Why `cumulative` decomposition is not as bad as it sounds

Andreas Schutt[1], Thibaut Feydy[1], Peter J. Stuckey[1], and Mark G. Wallace[2]

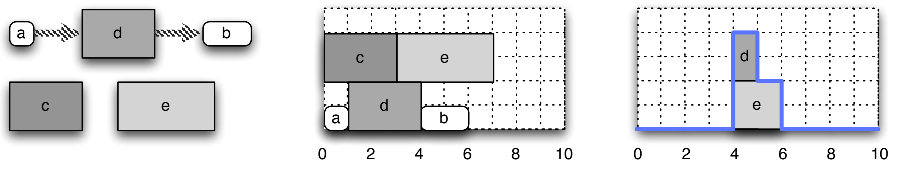
[1] National ICT Australia, Department of Computer Science & Software Engineering, The University of Melbourne, Australia
{aschutt,tfeydy,pjs}@csse.unimelb.edu.au
[2] School of Computer Science & Software Engineering, Monash University, Australia
mark.wallace@infotech.monash.edu.au

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 nogood learning and autonomous search capabilities. In this paper we show that using lazy clause generation where we model `cumulative` constraint by decomposition gives a very competitive implementation of cumulative resource problems. We are able to close a number of open problems from the well-established PSPlib benchmark library of resource-constrained project scheduling problems.

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 arises for example in the resource-constrained project scheduling problem Rcpsp, their variants, their extensions and their 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 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.

Example 1. Consider a simple resource scheduling problem. There are 5 tasks `a`, `b`, `c`, `d` and `e` to be scheduled to end before time 10. The tasks have respective durations 1, 2, 3, 3 and 4, each respective task requiring 1, 1, 2, 2 and 2 units of



(a) (b) (c)

Fig. 1. (a) A small cumulative resource problem, with 5 tasks to place in the 5x10 box, with task `a` before `d` before `b`, (b) a possible schedule, and (c) a profile under some further conditions.

resource, with a resource capacity of 5. Assume further that there are precedence constraints: task `a` must complete before task `d` begins, written `a` $\rightsquigarrow$ `d`, and similarly `d` $\rightsquigarrow$ `b`. Figure 1(a) shows the 5 tasks and precedences, while (b) shows a possible schedule, where the respective start times are: 0, 4, 0, 1, 3.

In 1993 Aggoun and Beldiceanu [2] 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 [16] and Baptiste and Le Pape [3].

The best known exact algorithm for solving Rcpsp is from Demeulemeester and Herroelen [6]. 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 comparison to Demeulemeester and Herroelen's specific nogoods 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.* [17] 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 of their behaviour. They show that this combination outperforms the best available constraint solvers on Open-Shop-Job problems which is a special case for Rcpsp.

Since 1993 little attention has been paid to decompositions of `cumulative` because decomposition cannot compete with the global propagator. 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 [9] where (in 2000) it is described as current work, and [18] which restricts attention to the `disjunctive` constraint (resource capacity 1).

In this paper we show how a decomposition based approach for solving complex scheduling problems can be competitive with state-of-the-art specialised methods from the CP and OR community. The G12 Constraint Programming Platform is used for implementation of decomposed cumulative constraint as a lazy clause generator. We evaluate our approach on RCPSP from the well-established and challenging benchmark library PSPLib [1].

2 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.

2.1 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 *tighter* than D_2, written $D_1 \sqsubseteq D_2$, if $D_1(v) \subseteq D_2(v)$ for all $v \in \mathcal{V}$. We use *range* notation: $[\,l \,..\, u\,]$ 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. $D \sqsubseteq D_{init}$.

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 $v \in vars(\theta)$.

A constraint c is a set of valuations over $vars(c)$ 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 $D \sqsubseteq D_{init}$: $f(D) \sqsubseteq D$ and $\{\theta \in D \mid \theta \in c\} = \{\theta \in f(D) \mid \theta \in c\}$. A *propagation solver* for a set of propagators F and current domain D, $solv(F, D)$, repeatedly applies all the propagators in F starting from domain D until there is no further change in resulting domain. $solv(F, D)$ is the weakest domain $D' \sqsubseteq D$ which is a fixpoint for all $f \in F$.

2.2 SAT Solving

DPLL SAT solvers can be understood as a form of propagation solver where variables are Boolean, and the only constraints are clauses C: $\vee_{l \in C} l$. The difference with an FD solver is that propagation engines are highly specialised and more importantly the reason for propagation is recorded, and on failure used to generate a nogood which explains the failure. This clause is added to the propagators to shortcircuit later search. It also helps direct backtracking to go above the cause of the failure.

2.3 Lazy Clause Generation

The lazy clause generation [17] 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 $D_{init}(x) = [\, l \,..\, u \,]$ 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.

Not every assignment of Boolean variables is consistent with the integer variable x, for example $\{[\![x = 3]\!], [\![x \leq 2]\!]\}$ 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 clauses $DOM(x)$ that encode $[\![x \leq d]\!] \rightarrow [\![x \leq d + 1]\!]$ and $[\![x = d]\!] \leftrightarrow ([\![x \leq d]\!] \wedge \neg[\![x \leq d - 1]\!])$. We let $DOM = \cup\{DOM(v) \mid v \in \mathcal{V}\}$.

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

Example 2. 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 $domain(A)(x_3) = [\, -1 \,..\, 10 \,]$. □

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 $domain(A) \sqsubseteq D$, then when clauses C that explain the propagation of f are added to the SAT database and unit propagation is performed, then the resulting assignment A' will be such that $domain(A') \sqsubseteq f(D)$.

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 [17]. The advantages over a normal FD solver 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.

3 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 [2] in 1993 is a constraint with Zinc [14] 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 also assume for simplicity 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`.

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

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

Example 4. Consider the cumulative resource problem defined in Example 1. This can be modelled by the cumulative constraint

$$\texttt{cumulative}\ ([s_{\mathtt{a}}, s_{\mathtt{b}}, s_{\mathtt{c}}, s_{\mathtt{d}}, s_{\mathtt{e}}], [1,2,3,3,4], [1,1,2,2,2], 5)$$

with precedence constraints $\mathtt{a} \rightsquigarrow \mathtt{d}$, $\mathtt{d} \rightsquigarrow \mathtt{b}$, modelled by $s_{\mathtt{a}} + 1 \leq s_{\mathtt{d}}$ and $s_{\mathtt{d}} + 3 \leq s_{\mathtt{b}}$. The propagator for the precedence constraints determines a domain D where $D(s_{\mathtt{a}}) = [\,0\,..\,3\,]$, $D(s_{\mathtt{b}}) = [\,4\,..\,8\,]$, $D(s_{\mathtt{c}}) = [\,0\,..\,7\,]$, $D(s_{\mathtt{d}}) = [\,1\,..\,5\,]$, $D(s_{\mathtt{e}}) = [\,0\,..\,6\,]$. The cumulative constraint does not determine any new information. If we add the constraints $s_{\mathtt{e}} \geq 2$, $s_{\mathtt{e}} \leq 4$, $s_{\mathtt{b}} \leq 7$, $s_{\mathtt{a}} \geq 1$, then precedence determines the domains $D(s_{\mathtt{a}}) = [\,1\,..\,3\,]$, $D(s_{\mathtt{b}}) = [\,4\,..\,6\,]$, $D(s_{\mathtt{c}}) = [\,0\,..\,7\,]$, $D(s_{\mathtt{d}}) = [\,2\,..\,4\,]$, $D(s_{\mathtt{e}}) = [\,2\,..\,4\,]$. We can determine that task `d` must use two resources between times 4 and 5, and task `e` must use two resources between times 4 and 6 (see Figure 1(c)). Hence task `c` cannot overlap these between times 4 and 5, and we can determine that $s_{\mathtt{c}} \neq 2$, $s_{\mathtt{c}} \neq 3$, $s_{\mathtt{c}} \neq 4$. If we restrict ourselves to bounds propagation then the cumulative constraint learns nothing. If we then add the constraint that $s_{\mathtt{c}} \geq 2$, then the bounds propagation on the cumulative constraint determines that $s_{\mathtt{c}} \geq 5$ and $D(s_{\mathtt{c}})$ becomes $[\,5\,..\,7\,]$.

Usually the `cumulative` constraint is implemented as a global propagator, since it can then take more information into account during propagation. In the remainder of this section we give two decompositions.

3.1 Time-Resource Decomposition

The time-resource decomposition (Time-RD) [2] 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 a 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} represents that task i is active at time t:

$$\begin{aligned}
\forall t \in [\,0\,..\,t_{max}-1\,], \forall i \in [\,1\,..\,n\,]: &\quad B_{it} \leftrightarrow [\![s[i] \leq t]\!] \wedge \neg [\![s[i] \leq t - d[i]]\!] \\
\forall t \in [\,0\,..\,t_{max}-1\,]: &\quad \sum_{i \in [\,1\,..\,n\,]} r[i] \cdot B_{it} \leq c
\end{aligned}$$

Note that since we are using lazy clause generation, the Booleans 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 $lb(s[i]) \leq t < ub(s[i]) + ub(d[i])$.

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

Note that if we have another cumulative constraint for a different resource on the same tasks then we can reuse the Boolean variables, and we just need to create t_{max} new sum constraints.

Example 5. Consider the problem of Example 4 after the addition of $s_{\mathtt{e}} \geq 2$, $s_{\mathtt{e}} \leq 4$, $s_{\mathtt{b}} \leq 7$, $s_{\mathtt{a}} \geq 1$, the decomposition determines that $B_{\mathtt{d}4}$ is true since $s_{\mathtt{d}} \leq 4$ and $\neg(s_{\mathtt{d}} \leq 4 - 4 = 0)$, similarly for $B_{\mathtt{e}4}$ and $B_{\mathtt{e}5}$. Using the sum constraint it determines that $B_{\mathtt{c}4}$ is false, and hence $\neg(s_{\mathtt{c}} \leq 4) \vee s_{\mathtt{c}} \leq 1$. This does not change any bounds. When we add that $s_{\mathtt{c}} \geq 2$ we determine that $\neg(s_{\mathtt{c}} \leq 4)$ or $s_{\mathtt{c}} \geq 5$.

We can expand the model to represent holes in the domains of start times. 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.

Example 6. Consider again the problem of Example 4, with the extended decomposition we also determine that $s_{\mathtt{c}} \neq 2$, $s_{\mathtt{c}} \neq 3$, $s_{\mathtt{c}} \neq 4$, so the resulting domain is $D(s_{\mathtt{c}}) = \{0, 1, 5, 6, 7\}$.

3.2 Task-Resource Decomposition

The Task-resource decomposition (Task-RD) is a relaxation of the Time-RD. 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 [7] 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]));
```

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.

$$\begin{aligned}
\forall j \in [\,1\,..\,n\,], \forall i \in [\,1\,..\,n\,] \setminus \{j\} : \quad && B_{ij} \leftrightarrow B^1_{ij} \wedge B^2_{ij} \\
&& B^1_{ij} \leftrightarrow s[i] \leq s[j] \\
&& B^2_{ij} \leftrightarrow s[j] < s[i] + d[i] \\
\forall j \in [\,1\,..\,n\,] : \quad && \sum_{i \in [\,1\,..\,n\,] \setminus \{j\}} r[i] \cdot B_{ij} \leq c - r[j]
\end{aligned}$$

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 $i, j \in [1 \mathinner{..} n]$ where $i < j$ in order to improve the propagation and the learning.

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

The size of this decomposition only depends on n whereas Time-RD depends on n and the number of points in the planning horizon t_{max}. At most $3n(n-1)$ Boolean variables, $3n(n-1)$ equivalence relations, n or relations, $3n$ 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 4 after the addition of $s_{\mathtt{e}} \geq 2$, $s_{\mathtt{e}} \leq 4$, $s_{\mathtt{b}} \leq 7$, $s_{\mathtt{a}} \geq 1$, after the precedence constraints are applied the decomposition learns $\neg B^2_{\mathtt{ad}}$, $\neg B^2_{\mathtt{db}}$ direct from precedence constraints and hence $\neg B_{\mathtt{ad}}$, $\neg B_{\mathtt{db}}$. From the start times it determines that $B^1_{\mathtt{ab}}$, $B^1_{\mathtt{db}}$, $B^1_{\mathtt{eb}}$, $\neg B^2_{\mathtt{ba}}$, $\neg B^2_{\mathtt{bd}}$, $\neg B^2_{\mathtt{be}}$, $\neg B_{\mathtt{ba}}$, $\neg B_{\mathtt{bd}}$, $\neg B_{\mathtt{be}}$. But nothing is determined from the sum constraints and no bounds changes are made by the cumulative. Adding $s_{\mathtt{c}} \geq 2$ does not change this. This illustrates the weaker propagation of the Task-RD.

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 Time-RD decomposition the Task-RD decomposition is stronger in its ability to relate to task information, but generates a weaker profile of resource usage, since no implicit profile is recorded.

3.3 Explanations

To see the advantage of decomposition in terms of explanations let us revisit Example 4. After the addition of $s_{\mathtt{e}} \geq 2$, $s_{\mathtt{e}} \leq 4$, $s_{\mathtt{b}} \leq 7$, $s_{\mathtt{a}} \geq 1$, together with $s_{\mathtt{c}} \geq 2$ a global (bounds consistent) cumulative constraint propagates that $s_{\mathtt{c}} \geq 5$. A minimal explanation of this is that $s_{\mathtt{e}} \geq 2 \wedge s_{\mathtt{e}} \leq 4 \wedge s_{\mathtt{d}} \geq 2 \wedge s_{\mathtt{d}} \leq 4 \wedge s_{\mathtt{c}} \geq 2 \rightarrow s_{\mathtt{c}} \geq 5$. This is recorded as the clause $[\![s_{\mathtt{e}} \leq 1]\!] \vee \neg[\![s_{\mathtt{e}} \leq 4]\!] \vee [\![s_{\mathtt{d}} \leq 1]\!] \vee [\![s_{\mathtt{d}} \leq 4]\!] \vee [\![s_{\mathtt{c}} \leq 1]\!] \vee \neg[\![s_{\mathtt{c}} \leq 4]\!]$.

Consider what happens in the Time-RD decomposition. After the addition of $s_{\mathtt{e}} \geq 2$, $s_{\mathtt{e}} \leq 4$, $s_{\mathtt{b}} \leq 7$, $s_{\mathtt{a}} \geq 1$, we learn $s_{\mathtt{d}} \leq 4 \wedge s_{\mathtt{d}} \geq 2 \rightarrow B_{\mathtt{d}4}$, $s_{\mathtt{e}} \leq 4 \wedge s_{\mathtt{e}} \geq 1 \rightarrow B_{\mathtt{e}4}$, $s_{\mathtt{e}} \leq 5 \wedge s_{\mathtt{e}} \geq 2 \rightarrow B_{\mathtt{e}5}$, and $B_{\mathtt{d}4} \wedge B_{\mathtt{d}5} \rightarrow \neg B_{\mathtt{c}4}$ With the addition of $s_{\mathtt{c}} \geq 2$ we learn that $s_{\mathtt{c}} \geq 2 \wedge \neg B_{\mathtt{c}4} \rightarrow s_{\mathtt{c}} \geq 5$. Each of the clauses is smaller and more reuseable. For example if we replace the constraint $s_{\mathtt{e}} \geq 2$ by $s_{\mathtt{e}} \geq 1$ then the same reasoning will apply. The crucial benefit of this is that nogoods are more reusable.

4 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 our decomposition on the well-known Rcpsp benchmark library PSPLib [1].

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 has a resource capacity $c[r]$.

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 following conditions

$$\begin{aligned} \forall i \rightsquigarrow j \in A : & \qquad s[i] + d[i] \leq s[j] \\ \forall t \in [\,0 \mathinner{..} t_{max} - 1\,], \forall k \in R : & \qquad \sum_{i \in T : s[i] \leq t < s[i]+d[i]} r[k, i] \leq c[r] \ , \end{aligned}$$

where t_{max} is the planning horizon. For our experiments we search for a schedule which minimises the makespan (i.e. latest end time). A basic Zinc model is given in Appendix C and at `http://www.cs.mu.oz.au/~pjs/rcpsp`.

In practice we share the Boolean variables generated inside the cumulative constraints as described in Section 3.1 (by common sub-expression elimination) and add redundant constraints as described in Section 3.2 when using the Task-RD 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 first solution found by labelling the smallest value of the start time variables in order. The initial domain of each variable $s[i]$ was determined as $D_{init}(s[i]) = [\,p[i] \mathinner{..} t_{max} - q[i]\,]$ 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 alternate search strategies.

4.1 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 [10, 8, 11].

Baptiste and Le Pape [3] 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 $ub(s[i])$. 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. 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, a constraint requiring a better solution is dynamically (globally) added during the search.

4.2 Search using Variable State Independent Decaying Sum

The SAT decision heuristic Variable State Independent Decaying Sum (VSIDS) [15] 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 [15] 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 optimization search.

Normally SAT solvers use dichotomic restart search for optimisation as the SAT solver itself does not have optimisation search built in, although in some cases it is possible to maintain the nogoods from the previous search. 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.

Restarting is shown to be beneficial in SAT solving (and CSP solving) in speeding up solution finding, and being more robust on hard problems. We also use VSIDS search with restarting, which we denote RESTART.[3]

[3] Note that restarting SGS search while possible is not attractive since the nogoods do not modify the search in most cases.

4.3 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 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.

5 Experiments

We carried out extensive experiments on RCPSP instances comparing our approach to decomposition without explanation, global `cumulative` propagators from SICStus and ECLiPSe, as well as a state-of-the-art exact solving algorithm [12]. Detailed results are available at `http://www.cs.mu.oz.au/~pjs/rcpsp`.

We use two suites of benchmarks. The library PSPLib [1] 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 [3].

The experiments were run on a X86-64 architecture running GNU/Linux and a 3.4 GHz processor. The code was written in G12 Constraint Programming Platform and compiled with the Mercury Compiler and grade hlc.gc.trseg. Each run was given a 10 minute limit.

5.1 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 three decompositions Time-RD (t), Task-RD (s), and an equivalent version to Task-RD on end times (e). The results are shown in Table 1. For J30 we show the number of problems solved (#svd), (cmpr(477)) the average solving time in seconds and number of choice points (#cp) on the 477 problems that all approaches solved, and (all(480)) average solving time in seconds and number of choice points on all 480 problems to find the best solution found.[4] Note that we shall use similar comparisons and notation in future tables. For BL we show the number of solved problems, (all(40)) average solving

[4] This means that for problems that time out this may be significantly smaller than the number of choice points explored before timeout.

Table 1. Results on J30 and BL instances

		J30					**BL**					
search	dec	#svd	cmpr(477)		all(480)		#svd	all(40)		#svd	cp(4000)	
			time	#cp	time	#cp		time	#cp		time	#cp
Sgs	s	477	3.25	3069	6.97	4114	**40**	4.18	9628	24	0.22	1261
	e	477	3.31	3054	7.04	4101	**40**	4.41	9443	24	0.19	1144
	t	**480**	1.36	2339	4.09	4230	**40**	1.40	5892	29	**0.05**	781
Vsids	s	**480**	1.82	2128	2.62	2984	**40**	1.24	4436	31	0.20	1115
	e	**480**	0.85	1504	1.45	2220	**40**	1.27	4104	30	0.20	1025
	t	**480**	0.43	1002	**0.54**	1271	**40**	0.30	2540	34	**0.05**	661
Restart	s	**480**	0.93	1504	1.73	2339	**40**	1.46	4597	31	0.23	1207
	e	**480**	0.82	1392	1.52	2153	**40**	2.61	5848	32	0.23	1177
	t	**480**	0.39	892	**0.54**	**1212**	**40**	0.17	1670	35	0.06	639
Hot Start	t	**480**	**0.34**	**782**	0.56	1223	**40**	**0.13**	**1456**	**36**	**0.05**	688
Hot Restart	t	**480**	0.42	892	0.59	1241	**40**	0.20	1850	35	0.07	733

time and number of choice points with 10 minute limit (on all 40 instances), as well as cp(4000) with a 4000 choice point limit.

Clearly the Time-RD decomposition is superior regardless of search, and the best search strategies are Restart and the hybrid ones.

The results on the BL instances show that approaches using Time-RD and Vsids could solve between 6 and 8 instances more than the base approach (FE) of Baptiste and Le Pape [3] within 4000 backtracking steps.[5] Their "left-shift/right-shift" approach could solve 40 instances in 30 minutes, with an average of 3634 steps and 39.4 seconds on a 200 MHz machine. All our approaches with Time-RD and Vsids find the optimal solution faster and in fewer backtracking steps (between a factor of 1.39 and 2.4).

Next we compare the Time-RD decomposition (Sgs+`t`) 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 Table 2. We can see that none of the other approaches compare to the lazy clause generation approach. The best is the sicstus cumulative with `global` flag. Clearly nogoods are very important to fathom search space.

While the Time-RD decomposition clearly outperforms Task-RD on these small examples, as the planning horizon grows at some point Task-RD should be better, since its model size is independent of the planning horizon. To investigate this we took 20 examples from J30 and multiplied the durations and planning horizon by 10 and 100. We compare the Time-RD decomposition versus the (`e`) end-time Task-RD decomposition (which is slightly better than start-time `s`).

[5] We count the number of choice points which is not smaller than the number of backtracking steps.

Table 2. Results of the FD solvers on the J30 and BL instances

		J30					BL				
	solver	#svd	cmpr(361)		all(480)		#svd	cmpr(6)		all(40)	
SICStus	default	417	0.24	268	89.00	13986	30	2.86	20865	213.59	489218
	global	411	0.43	263	96.85	6661	39	0.32	1265	19.19	10262
ECLiPSe	cumu	365	11.60	19529	158.30	42698	6	149.90	252462	532.31	123839
	ef	361	15.15	15438	161.32	22907	36	8.79	11265	117.21	89034
	ef3	362	13.37	12391	159.41	19186	37	7.17	7717	90.82	49114
G12	FD+`t`	403	1.93	5665	104.72	156598	30	2.23	34677	217.12	918287
	SGS+`t`	**480**	**0.02**	**75**	**4.09**	4230	**40**	**0.02**	**293**	**1.40**	5892

Table 3. Results on 20 modified instance from J30 instances

		SGS					VSIDS				
duration	dec	#svd	cmpr(12)		all(20)		#svd	cmpr(12)		all(20)	
1×	`e`	17	0.74	2383	152.49	67257	**20**	0.25	735	26.89	34563
	`t`	**20**	**0.44**	**1817**	**87.19**	72888	**20**	**0.11**	**404**	**6.59**	14405
10×	`e`	13	**4.25**	7493	212.75	47394	**20**	1.98	3971	117.27	68097
	`t`	**14**	4.74	**2516**	**201.27**	41081	**20**	**1.66**	**1622**	**94.09**	28250
100×	`e`	**13**	**22.03**	17620	**225.71**	35349	**14**	**10.52**	**6379**	**192.90**	24959
	`t`	**13**	55.78	**3017**	259.98	7175	**14**	22.42	9836	233.60	12618

The results are shown in Table 3. First we should note that simply increasing the durations makes the problems significantly more difficult for a decomposed cumulative. While the Time-RD decomposition is still just better than the Task-RD decomposition for the 10× extended examples, it is inferior for scheduling problems with very long durations.

5.2 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 approaches from the previous subsection: VSIDS + `t`, RESTART + `t`, HOT START + `t` and HOT RESTART + `t`. For this suite our solvers cannot solve all 480 instances within 10 minutes. The results are presented in the Table 4. For these examples we show the average distance of our best solution found from the best known solution from PSPLib (most of which are generated by specialised heuristic methods), as well as the usual time and number of choice points comparisons. Many of these are currently open problems. Our best approaches close 21 open instances. Clearly the hybrid search strategies are superior, although all of these approaches are quite competitive.

For the largest instances J90 and J120 we ran only HOT RESTART + `t`, since it is the most robust strategy. For J90 we can solve 396 of 480 instances, with an average solution distance of 7.6. The average for solved instances is 6.56s with 5077 #cp. We close 13 open instances in J90. For J120 we can solve 272 of 480 instances, with an average solution distance of 9.7, with average (on solved

Table 4. Results on J60 instances for Time-RD

solver	#svd	avg. dist.	cmpr(424)		all(480)	
VSIDS + t	424	4.5	5.77	6351	75.07	19781
RESTART + t	428	4.8	5.07	5010	69.70	24333
HOT START + t	**429**	9.3	**3.83**	**4111**	68.27	12072
HOT RESTART + t	**429**	**4.2**	4.66	4617	**68.25**	25810

Table 5. Comparison between Laborie's method and HOT RESTART + t

	J60			J90			J120		
1.4 GHz	45s	300s	1800s	45s	300s	1800s	45s	300s	1800s
Laborie	-	84.2	85.0	-	78.5	79.4	-	41.3	41.7
HOT RESTART + t	85.2	88.1	89.4	79.8	81.3	82.5	42.5	44.8	45.3
3.4 GHz	18s	120s	600s*	18s	120s	600s*	18s	120s	600s*

instances) times of 7.52s and 6136 #cp. We close 20 open instances in J120 (see Appendix A for more details).

We compare our best method HOT RESTART + t to the method by Laborie [12]. Their methods is the best published method so far on the J60, J90, and J120 instances.

Table 5 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: 3.4GHz vs. 1.4GHz. At the top of the table is the time cutoff for a 1.4GHz processor, and at the bottom the approximately equivalent cutoff times for a 3.4GHz machine. Note, that all * marked 3.4GHz times are much lower than the equivalent time for the 1.4GHz processor. Clearly this comparison can only be seen as indicative.

Our method clearly outperforms Laborie's method: for every class our method was able to solve more problems within 18s than they could solve in half an hour respectively on their machine. Interestingly, our solver could not solve six instances which were solved by others.

Finally we used HOT START + t 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 9 more problems and improved 76 lower bounds (see Appendix B).

6 Conclusion

We present a new approach solving RCPSP problems by modelling cumulative constraints by decomposition and using lazy clause generation. 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 63 open problems.

Acknowledgments. 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.

References

[1] PSPLib — project scheduling problem library. http://129.187.106.231/psplib/. 23.04.2009.

[2] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57–73, 1993.

[3] P. Baptiste and C. Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints*, 5(1-2):119–139, 2000.

[4] J. Carlier and E. Pinson. Jackson's pseudo-preemptive schedule and cumulative scheduling problems. *Discrete Applied Mathematics*, 145(1):80–94, 2004.

[5] Y. Caseau and F. Laburthe. Cumulative scheduling with task intervals. In *Procs. of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 363–377. MIT Press, 1996.

[6] E. L. Demeulemeester and W. S. Herroelen. New benchmark results for the resource-constrained project scheduling problem. *Management Science*, 43(11):1485–1492, 1997.

[7] A. O. El-Kholy. *Resource Feasibility in Planning.* PhD thesis, Imperial College, University of London, 1996.

[8] S. Hartmann and R. Kolisch. Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem. *EJOR*, 127(2):394–407, 2000.

[9] N. Jussien and V. Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of Techniques foR Implementing Constraint programming Systems*, pages 118–133, 2000.

[10] R. Kolisch. Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. *EJOR*, 90(2):320–333, 1996.

[11] R. Kolisch and S. Hartmann. Experimental investigation of heuristics for resource-constrained project scheduling: An update. *EJOR*, 174(1):23–37, 2006.

[12] P. Laborie. Complete MCS-based search: Application to resource constrained project scheduling. In L. P. Kaelbling and A. Saffiotti, editors, *Proceedings IJCAI 2005*, pages 181–186. Professional Book Center, 2005.

[13] O. Liess and P. Michelon. A constraint programming approach for the resource-constrained project scheduling problem. *Annals of Operations Research*, 157(1):25–36, 2008.

[14] K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. Garcia de la Banda, and M. G. Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.

[15] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, New York, NY, USA, 2001. ACM.

[16] W. P. M. Nuijten. *Time and Resource Constrained Scheduling.* PhD thesis, Eindhoven University of Technology, 1994.

[17] O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation = lazy clause generation. In *Procs. of the CP2007*, volume 4741 of *LNCS*, pages 544–558. Springer, 2007.

[18] P. Vilím. Computing explanations for the unary resource constraint. In *Procs of CPAIOR 2005*, volume 3524 of *LNCS*, pages 396–409. Springer, 2005.

A Closed instances

In the Table 6 we list all previously open instances (wrt. [1], [12], and [13]) with their optimal makespan which were closed by Hot Restart with Time-RD or by an other way. 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.

Table 6. Closed instances

J60	Instance	5_10[6]	9_2	9_4	14_1	14_10	17_8	21_9	25_3	25_9	30_5	30_7	30_10
	Makespan	81	82	87	61	72	85	89	113	99	76	86	86
	Instance	41_1	41_2	41_6	41_9	46_4	46_5	46_6	46_7	46_9	46_10		
	Makespan	122	113	134	131	74	91	90	78	69	88		
J90	Instance	5_1	5_2	21_4	21_6	21_9[7]	21_10	26_5	37_1	37_4	37_5	37_9	37_10[7]
	Makespan	78	93	106	106	121	109	85	110	123	126	123	123
	Instance	42_2	42_7	42_10									
	Makespan	102	87	90									
J120	Instance	1_8	1_10	2_2	8_3[8]	21_2	21_7	22_3	22_8	28_4	28_8[6]	28_9	28_10[7]
	Makespan	109	108	75	95	117	111	96	103	112	99	98	116
	Instance	29_4	41_2	41_9	42_5	42_8	48_1	48_5[6]	48_9	48_10	49_3	49_4[6]	49_5
	Makespan	80	141	121	120	113	100	110	113	111	96	96	89
	Instance	49_7	49_10[7]										
	Makespan	99	97										

In total 63 instances of 504 open instances could be closed where 54 instances by Hot Restart, 8 by the lower bound computation (see Appendix B) and 1 by the combination of both[8].

B New lower bounds

We tried to compute new lower bounds using Hot Start with Time-RD for the remaining open instances. These experiments were carried out on the same machine with the same time limit. We set the makespan to the best known lower bound, and tried to find a solution, if this failed we increased the makespan by one and resolved. If a solution was found this is the optimal, if we can prove

[6] Closed by lower bound computation with an optimal solution.

[7] Closed by lower bound computation by proof of the equality of lower and best known upper bound.

[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.

failure for a given makespan we have increased the lower bound. If the increased lower bound equals to the best known upper bound we prove the optimality of the upper bound and close the instance as well.

In totel this method closed 9 more instances (see Appendix A) and improved the lower bound by 76 instances of the remaining 441 open instances. The improved lower bounds are listed in Table 7.

Table 7. New lower bounds on all instances

J60	Instance	9_3	9_5	9_6	9_8	9_9	9_10	25_1	25_2	25_4	25_5	25_6	25_7
	LB	99	79	105	93	98	88	113	95	105	97	104	87
	Instance	25_8	25_10	30_2	41_3	41_5	41_10						
	LB	94	106	69	89	108	105						
J90	Instance	5_4	5_6	5_8	5_9	21_1	21_2	21_5	21_7	21_8	37_2	37_6	37_8
	LB	100	85	95	113	108	115	111	105	107	113	128	118
	Instance	41_3	41_7	46_4									
	LB	147	144	92									
J120	Instance	1_1	1_3	6_8	7_2	7_3	7_6	8_2	8_4	8_6	9_4	26_2	26_4
	LB	104	124	139	112	97	115	101	91	84	84	156	158
	Instance	26_8	26_10	27_2	27_5	27_7	27_10	28_7	29_3	34_8	42_1	46_1	46_2
	LB	165	174	109	105	118	110	107	95	86	106	168	183
	Instance	46_3	46_9	47_1	47_2	47_4	47_5	47_9	48_3	48_6	48_7	48_8	49_2
	LB	160	154	128	126	119	125	139	108	101	105	115	108
	Instance	50_4	53_3	53_4	53_9	54_7	54_10	60_2					
	LB	99	105	136	154	108	106	82					

C Zinc model of RCPSP

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
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] );
   % 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] );
   % Resource constraints
constraint
   forall ( k in Resources )
   ( cumulative(s, d, [ r[k,i] | i in Tasks], c[k]) );
   % Makespan constraints
constraint
   forall ( i in Tasks where suc[i] == {} )
   ( s[i] + d[i] <= makespan );
   % Objective function
solve minimize makespan;
```