# Realizing the Alternative Resources Constraint Problem with Single Resource Constraints

Armin Wolf and Hans Schlenker

Fraunhofer FIRST, Kekuléstraße 7, D-12489 Berlin, Germany
{Armin.Wolf|Hans.Schlenker}@first.fraunhofer.de

**Abstract.** Alternative resource constraint problems have to be solved in practical applications where several resources are available for the activities to be scheduled. In this paper, we present a modular approach to solve such problems which is based on single resource constraints. Furthermore, we present a new sweeping algorithm which performs some "global" overload checking for the alternative resource constraint problem. To our knowledge, this is the first presentation where "sweeping", a well-known technique in computational geometry, was used to perform this checking efficiently.
For a practical evaluation of our approach, we implemented and integrated it in our Java constraint engine `firstcs`. We compared our implementation with the more general `disjoint2` constraint in SICStus Prolog on some benchmark problems. These problems are random placement problems available online on the Internet.

## 1 Introduction

Often, in scheduling situations like course timetabling, medical tool use, or the allocation of railway tracks an activity can be scheduled on any one resource from a set of *alternative* resources. In all these cases it must be ensured that the selected resource is exclusively available during the execution of the activity.

In Constraint Programming, there are two main approaches to solve such alternative resource scheduling problems: either by an extension of some single resource constraints [1] or by abstraction and application of a more general non-overlapping rectangles constraint. Recent publications [3, 7] have shown that for both kinds of constraints some efficient pruning techniques based on "sweeping" [6] exist.

In this paper we combine both approaches: the extension and application of pruning algorithms developed for single resource constraints [7] and the development of a new sweeping algorithm for a non-overlapping rectangle constraint. This algorithm performs some overload checks for the early detection of some inconsistency after the application of the pruning algorithms.

## 2 The Alternative Resource Constraint Problem

Informally, the alternative resource constraint problem is the problem of finding allocation of non-interruptible tasks to be processed on one of its alternative

machines such that they are not overlapping on any machine. More formally, the problem is defined as follows:

**Definition 1 (Task).** *A* task *$t$ is a non-interruptible activity having a non-empty finite* set of potential start times $S_t \subset \mathbb{Z}$*, i.e. an integer set which is the domain of its variable start time. Furthermore, a task $t$ has a non-empty finite* set of possible durations $D_t \subset \mathbb{N}$*, i.e. a positive integer set which is the domain of its variable duration. Finally, a task $t$ has a non-empty finite* set of alternative resources $R_t \subset \mathbb{Z}$*, i.e. an integer set which is the domain of its variable resource.*

**Definition 2 (Alternative Resource Constraint Problem).** *Given a finite set of tasks $T = \{t_0, \ldots, t_n\}$ with at least two elements ($n > 0$). An* alternative resource constraint problem *is determined by such a set of tasks $T$. The problem is to find a* solution*, i.e. some start times $s(t_0) \in S_{t_0}, \ldots, s(t_n) \in S_{t_n}$, some durations $d(t_1) \in D_{t_1}, \ldots, d(t_n) \in D_{t_n}$ and some resources $r(t_1) \in R_{t_1}, \ldots, r(t_n) \in R_{t_n}$ such that for $1 \leq i < j \leq n$ it holds*

$$s(t_i) + d(t_i) \leq s(t_j) \vee s(t_j) + d(t_j) \leq s(t_i) \vee r(t_i) \neq r(t_j) \ .$$

*An alternative resource constraint problem is* solvable *if there is such a solution and* unsolvable*, otherwise.*[1]

Assuming that the durations are fixed and the (average) size of all sets of potential start times is $m$ and of all sets of alternative resources is $k$, the determination of an solution has in general an exponential time complexity of $O(m^n k^n)$. To reduce this complexity, in Constraint Programming (CP) *propagation* is used: An iteration over efficient algorithms pruning the variables' domains such that some – ideally all – values are eliminated not belonging to any solution.

Our aim is the development and implementation of such algorithms for alternative resource constraint problems. Keeping in mind that an alternative resource constraint problem corresponds to a conjunction of single resource constraint problems if the resource domains are singular, i.e. if

$$|R_{t_1}| = \cdots = |R_{t_n}| = 1$$

holds, our work focuses on an generalisation of our previous work performed for *non-preemptive one-machine constraint problems* (cf. [7] following the suggestions in [1]).

## 3 Forbidden Regions

Given a single resource constraint problem determined by a set of tasks $T$. A *forbidden region* of a task $t \in T$ to be scheduled on a resource $r$ is an integer interval $I$ such that for any start time $s(t) \in I$ it is impossible to schedule another task $u \in T \setminus \{t\}$ on the same resource $r$ either before or after the task $t$ (see Figure 1).

[1] Empty or singleton sets of tasks determine trivial problems.

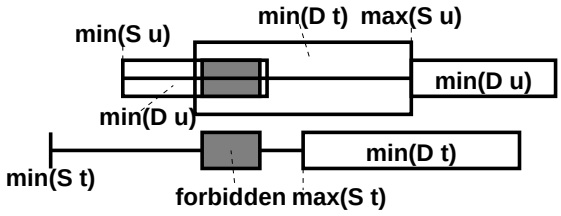


**Fig. 1.** A forbidden region of a task $t$ with respect to another task $u$.

Assuming that all tasks in $T$ must be scheduled on the same resource $r$, the application of the pruning rule

$$\begin{aligned} &\forall t \in T\, \forall u \in T \setminus \{t\} \\ &\quad : \ \max(S_u) - \min(D_t) + 1 \leq \min(S_u) + \min(D_u) - 1 \\ &\quad \Rightarrow S'_t := S_t \setminus [\max(S_u) - \min(D_t) + 1, \min(S_u) + \min(D_u) - 1] \end{aligned} \tag{1}$$

determines the *forbidden regions* of each task $t \in T$ locally with respect to another task $u \in T \setminus \{t\}$. The updating of the start times of the task $t$ (primed to notify the possible change) will prune the search space correctly (cf. [7] for the correctness poof.)

Considering an alternative resource constraint problem, this means that *after* an allocation of all tasks to their resources this pruning rule and all the other rules presented [1, 7] are applicable. However, any pruning and consistency checks must be delayed until at least all the tasks to one single resource are allocated. This seems to be rather late and contradicts the principle of *early pruning* in CP. Therefore, we investigated in a generalisation of the pruning rule (1) for alternative resource constraint problems. The result of this investigation is presented in the following:

**Definition 3.** *Given an alternative resource constraint problem determined by a set of tasks $T$. Then, for each task $t \in T$ and every resource $r \in R_t$, we define:*

- *a non-empty, finite* set of alternative start times $S_t^r := S_t$,
- *a non-empty, finite* set of alternative durations $D_t^r := D_t \cup \{0\}$.

*We also store the initial alternative resources for each task $t \in T$: $M_t := R_t$.*

Now the pruning rule (1) will be generalised in two directions. In both cases, a pair of two different tasks $t$ and $u$ are considered which may be allocated to the same resource $r \in R_t \cap R_u$:

1. If – under the assumption that the task $t$ is allocated to the resource $r$ – the forbidden region of the task $t$ with respect to the task $u$ subsumes its domain of potential start times, then the task $t$ must be scheduled on another resource, i.e. its duration on the resource $r$ becomes zero:

$$\forall t \in T_r\, \forall u \in T_r \setminus \{t\} \tag{2}$$

$$
\begin{aligned}
&: \ \max(S^r_u) - \min^\star(D^r_t \setminus \{0\}) + 1 \leq \min(S^r_t) \\
&\wedge \ \max(S^r_t) \leq \min(S^r_u) + \min(D^r_u) - 1 \\
&\Rightarrow {D'}^r_t := D^r_t \cap \{0\} \ ,
\end{aligned}
$$

where $T_r := \{t \in T \mid r \in R_t\}$ and for all $M \subseteq \mathbb{N}$ it holds

$$
\min^\star(M) := \begin{cases} 0 & \text{if } M = \emptyset, \\ \min(M) & \text{otherwise.} \end{cases}
$$

2. If – under the assumption that the task $t$ is allocated to the resource $r$ – the forbidden region of the task $t$ with respect to the task $u$ is not a superset of the domain of start times, then the task $t$ cannot be scheduled at start times in this forbidden region on the resource $r$:

$$
\begin{aligned}
&\forall t \in T_r \, \forall u \in T_r \setminus \{t\} \\
&\quad : \ \max(S^r_u) - \min(D^r_t \setminus \{0\}) + 1 \leq \min(S^r_u) + \min(D^r_u) - 1 \\
&\quad \wedge \ (\max(S^r_u) - \min(D^r_t \setminus \{0\}) + 1 > \min(S^r_t) \\
&\qquad \vee \max(S^r_t) > \min(S^r_u) + \min(D^r_u) - 1) \\
&\quad \Rightarrow {S'}^r_t := S^r_t \setminus [\max(S^r_u) - \min(D^r_t \setminus \{0\}) + 1, \min(S^r_u) + \min(D^r_u) - 1] \ ,
\end{aligned} \tag{3}
$$

where $T_r := \{t \in T \mid r \in R_t\}$.

The replacement the original pruning rule (1) by the these two rules for each single resource $r \in \bigcup_{t \in T} R_t$ will perform some additional pruning for the alternative resource constraint problem. Furthermore, these pruning are still correct, i.e. no solutions are lost: The second extension is a specialisation of the original rule; whenever the start times are pruned they will be pruned by the original rule, too. For the first extension we have to distinguish two cases: (1) $D^r_t = \{0\}$ and (2) $D^r_t \neq \{0\}$.

In the first case, nothing changes. Consequently, no solution will be lost.

In the second case, we further have to distinguish two sub-cases: (2.a) $0 \in D^r_t$ and (2.b) $0 \notin D^r_t$. In case (2.a), $D^r_t$ will be pruned to $\{0\}$ by the application of the extended rule (2) and $S^r_t$ will be pruned to the empty set by the original rule (1). In case (2.b) $D^r_t$ will be pruned to the empty set by the application of the extended rule while $S^r_t$ will also be pruned to the empty set by the original rule. In either case, there is no solution where the task $t$ is allocated to the resource $r$, especially because $0 \notin D_t$.

However, for further pruning the changes with respect to each single resource must be propagated to the original domains and from one resource to another. Especially, if the duration of task $t$ and a resource $r$ is set to zero (cf. pruning rule (2)), the task's potential resources must be updated:

$$
\forall t \in T \, \forall r \in R_t \ : \ D^r_t := \{0\} \Rightarrow {R'}_t := R_t \setminus \{r\} \ . \tag{4}
$$

If the possible durations of a task $t$ are restricted from "outside", e.g. during the search, these changes must be propagated to the duration sets of each single resource:

$$
\forall t \in T \, \forall r \in R_t : {D'}^r_t := D^r_t \cap (D_t \cup \{0\}) \ . \tag{5}
$$

If the potential start times are restricted from "outside", e.g. during the search, and/or by application of the pruning rule (3), these changes have to be propagated to the single resources:

$$\forall t \in T\, \forall r \in R_t \;:\; S_t^r \cap S_t = \emptyset \Rightarrow R'_t := R_t \setminus \{r\} \wedge D_t^r := D_t^r \cap \{0\} \;\;,$$
$$\forall t \in T\, \forall r \in R_t \;:\; S_t^r \cap S_t \neq \emptyset \Rightarrow S'^r_t := S_t^r \cap S_t \;\;,$$
$$\forall t \in T \;:\; S'_t := S_t \cap (\bigcup_{r \in R_t} S_t^r) \;\;.$$

In the first case, there is no potential start time left for a task $t$ and a resource $r$. Consequently, it will be impossible to allocate the task $t$ to the resource $r$. Thus, $r$ will be removed from the set of alternative resources. In the second case, the restriction of the start times is propagated to the start times on the alternative resources. Last but not least, the restrictions of the start times which are valid for all alternative resources are propagated to the potential start times.

The application of these three rules in the given order, supersedes an iteration over these rule for the computation of a local fix-point.

If the resource of a task $t$ is determined – from "outside", e.g. during the search, and/or by application of the pruning rules (2) and (4) – the possible duration for this resource must be positive and zero for all the other resources, i.e. the initial alternatives:

$$\forall t \in T \;:\; R_t = \{r\} \Rightarrow D'^r_t := D_t^r \setminus \{0\} \wedge$$
$$\forall s \in M_t \setminus \{r\} \;:\; D'^s_t := D_t^s \cap \{0\} \;\;.$$

## 4 Global Overload Checking

The application of several single resource constraints for the realisation of the alternative resource constraint benefits from overload checking performed "locally" for each resource (cf. [7]). However, these checks are weak as long as all activities are allocated to the resources: the minimal duration of an activity is zero on all its potential resources until the allocation is determined (see Section 3).

A necessary, global condition for the solubility of an alternative constraint problem determined by a set of tasks $T = \{t_0, \ldots, t_n\}$ is that each non-empty set of tasks $M \subseteq T$ is not overloaded, i.e. the occupied area is not greater than the available area:

$$\sum_{t \in M} \min(D_t) \times 1 \quad \leq$$
$$(\max_{t \in M}(\max(S_t) + \min(D_t)) - \min_{t \in M}(\min(S_t))) \times (\max_{t \in M}(R_t) + 1 - \min_{t \in M}(\min(R_t))) \;\;.$$

The naive overload checking of all $2^{n+1} - 1$ non-empty subsets of $T$ is not practical. Thus, we suppose to consider the set of at most $\sum_{l=0}^{n} l \times \sum_{m=0}^{n} m = \frac{(n+2)^2(n+1)^2}{4}$ *task rectangles*, i.e. the set of tasks defined by the cartesian product

of two integer intervals whose bounds are determined by some tasks:

$$\begin{aligned}&[llc(i,j)_X, urc(h,k)_X] \times [llc(i,j)_Y, urc(h,k)_Y]\\ &:= \{t \in T \mid llc(i,j)_X \leq \min(S_t) \wedge \max(S_t) + \min(D_t) \leq urc(h,k)_X\\ &\qquad \wedge\ llc(i,j)_Y \leq \min(R_t) \wedge \max(R_t) + 1 \leq urc(h,k)_Y\}\end{aligned}$$

where the *lower left corners* (the *llcs*) and the *upper right corners* (the *urcs*) are defined as follows:

$$\begin{aligned}llc(i,j)_X &:= \min(\min(S_{t_i}), \min(S_{t_j}))\\ llc(i,j)_Y &:= \min(\min(R_{t_i}), \min(R_{t_j}))\\ urc(h,k)_X &:= \max(\max(S_{t_h}) + \min(D_{t_h}), \max(S_{t_k}) + \min(D_{t_k}))\\ urc(h,k)_Y &:= \max(\max(R_{t_k}) + 1, \max(R_{t_k}) + 1)\end{aligned}$$

for $0 \leq i \leq j \leq n$ and $0 \leq h \leq k \leq n$ (see Figure 2).

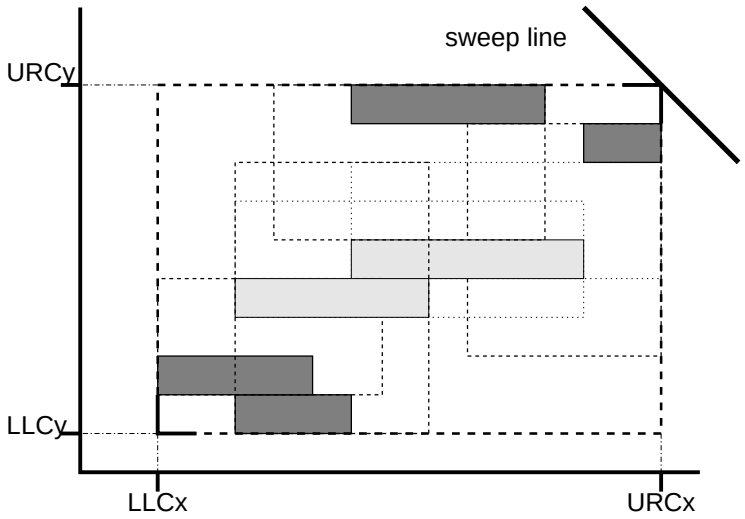


**Fig. 2.** Determining a possibly non-empty task rectangle.

To perform overload checking we use "sweeping" originated and used widely in computational geometry [6]. The recent publication [4] has shown that sweeping is also an efficient pruning technique when adapted and applied to finite domain constraint solving problems, especially for non-overlapping rectangles constraint problems.

While sweeping over task rectangles, it is assumed that the tasks in $T$ are numbered $t_0, \ldots, t_n$ according to an ascending sorting with respect to the order relation

$$\begin{aligned}t \preceq u :\Leftrightarrow\ &(\min(S_t) + \min(R_t) < \min(S_u) + \min(R_u))\\ &\vee\ (\min(S_t) + \min(R_t) = \min(S_u) + \min(R_u) \wedge \min(R_t) \leq \min(R_u))\end{aligned}$$

such that $t_0 \preceq \cdots \preceq t_n$ holds. Furthermore, we assume that all the corners (llcs and urcs) are also numbered $c_0, \ldots, c_m$ according to an ascending sorting with respect to the order relation

$$\begin{aligned} c \preceq d :\Leftrightarrow\ & (c_X + c_Y < d_X + d_Y) \vee (c_X + c_Y = d_X + d_Y \\ & \wedge ((c, d \text{ are both either llcs or urcs } \wedge c_Y \leq d_Y) \\ & \vee (c \text{ is an urc } \wedge d \text{ is an llc})) \end{aligned}$$

such that $c_0 \preceq \cdots \preceq c_m$ holds. Then, we are sweeping forward, i.e. in ascending order, over the sorted corners:

$\Diamond$ If the next event is $c_j$ $(0 \leq j \leq m)$ is an llc then

- append $c_j$ at the end of the sweep line.

$\Diamond$ If the next event $c_j$ $(0 \leq j \leq n)$ is an urc then

- let $p := 0$ and
- iterate forward over the llc $c_{i_0}, \ldots, c_{i_k}$ in the sweep line – for $l = 0, \ldots, k$:
  - if $(c_{i_l Y} > c_{j Y})$ then stop this iteration over the llcs; continue sweeping.
  - let $A := 0$ and $A_{i_l,j} := (c_{j X} - c_{i_l X}) \times (c_{j X} - c_{i_l X})$
  - while $((\min(S_{t_p}) + \min(R_{t_p}) < c_{i_l X} + c_{i_l Y})$
    $\vee\ (\min(S_{t_p}) + \min(R_{t_p}) = c_{i_l X} + c_{i_l Y} \wedge \min(R_{t_p}) < c_{i_l Y}))$
    - let $p := p + 1$.
  - iterate forward over the tasks $t_p, \ldots, t_n$ – for $m = p, \ldots, n$:
    - if $(\min(S_{t_m}) + \min(R_{t_m}) > c_{j X} + c_{j Y})$ then
      stop this iteration over the tasks; continue iteration over the llcs.
    - if $(\min(S_{t_m}) \geq c_{i_l X} \wedge \min(R_{t_m}) \geq c_{i_l Y}$
      $\wedge\ \max(S_{t_m}) + \min(D_{t_m}) \leq c_{j X} \wedge \max(R_{t_m}) + 1 \leq c_{j Y})$
      then let $A := A + (\min(D_{t_m} \times 1)$
    - if $A > A_{i_l,j}$ then there is no feasible schedule; exit.


(a) llc iteration (b) task iteration

**Fig. 3.** Breaking conditions for the iterations.

The algorithm works as follows: During the forward iteration over the sweep line all llc are considered which might define a possibly non-empty rectangle with respect to the current urc $c_j$ (cf. Figure 2). The chosen order relation '$\preceq$'

on corners ensures that all rectangles with a positive area $A_{i_l,j}$ are considered. Figure 3 (a) gives some evidence for stopping the iteration over the llcs. Then, the occupied area in the rectangle is determined. Therefore, some tasks that are not contained in the rectangle – they are are either "left" or "below" – are skipped. Again, the chosen order relations '$\preceq$' on tasks and corners as well as their compatibility with each other ensure that only irrelevant tasks are skipped. For all the other tasks which are contained the occupied area is accumulated. Figure 3 (b) gives some evidence for stopping the iteration over the tasks. An overload is detected if the total occupied area $A$ becomes greater than the available area $A_{i_l,j}$: Then, the algorithm stops because there is no feasible solution of the given alternative resource constraint problem.

## 5 Implementation

The extensions presented in Section 3 as well as the sweeping algorithm introduced in Section 4 are implemented in Java: Both implementations are integrated in our pure Java constraint engine, called `firstcs` [5]. Concretely, the forbidden region pruning algorithm in the existing single `Resource` constraint implementation was generalised. This algorithm is also realised as a sweeping algorithm (see [7] for details).

Then, we implemented an `AlternativeResource` constraint which generates for each resource $r \in \bigcup_{t \in T} R_t$ a single `Resource` constraint which performs pruning for each single resource constraint problem determined by the task set $T_r$. Thus, pruning for the alternative resource constraint problem benefits from additional pruning rules, like edge finding and not-first/not-last detection also implemented in the `Resource` constraint (cf. [7] for details).

Additionally, we realized in the `AlternativeResource` constraint the pruning rules presented in Section 3 performing propagation from and to the original domains and between the generated single `Resource` constraints. This implementation iterates over these rules until a local fix-point is computed, i.e. any application of the presented rules and algorithms will neither restrict the potential start times, the possible durations nor the alternative resources.

Finally, we implemented the sweeping algorithm for overload checking. This algorithm is applied after the computation of the fix-point because it will not change any domains.

## 6 Empirical Examinations

For empirical examinations, we used the *random placement problems (RPP)*, which are online available at `http://www.fi.muni.cz/~hanka/rpp/`. All the problem instances consist of 200 activities of randomly generated durations. Their potential start times as well as their alternative resources are randomly restricted to some finite integer intervals. For any of these instances the starting times and resources for the 200 activities must be determined such that the resources are exclusively available during the execution of these activities.

From a practical point of view, these RPP instances correspond to simple course timetabling problems [2]: each activity corresponds to a course and its alternative resources to the adequate classrooms, i.e. of sufficient capacity and with the needed equipment. We used the RPP instances to test our `AlternativeResource` constraint presented in Section 5. Furthermore, we compared our implementation based on the Java Standard Edition, version 1.4.0-03 against the more general `disjoint2` constraint in SICStus Prolog, version 3.11.0. The experiments were performed under Microsoft Windows XP on a PC Pentium 4, 2.8 GHz with 1 GByte RAM.

In both implementations we used standard labelling (simple depth-first search) to determine the start times and the resources: The activities were considered in their given order (activity 1 first, activity 200 last). During the search, for each activity the start time was assigned after selecting the resource. In both cases, the smallest available, not yet tried value was selected.

Our experiments have shown that all the instance with *filled area ratio* of 80% and 85% are solvable. Both implementations found their first solutions backtrack-free without any global propagation [2]: 400 assignments – 200 resources plus 200 start times. For all these problems, the SICStus Prolog implementation required on average no measurable time for constraint generation and initial propagation, our implementation required on average 150 milliseconds. The labelling process for one of these problems which triggers some further propagation took on average 50 milliseconds in SICStus Prolog and about 2200 milliseconds in our Java implementation.

Additional experiments on the instances with a *filled area ratio* of 100% have shown that all instances which are backtrack-free solvable using the SICStus Prolog implementation are backtrack-free solvable with our Java implementation, except the instance `100/gen38` which requires 5 backtracking steps in our Java implementation. Furthermore, all the instances which were detected to be unsolvable using SICStus Prolog are also detected by the use of our Java implementation. In any case an inconsistency was detected during the initial "global" propagation: SICStus Prolog's `disjoint2` with its `global` option switched on and our implementation with its global overload checking described in Section 4. In these unsolvable cases, SICStus Prolog used on average 15 milliseconds to detect an inconsistency, our Java implementation used on average 150 milliseconds. However, our Java implementation detects further unsolvable instances with *filled area ratio* of 100% during the initial "global" propagation: `100/gen6`, `100/gen13`, `100/gen32`, and `100/gen50`. It took only 230 milliseconds on average per instance to detect an overload.

## 7 Conclusion and Future Work

In this paper, we presented a modular approach for the alternative resource constraint problem based on single resource constraints. Furthermore, we presented

[2] In SICStus Prolog the `global` option was switch off as well as the global overload checking (see Section 4) in our Java implementation.

a new sweeping algorithm which performs overload checking for the alternative resource constraint problem. However, a proof that the consideration of task rectangles is sufficient as well as a proof that this algorithm performs correct is not yet given. This will the topic of future theoretical work.

Obviously, the presented "global" overload checking in Section 4 is applicable to non-overlapping rectangles problems, where the rectangles might have heights greater than one. Thus, our future work also focuses on the combination of our sweeping algorithm with to one presented in [3]) yielding better pruning for non-overlapping rectangles problems.

Last but not least, our implementation of the pruning rules and algorithm is successfully applied to some online available benchmark placement problems yielding some encouraging results: Compared to SICStus Prolog, our implementation's runtime is reasonable well, its pruning performance is comparable and even better for the unsolvable placement instances.

## References


1. Philippe Baptiste, Claude le Pape, and Wim Nuijten. *Constraint-Based Scheduling.* Number 39 in International Series in Operations Research & Management Science. Kluwer Academic Publishers, 2001.
2. Roman Barták, Tomáš Müller, and Hana Rudová. Minimal perturbation problem - a formal view. In *Proceedings of the Joint Annual Workshop of the ERCIM Working Group on Constraints and the CoLogNET area on Constraint and Logic Programming*, MTA SZTAKI, Budapest, Hungary, 30 June – 2 July 2003.
3. Nicolas Beldiceanu and Mats Carlsson. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In Toby Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming - CP2001*, number 2239 in Lecture Notes in Computer Science, pages 377–391. Springer Verlag, 2001.
4. Nicolas Beldiceanu and Mats Carlsson. A new multi-resource cumulatives constraint with negative heights. In Pascal van Hentenryck, editor, *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming - CP2002*, number 2470 in Lecture Notes in Computer Science, pages 63–79. Springer Verlag, 2002.
5. Matthias Hoche, Henry Müller, Hans Schlenker, and Armin Wolf. firstcs - A Pure Java Constraint Programming Engine. In Michael Hanus, Petra Hofstedt, and Armin Wolf, editors, *2nd International Workshop on Multiparadigm Constraint Programming Languages – MultiCPL'03*, 29th September 2003. Online available at `uebb.cs.tu-berlin.de/MultiCPL03/Proceedings.MultiCPL03.RCoRP03.pdf`.
6. Franco P. Preparata and Michael Ian Shamos. *Computational Geometry, An Introduction.* Texts and Monographs in Computer Science. Springer Verlag, 1985.
7. Armin Wolf. Pruning while sweeping over task intervals. In Francesca Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming – CP 2003*, number 2833 in Lecture Notes in Computer Science, pages 739–753, Kinsale, County Cork, Ireland, 30th September – 3rd October 2003. Springer Verlag.