Constraints (2011) 16:250–282
DOI 10.1007/s10601-010-9103-2

Explaining the `cumulative` propagator

Andreas Schutt · Thibaut Feydy · Peter J. Stuckey · Mark G. Wallace

Published online: 27 August 2010

Abstract The global `cumulative` constraint was proposed for modelling cumulative resources in scheduling problems for finite domain (FD) propagation. Since that time a great deal of research has investigated new stronger and faster filtering techniques for `cumulative`, but still most of these techniques only pay off in limited cases or are not scalable. Recently, the "lazy clause generation" hybrid solving approach has been devised which allows a finite domain propagation engine possible to take advantage of advanced SAT technology, by "lazily" creating a SAT model of an FD problem as computation progresses. This allows the solver to make use of SAT explanation and autonomous search capabilities. In this article we show how, once we use lazy clause generation, modelling the `cumulative` constraint by decomposition creates a highly competitive version of `cumulative`. Using decomposition into component parts automatically makes the propagator incremental and able to explain itself. We then show how, using the insights from the behaviour of the decomposition, we can create global `cumulative` constraints that explain their propagation. We compare these approaches to explaining the `cumulative` constraint on resource constrained project scheduling problems. All

A preliminary version of this paper appears as [35].

A. Schutt (✉) · T. Feydy · P. J. Stuckey
National ICT Australia, Department of Computer Science & Software Engineering,
The University of Melbourne, Melbourne, Victoria 3010, Australia
e-mail: aschutt@csse.unimelb.edu.au

T. Feydy
e-mail: tfeydy@csse.unimelb.edu.au

P. J. Stuckey
e-mail: pjs@csse.unimelb.edu.au

M. G. Wallace
School of Computer Science & Software Engineering, Monash University,
Clayton, Victoria 3800, Australia
e-mail: mark.wallace@infotech.monash.edu.au

our methods are able to close a substantial number of open problems from the well-established PSPlib benchmark library of resource-constrained project scheduling problems.

Keywords Cumulative constraint · Explanations · Nogood learning · Lazy clause generation · Resource-constrained project scheduling problem

1 Introduction

Cumulative resources are part of many real-world scheduling problems. A resource can represent not only a machine which is able to run multiple tasks in parallel but also entities such as: electricity, water, consumables or even human skills. Those resources arise for example in the resource-constrained project scheduling problem RCPSP, its variants, its extensions and its specialisations. A RCPSP consists of *tasks* (also called *activities*) consuming one or more resources, *precedences* between some tasks, and *resources*. In this paper we restrict ourselves to the case of non-preemptive tasks and renewable resources with a constant resource capacity over the planning horizon. A solution is a schedule of all tasks so that all precedences and resource constraints are satisfied. RCPSP is an NP-hard problem [3].

Example 1 Consider a simple resource scheduling problem. There are 6 tasks `a`, `b`, `c`, `d`, `e` and `f` to be scheduled to end before time 20. The tasks have respective durations 2, 6, 2, 2, 5 and 6, each respective task requiring 1, 2, 4, 2, 2 and 2 units of resource, with a resource capacity of 5. Assume further that there are precedence constraints: task `a` must complete before task `b` begins, written $\texttt{a} \ll \texttt{b}$, and similarly $\texttt{b} \ll \texttt{c}$, $\texttt{d} \ll \texttt{e}$. Figure 1a shows the five tasks and precedences, while Fig. 1b shows a possible schedule, where the respective start times are: 0, 2, 11, 0, 6, 0.

In 1993 Aggoun and Beldiceanu [1] introduced the global `cumulative` constraint in order to efficiently solve complex scheduling problems in a constraint programming framework. The `cumulative` constraint cannot compete with specific OR methods for restricted forms of scheduling, but since it is applicable whatever the side constraints are it is very valuable. Many improvements have been proposed to the `cumulative` constraint: see e.g. Caseau and Laburthe [5], Carlier and Pinson [4], Nuijten [29], Baptiste and Le Pape [2], and Vilím [38].

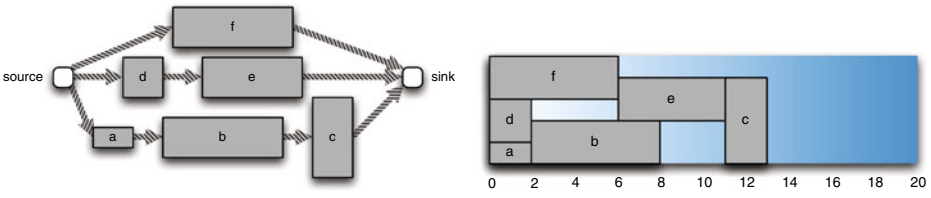

Fig. 1 **a** A small `cumulative` resource problem, with six tasks to place in the 5×20 box, with task `a` before `b` before `c`, and task `d` before `e`. **b** A possible schedule

The best known exact algorithm for solving RCPSP is from Demeulemeester and Herroelen [9]. Their specific method is a branch-and-bound approach relying heavily on dominance rules and cut sets, a kind of problem specific nogoods. They implicitly show the importance of nogoods to fathom the huge search space of RCPSP problems. Unfortunately, the number of cut sets grows exponentially in the number of tasks, so that this method is considered to be efficient only for small problems.

In general, *nogoods* are redundant constraints that are concluded during a conflict analysis of an inconsistent solution state. They are permanently or temporary added to the initial constraint system to reduce the search space, and/or to guide the search. Nogoods are inferred from explanations and/or search decisions where an *explanation* records the reason of value removals during propagation. Explanation can be also used to short circuit propagation.

In comparison to the specific nogoods of Demeulemeester and Herroelen SAT solvers records general nogoods. Since the introduction of `cumulative`, SAT solving has improved drastically. Nowadays, modern SAT solvers can often handle problems with millions of constraints and hundreds of thousands of variables. But problems like RCPSP are difficult to encode into SAT without breaking these implicit limits. Recently, Ohrimenko et al. [30, 31] showed how to build a powerful hybrid of SAT solving and FD solving that maintains the advantages of both: the high level modelling and small models of FD solvers, and the efficient nogood recording and conflict driven search of SAT. The key idea in this *lazy clause generation* approach is that finite domain propagators lazily generate a clausal representation which is an *explanation* of their behaviour. They show that this combination outperforms the best available constraint solvers on Open-Shop-Scheduling problems which is a special case of RCPSP.

Global constraint can almost always be *decomposed* into simpler constraints by introducing new intermediate variables to encode the meaning of the global constraint. Since the introduction of `cumulative` in 1993 [1], little attention has been paid to decompositions of `cumulative` because decomposition cannot compete with the global propagator because of the overheads of intermediate variables, and the lack of a global view of the problem. But once we consider explanation we have to revisit this. Decomposition of globals means that explanation of behaviour is more fine grained and hence more reusable. Also it avoids the need for complex explanation algorithms to be developed for the global. Note that there is some preliminary work on explanation generation for `cumulative`, in PaLM [16] where (in 2000) it is described as current work, and [37] which restricts attention to the `disjunctive` constraint (resource capacity 1).

In this paper we investigate how explanations for `cumulative` can improve the solving of complex scheduling problems. We show first how decomposition of `cumulative` can be competitive with state-of-the-art specialised methods from the CP and OR community. We then show that building a global `cumulative` propagator with specialised explanation capabilities can further improve upon the explaining decompositions. The G12 Constraint Programming Platform is used for implementation of the `cumulative` constraint as a lazy clause generator. We evaluate our approach on RCPSP from the well-established and challenging benchmark library PSPLib [22, 32].

This article is organised as follows: In Section 2 the related work to the `cumulative` constraint and explanation/nogoods is presented. In Section 3 lazy clause generation is explained and the terminology is introduced. In Section 4 we introduce the `cumulative` constraint. In Section 5 we show how to propagate the `cumulative` constraint using decomposition. In Section 6 we show how to modify a global `cumulative` constraint to explain its propagation. In Section 7 we introduce resource constrained project scheduling problems and discuss search strategies for tackling them. In Section 8 we give experimental results and finally in Section 9 we conclude.

2 Related work

2.1 `cumulative`

In 1993 Aggoun and Beldiceanu [1] introduced the global `cumulative` constraint in order to efficiently solve complex scheduling problems in a constraint programming framework. For `cumulative` a simple consistency check and filtering algorithm (time-table) with the time complexity $\mathcal{O}(n \log n)$ is based on compulsory parts [24]. The overload check [39], another consistency check, also runs with the same time complexity. There are more complex filtering techniques as follows.

In 1994 Nuijten [29] generalised the (extended) edge-finding, and not-first/not-last filtering algorithm for the `disjunctive` propagator to the `cumulative` propagator. All three algorithms detect the relationship between one task j and a subset of tasks Ω excluding j by considering the task's energy. (Extended) edge-finding infers if the task j must strictly end after (start before) all tasks in Ω, not-first if the task j must start after the end of at least one task in Ω and not-last if the task j must end before the start of at least one task in Ω.

Nuijtens edge-finding was corrected by Mercier and Van Hentenryck [27] and improved by Vilím [38] to the time complexity $\mathcal{O}(kn \log n)$ in 2009 where k is the number of different resource requirements of tasks. Nuijtens not-first/not-last algorithm was corrected by Schutt et al. [36] and improved by Schutt [34] to time complexity $\mathcal{O}(n^2 \log n)$ in 2006.

These filtering algorithms are based on reasoning about the energy of the tasks [12]. As a consequence, they are more likely to propagate if the slack, i.e., free available energy in some time interval, is small. Baptiste and Le Pape [2] showed that the use of these algorithms is beneficial for highly cumulative problems[1] but not for highly disjunctive problems. They also present the left-shift/right-shift filtering algorithm in the time complexity $\mathcal{O}(n^3)$ which subsumes (extended) edge-finding, but not not-first/not-last.

In 1996 Caseau and Laburthe [5] generalised the use of task intervals from the `disjunctive` constraint to the `cumulative` constraint. A task interval is characterised by two tasks i and j, and contains all tasks which earliest start time and latest end time are included in the time window $[\, start \,..\, end \,]$ where *start* is the earliest start time of i and *end* the latest end time of j. The number of task intervals is thus $\mathcal{O}(n^2)$.

[1] Problems are called highly cumulative if many tasks can be run in parallel.

The task intervals are incrementally maintained and used to check consistency, propagate the lower (upper) bound of the start (end) variable by rule which covers (extended) edge-finding, detect precedences between tasks and eliminate duration-usage pairs if the energy for a task is given. Additionally, the compulsory part profiles are incrementally tracked for the consistency check and the time-table filtering.

2.2 Explanations

There is a substantial body of work on explanations in constraint satisfaction (see e.g. [8], chapter 6), but there was little evidence until recently of success for explanations that combine with propagation (although see [17, 18]). The constraint programming community revisited this issue after the success of explanations in the SAT community.

Katsirelos and Bacchus [19] generalised the nogoods from the SAT community. Their generalised nogoods are conjunction of variable-value equalities and disequalities, *e.g.* $\{x_1 = 3, x_4 = 0, x_7 \neq 6\}$. For bound inference algorithms this representation is not suitable since one bound update of a variable must be explained by several nogoods.

The lazy clause generation approach proposed by Ohrimenko et al. [30] is a hybrid of SAT and FD solvers. It keeps the abstraction of the constraints and their propagators just explain their inferences to the SAT solvers. Moreover, their bounds of integer variables are represented by Boolean literals so it is more suitable for explaining a bound filtering algorithm. Feydy and Stuckey [13] re-engineered the lazy clause generator approach by swapping of the master-slave role from SAT-FD to FD-SAT.

In this paper we use lazy clause generation to implement a first version of an explaining global `cumulative` constraint consisting of the time-table consistency check and filtering algorithm. This constraint is compared to the two decompositions which also use lazy clause generation on the parts of their decomposition.

There has been a small amount of past work on explaining the `cumulative` constraint. Vilím [37] considered the disjunctive case (resource capacity 1) of `cumulative` where he presented a framework based on justification and explanation. The explanation consists of a subset of initial constraints, valid search decisions, and conflict windows for every tasks. He proposed explanations for an overload check concerning a task interval, edge- finding, not-first/not-last, and detectable precedence filtering algorithms. In this paper we consider the full `cumulative` case and in particular show how to explain time-table and edge-finding filtering. Explaining edge-finding for full `cumulative` is more complex than the disjunctive case, since we have to take into account the resource limit and that tasks can run in parallel. Moreover, Vilím does not consider explaining the filtering algorithms stepwise.

In the work [15] Jussien presents explanation for the time-table consistency and filtering algorithms in the context of the PaLM system [16]. The system explains inconsistency at time t by recording the set of tasks S_t whose compulsory part overlaps t and then requiring that at least one of them takes a value different from their current domain. These explanations are analogous to the naïve explanations we describe later which examine only the current bounds, but much weaker since they do not use bounds literals in the explanations. The time-table filtering explanations

are based on single time points but again use the current domains of the variables to explain.

3 Lazy clause generation

Lazy clause generation is a powerful hybrid of SAT and finite domain solving that inherits advantages of both: high level modelling, and specialised propagation algorithms from FD; nogood recording, and conflict driven search from SAT.

3.1 Finite domain propagation

We consider a set of integer variables $\mathcal{V}$. A *domain* D is a complete mapping from $\mathcal{V}$ to finite sets of integers. Let D_1 and D_2 be domains and $V \subseteq \mathcal{V}$. We say that D_1 is *stronger* than D_2, written $D_1 \sqsubseteq D_2$, if $D_1(v) \subseteq D_2(v)$ for all $v \in \mathcal{V}$. Similarly if $D_1 \sqsubseteq D_2$ then D_2 is *weaker* than D_1. We use *range* notation: $\left[\, l \,..\, u \,\right]$ denotes the set of integers $\{d \mid l \leq d \leq u, d \in \mathbb{Z}\}$. We assume an *initial domain* D_{init} such that all domains D that occur will be stronger i.e. $D \sqsubseteq D_{\text{init}}$.

A *valuation* θ is a mapping of variables to values, written $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$. We extend the valuation θ to map expressions or constraints involving the variables in the natural way. Let *vars* be the function that returns the set of variables appearing in an expression, constraint or valuation. In an abuse of notation, we define a valuation θ to be an element of a domain D, written $\theta \in D$, if $\theta(v) \in D(v)$ for all $v \in \mathit{vars}(\theta)$.

A constraint c is a set of valuations over $\mathit{vars}(c)$ which give the allowable values for a set of variables. In finite domain propagation constraints are implemented by propagators. A propagator f for c is a monotonically decreasing function on domains such that for all domains $D \sqsubseteq D_{\text{init}}$: $f(D) \sqsubseteq D$ and no solutions are lost, i.e. $\{\theta \in D \mid \theta \in c\} = \{\theta \in f(D) \mid \theta \in c\}$. A *propagation solver* for a set of propagators F and current domain D, $\mathit{solv}(F, D)$, repeatedly applies all the propagators in F starting from domain D until there is no further change in resulting domain. $\mathit{solv}(F, D)$ is the weakest domain $D' \sqsubseteq D$ which is a fixpoint (i.e. $f(D') = D'$) for all $f \in F$.

3.2 SAT solving

Davis-Putnam-Logemann-Loveland [7] SAT solvers can be understood as a form of propagation solver where variables are Boolean, and the only constraints are clauses. Each clause is in effect a propagator. The difference with an FD solver is that propagation of clauses is highly specialised and more importantly the reasons for propagations are recorded, and on failure used to generate a nogood clause which explains the failure. This nogood clause is added to the propagators to short circuit later search. It also helps to direct backtracking to go above the cause of the failure. See e.g. [6, 10] for more information about SAT solving.

We briefly introduce some SAT terminology. Let $\mathcal{B}$ be the set of Boolean variables. A *literal* is either b or $\neg b$ where $b \in \mathcal{B}$. A *clause* is a disjunction of literals. An *assignment*, A is a subset of literals over $\mathcal{B}$, such that $\{b, \neg b\} \not\subseteq A$.

3.3 Lazy clause generation

Lazy clause generation [30] works as follows. Propagators are considered as clause generators for the SAT solver. Instead of applying propagator f to domain D to obtain $f(D)$, whenever $f(D) \neq D$ we build a clause that encodes the change in domains. In order to do so we must link the integer variables of the finite domain problem to a Boolean representation.

We represent an integer variable x with domain $D_{\text{init}}(x) = [\, l \,..\, u \,]$ using the Boolean variables $[\![x = l]\!], \ldots, [\![x = u]\!]$ and $[\![x \leq l]\!], \ldots, [\![x \leq u - 1]\!]$ where the former is generated on demand. The variable $[\![x = d]\!]$ is true if x takes the value d, and false for a value different from d. Similarly the variable $[\![x \leq d]\!]$ is true if x takes a value less than or equal to d and false for a value greater than d. Note we sometime use the notation $[\![d \leq x]\!]$ for the literal $\neg[\![x \leq d - 1]\!]$.

Not every assignment of Boolean variables is consistent with the integer variable x, for example $\{[\![x = 3]\!], [\![x \leq 2]\!]\}$ (i.e. both Boolean variables are true) requires that x is both 3 and ≤ 2. In order to ensure that assignments represent a consistent set of possibilities for the integer variable x we add to the SAT solver the clauses $DOM(x)$ that encode $[\![x \leq d]\!] \rightarrow [\![x \leq d + 1]\!], l \leq d < u$, $[\![x = l]\!] \leftrightarrow [\![x \leq l]\!]$, $[\![x = d]\!] \leftrightarrow ([\![x \leq d]\!] \wedge \neg[\![x \leq d - 1]\!]), l < d < u$, and $[\![x = u]\!] \leftrightarrow \neg[\![x \leq u - 1]\!]$ where $D_{\text{init}}(x) = [\, l \,..\, u \,]$. We let $DOM = \cup\{DOM(v) \mid v \in \mathcal{V}\}$.

Any assignment A on these Boolean variables can be converted to a domain: $domain(A)(x) = \{d \in D_{\text{init}}(x) \mid \forall [\![c]\!] \in A, vars([\![c]\!]) = \{x\} : x = d \models c\}$, that is, the domain includes all values for x that are consistent with all the Boolean variables related to x. It should be noted that the domain may assign no values to some variable.

Example 2 Assume $D_{\text{init}}(x_i) = [\, -12 \,..\, 12 \,]$ for $i \in [\, 1 \,..\, 3 \,]$. The assignment $A = \{[\![x_1 \leq 10]\!], \neg[\![x_1 \leq 5]\!], \neg[\![x_1 = 7]\!], \neg[\![x_1 = 8]\!], [\![x_2 \leq 11]\!], \neg[\![x_2 \leq 5]\!], [\![x_3 \leq 10]\!], \neg[\![x_3 \leq -2]\!]\}$ is consistent with $x_1 = 6$, $x_1 = 9$ and $x_1 = 10$. Hence $domain(A)(x_1) = \{6, 9, 10\}$. For the remaining variables $domain(A)(x_2) = [\, 6 \,..\, 11 \,]$ and $domain(A)(x_3) = [\, -1 \,..\, 10 \,]$. Note that for brevity A is not a fixpoint of a SAT propagator for $DOM(x_1)$ since we are missing many implied literals such as $\neg[\![x_1 = 5]\!], \neg[\![x_1 = 12]\!]$, $\neg[\![x_1 \leq -4]\!]$ etc.

In lazy clause generation a propagator changes from a mapping from domains to domains to a generator of clauses describing propagation. When $f(D) \neq D$ we assume the propagator f can determine a set of clauses C which explain the domain changes.

Example 3 Consider the propagator f for $x_1 \leq x_2 + 1$. When applied to domain $D(x_1) = [\, 0 \,..\, 9 \,]$, $D(x_2) = [\, -3 \,..\, 5 \,]$ it obtains $f(D)(x_1) = [\, 0 \,..\, 6 \,]$, $f(D)(x_2) = [\, -1 \,..\, 5 \,]$. The clausal explanation of the change in domain of x_1 is $[\![x_2 \leq 5]\!] \rightarrow [\![x_1 \leq 6]\!]$, similarly the change in domain of x_2 is $\neg[\![x_1 \leq -1]\!] \rightarrow \neg[\![x_2 \leq -2]\!]$ ($x_1 \geq 0 \rightarrow x_2 \geq -1$). These become the clauses $\neg[\![x_2 \leq 5]\!] \vee [\![x_1 \leq 6]\!]$ and $[\![x_1 \leq -1]\!] \vee \neg[\![x_2 \leq -2]\!]$.

Assuming clauses C explain the propagation of f are added to the SAT database on which unit propagation is performed. Then if $domain(A) \sqsubseteq D$ then $domain(A') \sqsubseteq f(D)$ where A' is the resulting assignment after addition of C and unit propagation.

This means that unit propagation on the clauses C is as strong as the propagator f on the original domains.

Using the lazy clause generation we can show that the SAT solver maintains an assignment which is at least as strong as that determined by finite domain propagation [30]. The advantages over a standard FD solver (e.g. [33]) are that we automatically have the nogood recording and backjumping ability of the SAT solver applied to our FD problem. We can also use activity counts from the SAT solver to direct the FD search.

Propagation can be explained by different set of clauses. In order to get maximum benefit from the explanation we desire a "strongest" explanation as possible. A set of clauses C_1 is *stronger* than a set of clauses C_2 if C_1 implies C_2. In other words, C_1 restricts the search space at least as much as C_2.

Example 4 Consider the reified inequality constraint $b \Leftrightarrow x_1 \leq x_2 + 1$ which holds if $b = 1$ and the inequality holds, or $b = 0$ and $x_1 > x_2 + 1$. Assume the initial domains are $D_{\text{init}}(x_1) = [0..9]$, $D_{\text{init}}(x_2) = [0..19]$ and $D_{\text{init}}(b) = [0..1]$. Let us assume that after propagation of other constraints the domains become $D(x_1) = [0..8]$, $D(x_2) = [11..19]$, and $D(b) = [0..1]$. A propagator f of this constraint infers the new domain $f(D)(b) = \{1\}$. The propagator can explain the change in the domain by any of the singleton sets of clauses $\{[\![x_1 \leq 8]\!] \wedge \neg[\![x_2 \leq 10]\!] \rightarrow [\![b = 1]\!]\}$, $\{[\![x_1 \leq 8]\!] \wedge \neg[\![x_2 \leq 6]\!] \rightarrow [\![b = 1]\!]\}$, or $\{\neg[\![x_2 \leq 7]\!] \rightarrow [\![b = 1]\!]\}$. The second and third explanation are stronger than the first, but neither of the second or third explanation is stronger than the other.

4 Modelling the cumulative resource constraint

In this section we define the `cumulative` constraint and discuss two possible decompositions of it.

The `cumulative` constraint introduced by Aggoun and Beldiceanu [1] in 1993 is a constraint with Zinc [26] type

```
predicate cumulative(list of var int: s, list of var int: d,
                     list of var int: r,         var int: c);
```

Each of the first three arguments are lists of the same length n and indicate information about a set of *tasks*. $s[i]$ is the *start time* of the i^{th} task, $d[i]$ is the *duration* of the i^{th} task, and $r[i]$ is the *resource usage* (per time unit) of the i^{th} task. The last argument c is the *resource capacity*.

The `cumulative` constraints represent `cumulative` resources with a constant capacity over the considered planning horizon applied to non-preemptive tasks, i.e.,if they are started they cannot be interrupted. W.l.o.g. we assume that all values are integral and non-negative and there is a *planning horizon* $t_{\max}$ which is the latest time any task can finish.

We assume throughout the paper that each of d, r and c are fixed integers, although this is not important for much of the discussion. This is certainly the most common case of `cumulative`, and sufficient for the RCPSP problems we concentrate on.

The cumulative constraint enforces that at all times the sum of resources used by active tasks is no more than the resource capacity.

$$\forall t \in [\,0\,..\,t_{\max}-1\,]: \sum_{i\in[\,1\,..\,n\,]:s[i]\le t<s[i]+d[i]} r[i] \le c \tag{1}$$

Example 5 Consider the cumulative resource problem defined in Example 1. This can be modelled by the cumulative constraint

$$\texttt{cumulative}([s_\texttt{a}, s_\texttt{b}, s_\texttt{c}, s_\texttt{d}, s_\texttt{e}, s_\texttt{f}], [2, 6, 2, 2, 5, 6], [1, 2, 4, 2, 2, 2], 5)$$

with precedence constraints $\texttt{a} \ll \texttt{b}$, $\texttt{b} \ll \texttt{c}$, $\texttt{d} \ll \texttt{e}$, modelled by $s_\texttt{a} + 2 \le s_\texttt{b}$ $s_\texttt{b} + 6 \le s_\texttt{c}$, and $s_\texttt{d} + 6 \le s_\texttt{e}$. The propagator for the precedence constraints determines a domain D where $D(s_\texttt{a}) = [\,0\,..\,8\,]$, $D(s_\texttt{b}) = [\,2\,..\,10\,]$, $D(s_\texttt{c}) = [\,8\,..\,18\,]$, $D(s_\texttt{d}) = [\,0\,..\,13\,]$, $D(s_\texttt{e}) = [\,2\,..\,15\,]$, $D(s_\texttt{f}) = [\,0\,..\,14\,]$. The cumulative constraint does not determine any new information. If we add the constraints $s_\texttt{c} \le 9$, $s_\texttt{e} \le 4$, then precedence determines the domains $D(s_\texttt{a}) = [\,0\,..\,1\,]$, $D(s_\texttt{b}) = [\,2\,..\,3\,]$, $D(s_\texttt{c}) = [\,8\,..\,9\,]$, $D(s_\texttt{d}) = [\,0\,..\,2\,]$, $D(s_\texttt{e}) = [\,2\,..\,4\,]$. The cumulative constraint may be able to determine that task f cannot start before time 10 (See Example 9 for a detailed explanation of how).

5 Propagating the cumulative constraint by decomposition

Usually the cumulative constraint is implemented as a global propagator, since it can then take more information into account during propagation. But building a global constraint is a considerable undertaking which we can avoid if we are willing to encode the constraint using decomposition into primitive constraints. In the remainder of this section we give two decompositions.

5.1 Time decomposition

The time decomposition (*TimeD*) [1] arises from the Formula (1). For every time t the sum of all resource requirements must be less than or equal to the resource capacity. The Zinc encoding of the decomposition is shown below where: *index_set*(a) returns the index set of an array a (here $[\,1\,..\,n\,]$), $lb\,(x)$ ($ub\,(x)$) returns the declared lower (resp. upper) bound of an integer variable x, and *bool2int*(b) is 0 if the Boolean b is false, and 1 if it is true.

```
predicate cumulative(list of var int: s, list of var int: d,
                     list of var int: r,         var int: c) =
  let {set of int: tasks = index_set(s),
       set of int: times = min([lb(s[i]) | i in tasks]) ..
                           max([ub(s[i]) + ub(d[i]) - 1 |
              i in tasks])
  } in forall( t in times ) (
           c >= sum( i in tasks ) (
              bool2int( s[i] <= t /\ t < s[i] + d[i] ) * r[i]));
```

This decomposition implicitly introduces new Boolean variables B_{it}. Each B_{it} represents that task i is active at time t:

$$\forall t \in [\,0\,..\,t_{\max}-1\,], \forall i \in [\,1\,..\,n\,]: \qquad B_{it} \leftrightarrow [\![s[i] \leq t]\!] \wedge \neg [\![s[i] \leq t - d[i]]\!]$$

$$\forall t \in [\,0\,..\,t_{\max}-1\,]: \qquad \sum_{i \in [\,1\,..\,n\,]} r[i] \cdot B_{it} \leq c$$

Note that since we are using lazy clause generation, the Boolean variables for the expressions $[\![s[i] \leq t]\!]$ and $[\![s[i] \leq t - d[i]]\!]$ already exist and that for a task i we only need to construct variables B_{it} where $lb(s[i]) \leq t < ub(s[i]) + d[i]$ for the initial domain D_{init}.

At most $n \times t_{\max}$ new Boolean variables are created, $n \times t_{\max}$ conjunction constraints, and $t_{\max}$ sum constraints (of size n). This decomposition implicitly profiles the resource histograms for all times for the resource.

In order to add another `cumulative` constraint for a different resource on the same tasks we can reuse the Boolean variables, and we just need to create $t_{\max}$ new sum constraints.

The variable B_{it} records whether the task i must use its resources at time t. Hence B_{it} is true indicates a "compulsory part" of task i. It holds in the time interval $\big[\, lb(s[i])\,..\,lb(s[i]) + d[i] - 1\,\big] \cap \big[\, ub(s[i])\,..\,ub(s[i]) + d[i] - 1\,\big]$. The sum of the compulsory parts for all the tasks and times creates the resource *time table*. Figure 2a illustrates the diagrammatic notation we will use to illustrate the earliest start time, latest end time and compulsory part of a task.

Example 6 Consider the problem of Example 5 after the addition of $s_{\mathtt{c}} \leq 9$, $s_{\mathtt{e}} \leq 4$. The domains are $D(s_{\mathtt{a}}) = [\,0\,..\,1\,]$, $D(s_{\mathtt{b}}) = [\,2\,..\,3\,]$, $D(s_{\mathtt{c}}) = [\,8\,..\,9\,]$, $D(s_{\mathtt{d}}) = [\,0\,..\,2\,]$, $D(s_{\mathtt{e}}) = [\,2\,..\,4\,]$, $D(s_{\mathtt{f}}) = [\,0\,..\,14\,]$. Propagation on the decomposition determines that $B_{\mathtt{b}5}$ is true since $s_{\mathtt{b}} \leq 5$ and $\neg(s_{\mathtt{b}} \leq 5 - 6 = -1)$, similarly for $B_{\mathtt{e}5}$. Using the sum constraint propagation determines that $B_{\mathtt{f}5}$ is false, and hence $\neg(s_{\mathtt{f}} \leq 5) \vee s_{\mathtt{f}} \leq -1$.

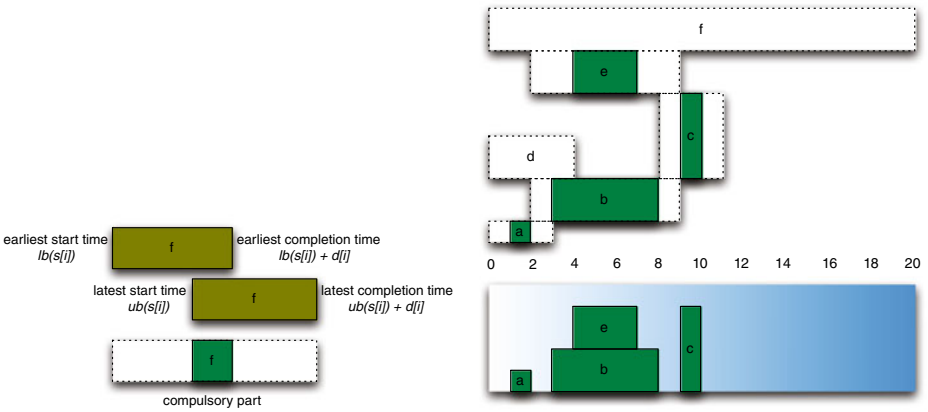

Fig. 2 **a** A diagram illustrating the calculation of the compulsory part of a task **b** and an example of propagation of the `cumulative` constraint

Since the second half of the disjunct is false already we determine that $s_{\mathtt{f}} \geq 6$. Similarly propagation on the decomposition determines that $B_{\mathtt{c}9}$ is true, and hence $B_{\mathtt{f}9}$ is false, and hence $\neg(s_{\mathtt{f}} \leq 9) \vee s_{\mathtt{f}} \leq 3$. Since the second disjunct must be false (due to $s_{\mathtt{f}} \geq 6$) we determine that $s_{\mathtt{f}} \geq 10$.

The top of Fig. 2b shows each task in a window from earliest start time to latest end time, and highlights the compulsory parts of each task. If there is no darkened part (as for `d` and `f`) then there is no compulsory part. Propagation of the decomposition by a finite domain solver will determine $B_{\mathtt{a}1}$, $B_{\mathtt{b}3}$, $B_{\mathtt{b}4}$, $B_{\mathtt{b}5}$, $B_{\mathtt{b}6}$, $B_{\mathtt{b}7}$, $B_{\mathtt{c}9}$, $B_{\mathtt{e}4}$, $B_{\mathtt{e}5}$, $B_{\mathtt{e}6}$. which corresponds to the compulsory parts of each task. The resulting resource time table is shown at the bottom of Fig. 2b, giving the required resource utilisation at each time. Clearly at times 5 and 9 there is not enough resource capacity for the resource usage 2 of task `f`.

We can expand the model to represent holes in the domains of start times.[2] The literal $[\![s[i] = t]\!]$ is a Boolean representing the start time of the i^{th} task is t. We add the constraint

$$[\![s[i] = t]\!] \rightarrow \bigwedge_{t \leq t' < t + d[i]} B_{it'}$$

which ensures that if $B_{it'}$ becomes false then the values $\{t' - d[i] + 1, t' - d[i] + 2, \ldots, t'\}$ are removed from the domain of $s[i]$. We do not use this constraint for our experiments since it was inferior in solving time to the model without it.

We tested this extended model on large instances RCPSP from the PSPLib, but it neither improved the search time, the number of choice points, nor the average distance to the best know upper bound in average. This is not surprising since for RCPSP there are no propagators that can take advantage of the holes in the domain generated to infer new information. The expanded model may be useful for cumulative scheduling problems with side constraints that can benefit from such reasoning.

5.2 Task decomposition

The task decomposition (*TaskD*) is a relaxation of the time decomposition. It ensures a non-overload of resources only at the start (or end) times which is sufficient to ensure non-overload at every time for the non-preemptive case. Therefore, the number of variables and linear inequality constraints is independent of the size of the planning horizon $t_{\max}$. It was used by El-Kholy [11] for temporal and resource reasoning in planning. The Zinc code for the decomposition at the start times is below.

```
predicate cumulative(list of var int: s, list of var int: d,
                     list of var int: r,         var int: c) =
   let { set of int: tasks = index_set(s) }
   in forall( j in tasks ) (
         c >= r[j] + sum( i in tasks where i != j ) (
            bool2int( s[i] <= s[j] /\ s[j] < s[i] + d[i] ) * r[i]));
```

[2] Usually CP representation of tasks does not encode holes.

The decomposition implicitly introduces new Boolean variables: $B^1_{ij} \equiv$ task j starts at or after task i starts, $B^2_{ij} \equiv$ task j starts before task i ends, and $B_{ij} \equiv$ task j starts when task i is running.

$$\forall j \in [1..n], \forall i \in [1..n] \setminus \{j\}: \quad B_{ij} \leftrightarrow B^1_{ij} \wedge B^2_{ij}$$

$$B^1_{ij} \leftrightarrow s[i] \leq s[j]$$

$$B^2_{ij} \leftrightarrow s[j] < s[i] + d[i]$$

$$\forall j \in [1..n]: \quad \sum_{i \in [1..n] \setminus \{j\}} r[i] \cdot B_{ij} \leq c - r[j]$$

Note not all tasks i must be considered for a task j, only those i which can overlap at the start times $s[j]$ wrt. precedence constraints, resource constraints and the initial domain D_{init}.

Since the SAT solver does not know about the relationship among the B^1_{**} and B^2_{**} the following redundant constraints can be posted for all $i, j \in [1..n]$ where $i < j$ in order to improve the propagation and the learning of reusable nogoods.

$$B^1_{ij} \vee B^2_{ij} \qquad B^1_{ji} \vee B^2_{ji} \qquad B^1_{ij} \vee B^1_{ji} \qquad B^1_{ij} \rightarrow B^2_{ji} \qquad B^1_{ji} \rightarrow B^2_{ij}$$

In addition for each precedence constraint $i \ll j$ we can post $\neg B_{ij}$.

The size of this decomposition only depends on n whereas *TimeD* depends on n and the number of time points in the planning horizon $t_{\max}$. At most $3n(n-1)$ Boolean variables, $3n(n-1)$ equivalence relations, $5(n-1)(n-2)/2$ redundant constraints and n sum constraints are generated. Again adding another `cumulative` resource constraints can reuse the Boolean variables and requires only adding n new sum constraints.

Example 7 Consider the problem of Example 5 after the addition of $s_{\mathtt{c}} \leq 9$, $s_{\mathtt{e}} \leq 4$. The domains from precedence constraints are $D(s_{\mathtt{a}}) = [0..1]$, $D(s_{\mathtt{b}}) = [2..3]$, $D(s_{\mathtt{c}}) = [8..9]$, $D(s_{\mathtt{d}}) = [0..2]$, $D(s_{\mathtt{e}}) = [2..4]$, $D(s_{\mathtt{f}}) = [0..14]$. Propagation on the decomposition learns $\neg B^2_{\mathtt{ab}}$, $\neg B^2_{\mathtt{bc}}$ and $\neg B^2_{\mathtt{de}}$ direct from precedence constraints and hence $\neg B_{\mathtt{ab}}$, $\neg B_{\mathtt{bc}}$, and $\neg B_{\mathtt{de}}$. From the start times propagation determines that $B^1_{\mathtt{ab}}$, $B^1_{\mathtt{db}}$, $B^1_{\mathtt{ae}}$, $B^1_{\mathtt{de}}$, $B^1_{\mathtt{ac}}$, $B^1_{\mathtt{bc}}$, $B^1_{\mathtt{dc}}$, $B^1_{\mathtt{ec}}$, and similar facts about B^2 variables, but no information about B variables. The sum constraints determine that $\neg B_{\mathtt{cf}}$ and $\neg B_{\mathtt{fc}}$ but there are no bounds changes. This illustrates the weaker propagation of the *TaskD* decomposition compared to the *TimeD* decomposition.

If we use end time variables $e[i] = s[i] + d[i]$, we can generate a symmetric model to that defined above.

In comparison to the *TimeD* decomposition, the *TaskD* decomposition is stronger in its ability to relate to task precedence information ($i \ll j$), but generates a weaker profile of resource usage, since no implicit profile is recorded. They are thus incomparable in strength of propagation, although in practice the *TimeD* decomposition it almost always stronger.

6 Explanations for the global `cumulative`

The global `cumulative` consists of consistency-checking and filtering algorithms. In order to gain maximum benefit from the underlying SAT solver (in term of nogood learning) these algorithms must explain inconsistency and domain changes of variables using Boolean literals that encode the integer variables and domains in the SAT solver.[3] The decomposition approaches to `cumulative` inherit the ability to explain "for free" from the explanation capabilities of the base constraints that make them up. The challenge in building an explaining global constraint is to minimise the overhead of the explanation generation and make the explanations as reusable as possible.

6.1 Consistency check

The `cumulative` constraint first has to perform a consistency check to determine whether the constraint is possibly satisfiable. Here we consider a consistency check based on the resource time table. If an overload in resource usage occurs on a resource with a maximal capacity c in the time interval $[\,\mathbf{s}\,..\,\mathbf{e}-1\,]$ involving the set of tasks Ω, the following condition holds:

$$\forall i \in \Omega : \quad ub\,(s[i]) \leq \mathbf{s} \wedge \mathbf{e} \leq lb\,(s[i]) + d[i] \qquad \sum_{i\in\Omega} r[i] > c$$

A *naïve explanation* explains the inconsistency using the current domain bounds on the corresponding variables from the tasks in Ω.

$$\bigwedge_{i\in\Omega} [\![lb\,(s[i]) \leq s[i]]\!] \wedge [\![s[i] \leq ub\,(s[i])]\!] \rightarrow \text{false}$$

The naïve explanation simply uses the current domains of all variables involved in the inconsistency, which is always a correct explanation for any constraint.

In some cases some task in Ω might have compulsory parts before or after the overload. These parts are not related to the overload, and give us the possibility to widen the bounds in the explanation. We can widen the bounds of all tasks in this way so that their compulsory part is only between $\mathbf{s}$ and $\mathbf{e}$. The corresponding explanation is called a *big-step explanation*.

$$\forall i \in \Omega : [\![\mathbf{e} - d[i] \leq s[i]]\!] \wedge [\![s[i] \leq \mathbf{s}]\!] \rightarrow \text{false}$$

The big-step explanation explains the reason for the overload over its entire width.

We can instead explain the overload by concentrating on a single time point t in $[\,\mathbf{s}\,..\,\mathbf{e}-1\,]$ rather than examining the whole time interval. This allows us to further strengthen the explanation which has the advantage that the resulting nogood is more reusable. The explanation has the same pattern as a big-step explanation except we

[3] When using lazy clause generation it is not strictly necessary that a propagator explains all its propagations, in which case the resulting propagated constraints are treated like decisions, and learning is not nearly as useful.

use t for $\mathbf{s}$ and $t+1$ for $\mathbf{e}$. We call these explanations *pointwise explanations*. The pointwise and big-step explanation coincide iff $\mathbf{s}+1=\mathbf{e}$.

The pointwise explanation is implicitly related to the *TimeD* where an overload is explained for one time point. That time point by *TimeD* depends on the propagation order of the constraints related to an overload whereas for the global `cumulative` we have the possibility of choosing a time point to use to explain inconsistency. This means that we have more control and flexibility about what explanation is generated which may be beneficial. In the experiments reported herein we always choose the mid-point of $[\,\mathbf{s}\,..\,\mathbf{e}-1\,]$.

There are many open questions related to time points for pointwise explanations in general and with respect to the possible search space reduction from derived nogoods or explanations: does it matter which time point is picked, if so, which is the best one?

Sometimes a resource overload is detected where the task set Ω of tasks which are compulsory at that time is not minimal with respect to the resource limit, i.e., there exists a proper subset of tasks $\Omega' \subset \Omega$ with $\sum_{i\in\Omega'} r[i] > c$. The same situation happens for the *TimeD* decomposition as well. Here, again with the global view on `cumulative`, we know the context of the tasks involved and can decide which subset Ω' is used in order to explain the inconsistency if there exists a choice. Here as well, it is an open question which subset is the best to restrict the search space most or whether it does not matter? For our experiments the lexicographic least set of tasks is chosen (where the order is given by the order of appearance in the `cumulative` constraint).[4]

Example 8 Consider the problem of Example 1 with the additional constraints $s_{\mathtt{c}} \le 9$, $s_{\mathtt{e}} \le 4$, and $s_{\mathtt{f}} \le 4$. The resulting bounds from precedence constraints are $D(s_{\mathtt{a}}) = [\,0\,..\,1\,]$, $D(s_{\mathtt{b}}) = [\,2\,..\,3\,]$, $D(s_{\mathtt{c}}) = [\,8\,..\,9\,]$, $D(s_{\mathtt{d}}) = [\,0\,..\,2\,]$, $D(s_{\mathtt{e}}) = [\,2\,..\,4\,]$, $D(s_{\mathtt{f}}) = [\,0\,..\,4\,]$. The time interval of positions where the tasks can fit and the resulting resource profile are shown in Fig. 3. There is an overload of the resource limit between time 4 and 6 with $\Omega = \{\mathtt{b}, \mathtt{e}, \mathtt{f}\}$. The naive explanation is $[\![2 \le s_{\mathtt{b}}]\!] \wedge [\![s_{\mathtt{b}} \le 3]\!] \wedge [\![2 \le s_{\mathtt{e}}]\!] \wedge [\![s_{\mathtt{e}} \le 4]\!] \wedge [\![0 \le s_{\mathtt{f}}]\!] \wedge [\![s_{\mathtt{f}} \le 4]\!] \rightarrow$ false. The big-step explanation is $[\![0 \le s_{\mathtt{b}}]\!] \wedge [\![s_{\mathtt{b}} \le 4]\!] \wedge [\![1 \le s_{\mathtt{e}}]\!] \wedge [\![s_{\mathtt{e}} \le 4]\!] \wedge [\![0 \le s_{\mathtt{f}}]\!] \wedge [\![s_{\mathtt{f}} \le 4]\!] \rightarrow$ false. A minimal explanation picking time 5 is $[\![-1 \le s_{\mathtt{b}}]\!] \wedge [\![s_{\mathtt{b}} \le 5]\!] \wedge [\![1 \le s_{\mathtt{e}}]\!] \wedge [\![s_{\mathtt{e}} \le 5]\!] \wedge [\![-1 \le s_{\mathtt{f}}]\!] \wedge [\![s_{\mathtt{f}} \le 5]\!] \rightarrow$ false. Note that each explanation is stronger than the previous one, and hence more reusable. Note also that some of the lower bounds (0 and -1) are universally true and can be omitted from the explanations.

6.2 Time-table filtering

Time-table filtering is based on the resource profile of the compulsory parts of all tasks. In a filtering without explanation the height of the compulsory parts concerning one time point or a time interval is given. For a task the profile is scanned through to detect time intervals where it cannot be executed. The lower (upper) bound of the task's start time is updated to the first (last) possible time point with

[4] In our experiments, successors of a task appear later in the order than the task.

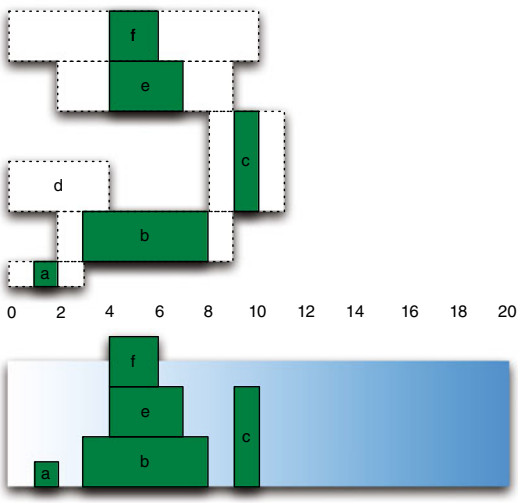

Fig. 3 An example of an inconsistent partial schedule for the `cumulative` constraint

respect to those time intervals. If we want to explain the new lower (upper) bound we need to know additionally which tasks have the compulsory parts of those time intervals.

A *profile* is a triple (A, B, C) where $A = [\,\mathbf{s}\,..\,\mathbf{e}-1\,]$ is a time interval, B the set of all tasks i with $ub(s[i]) \leq \mathbf{s}$ and $lb(s[i]) + d[i] \geq \mathbf{e}$ (that is a compulsory part in the time interval $[\,\mathbf{s}\,..\,\mathbf{e}-1\,]$), and C the sum of the resource requirements $r[i]$ of all tasks i in B. Here, we only consider profiles with a maximal time interval A with respect to B and C, i.e., there exists no other profile $(\left[\,\mathbf{s}'\,..\,\mathbf{e}'-1\,\right], B, C)$ where $\mathbf{s}' = \mathbf{e}$ or $\mathbf{e}' = \mathbf{s}$.

Let us consider the case when the lower bound of the start time variable for task j can be maximally increased from its current value $lb(s[j])$ to a new value $LB[j]$ using time-table filtering (the case of decreasing upper bounds in analogous and omitted). Then there exist a sequence of profiles $[D_1, \ldots, D_p]$ where $D_i = ([\,\mathbf{s}_i\,..\,\mathbf{e}_i-1\,], B_i, C_i)$ where $\mathbf{e}_0 = lb(s[j])$ and $\mathbf{e}_p = LB[j]$ such that

$$\forall 1 \leq i \leq p:\ C_i + r[j] > c \wedge \mathbf{s}_i \leq \mathbf{e}_{i-1} + d[j]$$

Hence each profile D_i pushes the start time of task j to $\mathbf{e}_i$.

A *naïve* explanation of the whole propagation would reflect the current domain bounds from the involved tasks.

$$\left([\![lb(s[j]) \leq s[j]]\!] \wedge \bigwedge_{1 \leq i \leq p, l \in B_i} [\![lb(s[l]) \leq s[l]]\!] \wedge [\![s[l] \leq ub(s[l])]\!] \right) \rightarrow [\![LB[j] \leq s[j]]\!]$$

As for the consistency check it is possible to use smaller (bigger) values in the inequalities to get a big-step explanation.

$$\left([\![\mathbf{s}_1 + 1 - d[j] \leq s[j]]\!] \wedge \bigwedge_{1 \leq i \leq p, l \in B_i} [\![\mathbf{e}_i - d[l] \leq s[l]]\!] \wedge [\![s[l] \leq \mathbf{s}_i]\!] \right) \rightarrow [\![LB[j] \leq s[j]]\!]$$

Both the above explanations are likely to be very large (they involve all start times appearing in the sequence of profiles) and hence are not likely to be very reusable.

One solution is to generate separate explanations for each profile D_i starting from the earliest time interval. An explanation for the profile $D_i = ([\,\mathbf{s}_i\,..\,\mathbf{e}_i - 1\,],\, B_i,\, C_i)$ which forces the lower bound of task j to move from $\mathbf{e}_{i-1}$ to $\mathbf{e}_i$ is

$$\left([\![\mathbf{s}_i + 1 - d[j] \leq s[j]]\!] \wedge \bigwedge_{l \in B_i} [\![\mathbf{e}_i - d[l] \leq s[l]]\!] \wedge [\![s[l] \leq \mathbf{s}_i]\!] \right) \rightarrow [\![\mathbf{e}_i \leq s[j]]\!]$$

This corresponds to a big-step explanation of inconsistency over the time interval $[\,\mathbf{s}_i\,..\,\mathbf{e}_i - 1\,]$.

Again we can use pointwise explanations based on single time points rather than a big-step explanation for the whole time interval. Different from the consistency case we may need to pick a set of time points no more than $d[j]$ apart to explain the increasing of the lower bound of $s[j]$ over the time interval. For a profile with length greater than the duration of task j we may need to pick more than one time point in a profile. Let $[t_1, \ldots, t_m]$ be a set of time points such that $t_0 = lb(s[j])$, $t_m + 1 = LB[j]$, $\forall 1 \leq j \leq m : t_{j-1} + d[j] \geq t_j$ and there exists a mapping $P(t_l)$ of time points to profiles such that $\forall 1 \leq l \leq m : \mathbf{s}_{P(t_l)} \leq t_l < \mathbf{e}_{P(t_l)}$ Then we build a pointwise explanation for each time point t_l, $1 \leq l \leq m$

$$\left([\![t_l + 1 - d[j] \leq s[j]]\!] \wedge \bigwedge_{k \in B_i} [\![t_l + 1 - d[l] \leq s[k]]\!] \wedge [\![s[k] \leq t_l]\!] \right) \rightarrow [\![t_l + 1 \leq s[j]]\!]$$

This corresponds to a set of pointwise explanations of inconsistency. We use these pointwise explanations in our experiments, by starting from $t_0 = lb(s[j])$ and for $j \in [\,1\,..\,m\,]$ we choose t_j as the greatest time maintaining the conditions above. The exception is that if we never entirely skip a profile D_i even if this is possible, but instead choose $\mathbf{e}_i - 1$ as the next time point and continue the process. Our experiments show this is slightly preferable to the skipping a profile entirely.

Example 9 Consider the example shown in Fig. 2b. which adds $s_{\mathtt{c}} \leq 9$, $s_{\mathtt{e}} \leq 4$ to the original problem. The profile filtering propagator can determine that task $\mathtt{f}$ can start earliest at time 10, since it cannot fit earlier. Clearly because there is one resource unit missing (one available but two required) in the period $[\,4\,..\,7\,]$ it must either end before or start after this period. Since it cannot end before it must start after this point. Similarly for the period $[\,9\,..\,10\,]$ is must either end before or start after, and since it cannot end before it must start after. So for this example $lb(s_{\mathtt{f}}) = 0$ and $LB[\mathtt{f}] = 10$ and there are two profiles used $[D_1, D_2] = [([\,4\,..\,7\,], \{\mathtt{b}, \mathtt{e}\}, 4), ([\,9\,..\,10\,], \{\mathtt{c}\}, 4)]$.

The naïve explanation is just to take the bounds of the tasks ($\mathtt{b}$, $\mathtt{e}$, $\mathtt{c}$) involved in the profile that is used: e.g. $[\![2 \leq s_{\mathtt{b}}]\!] \wedge [\![s_{\mathtt{b}} \leq 3]\!] \wedge [\![2 \leq s_{\mathtt{e}}]\!] \wedge [\![s_{\mathtt{e}} \leq 4]\!] \wedge [\![8 \leq s_{\mathtt{c}}]\!] \wedge [\![s_{\mathtt{f}} \leq 9]\!] \rightarrow [\![10 \leq s_{\mathtt{f}}]\!]$. (Note we omit redundant literals such as $[\![0 \leq s_{\mathtt{f}}]\!]$).

The iterative profile explanation explains each profile separately as $[\![1 \leq s_{\mathtt{b}}]\!] \wedge [\![s_{\mathtt{b}} \leq 4]\!] \wedge [\![2 \leq s_{\mathtt{e}}]\!] \wedge [\![s_{\mathtt{e}} \leq 4]\!] \rightarrow [\![7 \leq s_{\mathtt{f}}]\!]$ and $[\![4 \leq s_{\mathtt{f}}]\!] \wedge [\![8 \leq s_{\mathtt{c}}]\!] \wedge [\![s_{\mathtt{c}} \leq 9]\!] \rightarrow [\![10 \leq s_{\mathtt{f}}]\!]$.

The iterative pointwise explanation picks a set of time points, say 5 and 9, whose corresponding profiles are D_1 and D_2, and explains each time point minimally giving: $[\![s_{\mathtt{b}} \leq 5]\!] \wedge [\![1 \leq s_{\mathtt{e}}]\!] \wedge [\![s_{\mathtt{e}} \leq 5]\!] \rightarrow [\![6 \leq s_{\mathtt{f}}]\!]$ and $[\![4 \leq s_{\mathtt{f}}]\!] \wedge [\![8 \leq s_{\mathtt{c}}]\!] \wedge [\![s_{\mathtt{c}} \leq 9]\!] \rightarrow [\![10 \leq s_{\mathtt{f}}]\!]$. Note that this explanation is analogous to the explanation devised by the decomposition in Example 6, and stronger than the iterative profile explanation.

The global `cumulative` using time-table filtering and the *TimeD* decomposition have the same propagation strength. The advantages of the global approach is that we can control the times points we propagate on, while the decomposition in the worst case may propagate on every time point in every profile. The possible advantage of the decomposition is that it learns smaller nogoods related to the decomposed variables, but since B_{it} simply represents a fixed conjunction of bounds in practice the nogoods learned by the *TimeD* decomposition have no advantage.

6.3 (Extended) edge-finding filtering

The (extended) edge-finding filtering [29] is based on task intervals and reasoning about the task's energy, $\text{energy}_i = d[i] \times r[i]$, the area spanned by task's duration $d[i]$ and resource requirement $r[i]$. Edge finding finds a set of tasks Ω that all must occur in the time interval $[\,\mathbf{s}_\Omega \,..\, \mathbf{e}_\Omega\,]$ such that the total resources used by Ω is close to the amount available in that time interval $(c \times (\mathbf{e}_\Omega - \mathbf{s}_\Omega))$. If placing task j at its earliest start time will require more than the remaining amount of resources from this range, then task j cannot be strictly before any task in Ω. We can then update its lower bound accordingly.

Since the edge-finding and extended edge-finding are highly related we combine them in one rule and refer to them just as edge-finding for simplicity. Given a set of tasks Ω and task $j \notin \Omega$ where energy_Ω, $\mathbf{e}_\Omega$ and $\mathbf{s}_\Omega$ generalise the notation of the energy, the end time and start time from tasks to task sets, and o^j_Ω is the maximum resource usage for task j before $\mathbf{s}_\Omega$:

$$\text{energy}_\emptyset = 0 \quad \text{energy}_\Omega = \sum_{i \in \Omega} \text{energy}_i \quad \mathbf{e}_\emptyset = +\infty \quad \mathbf{e}_\Omega = \max_{i \in \Omega}(ub\,(s[i]) + d[i])$$

$$\mathbf{s}_\emptyset = -\infty \quad \mathbf{s}_\Omega = \min_{i \in \Omega}(lb\,(s[i])) \quad o^j_\Omega = r[j] \times \max(\mathbf{s}_\Omega - lb\,(s[j]), 0)$$

then

$$j \in T, \Omega \subseteq T \setminus \{j\} : \text{energy}_\Omega + \text{energy}_j > c \times (\mathbf{e}_\Omega - \mathbf{s}_\Omega) + o^j_\Omega \Rightarrow j \ll i . \forall i \in \Omega$$

If the rule holds the lower bound of the start time of the task j can be increased to

$$LB[j] := \max_{\Omega' \subseteq \Omega : rest(\Omega', r[j]) > 0} \mathbf{s}_{\Omega'} + \left\lceil \frac{\text{rest}(\Omega', r[j])}{r[j]} \right\rceil$$

where $rest(\Omega', r[j]) = \text{energy}_{\Omega'} - (c - r[j]) \times (\mathbf{e}_{\Omega'} - \mathbf{s}_{\Omega'})$. Figure 4 illustrates the (extended) edge-finding rule.

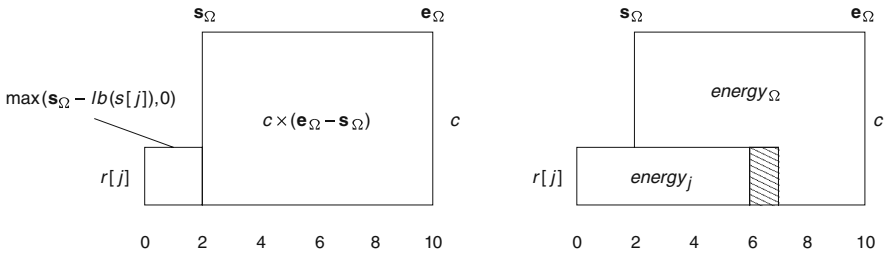

Fig. 4 The *left hand side* of the figure illustrates the available energy within the interval $[\,\mathbf{s}_\Omega \,..\, \mathbf{e}_\Omega\,]$ plus the additional energy o^j_Ω when task j starts earlier than $\mathbf{s}_\Omega$, while the *right hand side* illustrates the required energy if j starts earlier than all tasks in Ω. For the illustrated situation we have $j = \mathtt{f}$, $lb\,(s[j]) = 0$ and $\Omega = \{\mathtt{b}, \mathtt{c}, \mathtt{e}\}$. Since there is unused energy (*the shaded area*) no propagation occurs

A naïve explanation would be

$$\left(\llbracket lb\,(s[j]) \leq s[j] \rrbracket \wedge \bigwedge_{i \in \Omega} (\llbracket lb\,(s[i]) \leq s[i] \rrbracket \wedge \llbracket s[i] \leq ub\,(s[i]) \rrbracket)\right) \rightarrow \llbracket LB[j] \leq s[j] \rrbracket$$

In order to gain a stronger explanation we can maximise the bounds on the tasks in Ω. For that we have to consider that the condition for the edge-finding solely depends on Ω, and j and the new lower bound for j on a subset Ω' of Ω. We obtain a big-step explanation.

$$\left(\llbracket \mathbf{s}_{\Omega \cup \{j\}} \leq s[j] \rrbracket \wedge \bigwedge_{i \in \Omega \setminus \Omega'} (\llbracket \mathbf{s}_\Omega \leq s[i] \rrbracket \wedge \llbracket s[i] + d[i] \leq \mathbf{e}_\Omega \rrbracket) \wedge \bigwedge_{i \in \Omega'} (\llbracket \mathbf{s}_{\Omega'} \leq s[i] \rrbracket \wedge \llbracket s[i] + d[i] \leq \mathbf{e}_{\Omega'} \rrbracket)\right) \rightarrow \llbracket LB[j] \leq s[j] \rrbracket$$

If we look at the unused energy of the time interval $[\,\mathbf{s}_\Omega \,..\, \mathbf{e}_\Omega\,]$ and the energy needed for the task j in that time interval if j is scheduled at its earliest start time $lb\,(s[j])$ then the difference of the latter energy minus the former can be larger than 1. This would mean that the remaining energy can be used to strengthen the explanation in some way.

The remaining energy Δ concerning a task j and the Ω satisfying the edge-finding condition is

$$\Delta = \mathbf{e}_\Omega + r[j] \times (lb\,(s[j]) + d[j] - \max(\mathbf{s}_\Omega, s[j])) - c \times (\mathbf{e}_\Omega - \mathbf{s}_\Omega)$$

There exists different (non-exclusive) options to use this energy to strengthen the explanation where $\Omega' \subseteq \Omega$ maximises $LB[j]$.

1. By increasing the value of the end time $\mathbf{e}_\Omega$ if $\Delta / c > 1$, but not for Ω'.
2. By decreasing the value of the start time $\mathbf{s}_\Omega$ if $\Delta / c > 1$, but not for Ω'.
3. By decreasing the value of the start time $\mathbf{s}_j$ if $\Delta / r[j] > 1$.
4. By removing a task i from $\Omega \setminus \Omega'$ if $\Delta > \text{energy}_i$

Finally, if there exists several $\Omega' \subseteq \Omega$ with which the lower bound of task j could be improved then the explanation can be split into several explanation as in the

time-table filtering case. The filtering algorithm presented by Vilím [38][5] and Mercier [27] directly computes $LB[j]$ and its stepwise computation is hidden in the core of those algorithms. Hence, they have to be adjusted in a way that can increase the runtime complexity from $\mathcal{O}(kn\log(n))$ to $\mathcal{O}(kn^2\log(n))$ and $\mathcal{O}(kn^2)$ to $\mathcal{O}(kn^3)$ where n is the number of tasks and k the number of distinct resource usages.

For a pointwise explanation, rather than consider all $\Omega' \subseteq \Omega$ we restrict ourselves to explain the bounds update generated by Ω. The big-step explanation is generated as before (but $\Omega = \Omega'$). With the new bound the edge-finding rule (6.3) will now hold for Ω' and we can explain that. Clearly, the pointwise explanation is stronger than the big-step explanation, because we broaden the bounds requirements on the tasks in Ω' in the big-step explanation, and the later explanation only considers tasks in Ω'.

Example 10 Consider the example of Example 1 where we split the task e into two parts e1 of duration 2 and e2 of duration 3, but place no precedence constraints on these new tasks. Suppose the domains of the variables are $D(s_{\mathtt{a}}) = \{0\}$, $D(s_{\mathtt{b}}) = \{2\}$, $D(s_{\mathtt{c}}) = \{8\}$, $D(s_{\mathtt{d}}) = [0\,..\,2]$, $D(s_{\mathtt{e1}}) = [2\,..\,6]$, $D(s_{\mathtt{e2}}) = [2\,..\,5]$, $D(s_{\mathtt{f}}) = [2\,..\,14]$, so tasks a, b and c are fixed. The situation is illustrated in Fig. 5. The time-table filtering propagator cannot determine any propagation since f seems to fit at time 2, or after time 10. But in reality there is not enough resources for f in the time interval $[2\,..\,10]$ since this must include the tasks b, c, e1, and e2. The total amount of resource available here is $5 \times 8 = 40$, but 12 are taken by b and 8 are taken by c, e1 requires 4 resource units somewhere in this interval, and e2 required 6 resource units. Hence there are only 10 resource units remaining in the interval $[2\,..\,10]$. Starting f at time 2 requires at least 12 resource units be used within this interval hence this is impossible.

The edge-finding condition holds for $\Omega = \{\mathtt{b}, \mathtt{c}, \mathtt{e1}, \mathtt{e2}\}$ and f. Now $rest(\{\mathtt{c}\}, 2) = 10 - (5-2) \times (10-8) = 4$ for $\Omega' = \{\mathtt{c}\}$. The lower bound calculated is $LB[\mathtt{f}] = 8 + \lceil 4/2 \rceil = 10$. The naïve explanation is $[\![2 \le s_{\mathtt{f}}]\!] \wedge [\![2 \le s_{\mathtt{b}}]\!] \wedge [\![s_{\mathtt{b}} \le 2]\!] \wedge [\![2 \le s_{\mathtt{e1}}]\!] \wedge [\![s_{\mathtt{e1}} \le 6]\!] \wedge [\![2 \le s_{\mathtt{e2}}]\!] \wedge [\![s_{\mathtt{e2}} \le 5]\!] \wedge [\![8 \le s_{\mathtt{c}}]\!] \wedge [\![s_{\mathtt{c}} \le 8]\!] \rightarrow [\![10 \le s_{\mathtt{f}}]\!]$.

The big-step explanation ensures that each task in $\Omega \setminus \Omega'$ uses at least the amount of resources in the interval $[2\,..\,10]$ as the reasoning above. It is $[\![2 \le s_{\mathtt{f}}]\!] \wedge [\![2 \le s_{\mathtt{b}}]\!] \wedge [\![s_{\mathtt{b}} \le 4]\!] \wedge [\![2 \le s_{\mathtt{e1}}]\!] \wedge [\![s_{\mathtt{e1}} \le 8]\!] \wedge [\![2 \le s_{\mathtt{e2}}]\!] \wedge [\![s_{\mathtt{e2}} \le 7]\!] \wedge [\![8 \le s_{\mathtt{c}}]\!] \wedge [\![s_{\mathtt{c}} \le 8]\!] \rightarrow [\![10 \le s_{\mathtt{f}}]\!]$.

Now let us consider the pointwise explanation. If we restrict ourselves to use Ω for calculating the new lower bound we determine $LB[j] = 2 + \lceil (30 - (5-2) \times (10-2))/2 \rceil = 5$ and the big-step explanation is $[\![2 \le s_{\mathtt{f}}]\!] \wedge [\![2 \le s_{\mathtt{b}}]\!] \wedge [\![s_{\mathtt{b}} \le 4]\!] \wedge [\![2 \le s_{\mathtt{e1}}]\!] \wedge [\![s_{\mathtt{e1}} \le 8]\!] \wedge [\![2 \le s_{\mathtt{e2}}]\!] \wedge [\![s_{\mathtt{e2}} \le 7]\!] \wedge [\![2 \le s_{\mathtt{c}}]\!] \wedge [\![s_{\mathtt{c}} \le 8]\!] \rightarrow [\![5 \le s_{\mathtt{f}}]\!]$, which results in the situation shown in Fig. 5b. We then detect that edge-finding condition now holds for $\Omega' = \{\mathtt{c}\}$ which creates a new lower bound 10, and explanation $[\![5 \le s_{\mathtt{f}}]\!] \wedge [\![8 \le s_{\mathtt{c}}]\!] \wedge [\![s_{\mathtt{c}} \le 8]\!] \rightarrow [\![10 \le s_{\mathtt{f}}]\!]$. The original big-step explanation is broken into two parts, each logically stronger than the original.

For the original big-step explanation $\Delta = 30 + 2 \times (2 + 6 - 2) - 5 \times (10 - 2) = 2$. We cannot use this to improve the explanation since it is not large enough. But for the second explanation in the pointwise approach $\Delta = 10 + 2 \times (5 + 6 - 8) - 5 \times (10 - 8) = 6$. We could weaken the second explanation (corresponding to point 3 in the

[5] Vilím only presents the edge-finding algorithm, but the algorithm can be extended for extended edge-finding.

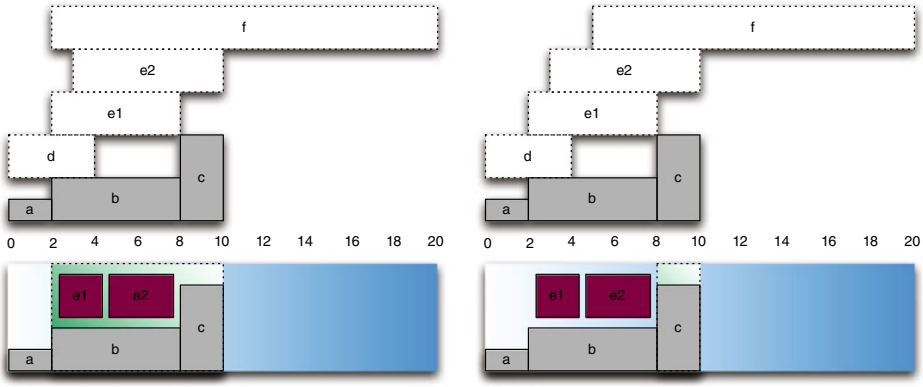

Fig. 5 **a** A example of propagation of the `cumulative` constraint using edge-finding. **b** The result of propagating after the first step of stepwise edge-finding

enumeration on the page 17) to $\underline{[\![3 \leq s_{\mathtt{f}}]\!]} \wedge [\![8 \leq s_{\mathtt{c}}]\!] \wedge [\![s_{\mathtt{c}} \leq 8]\!] \rightarrow [\![10 \leq s_{\mathtt{f}}]\!]$ because $\Delta/r[\mathtt{f}] = 6/2 = 3 > 1$.

We have not yet implemented edge-finding with explanation, since the problems we examine are not highly cumulative and for such problems edge-finding filtering cannot compete with time-table filtering. It remains interesting future work to experimentally compare different approaches to edge-finding filtering with explanation.

7 Resource-constrained project scheduling problems

Resource-constrained project scheduling problems (RCPSP) appear as variants, extensions and restrictions in many real-world scheduling problems. Therefore we test the time, task decomposition and the explaining `cumulative` propagator on the well-known RCPSP benchmark library PSPLib [22].

An RCPSP is denoted by a triple (T, A, R) where T is a set of tasks, A a set of precedences between tasks and R is a set of resources. Each task i has a duration $d[i]$ and a resource usage $r[k, i]$ for each resource $k \in R$. Each resource k has a resource capacity $c[k]$.

The goal is to find either a schedule or an optimal schedule with respect to an objective function where a schedule s is an assignment which meets the following conditions

$$\forall i \ll j \in A : \quad s[i] + d[i] \leq s[j]$$

$$\forall t \in [\,0\,..\,t_{\max} - 1\,], \forall k \in R : \quad \sum_{i \in T : s[i] \leq t < s[i]+d[i]} r[k, i] \leq c[k] \ ,$$

where $t_{\max}$ is the planning horizon. For our experiments we search for a schedule which minimises the makespan t_{ms} such that $s[i] + d[i] \leq t_{\text{ms}}$ holds for $i = 1, \ldots, n$.

The following code gives a basic Zinc model for the RCPSP problem.

RCPSP_basic_model.zinc

```
    % Parameters
int: t_max;     % planning horizon
enum Tasks;     % set of tasks
enum Resources; % set of resources
int: n = card(Tasks); % number of tasks
array[Tasks] of int: d; % durations of tasks
array[Tasks] of set of Tasks: suc;  % successors of each task
array[Resources, Tasks] of int: r;  % resource usage
array[Resources] of int: c; % resource limit
    % Variables
array[Tasks] of var 0..t_max: s; % start times of tasks
var 0..t_max: makespan;

    % Precedence constraints
constraint
    forall ( i in Tasks, j in suc[i] )
    ( s[i] + d[i] <= s[j] );
    % Resource constraints
constraint
    forall ( k in Resources )
    ( cumulative(array1d(1..n,s), array1d(1..n,d), [ r[k,i] |
     i in Tasks], c[k]));
    % Makespan constraints
constraint
    forall ( i in Tasks where suc[i] == {} )
    ( s[i] + d[i] <= makespan );
    % (Redundant) Non-overlap constraints
constraint
    forall ( i, j in Tasks, k in Resources
            where i < j /\ r[k,i] + r[k,j] > c[k] )
    ( s[i] + d[i] <= s[j]  \/  s[j] + d[j] <= s[i] );
    % Objective function
solve minimize makespan;
```

A Zinc data file representing the problem of Example 1 is

RCPSP_run_ex.dzn

```
t_max = 20;
enum Tasks = { ta, tb, tc, td, te, tf };
enum Resources = { single };
d = array1d(Tasks, [2,6,4,2,5,6]);
suc = array1d(Tasks, [{tb},{tc},{},{te},{},{}]);
r = array2d(Resources,Tasks,[1,2,4,2,2,2]);
c = array1d(Resources,[5]);
```

In practice we share the Boolean variables generated inside the `cumulative` constraints as described in Section 5.1 (by common sub-expression elimination) and add redundant constraints as described in Section 5.2 when using the *TaskD* decomposition. We also add redundant non-overlap constraints for each pair of tasks whose resource usages make them unable to overlap. Moreover, the planning horizon $t_{\max}$ was determined as the makespan of the first solution found by selection of the start time variable with the smallest lower bound (if a tie occurs then the lexicographic least variable) and assignment of the variable to its lower bound. The initial domain of each variable $s[i]$ was determined as $D_{\text{init}}(s[i]) = \left[\, p[i] \,..\, t_{\max} - q[i] \,\right]$ where $p[i]$ is the duration of the longest chain of predecessor tasks, and $q[i]$ is the duration of the longest chain of successor tasks.

In the remainder of this section we discuss alternative search strategies.

7.1 Search using serial scheduling generation

The serial scheduling generation scheme (serial SGS) is one of basic deterministic algorithms to assign stepwise a start time to an unscheduled task. It incrementally extends a partial schedule by choosing an *eligible* task—i.e. all of whose predecessors are fixed in the partial schedule—and assigns it to its earliest start time with respect to the precedence and resource constraints. For more details about SGS, different methods based on it, and computational results in Operations Research see [14, 20, 21].

Baptiste and Le Pape [2] adapt serial SGS for a constraint programming framework. For our experiments we use a form where we do not apply their dominance rules, and where we impose a lower bound on the start time instead of posting the delaying constraint "task i executes after at least one task in S".

1. *Select* an eligible unscheduled task i with the earliest start time $t = lb(s[i])$. If there is a tie between some tasks then select that one with the minimal latest start time $ub(s[i])$. If still tied then choose the lexicographic least task. Create a choice point.
2. *Left branch:* Extend the partial schedule by setting $s[i] = t$. If this branch fails then go to the right branch; Otherwise go to step 1.
3. *Right branch:* Delay task i by setting $s[i] \geq t'$ where $t' = \min\{lb(s[j]) + d[j] \mid j \in T : lb(s[j]) + d[j] > lb(s[i])\}$, that is, the earliest end time of the concurrent tasks. If this branch fails then backtrack to the previous choice point; Otherwise go to step 1.

The right branch uses the dominance rule that amongst all optimal schedules there exists one where every task starts either at the first possible time or immediately after the end of another task. Therefore, the imposing of the new lower bound is sound, no solution is lost for the considered problem. If we add side constraints then this assumption could be invalid.

Note that we use this search strategy with branch and bound, where whenever a new solution is found (with $makespan = t$), a constraint requiring a better solution ($makespan < t$) is dynamically (globally) added during the search.

7.2 Search using variable state independent decaying sum

The SAT decision heuristic Variable State Independent Decaying Sum (VSIDS) [28] is a generic search approach that is currently almost universally used in DPLL SAT solvers. Each variable is associated with a dynamic *activity* counter that is increased when the variable is involved in a failure. Periodically, all counters are reduced, thus *decaying*. The unfixed variable with the highest activity is selected to branch on at each stage. Benchmark results by Moskewicz [28] shows that VSIDS performs better on average on hard problems than other heuristics.

To use VSIDS in a lazy clause generation solver, we ask the SAT solver what its preferred literal for branching on is. This corresponds to an atomic constraint $x \leq d$ or $x = d$ and we branch on $x \leq d \vee x > d$ or $x = d \vee x \neq d$. Note that the search is still controlled by the FD search engine, so that we use its standard approach to implementing branch-and-bound to implement the optimisation search.

Normally SAT solvers use dichotomic restart search for optimisation as the SAT solver itself does not have optimisation search built in. That is assuming *minspan* is the current lower bound on makespan, and a new solution is found with $makespan = t$ we let $t' = \lfloor (t + minspan)/2 \rfloor$ and solve the satisfaction problem to find a *makespan* in the range $[\, minspan \,..\, t' \,]$. If we find a solution at t'' we continue the process, otherwise we reset *minspan* to $t' + 1$ and solve the satisfaction problem to find a *makespan* in the range $[\, minspan \,..\, t - 1 \,]$. Note that when a satisfaction search fails the nogoods generated in that search are not valid for subsequent searches (since they make use of the assumption $makespan \leq t'$.

The combination of VSIDS and branch and bound is much stronger since in the continuation of the search with a better bound, the activity counts at the time of finding a new better solution are used in the same part of the search tree, and all nogoods generated remain valid.

Restarting is shown to be beneficial in SAT solving (and CSP solving) in speeding up solution finding, and being more robust on hard problems. On restart the set of nogoods has changed as well as the activity of variables, so the search will take a very different path. We also use VSIDS search with restarting, which we denote RESTART.[6]

7.3 Hybrid search strategies

One drawback of VSIDS is that at the beginning of the search the activity counters are only related to the clauses occurring in the original model, and not to any conflict. This is exacerbated in lazy clause generation where many of the constraints of the problem may not appear at all in the clause database initially. This can lead to poor decisions in the early stages of the search. Our experiments support this, there are a number of "easy" instances which SGS can solve within a small number of choice points, where VSIDS requires substantially more.

In order to avoid these poor decisions we consider a hybrid search strategy. We use SGS for the first 500 choice points and then restart the search with VSIDS. The SGS search may solve the whole problem if it is easy enough, but otherwise it sets

[6]Note that restarting with SGS is not very attractive since we rarely learn anything that changes the SGS search decisions, we effectively just continue the same search.

the activity counters to meaningful values so that VSIDS starts concentrating on meaningful decisions. We denote this search as HOT START, and the version where the secondary VSIDS search also restarts as HOT RESTART.

8 Experiments

We carried out extensive experiments on RCPSP instances comparing our approaches to decomposition without explanation, global `cumulative` propagators from SICSTUS and ECLiPSe, as well as a state-of-the-art exact solving algorithm [23]. Detailed results are available at http://www.cs.mu.oz.au/~pjs/rcpsp.

We use two suites of benchmarks. The library PSPLib [22, 32] contains the four classes J30, J60, J90, and J120 consisting of 480 instances of 30, 60, 90, and 120 tasks respectively. We also use a suite (BL) of 40 highly cumulative instances with either 20 or 25 tasks constructed by Baptiste and Le Pape [2].

The experiments were run on a X86-64 architecture running GNU/Linux and a Intel(R) Xeon(R) CPU E54052 processor with 2 GHz. The code was written in Mercury using the G12 Constraint Programming Platform and compiled with the Mercury Compiler and grade hlc.gc.trseg. Each run was given a 10 min limit.

We compare 4 different implementations of `cumulative` with explanation: (`t`) the *TimeD* decomposition of Section 5.1, (`s`) the *TaskD* decomposition of Section 5.2, (`e`) a task decomposition using end times (`e`), and (`g`) a global `cumulative` using time-table filtering with explanation. The global `cumulative` uses pointwise explanations for consistency and iterative pointwise explanations for filtering. We experimented with other forms of explanations for the global `cumulative` but they were inferior for hard instances, although surprisingly not that bad (about 15% worse in average except for naïve explanations which behave terribly). The present implementation of the global `cumulative` recalculates the resource profile on each invocation which could be significantly improved by making it incremental. Profiling on a few instances showed that more than the half of the time was spent in propagation of `cumulative`.

8.1 Results on J30 and BL instances

The first experiment compares different decompositions and search on the smallest instances J30 and BL. We compare SGS, VSIDS, RESTART and the hybrid search approaches using our 4 different propagation with explanation approaches. The results are shown in Tables 1 and 2. For J30 we show the number of problems solved (#svd), (cmpr(477)) the average solving time in seconds and number of failures on the 477 problems that all approaches solved, and (all(480)) average solving time in seconds and number of failures on all 480 problems within the execution.[7] Note that we shall use similar comparisons and notation in future tables. The best results in each column are shown in italics. For the BL problems the results are shown in Table 2. We show the number of solved problems, (all(40)) average solving time

[7]This means that for problems that time out the number of failures is substantially larger than those which were solved before timeout.

Table 1 Results on J30 instances

Search	Model	#svd	cmpr(477)		all(480)	
			Time	Fails	Time	Fails
SGS	s	477	2.13	3,069	5.86	5,375
	e	477	2.19	3,054	5.93	5,331
	t	*480*	0.87	2,339	2.83	4,230
	g	*480*	0.73	1,977	3.04	3,919
VSIDS	s	*480*	1.20	2,128	1.63	2,984
	e	*480*	0.46	1,504	0.77	2,220
	t	*480*	0.26	1,002	0.33	1,271
	g	*480*	0.15	797	0.20	1,058
RESTART	s	*480*	0.50	1,483	0.93	2,317
	e	*480*	0.43	1,368	0.80	2,128
	t	*480*	0.24	856	0.33	1,174
	g	*480*	0.15	777	0.22	1,093
HOT START	t	*480*	0.21	779	0.34	1,220
	g	*480*	*0.12*	*706*	*0.17*	*956*
HOT RESTART	t	*480*	0.26	884	0.35	1,231
	g	*480*	0.13	727	0.21	1,058

and number of failures with a 10 min limit (on all 40 instances), as well as fails(4000) with a 4000 failure limit.

Of the decompositions the *TimeD* decomposition is clearly the best being almost twice as fast as the *TaskD* decompositions. This is presumably the effect of the stronger propagation. Note that these are the smallest problems where its relative disadvantage in size is least visible. The global is usually significantly better than the *TimeD* decomposition: it usually requires less search and can be up to twice as fast. Interestingly sometimes the *TimeD* decomposition is faster which may reflect the fact that it is (automatically) a completely incremental implementation of the `cumulative` constraint. For these small problems the best search strategy is HOT START since the overhead of restarting VSIDS does not pay off for these simple problems.

Table 2 Results on BL instances

Search	Model	#svd	all(40)		#svd	fails(4000)	
SGS	s	*40*	2.51	9,628	24	0.18	1,261
	e	*40*	2.63	9,443	24	0.15	1,144
	t	*40*	0.82	5,892	29	*0.04*	781
	g	*40*	0.88	5,723	30	0.05	860
VSIDS	s	*40*	0.79	4,436	31	0.16	1,115
	e	*40*	0.77	4,104	30	0.15	1,025
	t	*40*	0.22	2,540	34	*0.04*	661
	g	*40*	0.20	2,039	*37*	*0.04*	605
RESTART	s	*40*	0.88	4,549	31	0.17	1,169
	e	*40*	1.46	5,797	32	0.17	1,135
	t	*40*	0.13	1,626	35	0.05	603
	g	*40*	0.14	1,568	36	*0.04*	*546*
HOT START	t	*40*	*0.10*	*1,448*	36	*0.04*	680
	g	*40*	0.12	1,485	36	*0.04*	593
HOT RESTART	t	*40*	0.15	1,829	35	0.05	719
	g	*40*	0.25	2,460	36	0.05	680

Table 3 Results of the FD solvers on the J30 instances

Solver		#svd	cmpr(364)		all(480)	
SICStus	Default	418	0.22	337	87.43	141,791
	Global	415	0.40	331	94.01	76,533
ECLiPSe	cumu	368	13.98	26,469	154.79	365,364
	ef	366	18.33	21,717	157.43	173,445
	ef3	368	16.61	17,530	155.87	155,142
G12	FD + t	404	1.79	5,701	104.38	641,185
	Sgs + t	*480*	*0.01*	75	*2.83*	4,230
	Sgs + g	*480*	*0.01*	*70*	3.04	*3,919*

The results on the BL instances show that approaches using *TimeD* or the global propagator and Vsids could solve between six and nine instances more than the base approach (FE) of Baptiste and Le Pape [2] within 4000 failures. Their "left-shift/right-shift" approach could solve all 40 instances in 30 min, with an average of 3,634 failures and 39.4 s on a 200 MHz machine. All our approaches with *TimeD* and Vsids find the optimal solution faster and in fewer failures (between a factor of 1.39 and 2.4).

Next we compare the *TimeD* decomposition (Sgs + t) and global propagator (Sgs + g) against implementations of `cumulative` in SICStus v4.0 (default, and with the flag `global`) and ECLiPSe v6.0 (using its 3 `cumulative` versions from the libraries `cumulative`, `edge_finder` and `edge_finder3`). We also compare against (FD + t) a decomposition without explanation (a normal FD solver) executed in the G12 system. All approaches use the Sgs search strategy.

The results are shown in the Tables 3 and 4. Clearly the more expensive edge-finding filtering algorithms are not advantageous on the J30 examples, but they do become significantly beneficial on the highly cumulative BL instances. We can see that none of the other approaches compare to the lazy clause generation approaches. The best solver without learning is the SICStus `cumulative` with `global` flag. Clearly nogoods are very important to fathom search space.

While the *TimeD* decomposition clearly outperforms *TaskD* on these small examples, as the planning horizon grows at some point *TaskD* should be better, since its model size is independent of the planning horizon. We took the J30 examples and multiplied the durations and planning horizon by 10 and 100. We compare the *TimeD* decomposition versus the (e) end-time *TaskD* decomposition (which is slightly better than start-time (s)) and the global `cumulative` (g). The results are shown in Table 5. First we should note that simply increasing the durations makes the problems significantly more difficult. While the *TimeD* decomposition is still just better than the *TaskD* decomposition for the 10× extended examples,

Table 4 Results of the FD solvers on the BL instances

Solver		#svd	cmpr(7)		all(40)	
SICStus	Default	32	2.87	25,241	195.05	1,896,062
	Global	39	0.90	3,755	18.36	63,310
ECLiPSe	cumu	7	178.90	352,318	526.61	2,231,026
	ef	37	43.06	53,545	102.60	229,332
	ef3	37	35.56	39,836	81.47	144,051
G12	FD + t	30	6.71	72,427	216.87	2,650,886
	Sgs + t	40	*0.01*	*268*	*0.82*	5,892
	Sgs + g	40	*0.01*	278	0.88	*5,723*

Table 5 Results on the modified J30 instances

Search	Duration	Model	#svd	cmpr(462)		all(480)	
SGS	1×	e	477	0.11	542	5.93	5,331
		t	*480*	0.08	501	2.83	4,230
		g	*480*	*0.05*	*371*	*3.04*	*3,919*
	10×	e	471	0.58	1,812	14.76	9,512
		t	476	1.03	676	11.04	4,972
		g	*478*	*0.10*	*393*	*4.92*	*4,291*
	100×	e	466	4.87	4,586	23.40	10,813
		t	465	15.58	724	35.02	*1,582*
		g	*477*	*0.70*	*403*	*8.51*	3,684
VSIDS	1×	e	*480*	0.06	318	0.77	2,220
		t	*480*	0.04	249	0.33	1,271
		g	*480*	*0.02*	*151*	*0.20*	*1,058*
	10×	e	*480*	0.18	821	4.23	4,213
		t	*480*	0.63	1,210	4.84	2,714
		g	*480*	*0.06*	*284*	*0.39*	*1,215*
	100×	e	474	1.32	2,031	12.36	4,224
		t	469	9.88	9,229	27.35	9,707
		g	*480*	*0.62*	*1,296*	*3.15*	*2,360*

it is inferior for scheduling problems with very long durations. The most important result visible from this experiment is the advantage of the global propagator over the *TimeD* decomposition as the planning horizon gets larger. The global propagator is by far the best approach for the larger problems since it has the $\mathcal{O}(n^2)$ complexity of the *TaskD* decomposition but the same propagation strength as the much stronger *TimeD* decomposition. Note also how the failures get dramatically worse for the *TimeD* decomposition using VSIDS as the problem grows. This illustrates how the large number of Booleans in the decomposition makes the VSIDS heuristic less effective.

8.2 Results on J60, J90 and J120

We now examine the larger instances J60, J90 and J120 from PSPLib. For J60 we compare the most competitive search approaches from the previous subsection: VSIDS, RESTART, HOT START and HOT RESTART using the *TimeD* decomposition and global propagator. For this suite our solvers cannot solve all 480 instances within 10 min. The results are presented in Table 6. For these examples we show the average distance of the makespan from our best solution to the best known solution from

Table 6 Results on J60 instances for *TimeD* and global propagator

Search	Model	#svd	Avg. dist.	cmpr(425)		all(480)	
VSIDS	t	426	4.4	4.85	7,216	72.71	*41,891*
	g	*430*	6.2	2.99	4,943	67.13	52,016
RESTART	t	428	4.5	3.53	5,139	68.04	61,558
	g	*430*	*3.7*	*2.50*	*4,418*	*66.01*	60,518
HOT START	t	429	9.3	2.91	4,629	66.64	52,812
	g	428	18.1	2.93	4,848	66.19	57823
HOT RESTART	t	429	4.0	3.28	4,982	66.60	60,146
	g	*430*	3.9	2.60	4,658	66.18	60,652

Table 7 Results on J90 and J120 instances for *TimeD* and global propagator

J90					
Search	Model	#svd	Avg. dist.	cmpr(395)	all(480)
Hot Restart	t	396	*7.5*	4.16 4,364	108.90 90,582
	g	*397*	*7.5*	*3.34 3,950*	*108.57 90,134*

J120					
Search	Model	#svd	Avg. dist.	cmpr(274)	all(600)
Hot Restart	t	274	9.7	8.50 8,543	329.46 *234,897*
	g	*282*	*9.6*	*5.40 7,103*	*324.51* 242,343

PSPLib (most of which are generated by specialised heuristic methods), as well as the usual time and number of failures comparisons. Many of these are currently open problems. Our best approaches close 24 open instances. While all of the methods are quite competitive we see that restarting is valuable for improving the average distance from the best known optimal, and the hybrid approach Hot Restart is marginally more robust than the others. Interestingly Hot Start can clearly force the search into a less promising area than just plain Vsids.

For the largest instances J90 and J120 we ran only Hot Restart since it is the most robust search strategy, using the *TimeD* decomposition and the global propagator. The results are presented in the Table 7 which shows that the global propagator is superior to the *TimeD* decomposition. In total we close 15 and 27 open instances in J90 and J120 respectively (see Appendix A for more details).

We compare our best methods Hot Restart with either t or g to the method by Laborie [23] that uses minimal critical sets as a branching scheme and was implemented in ILOG Scheduler 6.1 using as filtering algorithms among others timetable and edge-finding. His method is the best published method so far on the J60, J90, and J120 instances.

Table 8 shows the percentage of solved instances within a maximal solve time. We give an equivalent time to our solver taking into account the speeds of the processors: 2.0 GHz vs. 1.4 GHz. At the top of the table is the time cutoff for a 1.4 GHz processor, and at the bottom the approximately equivalent cutoff times for a 2.0 GHz machine. Note, that all * marked 2.0 GHz times are much lower than the equivalent time for the 1.4 GHz processor. Clearly this comparison can only be seen as indicative.

Our methods clearly outperforms Laborie's method: for every class our methods were able to solve more problems within 10s than they could solve in half an hour respectively on their machine. Interestingly, our solver could not solve six instances

Table 8 Comparison between Laborie's method, Hot Restart + t and g

1.4 GHz	J60			J90			J120		
	15 s	300 s	1800 s	15 s	300 s	1800 s	15 s	300 s	1800 s
Laborie	–	84.2	85.0	–	78.5	79.4	–	41.3	41.7
Hot Restart + t	84.8	89.2	89.4	79.8	81.7	82.5	42.3	45.2	45.7
Hot Restart + g	85.8	89.0	89.6	80.0	81.9	82.7	42.7	45.8	47.0
2.0 GHz	10 s	200 s	600 s*	10 s	200 s	600 s*	10 s	200 s	600 s*

which were solved by others. We can also see the advantage of the global propagator increases with increasing problem size.

Finally we used Hot Start + g to try to improve lower bounds of the remaining open problems, by searching for a solution to the problem with the makespan varying from the best known lower bound to the best known upper bound from PSPLib. In this way we closed 6 more problems and improved 78 lower bounds (see Appendix B).

9 Conclusion

We present a new approach solving Rcpsp problems by using `cumulative` constraints with explanation in a lazy clause generation system. First we show that modelling `cumulative` constraints by decomposition and using lazy clause generation is highly competitive. We then improve this by building a global `cumulative` propagator with explanation. Benchmarks from the PSPLib show the strong power of nogoods and Vsids style search to fathom a large part of the search space. Without building complex specific global propagators or highly specialised search algorithms we are able to compete with highly specialised Rcpsp solving approaches and close 71 open problems.

Acknowledgements We would like to thank Phillipe Baptiste for suggesting this line of enquiry. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

Appendix A: Closed instances

In the Table 9 we list all previously open instances (wrt. [23, 32], and [25]) with their optimal makespan which were closed by Hot Restart with the global `cumulative` or some other method. In the last case a footnote is given for these instances. The optimal makespan of almost all closed instances correspond to the previously best known upper bound found by local search heuristics, except for the J120 instances 8_3 and 48_5 where our solver could reduce them by 1 to 95 and 110 resp. Note that Hot Restart with the *TimeD* decomposition was also capable of closing 63 of these instances.

In total 71 instances of 504 open instances are closed with 64 instances by Hot Restart with `cumulative` (`g`), one by Hot Restart with *TimeD*, one by Vsids with `cumulative`, four by the lower bound computation (see Appendix B) and one by a combination of two methods.[8]

Appendix B: New lower bounds

We tried to compute new lower bounds using Hot Start with global `cumulative` for the remaining open instances. These experiments were carried out on the same machine with the same (overall) time limit. (cf. Section 8). We set the makespan to

[8]Closed by Hot Restart and lower bound computation. Hot Restart decreased the previously best known upper bound to 95 and the lower bound computation proved the optimality of this new bound.

Table 9 Closed instances

J60	Instance	5_10	9_2	9_4	14_1	14_10	17_8	21_9	25_1
	Makespan	81	82	87	61	72	85	89	114
	Instance	25_3	25_5[a]	25_9	30_5	30_7	30_10	41_1	41_2
	Makespan	113	98	99	76	86	86	122	113
	Instance	41_6	41_9	46_4	46_5	46_6	46_7	46_9	46_10
	Makespan	134	131	74	91	90	78	69	88
J90	Instance	5_1	5_2	21_2[c]	21_4	21_5	21_6	21_9	21_10
	Makespan	78	93	116	106	112	106	121	109
	Instance	26_5[b]	37_1	37_4	37_5	37_8[c]	37_9	37_10[c]	42_2
	Makespan	85	110	123	126	119	123	123	102
	Instance	42_7	42_10						
	Makespan	87	90						
J120	Instance	1_3	1_8	1_10	2_2	8_3[d]	21_2	21_7	22_3
	Makespan	125	109	108	75	95	117	111	96
	Instance	22_8	28_4	28_8	28_9	28_10	29_4	41_2	41_9
	Makespan	103	112	99	98	116	80	141	121
	Instance	42_5	42_8	48_1	48_5	48_8	48_9	48_10	49_3
	Makespan	120	113	100	110	116	113	111	96
	Instance	49_4	49_5	49_7	49_10	50_4[c]			
	Makespan	96	89	99	97	100			

[a] Closed by VSIDS + g
[b] Closed by HOT RESTART + t
[c] Closed by lower bound computation by proof of the equality of lower and best known upper bound
[d] See footnote 8

Table 10 New lower bounds on all instances

J60	Instance	9_3	9_5	9_6	9_8	9_9	9_10	25_2	25_4
	LB	99	80	105	94	98	88	95	105
	Instance	25_6	25_7	25_8	25_10	29_2	29_9	30_2	41_3
	LB	105	88	95	107	123	105	69	89
	Instance	41_5	41_10	45_3	45_4				
	LB	109	105	133	101				
J90	Instance	5_4	5_6	5_8	5_9	21_1	21_7	21_8	37_2
	LB	101	85	96	113	109	105	107	113
	Instance	37_6	41_3	41_7	46_4				
	LB	129	147	144	92				
J120	Instance	1_1	6_8	7_2	7_3	7_6	8_2	8_4	8_6
	LB	104	140	113	97	115	101	91	84
	Instance	9_4	26_2	26_4	26_5	26_7	26_8	26_9	26_10
	LB	84	158	160	138	144	167	160	177
	Instance	27_2	27_5	27_7	27_10	28_7	29_3	34_8	42_1
	LB	109	105	118	110	108	96	86	106
	Instance	46_1	46_2	46_3	46_5	46_7	46_9	46_10	47_1
	LB	171	186	162	135	155	156	174	129
	Instance	47_2	47_4	47_5	47_9	48_3	48_6	48_7	49_2
	LB	126	119	125	140	109	102	105	108
	Instance	53_3	53_4	53_9	54_7	54_10	60_2		
	LB	105	137	155	108	107	82		

the best known lower bound, and tried to find a solution, if this failed we increased the makespan by one and re-solved. If a solution was found this is the optimal, if we can prove failure for a given makespan we have increased the lower bound. If the increased lower bound equaled the best known upper bound we have proved the optimality of the upper bound and closed the instance as well.

In total this method closed 6 more instances (see Appendix A) and improved the lower bound by 78 instances of the remaining 433 open instances. The improved lower bounds are listed in Table 10.

References

1. Aggoun, A., & Beldiceanu, N. (1993). Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling, 17*(7), 57–73.
2. Baptiste, P., & Le Pape, C. (2000). Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints, 5*(1–2), 119–139.
3. Blazewicz, J., Lenstraand, J. K., & Rinnooy Kan, A. H. G. (1983). Scheduling subject to resource constraints: Classification and complexity. *Discrete Applied Mathematics, 5*, 11–24.
4. Carlier, J., & Pinson, E. (2004). Jackson's pseudo-preemptive schedule and cumulative scheduling problems. *Discrete Applied Mathematics, 145*(1), 80–94. doi:10.1016/j.dam.2003.09.009.
5. Caseau, Y., & Laburthe, F. (1996). Cumulative scheduling with task intervals. In *Procs. of the 1996 Joint International Conference and Symposium on Logic Programming* (pp. 363–377). MIT. citeseer.ist.psu.edu/caseau94cumulative.html.
6. Claessen, K., Een, N., Sheeran, M., Sörensson, N., Voronov, A., & Åkesson, K. (2009). Sat-solving in practice, with a tutorial example from supervisory control. *Discrete Event Dynamic Systems, 19*(4), 495–524. doi:10.1007/s10626-009-0081-8.
7. Davis, M., Logemman, G., & Loveland, D. (1962). A machine program for theorem proving. *Communications of the ACM, 5*(7), 394–397.
8. Dechter, R., Meiri, I., & Pearl, J. (1991). Temporal constraint networks. *Artificial Intelligence, 49*, 61–95.
9. Demeulemeester, E. L., & Herroelen, W. S. (1997). New benchmark results for the resource-constrained project scheduling problem. *Management Science, 43*(11), 1485–1492.
10. Eén, N., & Sörensson, N. (2003). An extensible SAT-solver. In E. Giunchiglia & A. Tacchella (Eds.), *Proceedings of SAT 2003, LNCS* (Vol. 2919, pp. 502–518). Heidelberg: Springer.
11. El-Kholy, A. O. (1996). *Resource feasibility in planning*. Ph.D. thesis, Imperial College, University of London.
12. Erschler, J., & Lopez, P. (1990). Energy-based approach for task scheduling under time and resources constraints. In *2nd International Workshop on Project Management and Scheduling* (pp. 115–121). France: Compiègne.
13. Feydy, T., & Stuckey, P. J. (2009). Lazy clause generation reengineered. In I. Gent (Ed.), *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming, LNCS* (Vol. 5732, pp. 352–366). Springer-Verlag. doi:10.1007/978-3-642-04244-7_29.
14. Hartmann, S., & Kolisch, R. (2000). Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research, 127*(2), 394–407. doi:10.1016/S0377-2217(99)00485-3.
15. Jussien, N. (2003). *The versatility of using explanations within constraint programming*. Research Report 03-04-INFO, École des Mines de Nantes, Nantes, France. http://www.emn.fr/jussien/publications/jussien-RR0304.pdf.
16. Jussien, N., & Barichard, V. (2000). The PaLM system: Explanation-based constraint programming. In *Proceedings of Techniques foR Implementing Constraint Programming Systems (TRICS 2000)* (pp. 118–133). http://www.emn.fr/jussien/publications/jussien-WCP00.pdf.
17. Jussien, N., & Lhomme, O. (2002). Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence, 139*(1), 21–45.

18. Jussien, N., Debruyne, R., & Boizumault, P. (2000). Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming—CP 2000, no. 1894 in Lecture Notes in Computer Science* (pp. 249–261). Singapore: Springer-Verlag.
19. Katsirelos, G., & Bacchus, F. (2005). Generalized nogoods in csps. In M. M. Veloso & S. Kambhampati (Eds.), *National Conference on Artificial Intelligence* (pp. 390–396). AAAI Press/The MIT.
20. Kolisch, R. (1996). Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. *European Journal of Operational Research, 90*(2), 320–333. doi:10.1016/0377-2217(95)00357-6.
21. Kolisch, R., & Hartmann, S. (2006). Experimental investigation of heuristics for resource-constrained project scheduling: An update. *European Journal of Operational Research, 174*(1), 23–37. doi:10.1016/j.ejor.2005.01.065.
22. Kolisch, R., & Sprecher, A. (1997). PSPLIB—A project scheduling problem library. *European Journal of Operational Research, 96*(1), 205–216. doi:10.1016/S0377-2217(96)00170-1.
23. Laborie, P. (2005). Complete MCS-based search: Application to resource constrained project scheduling. In L. P. Kaelbling & A. Saffiotti (Eds.), *Proceedings IJCAI 2005* (pp. 181–186). Professional Book Center. http://ijcai.org/papers/0571.pdf.
24. Lahrichi, A. (1982). Scheduling: The notions of hump, compulsory parts and their use in cumulative problems. *Comptes Rendus de l'Académie des Sciences. Paris, Série 1, Mathématique, 294*(2), 209–211.
25. Liess, O., & Michelon, P. (2008). A constraint programming approach for the resource-constrained project scheduling problem. *Annals of Operations Research, 157*(1), 25–36. doi:10.1007/s10479-007-0188-y.
26. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P. J., Garcia de la Banda, M., & Wallace, M. G. (2008). The design of the Zinc modelling language. *Constraints, 13*(3), 229–267. doi:10.1007/s10601-008-9041-4.
27. Mercier, L., & Van Hentenryck, P. (2008). Edge finding for cumulative scheduling. *INFORMS Journal on Computing, 20*(1), 143–153. doi:10.1287/ijoc.1070.0226.
28. Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Design automation conference* (pp. 530–535). New York: ACM. doi:10.1145/378239.379017.
29. Nuijten, W. P. M. (1994). *Time and resource constrained scheduling*. Ph.D. thesis, Eindhoven University of Technology.
30. Ohrimenko, O., Stuckey, P., & Codish, M. (2009). Propagation via lazy clause generation. *Constraints, 14*(3), 357–391.
31. Ohrimenko, O., Stuckey, P. J., & Codish, M. (2007). Propagation = lazy clause generation. In *Procs. of the CP2007, LNCS* (Vol. 4741, pp. 544–558). Springer-Verlag. doi:10.1007/978-3-540-74970-7_39.
32. PSPLib—project scheduling problem library (2009). http://129.187.106.231/psplib/. Accessed 23 April 2009.
33. Schulte, C., & Stuckey, P. (2008). Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems, 31*(1), 1–43.
34. Schutt, A. (2006). Entwicklung suchraumeinschränkender Verfahren zur Constraint-basierten Lösung kumulativer Ressourcenplanungsprobleme. Master's thesis, Humboldt-Universität zu Berlin.
35. Schutt, A., Feydy, T., Stuckey, P. J., & Wallace, M. G. (2009). Why cumulative decomposition is not as bad as it sounds. In I. Gent (Ed.), *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming, LNCS* (Vol. 5732, pp. 746–761). Springer-Verlag.
36. Schutt, A., Wolf, A., & Schrader, G. (2006). Not-first and not-last detection for cumulative scheduling in $\mathcal{O}(n^3 \log n)$. In *Declarative programming for knowledge management, Lecture Notes in Computer Science* (Vol. 4369, pp. 66–80). Springer-Verlag. doi:10.1007/11963578. INAP 2005—16th international conference on applications of declarative programming and knowledge management.
37. Vilím, P. (2005). Computing explanations for the unary resource constraint. In *Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization problems, LNCS* (Vol. 3524, pp. 396–409). Springer-Verlag. doi:10.1007/11493853_29. http://kti.ms.mff.cuni.cz/~vilim/cpaior2005.pdf.

38. Vilím, P. (2009). Edge finding filtering algorithm for discrete cumulative resources in $\mathcal{O}(kn \log n)$. In *Principles and Practice of Constraint Programming—CP 2009, LNCS* (Vol. 5732, pp. 802–816). Springer-Verlag. doi:10.1007/978-3-642-04244-7_62.
39. Wolf, A., & Schrader, G. (2006). $\mathcal{O}(n \log n)$ overload checking for the cumulative constraint and its application. In *Declarative programming for knowledge management, Lecture Notes in Computer Science* (Vol. 4369, pp. 88–101). Springer-Verlag. doi:10.1007/11963578_8. INAP 2005—16th international conference on applications of declarative programming and knowledge management.