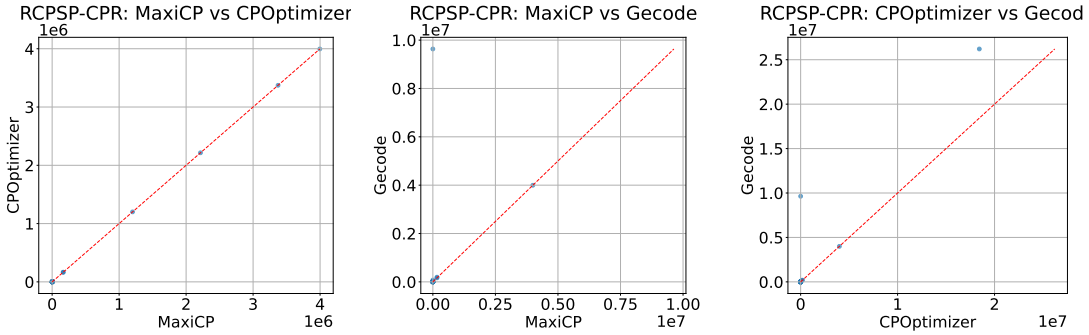

Fig. 10. Failure comparison for the RCPSP-CPR

5.2 The Single Machine with Inventory Constraints (SMIC)

The Single Machine with Inventory Constraints (SMIC) problem was introduced in (Davari et al. 2020). Each task has a release date $release_i$, a fixed duration, and a positive or negative inventory consumption $invent_i$ that occurs at the start of the activity (modeled with stepAtStart). The inventory starts at a given value *initInventory* and must stay within a specified range ($[0, capaInventory]$). Activities cannot overlap in time. The objective is to minimize the makespan.

Model. The model is written as:

$$minimize \ \max_{i \in T} e_i \tag{17}$$

subject to

$$s_i \geq release_i, \qquad \forall i \in T \tag{18}$$

$$res = step(0, initInventory) + \sum_{i \in T} stepAtStart(i, invent_i), \qquad \forall i \in T \tag{19}$$

$$0 \leq res \leq capaInventory \tag{20}$$

$$noOverlap = \sum_{i \in T} pulse(i, 1), \qquad \forall i \in T \tag{21}$$

$$noOverlap \leq 1 \tag{22}$$

The objective (17) is to minimize the makespan (i.e., the time at which all tasks are completed). The release dates are ensured by the relation (18). The reservoir resource is constrained in relations (19) and (20) to ensure that the capacity of the inventory is not violated. The noOverlap constraint is enforced with the relations (21) and (22) to ensure the no-overlapping of activities. Notice that those are modeled using cumulative functions rather than with global dedicated non-overlap constraints on task intervals to ensure that the results are solely impacted by the filtering of the cumulative constraint rather than by the possibly different filtering of the non-overlap constraint. This, unfortunately, did not appear possible for CPOptimizer which recognizes that the cumulative function is used to model a non-overlap constraint and automatically reformulates it using a non-overlap constraint with stronger filtering.

The search consists in a static branching heuristic that selects variables in the order in which tasks are declared in the instance file. The first un-assigned start variables is selected and assigned to its minimum value in the left branch. The right branch removes the minimum value from the domain.

Results. We experiment on instances with 10 tasks from (Davari et al. 2020). A time limit of 600s is set per instance for finding and proving optimality. As shown in Figure 11 CPOptimizer solves more instances while MaxiCP and Gecode roughly have the same behavior on this problem.

Figure 12 reports for each pair of solvers, for each instance solved commonly by both solvers, an x, y- plot of the number of backtracks. This analysis shows that CPOptimizer requires significantly fewer backtracks on this problem, while Gecode and MaxiCP are not on par for this problem in terms of time and number of backtracks. Unfortunately the results for CPOptimizer for this problem are biased as it appears to reformulate the cumulative function for modeling the non-overlap into a standard disjunctive constraint for which strong dedicated filtering exist such as (Vilím 2004).

5.3 The Maximum Energy Scheduling Problem (MESP)

The Maximum Energy Scheduling Problem (MESP) consists in scheduling optional tasks with variable durations and resource demands, which can be positive or negative, on a single resource with fixed capacity $\overline{C}$. The energy of a task is the product of the duration and the demand. The objective is to maximize the total positive energy consumption of the resources while ensuring that the resource capacity is not exceeded at any time. All filtering rules are beneficial for this problem, making it the most relevant benchmark for evaluating the proposed filtering algorithm.

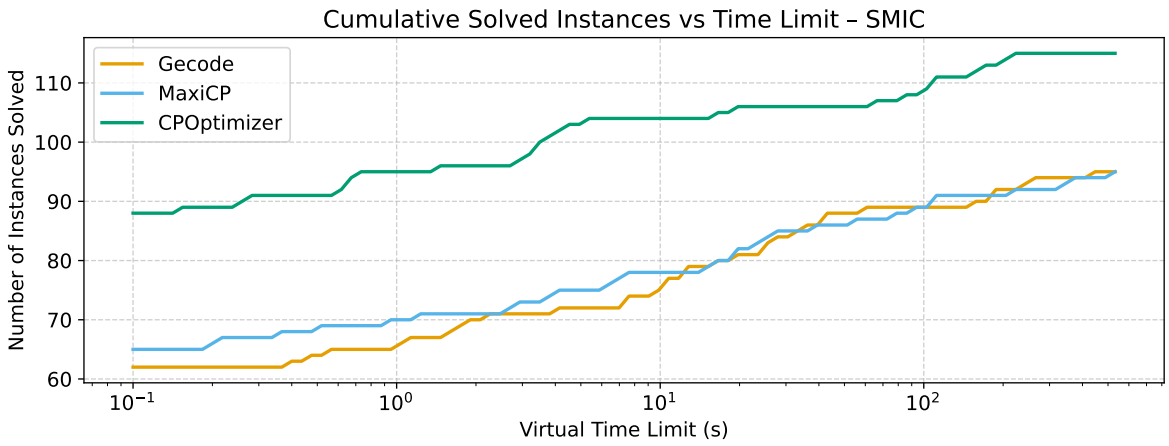

Fig. 11. Number of solved instances in function of time for the SMIC Problem

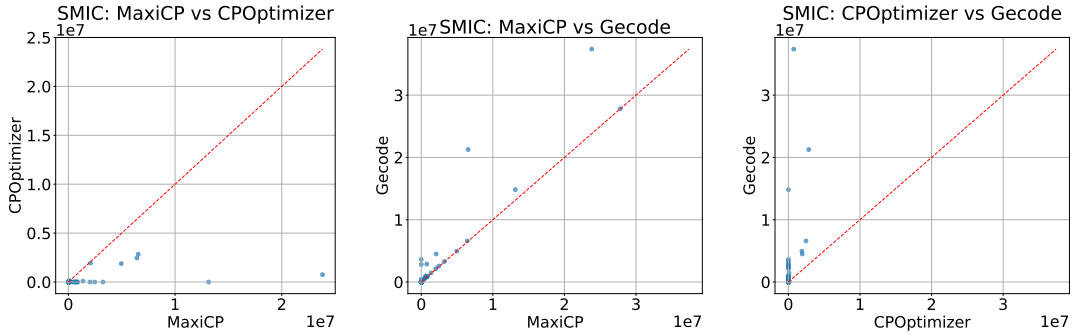

Fig. 12. Failure comparison for the SMIC

The model is written as:

$$maximize \sum_{i \in T | p_i = true} \max(c_i, 0) \cdot d_i \tag{23}$$

subject to

$$resource = \sum_{i \in T} pulse(i, c_i) \tag{24}$$

$$resource \leq \overline{C} \tag{25}$$

The objective (23) is to maximize the sum of the positive energies of the tasks that are present. The constraint (24) sets up the reservoir resource and its capacity is constrained by (25).

Note that the resource demand variables of tasks c_i may have negative values in their domain in this problem. As explained in *Filtering Differences* section, CPOptimizer does not allow cumulative functions to have a negative height. In order to accommodate this restriction, the model for CPOptimizer has been adapted in the following

way: The cumulative function *resource* is shifted up in height at the time 0 by a step function of height $neg^* = \sum_{i \in T} - \min(\underline{c_i}, 0)$ which corresponds to the sum of all maximum negative values of the tasks resources demands. The capacity $\overline{C}$ of the resource is also shifted by the same value. In formal terms, constraint (24) is changed to: $resource = step(0, neg^*) + \sum_{i \in T} pulse(i, c_i)$ and constraint (25) is changed to $resource \leq \overline{C} + neg^*$. This ensures that the cumul function *resource* remains non negative, as required by CPOptimizer.

The search heuristic consists in selecting tasks in the fixed order of their declaration in the instance file. For each task, the following variables are fixed in this order:

- Its presence p_i to true
- Its demand c_i to the maximum value in the domain
- Its duration d_i to the maximum value in the domain
- Its end time e_i to the maximum value in the domain

Two branches are created: The left branch fixes the value while the right branch removes it from the domain. The search stops at the first solution found.

Results. We generated 60 instances with a number of tasks ranging between 6 and 12800. A time limit of 2,000s and a memory limit of 16GB is set per instance for finding a feasible solution. Given the large number of tasks in some instances, this experiment is designed to test the scalability of the filtering. As can be observed in Figure 13, our approach performs very well with more instances solved. Both Gecode and CP Optimzer fail to solve largest instances due to timeouts and memory limits respectively.

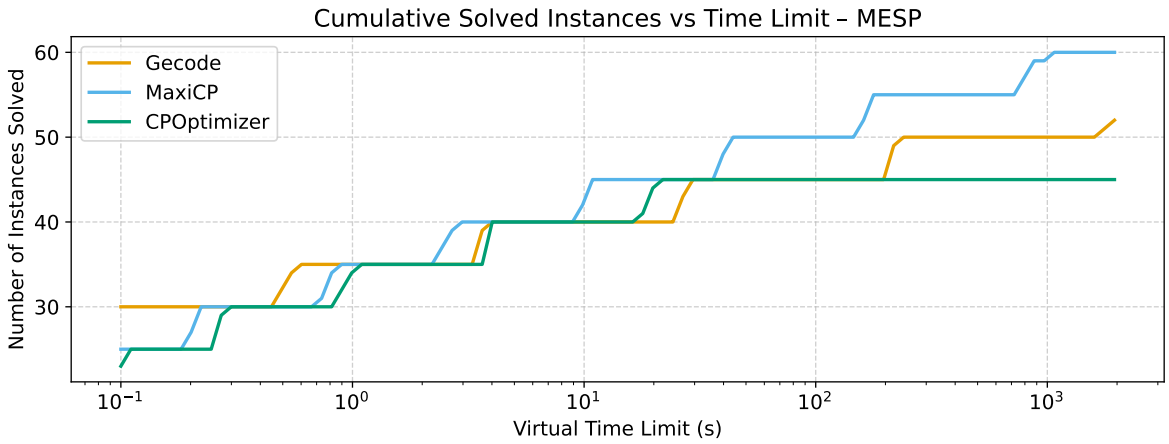

Fig. 13. Number of solved instances in function of time for the MESP Problem.

Figure 14 reports for each pair of solvers, for each instance solved commonly by both solvers, an x, y- plot of the number of backtracks. As can be seen, our approach avoids backtracking, unlike the other solvers. Interestingly, CPOptimizer requires fewer backtracks than Gecode although it solves less instances.

5.4 Why not compare with solvers using Minizinc?

The MiniZinc modeling language provides a `cumulatives` constraint that allows for negative consumption. Unfortunately, with the exception of Sicstus (Carlsson and Mildner 2012) and Gecode (Schulte et al. 2006), this

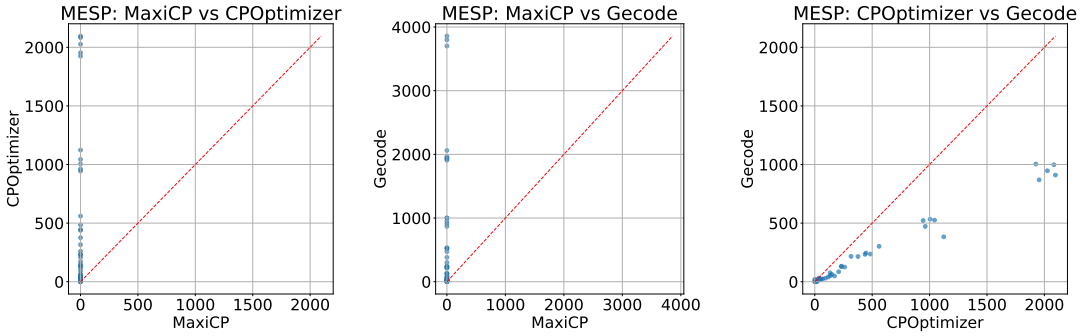

Fig. 14. Failure comparison for the MESP

constraint is decomposed in other solvers. Also the MiniZinc API for this constraint does not support explicit access to the *end* variables. Being able to access the *end* variables directly rather than relying on the relation *start + duration = end* is crucial for the problems considered, as the *end* variables can be constrained independently, as explained in the next example.

Example 8. *Assume a task with the following properties:* $s = [3, 7]$, $d = [1, 5]$, $e = [8, 8]$. *The mandatory part of this task is* $[\overline{s}, \underline{e} - 1] = [7, 7]$. *If the end variable is not available, the mandatory part inferred from* s *and* d *is* $[\overline{s}, \underline{s} + \underline{d} - 1] = \phi$.

As a result, the propagation when using MiniZinc for these problems can be much weaker, making the comparison unfair. This was confirmed experimentally and can be verified with the provided MiniZinc models in the appendix.

6 Conclusion

We presented an implementation of the cumulative functions modeling paradigm for scheduling problems. Our implementation applies a flattening procedure to cumulative function expressions, producing a set of activities with positive or negative resource consumption, which are then passed to a `GeneralizedCumulative` constraint.

We introduced a novel time-tabling filtering algorithm for the `GeneralizedCumulative` constraint that operates on conditional task intervals and supports negative resource consumption. This algorithm is simpler and achieves stronger pruning than the one proposed by (Beldiceanu and Carlsson 2002).

Experimental results demonstrate that the proposed approach is scalable and competitive with other solvers on three cumulative scheduling problems with diverse characteristics, including optional tasks and negative resource consumption.

As future work, we plan to extend the filtering algorithm to generalize the Overload/Underload checking and edge-finding rules in the context of producers and consumers. We also intend to explore explanations for the generalized cumulative constraint, similar to those proposed in (Schutt et al. 2011), and to investigate whether a CP-SAT solver such as OR-Tools (Perron et al. 2023) could benefit from them.

References

N. Beldiceanu and M. Carlsson. 2002. "A New Multi-resource cumulatives Constraint with Negative Heights." In: *CP* (Lecture Notes in Computer Science). Vol. 2470. Springer, 63–79.

Q. Cappart and P. Schaus. 2017. "Rescheduling railway traffic on real time situations using time-interval variables." In: *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*. Springer, 312–327.

Q. Cappart, C. Thomas, P. Schaus, and L.-M. Rousseau. 2018. "A constraint programming approach for solving patient transportation problems." In: *International Conference on Principles and Practice of Constraint Programming (CP)*. Springer, 490–506.

M. Carlsson et al.. June 2025. *SICStus Prolog User's Manual*. (Release 4.10.1 ed.). Version 4.10.1. RISE Research Institutes of Sweden AB. Kista, Sweden, (June 2025). https://sicstus.sics.se/sicstus/docs/latest4/pdf/sicstus.pdf.

M. Carlsson and P. Mildner. 2012. "SICStus Prolog - The first 25 years." *Theory Pract. Log. Program.*, 12, 1-2, 35–66.

M. Davari, M. Ranjbar, P. De Causmaecker, and R. Leus. 2020. "Minimizing makespan on a single machine with release dates and inventory constraints." *European Journal of Operational Research*, 286, 1, 115–128.

S. H. Dike. 1964. "Project scheduling with resource constraints." *IEEE Transactions on Engineering Management*, EM-11, 4, 155–157. doi:10.1109/TEM.1964.6446423.

H. Ding, C. Zhuang, and J. Liu. 2023. "Extensions of the resource-constrained project scheduling problem." *Automation in Construction*, 153, 104958. doi:https://doi.org/10.1016/j.autcon.2023.104958.

H. Fahimi, Y. Ouellet, and C. Quimper. 2018. "Linear-time filtering algorithms for the disjunctive constraint and a quadratic filtering algorithm for the cumulative not-first not-last." *Constraints*, 23, 3, 272–293.

M. R. Garey and D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

S. Gay, R. Hartert, and P. Schaus. 2015a. "Simple and Scalable Time-Table Filtering for the Cumulative Constraint." In: *CP* (Lecture Notes in Computer Science). Vol. 9255. Springer, 149–157.

S. Gay, R. Hartert, and P. Schaus. 2015b. "Time-table disjunctive reasoning for the cumulative constraint." In: *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*. Springer, 157–172.

R. Gedik, D. Kalathia, G. Egilmez, and E. Kirac. 2018. "A constraint programming approach for solving unrelated parallel machine scheduling problem." *Computers & Industrial Engineering*, 121, 139–149.

V. Gingras and C. Quimper. 2016. "Generalizing the Edge-Finder Rule for the Cumulative Constraint." In: *IJCAI*. IJCAI/AAAI Press, 3103–3109.

J. Kinable, T. Wauters, and G. V. Berghe. 2014. "The concrete delivery problem." *Computers & Operations Research*, 48, 53–68.

D. Kizilay, D. T. Eliiyi, and P. Van Hentenryck. 2018. "Constraint and mathematical programming models for integrated port container terminal operations." In: *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*. Springer, 344–360.

O. Koné, C. Artigues, P. Lopez, and M. Mongeau. 2013. "Comparison of mixed integer linear programming models for the resource-constrained project scheduling problem with consumption and production of resources." *Flexible Services and Manufacturing Journal*, 25, 25–47.

R. Kumar, G. Sen, S. Kar, and M. K. Tiwari. 2018. "Station dispatching problem for a large terminal: A constraint programming approach." *Interfaces*, 48, 6, 510–528.

P. Laborie and J. Rogerie. 2008. "Reasoning with Conditional Time-Intervals." In: *FLAIRS*. AAAI Press, 555–560.

P. Laborie, J. Rogerie, P. Shaw, and P. Vilím. 2009. "Reasoning with Conditional Time-Intervals. Part II: An Algebraical Model for Resources." In: *FLAIRS*. AAAI Press.

P. Laborie, J. Rogerie, P. Shaw, and P. Vilím. 2018. "IBM ILOG CP optimizer for scheduling: 20+ years of scheduling with constraints at IBM/ILOG." *Constraints*, 23, 210–250.

A. Letort, M. Carlsson, and N. Beldiceanu. 2015. "Synchronized sweep algorithms for scalable scheduling constraints." *Constraints*, 20, 2, 183–234.

C. Liu, D. M. Aleman, and J. C. Beck. 2018. "Modelling and solving the senior transportation problem." In: *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*. Springer, 412–428.

C. Mears, A. Schutt, P. J. Stuckey, G. Tack, K. Marriott, and M. Wallace. 2014. "Modelling with option types in MiniZinc." In: *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*. Springer, 88–103.

L. Michel, P. Schaus, and P. Van Hentenryck. 2021. "MiniCP: a lightweight solver for constraint programming." *Mathematical Programming Computation*, 13, 1, 133–184. ISBN: 1867-2957. doi:10.1007/s12532-020-00190-7.

Y. Ouellet and C.-G. Quimper. 2019. "Processing times filtering for the Cumulative constraint." In: *Doctoral Program of the International Conference on Principles and Practice of Constraint Programming (CP)*.

L. Perron, F. Didier, and S. Gay. 2023. "The CP-SAT-LP Solver." In: *International Conference on Principles and Practice of Constraint Programming (CP)* (Leibniz International Proceedings in Informatics (LIPIcs)). Ed. by R. H. C. Yap. Vol. 280. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 3:1–3:2. ISBN: 978-3-95977-300-3. doi:10.4230/LIPIcs.CP.2023.3.

P. Schaus, G. Derval, A. Delecluse, L. Michel, and P. V. Hentenryck. 2024. *MaxiCP: A Constraint Programming Solver for Scheduling and Vehicle Routing*. (2024). https://github.com/aia-uclouvain/maxicp.

C. Schulte, M. Lagerkvist, and G. Tack. 2006. "Gecode." *Software download and online material at the website: http://www. gecode. org*, 11–13.

A. Schutt, T. Feydy, P. J. Stuckey, and M. G. Wallace. 2011. "Explaining the cumulative propagator." *Constraints*, 16, 3, 250–282.

C. Thomas and P. Schaus. 2024. "A Constraint Programming Approach for Aircraft Disassembly Scheduling." In: *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer, 211–220.

P. Vilím. 2004. "O (nlog n) filtering algorithms for unary resource constraint." In: *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*. Springer, 335–347.

P. Vilím and D. O. F. Pons. 2023. *OptalCP*. OptalCP Scheduling Solver. Available online. (2023). https://optalcp.com.