Abstract

Some constraint programming solvers and constraint modelling languages feature the $\text{Sort}(L, P, S)$ constraint, which holds if $S$ is a nondecreasing rearrangement of the list $L$, the permutation being made explicit by the optional list $P$. However, such sortedness constraints do not seem to be used much in practice. We argue that reasons for this neglect are that it is impossible to require the underlying sort to be stable, so that $\text{Sort}$ cannot be guaranteed to be a total-function constraint, and that $L$ cannot contain tuples of variables, some of which form the key for the sort. To overcome these limitations, we introduce the $\text{StableKeysort}$ constraint, decompose it using existing constraints, and propose a propagator. This new constraint enables a powerful modelling idiom, which we illustrate by elegant and scalable models of two problems that are otherwise hard to encode as constraint programs.

1 Motivation

In order to motivate our work, we consider the following human resource planning problem, which will serve as running example throughout this paper. Consider $n$ tasks that are to be assigned to a given set of $m$ employees. For each task $i \in 1..n$, let attribute $A_i$ denote the initially unknown employee performing it, $B_i$ its fixed beginning time, $D_i$ its fixed duration, and $E_i$ its fixed end time, with $E_i = B_i + D_i$ for all $i \in 1..n$. A first constraint is that tasks assigned to the same employee should not overlap in time. A second constraint deals with the allocation of breaks to each employee. This is done by stating a $\text{Shift}$ constraint, which we now describe. A shift is a maximum sequence of tasks assigned to a same employee such that the time gap between any two consecutive tasks is shorter than a given threshold $\text{minBreak}$: that is, consecutive shifts of an employee are separated by a break of minimum duration $\text{minBreak}$, and consecutive tasks of a shift are separated by a gap shorter than $\text{minBreak}$. The span of a shift is the difference between the end time of its last task and the beginning time of its first task: see Figure 1. The $\text{Shift}$ constraint holds if and only if the span of each shift of each employee is at most a given threshold $\text{maxSpan}$.
Concisely expressing constraints like \textbf{SHIFT} is hard. We show that sortedness constraints, possibly upon some extensions, enable elegant models of such constraints and problems.

The \textit{contributions} and \textit{organisation} of the rest of this paper are as follows. In Section 2, we discuss the existing sortedness constraints and their limitations. In Section 3, we introduce the \textsc{StableKeysort} constraint, which overcomes those limitations. In Section 4, we argue that composing the new \textsc{StableKeysort} constraint with other constraints is a powerful modelling idiom. We illustrate this idiom by elegant models of the motivating problem of this introduction and another problem. We empirically evaluate our results in Section 5 and conclude in Section 6.

2 The \textsc{Sort} Constraints and Their Limitations

Constraint programming (CP, e.g., [23]) is a problem solving technology where relations between \textit{variables} (or: unknowns) are stated in the form of \textit{constraints}, which together form a \textit{model} of a constraint problem. Each variable \( x \) has a finite \textit{domain} of possible values, usually a set of integers, whose lower and upper bounds are here respectively denoted by \( x \) and \( \bar{x} \). Each constraint has an associated \textit{propagator}, which removes domain values that cannot be part of a solution to that constraint. Depending on the propagators used, the domain reduction for all constraints of the model, called \textit{propagation}, can be more or less effective: the notion of \textit{consistency} characterises propagator effectiveness. Some propagators only adjust the domain bounds, achieving \textit{bounds(Z) consistency} [12] for example. A propagator that only keeps domain values that participate in a solution to its constraint is said to achieve \textit{domain consistency} [23]. A \textit{solution to a constraint} is obtained when the domains of all its variables are singletons: we say that the variables are \textit{fixed}. If propagation alone does not lead to a \textit{solution to the constraint problem}, that is to a solution to each of its constraints, then \textit{search} is interleaved with further propagation at every node of the search tree, with \textit{backtracking} whenever some domain is shrunk to the empty set. A constraint, especially when no propagator is available for it in the used CP solver, can be represented by a conjunction of other constraints, called a \textit{decomposition} of the constraint.

The \textsc{Sort}(L, S) constraint was introduced, without exhibiting a propagator, by Older \textit{et
A Modelling Pearl with Sortedness

Beldiceanu, Carlsson, Flener, Lorca, Pearson, Petit, and Prud’homme

For expressing disjunctive constraints in job-shop scheduling problems: it holds if and only if the list $S$ of variables is a nondecreasing rearrangement of the list $L$ of variables. Also motivated by the job-shop problem, Zhou [30, 31] introduced, also without exhibiting a propagator, the $\text{SORT}(L, P^{-1}, S)$ constraint, where the permutation $P^{-1}$ of the integer interval $1..n$ linking the variables of $S$ to those of $L$ is made explicit: we have $L_i = S_{P^{-1}_i}$ for all positions $i$. (Our reason for referring to the permutation as $P^{-1}$ will become clear in Section 3.1.) For example, $\text{SORT}([5, 4, 4], [3, 1, 2], [4, 4, 5])$ holds. Introducing the variables of $P^{-1}$ and fixing them during search is convenient for expressing a branching heuristic that selects which task is scheduled before or after a group of tasks, without explicitly adding precedence constraints during search [30, 31]. A modelling use of $P^{-1}$ will be given at the end of Section 3.2.

Bleuzen-Guernalec and Colmerauer [5, 6] proposed the first bounds($Z$) consistency [12] propagator for $\text{SORT}(L, S)$: it has a worst-case time complexity of $O(n \log n)$, where $n$ is the length of the lists $L$ and $S$. When sorting can be achieved in linear time, an $O(n)$ bounds($Z$) consistency propagator for $\text{SORT}(L, S)$ was introduced by Mehlhorn and Thiel [15], maintaining convexity [14] of the underlying bipartite graph. It was generalised [27] for the $\text{SORT}(L, P^{-1}, S)$ constraint of SICStus [11, 10], ECLiPSe [26], and Gecode [13, 19], at the expense of bounds consistency. The latter two provide both versions, Choco [21] and or-tools [18] only provide the two-argument version, whereas all other CP solvers we checked provide neither of them.

Even if the $\text{SORT}$ constraint, with or without the permutation variables, is available in a few CP solvers, as well as in the constraint modelling language MiniZinc [16], it does not seem to be used much. Beyond the original papers, we found very few published uses of $\text{SORT}$, notably [2, 8], and very few uses in public repositories of constraint models. In [7], a mathematical model using $\text{SORT}$ is given for a meeting scheduling problem, but the translation into MiniZinc overlooks the $\text{SORT}$ constraint of MiniZinc. The $\text{SORT}(L, S)$ constraint was mentioned in [4] as a way to help decompose constraints. For example, the constraint $\text{NVALUE}(\{v_1, \ldots, v_n\}, N)$, which holds if and only if there are $N$ distinct values in the sequence $[v_1, \ldots, v_n]$, can be decomposed by constraining $N$ to be the number of strict increases between adjacent variables in the sorted rearrangement of $[v_1, \ldots, v_n]$. While this is an important observation, the use of the $\text{SORT}$ constraint for decomposing constraints reveals two limitations that we discuss now.

### 2.1 The $\text{SORT}(L, P^{-1}, S)$ Constraint Is Not a Total-Function Constraint

The $\text{SORT}(L, P^{-1}, S)$ constraint is not a total-function constraint [4], as a ground list $L$ does not uniquely determine $P^{-1}$ and $S$. For example, $\text{SORT}([5, 4, 4], [3, 1, 2], [4, 4, 5])$ holds, but $\text{SORT}([5, 4, 4], [3, 2, 1], [4, 4, 5])$ also holds, with a different permutation for the duplicate elements inside $L$ and $S$. The problem is that the $\text{SORT}(L, P^{-1}, S)$ constraint does not require the sort to be stable, that is to use the position of a variable in $L$ in order to break ties deterministically. Not uniquely determining $P^{-1}$ from a list $L$ with duplicates is problematic for constraint decompositions involving $\text{SORT}(L, P^{-1}, S)$ where the variables of $P^{-1}$ are used in other constraints whose propagation is hindered until all variables of $P^{-1}$ are fixed. For example, $\text{SORT}([5, 4, 4], [P^{-1}_1, P^{-1}_2, P^{-1}_3], [S_1, S_2, S_3])$ yields $S_1 = 4$, $S_2 = 4$, $S_3 = 5$, $P^{-1}_1 = 3$, $P^{-1}_2 \in 1..2$, and $P^{-1}_3 \in 1..2$: the last two variables are not fixed. In Section 4.2, we present a model where stability is crucial for effective propagation.

To overcome this difficulty, the modeller must be aware of the variables of $P^{-1}$ when they are not part of the overall problem. While this is possible, the effect is that the decomposition cannot be used in a transparent way, as the variables of $P^{-1}$ must be considered during search. Determining when to fix the variables of $P^{-1}$ during search is not straightforward: on the one hand, fixing them too late may be inefficient since subproblems may be recognised belatedly as
infeasible when these variables are not fixed; on the other hand, fixing them too early and in a wrong order may lead to a lot of backtracks.

In this paper, we argue that it is preferable to make \( \text{SORT}(L, P^{-1}, S) \) a total-function constraint, based on a stable sort.

### 2.2 The Sort Constraints Cannot Handle Lists of Keyed Tuples

Quite often, rather than constraining a sorted list of variables, one needs to constrain a lexicographically sorted list of tuples of variables, where each tuple typically corresponds to the attributes of an object, some of which form the sorting key. Without loss of generality, we assume the first \( k \geq 1 \) fields of a tuple form the sorting key.

For example, a task \( i \) of the motivating problem in Section 1 yields a tuple \( \langle A_i, B_i, D_i, E_i \rangle \) of \( q = 4 \) fields: modelling the \textsc{Shift} constraint is facilitated when constraining the list of task tuples that is lexicographically sorted on the key \( \langle A_i, B_i \rangle \) of their first \( k = 2 \) fields, thereby bundling the tasks assigned to a same employee and ordering them by increasing beginning time, so that the shifts of each employee become apparent. The full details of this decomposition will be given in Section 4.1.

In this paper, we argue for the generalisation of \textsc{Sort} to a constraint on lists of keyed tuples.

### 3 The StableKeysort constraint

To overcome the limitations of the existing \textsc{Sort} constraints, as outlined in Section 2, we now introduce the \textbf{StableKeysort} constraint (Section 3.1), decompose it using existing constraints (Section 3.2), and discuss a dedicated propagator (Section 3.3).

#### 3.1 Specification of StableKeysort

The \textbf{StableKeysort}\((L, P, S, k)\) constraint, where \( L \) and \( S \) are lists of \( n \geq 0 \) tuples of \( q \geq 1 \) variables per tuple, \( P \) is an optional list of \( n \) variables, and \( k \) is an integer value in \( 1 \ldots q \), holds if and only if the following three conditions hold:

1. All variables of \( P \) are assigned distinct values in the interval \( 1 \ldots n \).
2. The tuples of \( S \) correspond to those of \( L \) according to the permutation \( P \): we have \( S_i = L_{P_i} \) for all positions \( i \). Note that we here prefer, for ease of modelling the decompositions below, to define \( P \) as the inverse of \( P^{-1} \) in \textsc{Sort}(\( L, P^{-1}, S \)), where \( L_i = S_{P_i^{-1}} \) for all \( i \) (see Section 2).
3. The tuples of \( S \) form a rearrangement of \( L \) that is stably sorted by nondecreasing lexicographical order on the first \( k \) positions in the tuples.

If the permutation argument \( P \) is omitted, then only condition 3 needs to hold. As we will see in Section 4, (the inverse of) \( P \) is often unnecessary for modelling purposes when one uses \textbf{StableKeysort} instead of \textsc{Sort}, so that the main use of \( P \) may lie in expressing search heuristics.

#### 3.2 Decomposition of StableKeysort

We now give a decomposition of \textbf{StableKeysort}\((L, P, S, k)\) in terms of \textsc{Sort}, \textsc{Element}, and arithmetic constraints. The idea is based on having a unique key \( K_i \) that is a linear combination of the \( k \) first fields of tuple \( L_i \) as well as position \( i \) itself, for stability: the permutation \( P \) linking
A Modelling Pearl with Sortedness

Beldiceanu, Carlsson, Flener, Lorca, Pearson, Petit, and Prud’homme

\textbf{StableKeysort}((L_1, \ldots, L_n), [P_1, \ldots, P_n], [S_1, \ldots, S_n], k) \iff (1) \land (2) \land (3) \land (4)

\begin{align*}
K_i &= \sum_{j \in 1..k} c_j \cdot L_{i,j} + i - 1, \; \forall i \in 1..n \quad (1) \\
\text{Sort}([K_1, \ldots, K_n], [K'_1, \ldots, K'_n])
\end{align*}

\begin{align*}
P_i &= (K'_i \mod n) + 1, \; \forall i \in 1..n \quad (2) \\
S_{i,j} &= L_{P_{i,j}} + i - 1, \; \forall i \in 1..n, \forall j \in 1..q \quad (3)
\end{align*}

where the constant bounds \( \ell_j \) and \( u_j \) and the constant coefficients \( c_j \) are determined as follows:

\begin{align*}
\ell_j &= \min\{L_{i,j} \mid i \in 1..n\}, \; \forall j \in 1..k \\
u_j &= \max\{L_{i,j} \mid i \in 1..n\}, \; \forall j \in 1..k \\
c_j &= \begin{cases} 
(u_j+1 - \ell_j+1) \cdot c_{j+1} & \text{if } 1 \leq j < k \\
\frac{n}{n} & \text{if } j = k
\end{cases}
\end{align*}

\textbf{Decomposition 1}: Decomposition of the \textbf{StableKeysort} constraint.

Using the remark on condition 2 of the specification of \textbf{StableKeysort}, the constraints (2) and (3) can be reformulated without arithmetic as

\begin{align*}
\text{Sort}([K_1, \ldots, K_n], [P_1^{-1}, \ldots, P_n^{-1}], [K'_1, \ldots, K'_n]) \\
\land \text{Inverse}([P_1, \ldots, P_n], [P_1^{-1}, \ldots, P_n^{-1}])
\end{align*}

when the constraint \text{Sort}(L, P^{-1}, S) is available, and where the constraint \text{Inverse}(P, P^{-1}) holds if \( P_i = j \iff P_j^{-1} = i \), for all \( i, j \in 1..n \).

\section{3.3 Propagator for \textbf{StableKeysort}}

We designed and implemented a propagator for \textbf{StableKeysort}, using the Mehlhorn and Thiel propagator [15] (MT) as a starting point. Our propagator runs in \( O(nk \log nk) + O(nq) \).
time. It does not guarantee bounds(\(\mathbb{Z}\)) consistency, as shown at the end of this section. Let key(\(L_i\)) denote the key part of tuple \(L_i\), consisting of the first \(k\) fields of \(L_i\) extended by \(i\). Let key(\(S_j\)) denote the key part of tuple \(S_j\), consisting of the first \(k\) fields of \(S_j\) extended by \(P_i\). The function of each last position is twofold: it breaks ties among otherwise equal keys, ensuring stability, and it allows \(P\) to be obtained from the sorted key parts.

The core of MT for \(\text{Sort}(L,S)\) is the construction of a bipartite intersection graph, with an arc from \(L_i\) to \(S_j\) if \(L_i, \overline{L_i}\) and \(S_j, \overline{S_j}\) have a nonempty intersection. Then, the propagator computes a perfect matching, a reduced intersection graph, and its strongly connected components (SCCs), which are used for pruning the domains of the variables of \(L\). Throughout the algorithm, the domains of the variables of \(S\) are kept normalised, i.e., \(S_i \leq S_{i+1}\) and \(S_i \leq S_{i+1}\) for all \(i\). Normalisation allows MT to use a convex intersection graph, which admits a compact representation and linear-time operations. For details, see Section 1.2 of [15].

Our propagator for \(\text{ StableKeysort} \) is an adaption of MT, using exactly the same phases, followed by one additional phase. In the following, if \(x\) is a tuple of variables, then \(\pi\) and \(\pi\) respectively denote the tuple of lower bounds and the tuple of upper bounds of the domains of those variables. Whenever MT uses a scalar comparison among \(L_i, \overline{L_i}, S_j, \overline{S_j}\) and \(\overline{L_i}\), for example when sorting \(L\) on lower and upper bounds, during normalisation, or during SCC computation, we use a lexicographic comparison among \(\text{key}(L_i)\), \(\text{key}(L_i)\), \(\text{key}(S_j)\), and \(\text{key}(S_j)\). Whenever MT adjusts a lower bound, enforcing \(L_i \geq c\), we enforce \(\text{key}(L_i) \geq_{\text{lex}} c\), using methods from [9], and similarly for \(S_j\) and upper bounds. Our intersection graph contains an arc from \(L_i\) to \(S_j\) if:

\[
\text{key}(L_i) \leq_{\text{lex}} \text{key}(S_j) \land \text{key}(S_j) \leq_{\text{lex}} \text{key}(L_i).
\]

Thus, as described so far, our propagator generalises MT by replacing scalar operations on the bounds of \(L_i\) and \(S_j\) by lexicographic operations on the bounds of \(\text{key}(L_i)\) and \(\text{key}(S_j)\). This, however, ignores the domains of the individual tuple fields and of the variables of \(P\). This is why we need an additional pruning phase, to ensure that every tuple of \(L\) can be equal to at least one tuple of \(S\) in the same SCC, and vice versa. The additional phase removes infeasible values in four steps, for all \(i \in 1..n\) and \(j \in 1..q\), where \(i \sim j\) denotes that \(L_i\) and \(S_j\) belong to the same SCC:

1. The bounds of any position \(j\) of any tuple of \(S\) must be inside the bounds of the same position \(j\) of the tuples of \(L\). That is, enforce:

\[
\overline{S}_i,j \leq \max\{L_{p,j} \mid p \in 1..n \land p \sim i\}
\]

\[
S_{i,j} \geq \min\{L_{p,j} \mid p \in 1..n \land p \sim i\}
\]

2. The bounds of any position \(j\) of any tuple of \(L\) must be inside the bounds of the same position \(j\) of the tuples of \(S\). That is, enforce:

\[
\overline{L}_{i,j} \leq \max\{\overline{S}_{p,j} \mid p \in 1..n \land i \sim p\}
\]

\[
L_{i,j} \geq \min\{\overline{S}_{p,j} \mid p \in 1..n \land i \sim p\}
\]

3. The bounds of any permutation variable must be inside the possible tuple positions. That is, enforce:

\[
\overline{P}_i \leq \max\{p \mid p \in 1..n \land p \sim i\}
\]

\[
P_i \geq \min\{p \mid p \in 1..n \land p \sim i\}
\]
4. Any tuple position must be inside the bounds of the permutation variables. That is, check:

\[ i \leq \max \{ P_p \mid p \in 1..n \land i \sim p \} \]
\[ i \geq \min \{ P_p \mid p \in 1..n \land i \sim p \} \]

The phases that were generalised from MT take \( O(nk) \) time, except for sorting \( L \), which takes \( O(nk \log nk) \) time. The additional phase takes \( O(nq) \) time, yielding a total time complexity of \( O(nk \log nk) + O(nq) \).

The fact that the matching ignores the domains of individual tuple positions and of permutation variables is the main weakness of our propagator and the main reason why it sometimes does not achieve bounds(\( Z \)) consistency. For example:

\[ x, y, z \in 1..3 \land \text{StableKeysort}([[x, 2], [y, 3]], [[2, 3], [z, 2]], 1) \]

will prune the domains to \( x, y, z \in 2..3 \) instead of \( x = 3, y = 2, z = 3 \), which is the only solution. Unfortunately, taking domain information into account would mean losing the convexity of the bipartite graph, but convexity is the property that allows all phases of MT except the initial sort to run in linear time.

Achieving domain consistency for \text{StableKeysort} is NP-hard [24], but is more interesting than achieving bounds consistency, as fields of the key can be attributes such as an employee identifier, for which bounds propagation is meaningless. The tractability of achieving bounds(\( Z \)) consistency for \text{Sort}(L, P^{-1}, S) \text{ and StableKeysort} is still open [24].

4 A Modelling Pearl with \text{StableKeysort}

Our motivation for introducing the \text{StableKeysort} constraint, with or without the permutation argument, is to maintain the relation between a given list of tuples and the corresponding lexicographically sorted tuples on the \( k \) first positions of the given tuples, so that a constraint on the given tuples can be expressed on the sorted tuples instead. Typically, by scanning the sorted tuples we can directly express the required constraint. In other words, the modelling idiom that we want to convey is:

\[
\text{Constraint}(L) \iff \text{StableKeysort}(L, S, k) \land \text{ConstraintOnSequence}(S)
\]

where it is awkward or unknown how to express \text{Constraint} by other means, but well known how to express \text{ConstraintOnSequence}. We now show two instances of this modelling pearl.

4.1 The Shift Constraint

We now revisit the motivating problem of Section 1 and model \text{Shift} as a decomposition into \text{StableKeysort} and simple arithmetical and logical constraints: see Decomposition 2. The decomposition introduces the auxiliary variables \( A'_i, B'_i, D'_i, E'_i \) for a sorted list of task attribute tuples \( (q = 4) \), as well as auxiliary 0..1 variables \( Y_i \), denoting whether two consecutive tasks \( i - 1 \) and \( i \) are performed by the same employee, and \( X_i \), denoting whether task \( i \) is a subsequent task of the current shift. A last set of auxiliary variables are the \( R_i \) over domain \( 1..\text{maxSpan} \), denoting the shift length up to the end of task \( i \). In the sortedness constraint (6), the permutation argument \( P \) is not needed, and stability is not needed (since the tasks assigned to a same employee are not allowed to overlap, so that their beginning times break ties), but crucial use is made of tuples, keyed on the assigned employee identity \( A_i \) and the beginning time
A Modelling Pearl with Sortedness  

### Decomposition 2: Decomposition of the Shift constraint.

$B_i$ of each task ($k = 2$). Constraint (7) prevents two consecutive tasks with the same employee from overlapping. Constraints (9) and (10) channel between the sorted task attributes and the $X_i$ and $R_i$.

#### 4.2 The Cumulative Smooth Constraint

The cumulative scheduling problem (CuSP) is frequently modelled using the Cumulative($T$, $c$) constraint [1]. A CuSP is defined on a set $T$ of $n$ tasks consuming the same resource, with a fixed integer capacity, $c$, of the resource. Each task $i \in 1..n$ is here defined by the following attributes: its beginning time $B_i$, its duration $D_i$, its end time $E_i$, with $E_i = B_i + D_i$, and its height $H_i$, that is the quantity of resource necessary to perform it. Tasks may overlap in time. A solution to a CuSP is a schedule that satisfies the following constraint, requiring that the resource capacity never be exceeded:

$$
\sum_{i \in 1..n} H_i \leq c, \ \forall t \in \mathbb{Z}
$$

In many CuSP applications, in addition to the resource capacity, the profile of resource consumption throughout the schedule is constrained: a frequent constraint is a limit on the number of large resource consumption variations between consecutive time points in this profile. For example, teams often cannot vary at arbitrary times in personnel scheduling: the human resource profile needs to be smoothened; other examples are in [29, Section 5.2]. The Smooth($N, [X_1, X_2, \ldots, X_n], \tau$) constraint [3] holds if and only if $|X_i - X_{i+1}| > \tau$ is satisfied.
A Modelling Pearl with Sortedness

Beldiceanu, Carlsson, Flener, Lorca, Pearson, Petit, and Prud’homme

CumulativeSmooth\(\left(\left\{\left(B_i, D_i, E_i, H_i\right) \mid i \in 1..n\right\}, c, N, \tau\right) \Leftrightarrow (11) \land (12) \land (13) \land (14) \land (15)\)

\[
\text{Cumulative}\left(\left\{\left(B_i, D_i, E_i, H_i\right) \mid i \in 1..n\right\}, c\right) \tag{11}
\]

\[
\text{StableKeysort}\left(\left\{\left(B_i, H_i\right), \left(E_i, -H_i\right) \mid i \in 1..n\right\}, \left(\left(\text{Instant}_i, \Delta_i\right) \mid i \in 1..2n\right), 1\right) \tag{12}
\]

\[
\text{Current}_i = \begin{cases} 
0 & \text{if } i = 0 \lor i = 2n \\
\text{Current}_{i-1} + \Delta_i & \text{if } 1 \leq i < 2n
\end{cases} \tag{13}
\]

\[
\text{Profile}_i = \begin{cases} 
0 & \text{if } i = 0 \lor i = 2n \\
\text{Profile}_{i+1} & \text{if } 1 \leq i < 2n \land \text{Instant}_i = \text{Instant}_{i+1} \\
\text{Current}_i & \text{if } 1 \leq i < 2n \land \text{Instant}_i < \text{Instant}_{i+1}
\end{cases} \tag{14}
\]

\[
\text{Smooth}(N, \left[\text{Profile}_i \mid i \in 0..2n\right], \tau) \tag{15}
\]

**Decomposition 3:** Decomposition of the CumulativeSmooth constraint.

\(N\) times for \(1 \leq i < n\), where \(N\) is an integer variable and \(\tau\) a positive integer. Unfortunately, Cumulative usually does not provide variables representing the resource profile, although the ECLiPSe\cite{26} constraint Profile does, so in most CP solvers, we cannot use Smooth in a CuSP context. In order to address this issue, we introduce the CumulativeSmooth\((T, c, N, \tau)\) constraint, which combines a Cumulative constraint with a Smooth constraint on variables representing the profile. At time \(t\), the profile height is the accumulated height, that is \(\sum_{i \in 1..n: B_i \leq t < E_i} H_i\). Such a profile is built in three steps:

1. Construct for each task two pairs \(\left(q = 2\right)\): one with the task beginning time and height, the other with the task end time and negated height: see the first argument of the StableKeysort constraint (12).

2. Sort these pairs on their first item \(\left(k = 1\right)\), i.e., on the beginning and end times of the tasks: see constraint (12).

3. Build the profile from the sorted list obtained in phase 2: see constraints (13) and (14).

The profile height variables admit the statement of a Smooth constraint on the number of big variations between consecutive time points: see constraint (15). All this leads to a decomposition of CumulativeSmooth that uses StableKeysort: see Decomposition 3 and note how all arithmetic constraints are elegantly stated by reasoning on the sorted lists of variable tuples.

In the sortedness constraint (12), the permutation argument \(P\) is not needed, but crucial use is made of keyed tuples and stability: without stability, the \(\Delta_i\) associated to identical instants in constraint (12) will only be fixed during search, leading to poor propagation.

Figure 2 shows an example CuSP, the variables that arise from the decomposition of its CumulativeSmooth instance, and their values.
Figure 2: A cumulative scheduling problem and the values of the variables that arise from the decomposition of CumulativeSmooth([⟨Bi, Di, Ei, Hi⟩ | i ∈ 1..6], 6, 0, 3): the cumulative profile should never exceed c = 6 and the profile variation (shown by vertical arrows) should never (N = 0) exceed τ = 3.

4.3 Synthesis

Both the Shift and CumulativeSmooth examples can be seen as typical instances of problems in which one needs to generate profiles represented by sorted events where side constraints, such as work regulation rules for employees in the case of Shift, or preferences on the resource profile in the case of CumulativeSmooth, should be added to the model before search. As work regulation rules and preferences on the resource profile can vary a lot, depending on the peculiarity of the application, a modelling idiom makes sense since implementing a dedicated propagator for each variant would be very time consuming. One may observe that Decomposition 3, for instance, can easily be generalised: the Smooth constraint (15) can be replaced by other constraints, such as a balancing constraint [25]. Note also that our modelling idiom leads to constraint decompositions that are linear in memory with respect to the number of tasks.

5 Empirical Evaluation

For an empirical evaluation, we revisit the human resource planning problem introduced in Section 1. The time line is split into t 15-minute time slots. The objective is to minimise labour cost, subject to the Shift constraint as defined in Section 4.1. Let dayDuration denote the maximum amount of time slots per day for an employee, and let costa,w denote the cost of employee a when at work for w time slots in one day.

An elegant formulation, with the Shift constraint, of the problem is provided as Model 1. The working time of the employees are maintained (16) in order to compute their costs (17). In our evaluation, Shift is implemented in terms of the proposed new StableKeysort constraint as shown in Decomposition 2.

A less elegant formulation, without the Shift constraint, of the problem is provided as Model 2. Let variable aj,k denote the activity of employee j in slot k: a task is represented by its index from 1 to n; a break shorter than minBreak is represented by the special value n + 1; a long break (between minBreak and maxSpan) is represented by the special value n + 2; and any other inactivity is represented by the special value n + 3. Constraint (18) applies to the sequence of aj,k for each employee j, where the Regular(X, A) constraint [20] holds if and only if the
Minimise

\[ \sum_{j=1}^{n} C_j \]

subject to

\[ \text{SHIFT}([\langle A_i, B_i, D_i, E_i \rangle | i \in 1..n], [\langle A'_i, B'_i, D'_i, E'_i, Y_i \rangle | i \in 1..n], \text{minBreak}, \text{maxSpan}) \]

\[ W_j = \begin{cases} D'_j & \text{if } j = 1 \\ D'_j + Y_j \cdot (W_{j-1} + B'_j - E'_{j-1}) & \text{if } j \in 2..n \end{cases} \quad (16) \]

\[ C_j = \begin{cases} (1 - Y_{j+1}) \cdot \text{cost}_{A'_j, W_j} & \text{if } j \in 1..n-1 \\ \text{cost}_{A'_j, W_j} & \text{if } j = n \end{cases} \quad (17) \]

where

\[ W_j \in 0..\text{dayDuration}, \forall j \in 1..n \]

Model 1: Formulation of the shift problem with the Shift constraint.

word represented by the sequence \( X \) of variables is accepted by the given finite automaton \( A \), where double circles denote accepting states. Here, the automaton is the intersection of one automaton encoding the break rules (depicted in the inner box of Model 2) and one automaton \( \Pi \) encoding the task successor relation, which stems from the fixed beginning and end times. Constraints (19) and (20) ensure that, for each slot, every task is performed by at most one employee, where the GCC(\( X, V, O \)) constraint [22] holds if and only if there are \( O_i \) occurrences of value \( V_i \) within the sequence \( X \) of variables. The non-inactivities of the employees serve to compute their working times, via constraints (21) and (22), and their costs, with \( 0..1 \) variable \( w_{j,k} \) denoting whether employee \( j \) is incurring cost in time slot \( k \).

We compare Model 2 against two implementations of Model 1, namely one where the required StableKeysort constraint is implemented by Decomposition 1, and one where it is implemented by the propagator of Section 3.3. As stated in Section 3.2, we expect the implementation based on Decomposition 1 to produce integer overflows for problem instances where the key fields (namely the employee and the beginning time of a task) have large domains. Neither implementation of StableKeysort will even achieve bounds(\( Z \)) consistency, as noted in Section 3.2 and Section 3.3, respectively, whereas domain consistency can be achieved on Regular and GCC.

The used real-world problem instances involve up to 3,200 tasks and originate from Eurodecision and express a bus driver planning problem. The models above are simplified in order to hide industrial constraints that fall within the competitive advantage of Eurodecision. The experiments were run under Choco [21] on a Mac Pro with 8-core Intel Xeon E5 at 3 GHz under MacOS 10.10 and Java 1.8.0_25. Each instance was run with a 30-minute limit on its own core, each with up to 4 GB of memory. For every model, the search heuristic tries to assign the employees one by one, in lexicographic index order, to the most suitable free task, considering their current workload. This implies that the first solutions are the same for all models, but alternative solutions may come in a different order.

The results are given in Table 1, where “out of memory” denotes that the Java Virtual Machine ran out of memory and could not allocate more memory, while “integer overflow”...
Minimise

\[ \sum_{j=1}^{m} \text{cost}_j W_j \]

subject to

\[
\text{Regular}([a_{j,k} \mid k \in 1..t]), \quad \forall j \in 1..m \quad (18)
\]

\[
\text{GCC}([a_{j,k} \mid j \in 1..m, [1, \ldots, n + 3], [O_1, \ldots, O_{n+3}]], \quad \forall k \in 1..t \quad (19)
\]

\[ O_i \leq 1, \quad \forall i \in 1..n \quad (20) \]

\[ w_{j,k} = 1 \iff a_{j,k} < n + 3, \quad \forall j \in 1..m, \quad \forall k \in 1..t \quad (21) \]

\[ W_j = \sum_{k=1}^{t} w_{j,k}, \quad \forall j \in 1..m \quad (22) \]

where

\[ W_j \in 0..\text{dayDuration}, \quad \forall j \in 1..m \]

**Model 2:** Formulation of the shift problem without the **Shift** constraint.

\[ \text{denotes that an integer overflow occurred during run time. We observe that:} \]

- Model 2 without the **Shift** constraint reaches better solutions for some small instances, but building it is time-consuming and needs a huge amount of memory, until it finally becomes too large to load, mainly due to the memory requirement of (18). Even though this model appears to be more appropriate for small instances, in terms of the quality of the best solutions found, it is clearly unusable in practice for medium and large instances.

- Model 1 with Decomposition 2 of the **Shift** constraint and with Decomposition 1 of the **StableKeysort** constraint takes linear building time and memory, but the domain creation of key \( K_i \) throws an integer overflow when the instance size \( n \) becomes large; indeed, the upper bound of \( K_i \) is approximately \( n^{k+1} \), with \( k = 2 \) here.

- Model 1 with Decomposition 2 of the **Shift** constraint and with the **StableKeysort** propagator of Section 3.3 also takes linear building time and memory (and is slightly better), and does not suffer from memory or integer overflows.
6 Conclusion

The Sort constraints, especially their StableKeysort generalisation introduced here, significantly ease the modelling of several constraints, as illustrated with our elegant decompositions of the Shift and CumulativeSmooth constraints. However, perhaps due to the identified limitations of the original Sort constraints, only a few CP solvers implement them. We argue that Sort, or ideally the new StableKeysort constraint of this paper, should be provided in every CP solver, as that will enable the widespread use of the modelling idiom we have identified. Sort is already identified as a core constraint in [4], due to its key role for constraint reification and negation purposes. Recently, MiniZinc 2 took a step in this direction by introducing sortedness constraints with stability, but without keyed tuples. Our StableKeysort constraint could be added to MiniZinc, with Decomposition 1 as default, but non-scalable, implementation.

Acknowledgements

Pierre Flener and Justin Pearson are supported by grants 2011-6133 and 2012-4908 of the Swedish Research Council (VR). Xavier Lorca and Charles Prud’homme are supported by the French common-laboratory grant TransOp involving the TASC team and Eurodecision. We thank Jean-Noël Monette and the anonymous reviewers for their valuable comments, and Irena Rusu for classifying the complexity of domain consistency for StableKeysort.

References


---

Table 1: Empirical results on the shift problem: memory consumption (mem., in MB), time for creating and posting all constraints (build, in seconds), and the best solution found in 30 minutes (best).

<table>
<thead>
<tr>
<th>n</th>
<th>mem.</th>
<th>build</th>
<th>best</th>
<th>mem.</th>
<th>build</th>
<th>best</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>23</td>
<td>0.398</td>
<td>3144</td>
<td>4</td>
<td>0.119</td>
<td>3144</td>
</tr>
<tr>
<td>50</td>
<td>45</td>
<td>0.707</td>
<td>4948</td>
<td>5</td>
<td>0.124</td>
<td>4958</td>
</tr>
<tr>
<td>100</td>
<td>102</td>
<td>1.672</td>
<td>10586</td>
<td>8</td>
<td>0.155</td>
<td>10693</td>
</tr>
<tr>
<td>200</td>
<td>246</td>
<td>5.999</td>
<td>21386</td>
<td>15</td>
<td>0.223</td>
<td>21386</td>
</tr>
<tr>
<td>400</td>
<td>689</td>
<td>30.891</td>
<td>42772</td>
<td>30</td>
<td>0.338</td>
<td>42961</td>
</tr>
<tr>
<td>800</td>
<td>2334</td>
<td>192.923</td>
<td>86678</td>
<td>64</td>
<td>0.472</td>
<td>86867</td>
</tr>
<tr>
<td>1600</td>
<td>out of memory</td>
<td>integer overflow</td>
<td>141</td>
<td>1.447</td>
<td>174112</td>
<td></td>
</tr>
<tr>
<td>3200</td>
<td>out of memory</td>
<td>integer overflow</td>
<td>373</td>
<td>2.882</td>
<td>348224</td>
<td></td>
</tr>
</tbody>
</table>

1See Arg_Sort at http://www.minizinc.org/2.0/doc-lib/doc-globals-sort.html
A Modelling Pearl with Sortedness

Beldiceanu, Carlsson, Flener, Lorca, Pearson, Petit, and Prud’homme


