FP6-034691

Net-WMS

SPECIFIC TARGETED RESEARCH OR INNOVATION PROJECT

Networked Businesses

D4.2 – A prototype handling geometrical constraints and placement rules

Due date of deliverable: 15 July 2008 (M21)

Actual submission date: 15 July 2008

Start date of project: 1 September 2006 Duration: 36 months

Organisation name of lead contractor for this deliverable: EMN (ARMINES) Version 1

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

Dissemination Level : PU

COVER AND CONTROL PAGE OF DOCUMENT	
Project Acronym:	**Net-WMS**
Project Full Name:	Towards integrating Virtual Reality and optimisation techniques in a new generation of **Net**worked businesses in **W**arehouse **M**anagement **S**ystems under constraints
Document id:	D4.2
Document name:	A prototype handling geometrical constraints and placement rules
Document type (PU, INT, RE, CO)	PU
Version:	1
Submission date:	15-07-2008
Authors: **Organisations:** **Emails:**	Abder Aggoun[2], Nicolas Beldiceanu[1], Mats Carlsson[3], Rida Sadek[1], Mohammed Sbihi[1] Armines[1], KLS-Optim[2], SICS[3] abder.aggoun@klsoptim.com {Nicolas.Beldiceanu,Rida.Sadek,Mohammed.Sbihi}@emn.fr matsc@sics.se

Document type PU = public, INT = internal, RE = restricted, CO = confidential

ABSTRACT:

This document describes the contribution to the software deliverable D4.2 regarding the development of the geometrical kernel *geost*. The *geost* constraint was integrated within the constraint programming platform of EMN (i.e., the CHOCO library), as well as in the constraint programming platform of SICS (i.e., SICStus Prolog).

KEYWORD LIST:

software, specification, integration, prototype, geometric constraint

MODIFICATION CONTROL			
Version	Date	Status	Author
0	02-06-2008	Initial	N.Beldiceanu
1	07-07-2008	Draft	N.Beldiceanu
2	15-07-2008	Final	N.Beldiceanu

Deliverable manager

- N.Beldiceanu, EMN (ARMINES)

List of Contributors

- Mats Carlson, SICS
- Abder Aggoun, KLS-OPTIM
- Rida Sadek, EMN (ARMINES)
- Mohammed Sbihi EMN (ARMINES)

List of Evaluators

- François Fages, INRIA
- Abder Aggoun, KLS-OPTIM

TABLE OF CONTENT

SUMMARY

This document describes the contribution to the software deliverable D4.2 regarding the developpement of the geometrical kernel *geost*. The *geost* constraint was integrated within the constraint programming platform of EMN (i.e., the CHOCO library), as well as in the constraint programming platform of SICS (i.e., SICStus Prolog).

Both versions can be dowloaded from the web :

CHOCO is available from
http://choco.emn.fr

while the SICStus version containing *geost* is available from
http://www.sics.se/sicstus/products4/sicstus/4.0.2-NETWMS-2/binaries/x86-linux-glibc2.3/sp-4.0.2-NETWMS-2-x86-linux-glibc2.3.tar.gz

The document is composed of four parts that respectively correspond to :

(Part A) An updated version of the specification of the geometrical kernel that integrates the **greedy mode** of the geometrical kernel. It extends the SICS report T2007-08 (A Generic Geometrical Constraint Kernel in Space and Time for Handling Polymorphic k-Dimensional Objects).

(Part B) The CHOCO documentation of the *geost* constraint.

(Part C) The SICStus documentation of the *geost* constraint.

(Part D) An example of exploitation of the *geost* constraint by KLS-Optim.

Part A: Geometrical kernel and greddy mode

Armines & SICS

A Generic Geometrical Constraint Kernel in Space and Time for Handling Polymorphic k-Dimensional Objects

Nicolas Beldiceanu[1], Mats Carlsson[2], Rida Sadek[1], and Mohammed Sbihi[1]

[1] École des Mines de Nantes, LINA FRE CNRS 2729, FR-44307 Nantes, France
{Nicolas.Beldiceanu,Emmanuel.Poder,Rida.Sadek}@emn.fr
[2] SICS, P.O. Box 1263, SE-164 29 Kista, Sweden
Mats.Carlsson@sics.se

Abstract. This article introduces a generic geometrical constraint kernel for handling the location in space and time of polymorphic k-dimensional objects subject to various geometrical and time constraints. It first describes a reduced set of standard primitives one has to provide for plugging any new geometrical/temporal constraint. Based on these primitives, it develops a generic k-dimensional lexicographic sweep-point algorithm for filtering the attributes of an object (i.e., the coordinates of its origin as well as its start and end in time) according to all constraints where the object occurs. Experiments are provided in the context of *inclusion* and *non-overlapping* constraints in dimensions 2, 3 and 4, both for simple shapes (i.e., rectangles, parallelepipeds) as well as for more complex shapes.

1 Introduction

This article introduces a global constraint $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{C})$ for handling in a generic way a variety of geometrical constraints $\mathcal{C}$ in space and time between polymorphic k-dimensional objects $\mathcal{O}$ ($k \in \mathbb{N}^+$), each of which taking a shape among a set of shapes $\mathcal{S}$ during a given time interval and at a given position in space.

Each shape from $\mathcal{S}$ is defined as a finite set of shifted boxes, where each shifted box is described by a box in a k-dimensional space at the given offset with the given sizes. More precisely a *shifted box* $s = \mathrm{shape}(sid, t[], l[]) \in \mathcal{S}$ is an entity defined by its shape id $s.sid$, shift offset $s.t[d]$, $0 \leq d < k$, and sizes $s.l[d]$ ($s.l[d] > 0, 0 \leq d < k$). All attributes of a shifted box are integer values. Then, a *shape* is a collection of shifted boxes sharing all the same shape id. [1] Each *object* $o = \mathrm{object}(id, sid, x[], start, duration, end)$ from $\mathcal{O}$ is an entity defined by its unique object id $o.id$ (an integer), shape id $o.sid$, origin $o.x[d]$, $0 \leq d < k$, start in time $o.start$, duration in time $o.duration$ ($o.duration \geq 0$) and end in time $o.end$. [2] All attributes sid, $x[0], x[1], \ldots, x[k-1]$, $start$, $duration$

[1] Note that the shifted boxes associated with a given shape may or may not overlap. This sometimes allows a drastic reduction in the number of shifted boxes needed to describe a shape.

[2] A first reason why the time dimension is treated specially comes from the fact that the $duration$ attribute may not fixed, which is actually not the case for the sizes of a shifted box. A second reason to distinguish the time dimension from the geometrical dimensions is that all geometrical constraints only apply on objects that intersect in time.

end correspond to domain variables.[3] Typical constraints from the list of constraints $\mathcal{C}$ are for instance the fact that a given subset of objects from $\mathcal{O}$ do not pairwise overlap or that they are all included within a given bounding box. Constraints of the list of constraints $\mathcal{C}$ have always two first arguments $\mathcal{A}_i$ and $\mathcal{O}_i$ (followed by possibly some additional arguments) which respectively specify:

- A list of dimensions (integers between 0 and $k-1$), or attributes of the objects of $\mathcal{O}$, or attributes of the shifted boxes of $\mathcal{S}$ the constraint considers.
- A list of identifiers of the objects to which the constraint applies.

Example 1. Assume we have a 3-dimensional placement problem involving a set of parallelepipeds $\mathcal{P}$ and one subset $\mathcal{P}'$ of $\mathcal{P}$, where we want to express the fact that (1) all parallelepipeds of $\mathcal{P}$ should not overlap, and (2) no parallelepipeds of $\mathcal{P}'$ should be piled. We have a placement problem where $k = 3$. Constraints (1) and (2) respectively correspond to $non\text{-}overlapping([0,1,2],\mathcal{P})$ and to $non\text{-}overlapping([0,1],\mathcal{P}')$. Within the first $non\text{-}overlapping$ constraint, the argument $[0,1,2]$ expresses the fact that we consider a non-overlapping constraint according to dimensions 0, 1 and 2 (i.e., given any pair of parallelepipeds p' and p'' of $\mathcal{P}$ there should exist at least one dimension d ($d \in \{0,1,2\}$) where the projections of p' and p'' on d do not overlap). Similarly, the argument $[0,1]$ of the second non-overlapping constraint expresses the fact that, given any pair of parallelepipeds p' and p'' of $\mathcal{P}'$, there should exist at least one dimension d ($d \in \{0,1\}$) where p' and p'' do not overlap).

The $geost$ constraint is defined in the following way: given a constraint $ctr_i(\mathcal{A}_i,\mathcal{O}_i)$ from the list of constraints $\mathcal{C}$ between a subset of objects $\mathcal{O}_i \subseteq \mathcal{O}$ according to the attributes $\mathcal{A}_i$, let $\mathcal{MC}_i$ denotes the sets of maximum cliques stemming from the objects of $\mathcal{O}_i$ which all overlap in time.[4] The $geost(k,\mathcal{O},\mathcal{S},\mathcal{C})$ constraint holds if and only if $\forall ctr_i \in \mathcal{C}, \forall \mathcal{O}_{\mathcal{MC}_i} \in \mathcal{MC}_i : ctr_i(\mathcal{O}_{\mathcal{MC}_i})$ holds.

Example 2. Figure 1 presents a typical example of a dynamic two-dimensional placement problem where one has to place four objects, both in time as well as within a given box, so that objects that overlap in time do not overlap within the box. Parts (A), (B), (C) and (D) respectively represent the potential shapes associated with the four objects to place, where the origin of each object is stressed in bold. Part (E) shows the position of the four objects of the example as the time vary, where the first, second, third and fourth objects were respectively assigned shapes 1, 5, 8 and 9:

- During the first time interval $[2,9]$ we have only object O_1 at position $(1,2)$.
- Then, at instant 10 objects O_2 and O_3 both appear. Their origins are respectively placed at positions $(2,1)$ and $(4,1)$.
- At instant 14 object O_1 disappears and is replaced by object O_4. The origin of O_4 is fixed at position $(1,1)$. Finally at instant 22 all three objects O_2, O_3 and O_4 disappear.

The arguments of the corresponding $geost$ constraint are:

[3] A *domain variable* v is a variable ranging over a finite set of integers denoted by $\mathrm{dom}(v)$; let $\underline{v}$ and $\overline{v}$ respectively denote the minimum and maximum possible values for v.

[4] In fact these maximum cliques are only used for defining the declarative semantics of the $geost$ constraint.

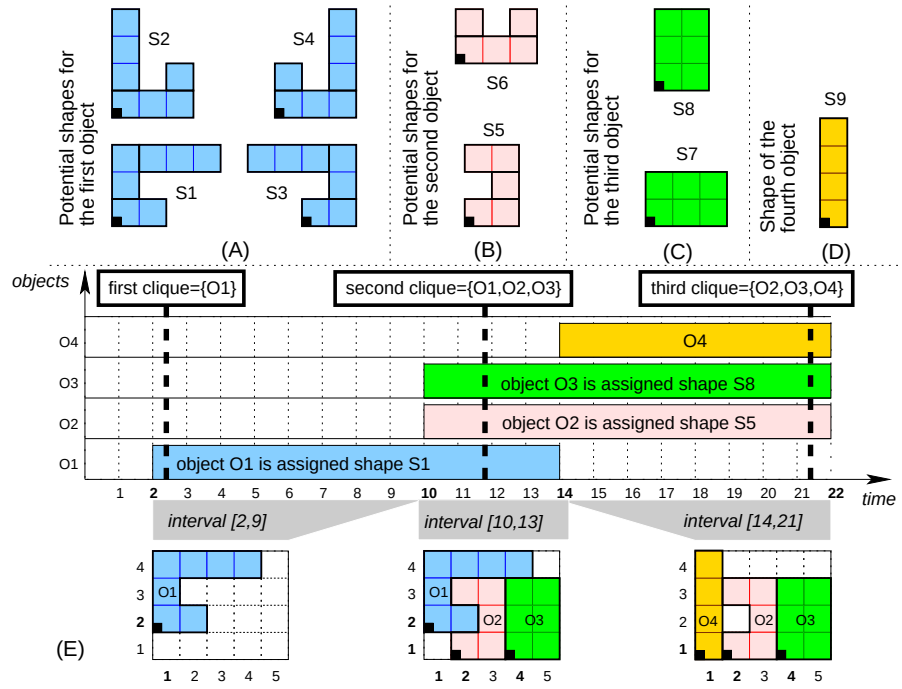


Fig. 1. Example with 4 objects, 9 shapes, one *non-overlapping* and one *included* constraints

```
geost(2,
      [object(1,1,[1,2], 2,12,14),
       object(2,5,[2,1],10,12,22),
       object(3,8,[4,1],10,12,22),
       object(4,9,[1,1],14, 8,22)],
      [shape(1,[0,0],[2,1]), shape(1,[0,1],[1,2]), shape(1,[1,2],[3,1]),
       shape(2,[0,0],[3,1]), shape(2,[0,1],[1,3]), shape(2,[2,1],[1,1]),
       shape(3,[0,0],[2,1]), shape(3,[1,1],[1,2]), shape(3,[2,2],[3,1]),
       shape(4,[0,0],[3,1]), shape(4,[0,1],[1,1]), shape(4,[2,1],[1,3]),
       shape(5,[0,0],[2,1]), shape(5,[1,1],[1,1]), shape(5,[0,2],[2,1]),
       shape(6,[0,0],[3,1]), shape(6,[0,1],[1,1]), shape(6,[2,1],[1,1]),
       shape(7,[0,0],[3,2]),
       shape(8,[0,0],[2,3]),
       shape(9,[0,0],[1,4])],
      [non-overlapping([0,1],[1,2,3,4]),included([0,1],[1,2,3,4],[1,1],[5,4])])
```

Its first argument 2 is the number dimensions of the placement space we consider. Its second and third arguments respectively describe the four objects and the nine shapes we have. Finally its last argument gives the list of geometrical constraints imposed by the $geost$ constraint: the first constraint expresses a non-overlapping constraint between the four objects, while the second constraint imposes the four objects to be located within the box containing all points (x, y) such that $1 \leq x \leq 5$ and $1 \leq y \leq 4$. The $geost$ constraint holds since the four objects do not both simultaneously overlap in time and in space and since they are completely included within the previous box (i.e., see Part (E) of Figure 1).

Within the scope of the $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{C})$ constraint, this article presents a filtering algorithm that adjusts the minimum and maximum value of each coordinate $o.x[d]$,

$0 \leq d < k$ of the origin of an object $o \in \mathcal{O}$, adjusts also the minimum and maximum value of its start $o.start$, its duration $o.duration$ and its end $o.end$ in time, and finally prunes its shape variable $o.sid$. The approach presented in this article offers a number of advantages:

- The main theoretical advantages are fourfold:
  - First, the geometrical kernel makes it possible to integrate new geometrical constraints as new applications and/or requirements show up. This is achieved by providing for each geometrical constraint an API without knowing any details about the geometrical kernel. This contrasts with traditional approaches where one has to come up with a rather involved filtering algorithm for each global constraint.
  - Second, while pruning the attribute of an object, the geometrical kernel takes direct advantage of all geometrical constraints involving that object in order to perform more deduction. This is a fundamental progress over the traditional approach where constraints only co-operate through the domains of their shared variables.
  - Even when we have three or four dimensions, the approach scales well since it does not rely on building complex multi-dimensional data structure (e.g., like quadtrees or octrees). It only stores a number of points in the order of $O(m \cdot k)$ where m is the total number of objects and k is the number of dimensions.
  - Even if complex objects could be decomposed into boxes for which one links the coordinates by external equality constraints this weakens a lot the deduction process as illustrated by the following example of Figure 2: if the shape s (see Part (A)) is decomposed into two rectangles $r4$ and $r5$ (see Part (C)) and if the constraints linking the coordinates of the origins of $r4$ and $r5$ are not integrated within the sweep process, infeasibility cannot be directly derived (see Part (M)). In contrast our approach allows to detect infeasibility directly by reasoning only on the coordinates of the origin of s.
- The main practical advantages are as follows:
  - Having k dimensions allows to come up with a single constraint that can be used for handling general non-overlapping constraints. This was originally motivated by a warehouse management problem where both two-dimensional and three-dimensional sub-problems had to be solved. In the context of three-dimensional packing problems having an extra dimension also makes sense for modelling the fact that we want to assign objects to a truck (in this context we speak about an *assignment dimension* – see Part (I) of Figure 3) or the fact that we do not want to place all the objects since there may simply be not enough room (in this context we speak about a *relaxation dimension*).
  - Factoring out the description of the shapes from the description of an object makes sense in a lot of practical problems where a number of instances of the same shape have to be considered (this is illustrated by Part (J) of Figure 3 where we have five objects but only three shapes: in fact the first, third and fifth objects correspond to the first shape). This again occurs in the warehouse management problem that originally motivated the constraint, where a major car manufacturer has to pack within the same container the parts associated

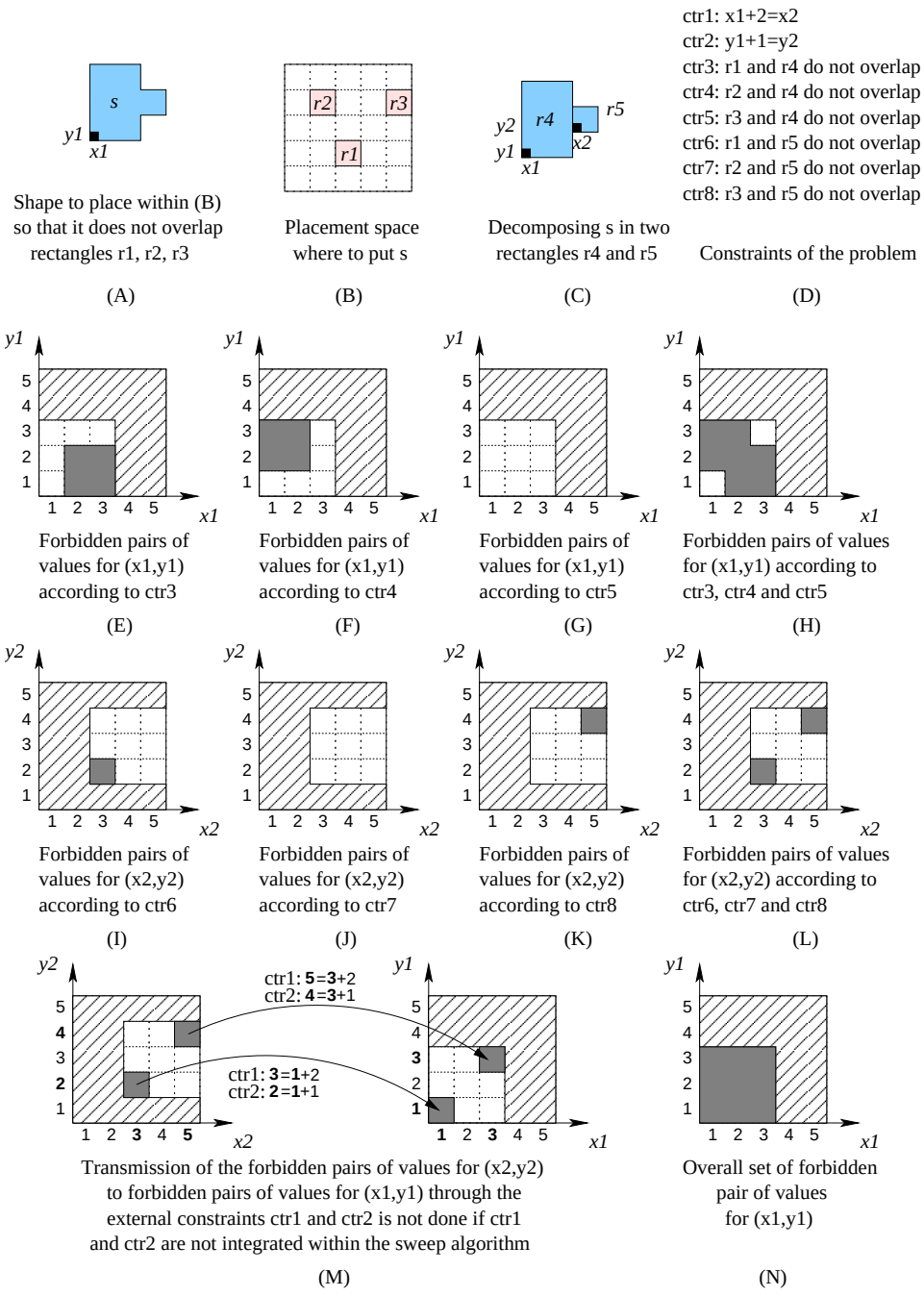


Fig. 2. Reasoning for detecting the infeasibility of the placement problem (dashed areas correspond to initially forbidden pairs of values, while grey areas represent forbidden pairs of values related to some non-overlapping constraints)

with 24 instances of the same car model. By doing so we can decrease the memory requirement (i.e., each complex shape is represented only once), but we can also reduce the running time of the algorithm as we will see later on.
- Having a set of potential shapes for an object offers an extra modelling power for representing directly the fact that objects may rotate (see Part (E) of Figure 3 where objects 2 and 3 can rotate from 90 degrees), or for dealing with tasks for which the duration depends on the machine where the task is actually assigned (see Part (C) of Figure 3).
- Having a temporal dimension allows to tackle dynamic placement problems where objects are moving in time. Consider for instance a pick-up delivery problem where objects are loaded or unloaded from a truck while visiting different locations. In this context the non-overlapping constraint applies only for those objects which overlap in time. This is illustrated by Part (J) of Figure 3.

The article is organised as follows. Section 2 provides an overview of placement problems that can be modelled with the $geost$ constraint. Section 3 introduces the terminology and notation used throughout this article. Section 4 presents the overall architecture of the geometrical kernel. It explains how to define geometrical constraints in terms of a reduced set of standard primitives that are used by the geometrical kernel. Finally, Section 4 describes the set of geometrical constraints currently available. Section 5 presents a multi-dimensional lexicographic sweep algorithm used for filtering the attributes of an object of the $geost$ constraint. Finally Section 6 shows how to adapt the sweep kernel to a greedy mode that is fully compatible with search and backtrack.

2 Modelling Problems with the *geost* Constraint

One advantage of the $geost$ constraint is that is allows to model directly a large number of placement problems by using one single global constraint. Figure 3 sketches ten typical use of the $geost$ constraint, which all mention the $non\text{-}overlapping$ constraint:

- The first case (A) corresponds to a non-overlapping constraint among three segments.
- The second, third and fourth cases (B,C,D) correspond to a non-overlapping constraint between rectangles where (B) and (C) are special cases where the sizes of all rectangles in the second dimension are equal to 1; this can be interpreted as a *machine assignment problem* where each rectangle corresponds to a non-pre-emptive task that has to be placed in time and assigned to a specific machine so that no two tasks assigned to the same machine overlap in time. In Part (B) the duration of each task is fixed, while in Part (C) the duration depends on the machine to which the task is actually assigned. This dependence is expressed by the $compatible$ constraint, which specifies the dependence between the shape variable and the assignment variable of each task.
- The fifth case (E) corresponds to a non-overlapping constraint between rectangles where each rectangle can have two orientations. This is achieved by associating with each rectangle two shapes of respective sizes $l \cdot h$ and $h \cdot l$. Since their orientation is not initially fixed, the $included$ constraint enforces the three rectangles to be

included within the bounding box defined by the origin's coordinates $1, 1$ and sizes $8, 3$.
- The sixth case (F) corresponds to a non-overlapping constraint between more complex objects where each object is described by a given set of rectangles.
- The seventh case (G) describes a rectangle placement problem where one has to first assign each rectangle to a strip so that all rectangles that are assigned to the same strip do not overlap.
- The eighth case (H) corresponds to a non-overlapping constraint between parallelepipeds.
- The ninth case (I) can be interpreted as a non-overlapping constraint between parallelepipeds that are assigned to the same container. The first dimension corresponds to the identifier of the container, while the next three dimensions are associated with the position of a parallelepiped inside a container.
- Finally the tenth case (J) describes a rectangle placement problem over three consecutive time-slots: rectangles assigned to the same time-slot should not overlap in time. We initially start with the three rectangles 1, 2 and 3. Rectangle 3 is no more present at instant 2 (the arrow $\downarrow$ within rectangle 3 at time 1 indicates that rectangle 3 will disappear at the next time-point), while rectangle 4 appears at instant 2 (the arrow $\uparrow$ within rectangle 4 at time 2 denotes the fact that the rectangle 4 appears at instant 2). Finally rectangle 2 disappears at instant 3 and is replaced by rectangle 5.

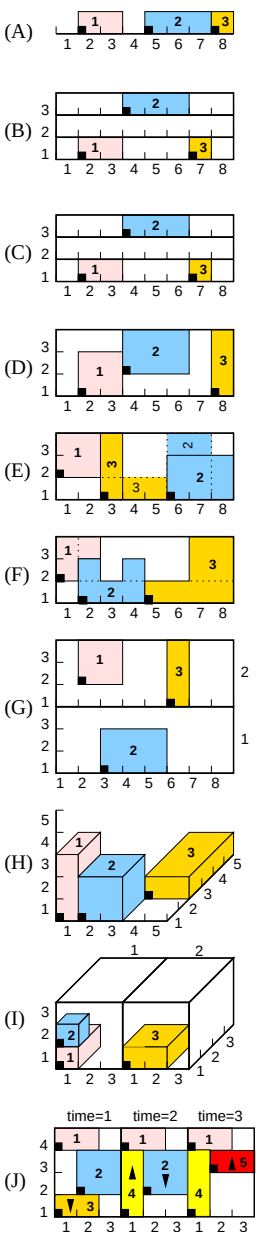
Before continuing, the next section introduces some notation used throughout this article.

3 Notation

We will be using the following terminology and notation:

Point A *point* is a k-dimensional coordinate.
Shifted Box A *shifted box* s is an entity defined by its shape id $s.\mathrm{sid}$, shift offset $s.t[d], 0 \leq d < k$, and sizes $s.l[d], 0 \leq d < k$. It denotes a box in k-dimensional space at the given offset with the given sizes.
Shape A *shape* is a collection of shifted boxes sharing the same shape id.
Object An *object* o is an entity defined by its unique object id $o.\mathrm{oid}$, shape id $o.\mathrm{sid}$, and coordinates $o.x[d], 0 \leq d < k$.
Region A *region* r in k-dimensional space is defined by its object id $r.\mathrm{oid}$ and boundaries $r.\min[d], r.\max[d], 0 \leq d < k$.

Assume v and w are vectors of scalars of k components. Then $v \leftarrow w$ denotes the element-wise assignment of w to v, $w + d$ ($w - d$) denotes the element-wise addition of d ($-d$) to w, $\min(v, w)$ ($\max(v, w)$) denotes the element-wise min (max) of v and w, $\min_{\mathrm{lex}}(v, w)$ ($\max_{\mathrm{lex}}(v, w)$) denotes the lexicographic min (max) of v and w, and w, $v \circ w$ for $\circ \in \{<, \leq, \geq, >\}$ holds if the comparison holds for every element. Given a scalar d, $0 \leq d \leq k - 1$, $\mathrm{rot}(v, d, k)$ denotes the vector $(v[d], v[(d + 1) \mod k], \ldots, v[(d - 1) \mod k])$. Finally, the notation $v \leq_k^d w$ denotes the fact that vector $\mathrm{rot}(v, d, k)$ is lexicographically less than or equal to vector $\mathrm{rot}(w, d, k)$.



geost(1, [object(1,1,[2],0,1,1),object(2,2,[5],0,1,1),object(3,3,[8],0,1,1)],
[shape(1,[0],[2]),shape(2,[0],[3]),shape(3,[0],[1])],
[non−overlapping([0],[1,2,3])])

geost(2, [object(1,1,[2,1],0,1,1),object(2,2,[4,3],0,1,1),object(3,3,[7,1],0,1,1)],
[shape(1,[0,0],[2,1]),shape(2,[0,0],[3,1]),shape(3,[0,0],[1,1])],
[non−overlapping([0,1],[1,2,3])])

geost(2, [object(1,1,[2,1],0,1,1),object(2,4,[4,3],0,1,1),object(3,5,[7,1],0,1,1)],
[shape(1,[0,0],[2,1]),shape(2,[0,0],[3,1]),
[shape(3,[0,0],[2,1]),shape(4,[0,0],[3,1]),
[shape(5,[0,0],[1,1]),shape(6,[0,0],[2,1]),
[non−overlapping([0,1],[1,2,3]),compatible([sid,2],[1,2,3],[1−1,2−2,3−2,4−3,5−1,6−2])])

geost(2, [object(1,1,[2,1],0,1,1),object(2,2,[4,2],0,1,1),object(3,3,[8,1],0,1,1)],
[shape(1,[0,0],[2,2]),shape(2,[0,0],[3,2]),shape(3,[0,0],[1,3])],
[non−overlapping([0,1],[1,2,3])])

geost(2, [object(1,1,[1,2],0,1,1),object(2,2,[6,1],0,1,1),object(3,4,[3,1],0,1,1)],
[shape(1,[0,0],[2,2]),shape(2,[0,0],[3,2]),shape(3,[0,0],[2,3]),
shape(4,[0,0],[1,3]),shape(5,[0,0],[3,1])],
[non−overlapping([0,1],[1,2,3]),included([0,1],[1,2,3],[1,1],[8,3])])

geost(2, [object(1,1,[1,2],0,1,1),object(2,2,[2,1],0,1,1),object(3,3,[5,1],0,1,1)],
[shape(1,[0,0],[1,2]),shape(1,[1,1],[1,1]),
shape(2,[0,0],[3,1]),shape(2,[0,1],[1,1]),shape(2,[2,1],[1,1]),
shape(3,[0,0],[4,1]),shape(3,[2,1],[2,2])],
[non−overlapping([0,1],[1,2,3])])

geost(3, [object(1,1,[2,2,2],0,1,1),object(2,2,[1,3,1],0,1,1),object(3,3,[2,6,1],0,1,1)],
[shape(1,[0,0,0],[1,2,2]),shape(2,[0,0,0],[1,3,2]),shape(3,[0,0,0],[1,1,3])],
[non−overlapping([0,1,2],[1,2,3])])

geost(3, [object(1,1,[1,1,1],0,1,1),object(2,2,[2,1,1],0,1,1),object(3,3,[4,3,1],0,1,1)],
[shape(1,[0,0,0],[1,2,3]),shape(2,[0,0,0],[2,2,2]),shape(3,[0,0,0],[2,4,1])],
[non−overlapping([0,1,2],[1,2,3])])

geost(4, [object(1,1,[1,1,1,1],0,1,1),object(2,2,[1,1,1,2],0,1,1),object(3,3,[2,1,1,1],0,1,1)],
[shape(1,[0,0,0,0],[1,1,2,1]),
shape(2,[0,0,0,0],[1,1,1,1]),
shape(3,[0,0,0,0],[1,2,2,1])],
[non−overlapping([0,1,2,3],[1,2,3])])

geost(2, [object(1,1,[1,4],1,3,4),object(2,2,[2,2],1,2,3),object(3,1,[1,1],1,1,2)
object(4,3,[1,1],2,2,4),object(5,1,[2,3],3,1,4)],
[shape(1,[0,0],[2,1]),shape(2,[0,0],[2,2]),shape(3,[0,0],[1,3])],
[non−overlapping([0,1],[1,2,3,4,5])])

Fig. 3. Ten typical examples of use of the *geost* constraint (ground instances)

4 Standard Representation of Geometrical Constraints

The key idea is that one has to provide for each kind of geometrical constraint found in $\mathcal{C}$ a reduced set of standard primitives that will be used by the geometric constraint kernel. But before describing these functions, let us first introduce the notion of *internal geometrical constraint* (as opposed to the *external geometrical constraints* present in $\mathcal{C}$). Given an external geometrical constraint $ectr_i(\mathcal{O}_i)$ $(\mathcal{O}_i \subseteq \mathcal{O})$ and its frame $FRAME[ectr_i]$, one of its object $o \in \mathcal{O}_i$ and one potential shape s of o, we associate with the triple $(ectr_i, o, s)$ a number (possibly 0 if the constraint is entailed) of internal geometrical constraints (this is concretely done by providing function $\mathrm{GenInternalCtrs}$ $(ectr_i, o, s, FRAME[ectr_i]) : (ictrs)$). This stems from the fact that external geometrical constraints are usually decomposed into a conjunction of smaller internal geometrical constraints (and to some extend the k-dimensional lexicographic sweep algorithm presented in next section handles them globally). Internal geometrical constraints are also used for representing implicit constraints (such as holes in the domain of the coordinates of the origin of an object) or for representing redundant constraints derived from external geometrical constraints (such as preventing the formation of too small holes in the context of non-overlapping constraints when the allowed waste is very small).

The purpose of an internal geometrical constraint $ictr$ is to make available to the geometrical kernel a set of infeasible points for the origin of o under the assumption that o will be assigned shape s and that constraint $ctr_i(\mathcal{O}_i)$ holds. In order to have a compact representation which can be used efficiently by the geometrical kernel this set of infeasible points is defined implicitly by providing the following functions:

- $\mathrm{LexInfeasible}(ictr, minlex, d, k, o) : (found, p)$ when $minlex = \mathbf{true}$ (respectively **false**), returns the smallest (respectively largest) infeasible lexicographical point p associated with the internal geometrical constraint $ictr$ (according to the fact that we prune the d^{th} coordinate of the origin of object o, i.e., the ordering among the different dimensions is $d, (d+1) \mod k, \ldots, (d-1) \mod k$) compatible with the domains of the coordinates of the origin of o. If no such point exists, $found$ is set to **false** (otherwise $found$ is set to $\mathbf{true}$).
- $\mathrm{IsFeasible}(ictr, min, d, k, o, c) : (feasible, f)$ sets $feasible$ to $\mathbf{true}$ if point c is feasible according to the internal constraint $ictr$; if this is not the case, sets $feasible$ to **false**, and computes the forbidden region f according to the fact that we prune the minimum ($min = \mathbf{true}$) or the maximum ($min = \mathbf{false}$) value of the d^{th} coordinate of o: we first maximise the size of f in dimension $(d-1) \mod k$, then maximise the size of f in dimension $(d-2) \mod k$ and so on until we reach the most significant dimension d. Part (A) (respectively Part (B)) of Figure 4 illustrates the computation of the forbidden region f in the context of $k = 2$ and $d = 0$ (respectively $d = 1$).
- $\mathrm{CardInfeasible}(ictr, k, o) : (n)$ returns an estimation of the number n of infeasible points for the origin of object o under the assumption that constraint $ictr$ holds. This information is used as a heuristics for ordering the internal constraints checked by the geometrical kernel.

Figure 5 provides the overall architecture of the system. As illustrated by the figure, the system is decomposed into three parts respectively handling the external geometrical

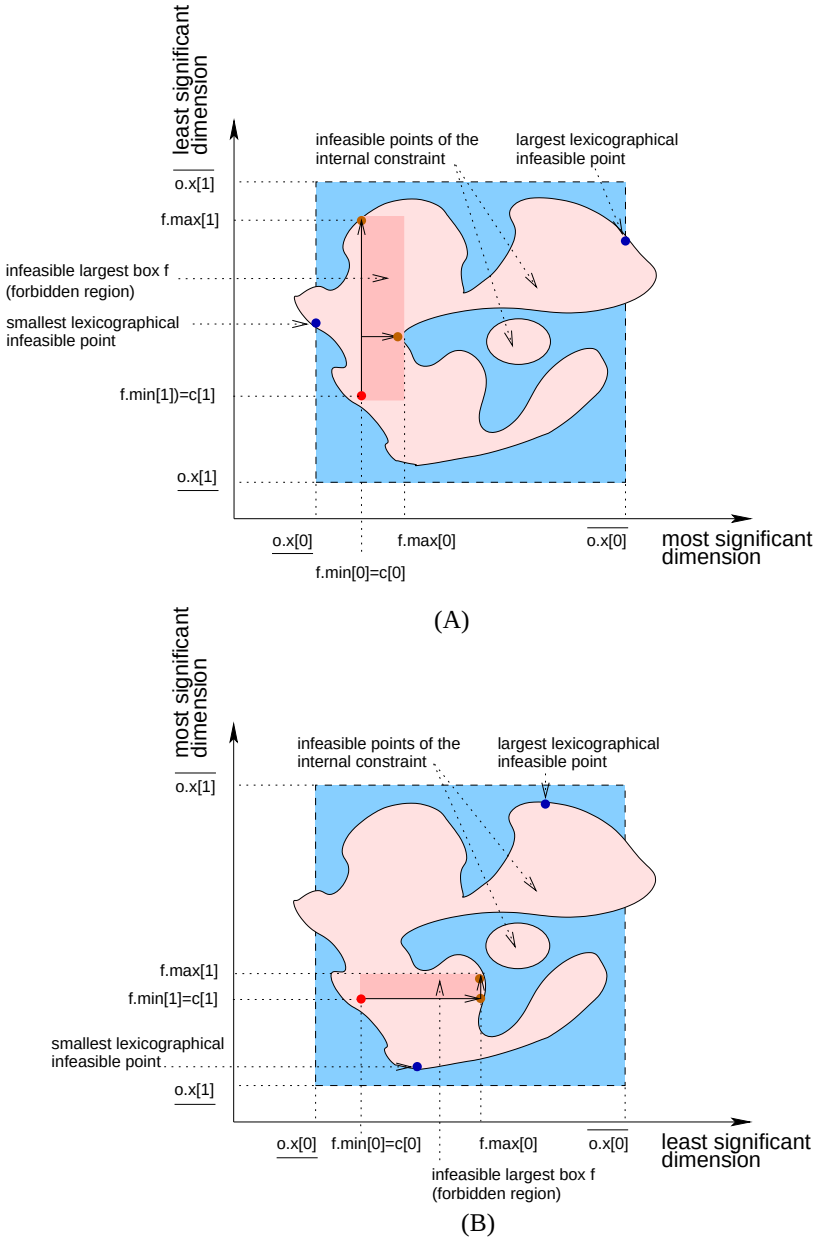


Fig. 4. Illustration of an internal geometrical constraint and of how to compute the forbidden region according to the dimension for which we want to prune (i.e., the most significant dimension)

constraints, the internal geometrical constraints and the geometrical kernel itself. Within each part, pink boxes represent specific internal or external geometrical constraints that will be explained in the two next sections, blue boxes represent the services that have to be provided in order to describe a concrete constraint so that it can be used by the geometrical kernel, and finally green boxes describe the purpose of the corresponding services.

We now describe all internal and external geometrical constraints that are currently available within the constraint kernel. For each internal constraint we provide the set of functions that was just presented, while for each external constraint we show how to reformulate it into a set of internal constraints.

4.1 Internal Geometrical Constraints Currently Available

The inbox constraint The $\mathrm{inbox}(t, l)$ constraint (according to an object o of the $geost$ constraint) is an internal constraint which enforces the point $o.x$ to be located inside the shifted box defined by its shift offset $t[d]$, $0 \leq d < k$, and sizes $l[d]$, $0 \leq d < k$ (i.e., $\forall d \in [0, k-1] : t[d] \leq o.x[d] \leq t[d] + l[d] - 1$).

The outbox constraint The $\mathrm{outbox}(t, l)$ constraint (according to an object o of the $geost$ constraint) is an internal constraint which enforces the point $o.x$ to be located outside the shifted box defined by its shift offset $t[d]$, $0 \leq d < k$, and sizes $l[d]$, $0 \leq d < k$ (i.e., $\exists d \in [0, k-1] : o.x[d] < t[d] \vee o.x[d] > t[d] + l[d] - 1$).

4.2 External Geometrical Constraints Currently Available

This section presents all the external geometrical constraints currently available. For each external constraint we first provide its declarative semantics and then indicate how to generate its corresponding internal constraints. Each internal constraint defines in an implicit way a set of forbidden points for the origin of an object in the k-dimensional space. As we saw in the introduction, an external constraint has always at least two arguments that respectively correspond to dimensions and/or attributes of the object and/or shapes and to the objects (i.e., objects identifiers) for which the constraint apply. In order to simplify the presentation we assume without loss of generality that all the dimensions $0, 1, \ldots, k-1$ are mentioned in the first argument of an external constraint.[5]

The $non\text{-}overlapping$ Constraint

The $non\text{-}overlapping(attributes, iobjects)$ external constraint takes as input a list of distinct dimensions in $\{0, 1, \ldots, k-1\}$ and a list of distinct object's identifiers of the $geost$ constraint. It enforces the following condition: given two distinct objects $o_i =$

[5] If this is not the case we can proceed as follows. Assume the external constraint mentions a subset of dimensions $\mathcal{D} \subseteq \{0, 1, \ldots, k-1\}$. Any infeasible point p according to the dimensions of $\mathcal{D}$ can be extended to a set of infeasible points $\mathcal{P}$ by taking the Cartesian product of the coordinates of p and each interval associated with the domain of the object we currently consider and with the dimensions that do not belong to $\mathcal{D}$.

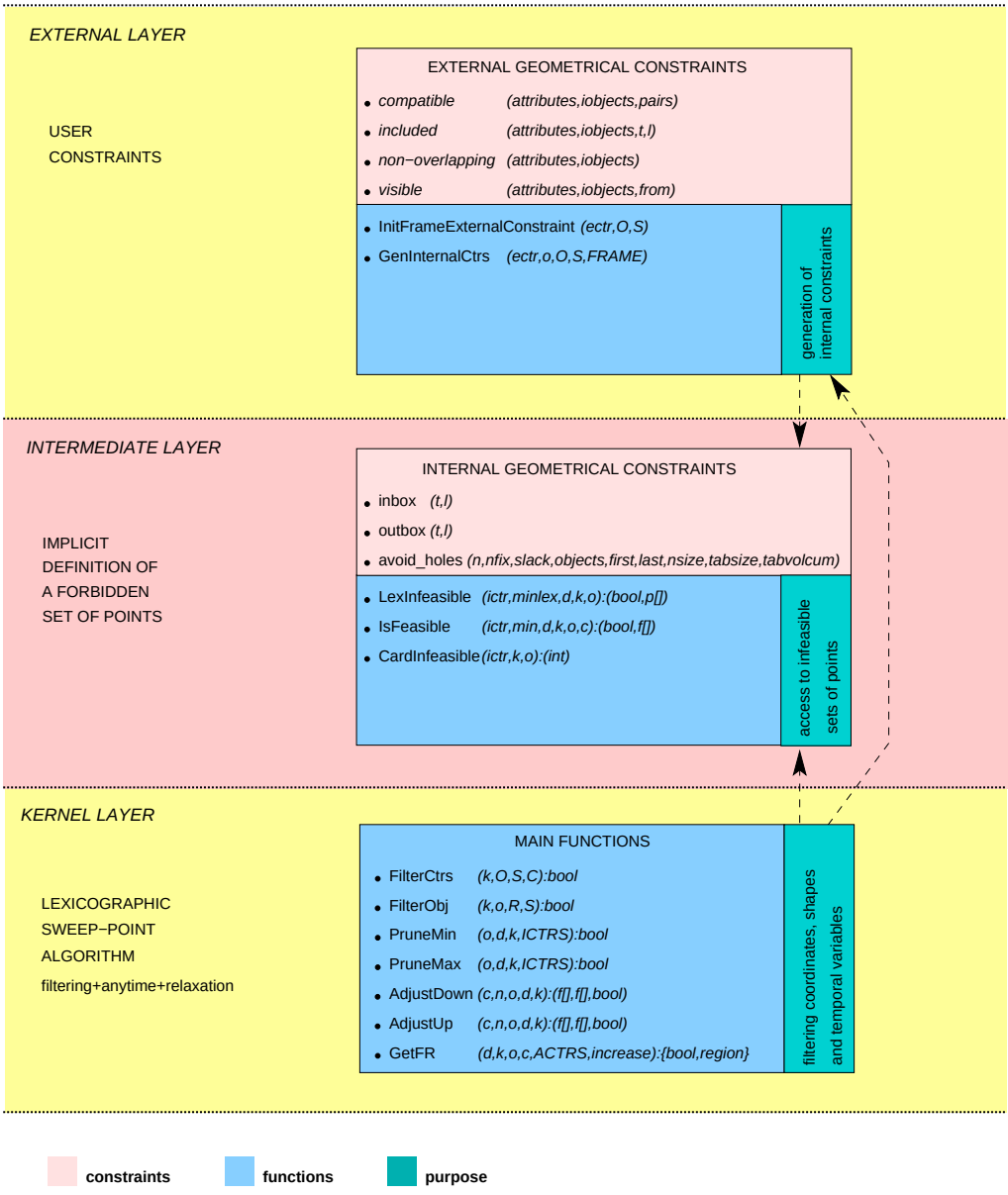


Fig. 5. Overall architecture of the system

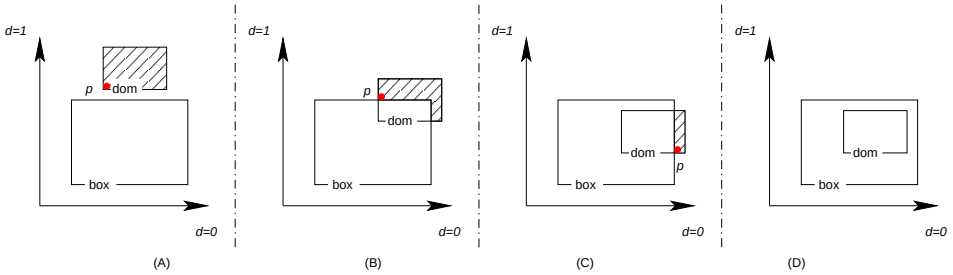


Fig. 6. Illustration of algorithm LexInfeasible in the context of the inbox internal constraint, assuming $minlex = \mathbf{true}$ and $d = 0$ (i.e., 1 is the least significant dimension, 0 is the most significant dimension): (A) $found = \mathbf{true}$ (line 29), (B) $found = \mathbf{true}$ (line 18, first iteration), (C) $found = \mathbf{true}$ (line 18, second iteration), (D) $found = \mathbf{false}$ (line 27); dashed areas represent set of infeasible points for the origin of the object we want to place, while red points correspond to the smallest infeasible lexicographical point p returned by the algorithm.

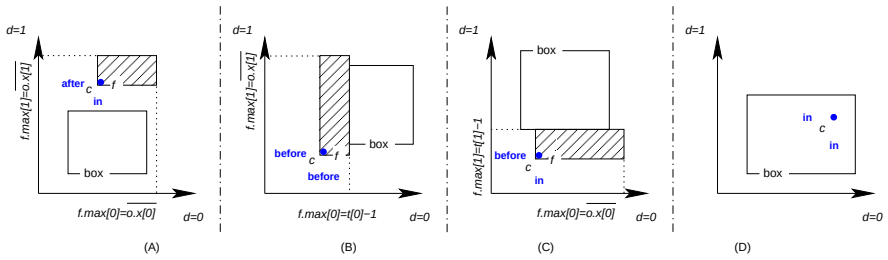


Fig. 7. Illustration of algorithm IsFeasible in the context of the inbox internal constraint, assuming $min = \mathbf{true}$ and $d = 0$ (i.e., 1 is the least significant dimension, 0 is the most significant dimension): (A) $feasible = \mathbf{false}$ (since their is one dimension 1 where $c[1] > t[1] + l[1] - 1$, we have $f.\max[j'] = \overline{o.x[j']}$), (B) $feasible = \mathbf{false}$ (since no dimension j' such that $c[j'] > t[j'] + l[j'] - 1$, we have $f.\max[j'] = t[j'] - 1$ for the the most significant dimension $j' = 0$ such that $c[j'] < t[j']$), (C) $feasible = \mathbf{false}$ (since no dimension j' such that $c[j'] > t[j'] + l[j'] - 1$, we have $f.\max[j'] = t[j'] - 1$ for the the most significant dimension $j' = 1$ such that $c[j'] < t[j']$), (D) $feasible = \mathbf{true}$ (since the point c is located within the box); dashed areas represent the computed forbidden region, while blue points correspond to the point c given to the algorithm.

```
PROCEDURE LexInfeasible(inbox(t, l), minlex, d, k, o) : {found, p}
 1: in ← true
 2: for j ← 0 to k − 1 do
 3:    if minlex then
 4:       p[j] ← o.x[j]                          // copy smallest lexicographical point of o.x to p
 5:    else
 6:       p[j] ← o.x[j]                          // copy largest lexicographical point of o.x to p
 7:    end if
 8:    if p[j] < t[j] ∨ p[j] > t[j] + l[j] − 1 then
 9:       in ← false        // outside the box since outside interval associated with dimension j
10:    end if
11: end for
12: if in then
13:    for j ← k − 1 downto 0 do
14:       j' ← (j + d) mod k                     // scan dimensions by increasing order of priority
15:       if minlex then
16:          if t[j'] + l[j'] ≤ o.x[j'] then
17:             p[j'] ← t[j'] + l[j'] // stops when find a dimension where the upper border (+1) of
18:             return {true, p}                // the box is in the range of the j'^th coordinate of o.x
19:          end if
20:       else
21:          if t[j'] − 1 ≥ o.x[j'] then
22:             p[j'] ← t[j'] − 1    // stops when find a dimension where the lower border (-1) of
23:             return {true, p}                // the box is in the range of the j'^th coordinate of o.x
24:          end if
25:       end if
26:    end for
27:    return {false, p}
28: else
29:    return {true, p}
30: end if
```

Algorithm 1: When $minlex = \mathbf{true}$ (respectively $minlex = \mathbf{false}$) returns the smallest (respectively largest) infeasible lexicographical point p associated with the $\mathrm{inbox}(t, l)$ constraint according to the fact that we prune the d^{th} coordinate of the origin of object o. Set $found$ to $\mathbf{true}$ if such a point exits and to $\mathbf{false}$ otherwise.

```
PROCEDURE IsFeasible(inbox(t, l), min, d, k, o, c) : {feasible, f}
 1: before ← false                          // initially no dimension j' where c[j'] < t[j']
 2: after ← false                           // initially no dimension j' where c[j'] > t[j'] + l[j'] − 1
 3: for j ← 0 to k − 1 do
 4:    j' ← (j + d) mod k                   // scan dimensions by decreasing order of priority
 5:    if min then
 6:       f.min[j'] ← c[j']                 // set to the j'^th coordinate of c
 7:       if c[j'] < t[j'] ∧ ¬before then
 8:          f.max[j'] ← t[j'] − 1          // set to the lower limit of the box minus 1
 9:          before ← true
10:       else
11:          f.max[j'] ← o.x[j']            // set to infinity
12:          if c[j'] > t[j'] + l[j'] − 1 then
13:             after ← true
14:          end if
15:       end if
16:    else
17:       f.max[j'] ← c[j']                 // set to the j'^th coordinate of c
18:       if c[j'] > t[j'] + l[j'] − 1 ∧ ¬after then
19:          f.min[j'] ← t[j'] + l[j']      // set to the upper limit of the box plus 1
20:          after ← true
21:       else
22:          f.min[j'] ← o.x[j']            // set to minus infinity
23:          if c[j'] < t[j'] then
24:             before ← true
25:          end if
26:       end if
27:    end if
28: end for
29: feasible ← ¬(before ∨ after)            // feasible is true if c is located
30: return (feasible, f)                    // within the box depicted by inbox(t, l)
```

Algorithm 2: Set $feasible$ to $\mathbf{true}$ if point c is feasible according to the $\mathrm{inbox}(t, l)$ constraint; if this is not the case, sets $feasible$ to $\mathbf{false}$, and computes the forbidden region f according to the fact that we prune the minimum ($min = \mathbf{true}$) or maximum ($min = \mathbf{false}$) value of the d^{th} coordinate of o.

```
PROCEDURE CardInfeasible(inbox(t, l), k, o) : {n}
1: n ← 1                                    // volume of the domain of the origin of o
2: for j ← 0 to k − 1 do
3:    n ← n · (o.x[j] − o.x[j] + 1)
4: end for
5: m ← 1                                    // volume of the intersection between the box and the origin of o
6: for j ← 0 to k − 1 do
7:    m ← m · max(0, min(o.x[j], t[j] + l[j] − 1) − max(o.x[j], t[j]) + 1)
8: end for
9: return n − m
```

Algorithm 3: Returns the number n of infeasible points for the origin of object o under the assumption that the $\mathrm{inbox}(t, l)$ constraint holds.

PROCEDURE LexInfeasible($\mathrm{outbox}(t,l), minlex, d, k, o) : \{found, p\}$

```
1: for j ← 0 to k − 1 do
2:    if ō.x[j] < t[j] ∨ o.x[j] > t[j] + l[j] − 1 then
3:       return {false, p}
4:    end if
5:    if minlex then
6:       p[j] ← max(t[j], o.x[j])
7:    else
8:       p[j] ← min(t[j] + l[j] − 1, ō.x[j])
9:    end if
10: end for
11: return {true, p}
```

Algorithm 4: When $minlex = \mathbf{true}$ (respectively $minlex = \mathbf{false}$) returns the smallest (respectively largest) infeasible lexicographical point p associated with the $\mathrm{outbox}(t,l)$ constraint according to the fact that we prune the d^{th} coordinate of the origin of object o. Set $found$ to $\mathbf{true}$ if such a point exits and to $\mathbf{false}$ otherwise.

PROCEDURE IsFeasible($\mathrm{outbox}(t,l), min, d, k, o, c) : \{feasible, f\}$

```
1: for j ← 0 to k − 1 do
2:    if c[j] < t[j] ∨ c[j] > t[j] + l[j] − 1 then
3:       return (true, f)      // exit since point c is feasible according to the outbox constraint
4:    end if
5:    if min then
6:       f.min[j] ← c[j]                                  // set to the j^th coordinate of c
7:       f.max[j] ← min(ō.x[j], t[j] + l[j] − 1)          // set to the j^th upper limit of outbox(t, l)
8:    else
9:       f.max[j] ← c[j]                                  // set to the j^th coordinate of c
10:      f.min[j] ← max(o.x[j], t[j])                     // set to the j^th lower limit of outbox(t, l)
11:   end if
12: end for
13: return (false, f)
```

Algorithm 5: Set $feasible$ to $\mathbf{true}$ if point c is feasible according to the $\mathrm{inbox}(t,l)$ constraint; if this is not the case, sets $feasible$ to $\mathbf{false}$, and computes the forbidden region f according to the fact that we prune the minimum ($min = \mathbf{true}$) or maximum ($min = \mathbf{false}$) value of the d^{th} coordinate of o.

PROCEDURE CardInfeasible($\mathrm{outbox}(t,l), k, o) : \{n\}$

```
1: n ← 1
2: for j ← 0 to k − 1 do
3:    n ← n · (min(ō.x[j], t[j] + l[j] − 1) − max(o.x[j], t[j]) + 1)
4: end for
5: return n
```

Algorithm 6: Returns an estimation about the number n of infeasible points for the origin of object o under the assumption that the $\mathrm{outbox}(t,l)$ constraint holds.

$\mathrm{object}(id_i, sid_i, x_i[], start_i, duration_i, end_i)$ and $o_j = \mathrm{object}(id_j, sid_j, x_j[], start_j, duration_j, end_j)$ ($id_i, id_j \in iobjects$) such that both objects overlap in time (i.e., $end_i > start_j \wedge end_j > start_i \wedge duration_i \cdot duration_j > 0$), no shifted box $s_i = \mathrm{shape}(sid_i, t_i[], l_i[])$ (with shape identifier sid_i) should overlap any shifted box $s_j = \mathrm{shape}(sid_j, t_j[], l_j[])$ (with shape identifier sid_j) (i.e., $\exists d \in [0, k-1] : o_i.x_i[d] + s_i.t_i[d] + s_i.l_i[d] \leq o_j.x_j[d] + s_j.t_j[d] \vee o_j.x_j[d] + s_j.t_j[d] + s_j.l_j[d] \leq o_i.x_i[d] + s_i.t_i[d] \vee s_i.l_i[d] \cdot s_j.l_j[d] = 0$). Before describing the internal constraints associated with the $non\text{-}overlapping$ constraint, we must first define the notion of *relative forbidden regions* and of *absolute forbidden regions*.

4.2.3.1 Forbidden Regions

Let $\mathrm{line}(x, l)$ denote a line segment with origin x and length l. Two line segments with fixed boundaries $\mathrm{line}(x, l)$ respectively $\mathrm{line}(x', l')$ *overlap* if and only if

$$x + l > x' \wedge x' + l' > x \tag{1}$$

Suppose now that x' varies over the interval $[\underline{x'}, \overline{x'}]$. Then if $\mathrm{line}(x, l)$ overlaps with both $\mathrm{line}(\underline{x'}, l')$ and $\mathrm{line}(\overline{x'}, l')$, then $\mathrm{line}(x, l)$ also overlaps with $\mathrm{line}(y, l')$ for any value $\underline{x'} < y < \overline{x'}$. From (1) we get that $\mathrm{line}(x, l)$ overlaps with both $\mathrm{line}(\underline{x'}, l')$ and $\mathrm{line}(\overline{x'}, l')$ if and only if:

$$x + l > \underline{x'} \wedge \underline{x'} + l' > x \wedge x + l > \overline{x'} \wedge \overline{x'} + l' > x$$

which simplifies to:

$$\underline{x'} + l' > x \wedge x + l > \overline{x'}$$

i.e.:

$$x \in [\overline{x'} - l + 1, \underline{x'} + l' - 1] \tag{2}$$

We call the interval in the right hand side of (2) the *forbidden region of* $\mathrm{line}(x, l)$ *wrt.* $\mathrm{line}(x', l')$, i.e. the set of values V such that if $x \in V$ then for any value x' in its interval, the two lines will overlap.

Note that the forbidden region does not depend on x, but does depend on l in its left boundary. For purposes of factoring out parts of the computation, we introduce the notion *relative forbidden region of* $\mathrm{line}(x', l')$ as the interval $[\overline{x'} + 1, \underline{x'} + l' - 1]$. That is, it does not depend on $\mathrm{line}(x, l)$ at all (i.e., in fact it assumes that $\mathrm{line}(x, l)$ is reduced to a single point, $l = 0$).

The notion of (relative) forbidden region generalises naturally to k dimensions, denoting a region where the origin of an object cannot be placed without causing it to overlap with some other object.

4.2.3.2 Generating the Internal Constraints Associated with the $non\text{-}overlapping$ Constraint

Within the filtering algorithms, the function $\mathrm{RelForbReg}(iobjects, \mathcal{S})$ computes the set R of all relative forbidden regions. The procedure

$\mathrm{InitFrameExternalConstraint}(\mathit{non\text{-}overlapping}(\mathit{attributes}, \mathit{iobjects}), \mathcal{O}, \mathcal{S})$ calls the function $\mathrm{RelForbReg}(\mathit{iobjects}, \mathcal{S})$. $\mathrm{InitFrameExternalConstraint}$ is called each time we wake the geost constraint (see line 5 of procedure $\mathrm{FilterCtrs}$).

$$\begin{aligned}
\mathrm{RelForbReg}(\mathit{iobjects}, \mathcal{S}) = \Big\{ r \;\Big|\; & \exists o \in \mathit{iobjects} \wedge \exists s \in \mathcal{S} \wedge s.\mathrm{sid} \in \mathrm{dom}(o.\mathrm{sid}) \wedge \\
& r.\mathrm{oid} = o.\mathrm{oid} \wedge \\
& r.\min = \overline{o.x} + s.t + 1 \wedge \\
& r.\max = \underline{o.x} + s.t + s.l - 1 \Big\} \\
\mathrm{AbsForbReg}(o, R, \mathcal{S}) = \Big\{ r' \;\Big|\; & \exists r \in R \wedge \exists s \in \mathcal{S} \wedge s.\mathrm{sid} = o.\mathrm{sid} \wedge \\
& r'.\mathrm{oid} = r.\mathrm{oid} \neq o.\mathrm{oid} \wedge \\
& r'.\min = r.\min - s.t - s.l \wedge \\
& r'.\max = r.\max - s.t \wedge \\
& \mathrm{dom}(o.x) \cap r' \neq \emptyset \Big\}
\end{aligned}$$

The function $\mathrm{AbsForbReg}(o, R, \mathcal{S})$ computes the set of relevant forbidden regions, associated with each object and the shifted boxes belonging to its potential shapes, for a given object o. To each relevant forbidden region corresponds an outbox constraint. When we try to filter the coordinates of an object o, the function $\mathrm{AbsForbReg}(o, R, \mathcal{S})$ is called by $\mathrm{GenInternalCtrs}(\mathit{non\text{-}overlapping}(\mathit{attributes}, \mathit{iobjects}), o, \mathcal{O}, \mathcal{S}, \mathit{FRAME})$ for computing the absolute forbidden regions of o according to all other objects and generate the corresponding outbox constraints (see line 6 of procedure $\mathrm{FilterObj}$).

5 The Geometrical Kernel: a Generic k-Dimensional Lexicographic Sweep Algorithm

This section first presents the sweep algorithm used for filtering the co-ordinates of the origin of an object of the geost constraint, then shows how to extend the algorithm in order to prune the origin, duration and end attributes of an object. Finally it shows how to control the sweep algorithm in order to rapidly find a solution or a relaxed solution.

5.1 The Sweep Algorithm

This algorithm first considers all internal geometrical constraints $\mathcal{IC}_o$ derived from $\mathcal{C}$ where o actually occurs, and then performs a recursive traversal of the placement space for each coordinate, each direction (i.e., $\min$ or $\max$) and each potential shape $s \in \mathrm{dom}(o.sid)$. Without loss of generality, assume we want to adjust the minimum value of the d^{th} coordinate $o.x[d]$ $0 \leq d < k$ of the origin of o according to the hypothesis that the shape of o is fixed to s. The algorithm starts its recursive traversal of the placement space at point

$$c = \mathrm{rot}(\underline{o.x}, d, k) = (\underline{o.x}[d], \underline{o.x}[(d+1) \mod k], \ldots, \underline{o.x}[(d-1) \mod k])$$

and could in principle explore all points of the domains of $o.x$, one by one, in increasing lexicographic order, until a point is found that is not infeasible for any internal constraint, in which case $c[0]$ is the computed lower bound. To make the search efficient, instead of moving each time to the successor point, we arrange the search so that it skips points that are known to be infeasible for some internal constraint.[6]

Example 3. To illustrate the idea on a 2-D case, consider Figure 8. Suppose that both coordinates (x, y) range over $[2, 6]$ and that we have two sets of infeasible points corresponding to rectangular regions r_1 and r_2:

$$r_1.\min = (2, 2), r_1.\max = (3, 4)$$
$$r_2.\min = (1, 4), r_2.\max = (4, 6)$$

Let us simulate a search for a point that is not inside r_1 or r_2.

1. **Initialise.** We start at the minimal value of the coordinates. $(x, y) \leftarrow (2, 2)$.
2. **Check and move.** $(x, y) \in r_1$, and the next point that is outside r_1 is $(2, 5)$, so $(x, y) \leftarrow (2, 5)$.
3. **Check and move.** $(x, y) \in r_2$, and the next point after that is outside r_2 is $(2, 7)$, which is outside the coordinate domain. So now we know that there is no feasible point for $x = 2$. We also know that the forbidden regions encountered during the search where $x = 2$ cover all points for $x = 3$ as well. So $(x, y) \leftarrow (4, 2)$, the next point that is inside the coordinate domain and not yet known to be in any forbidden region.
4. **Check.** $(x, y) \notin r_1, (x, y) \notin r_2$, and we are done.

□

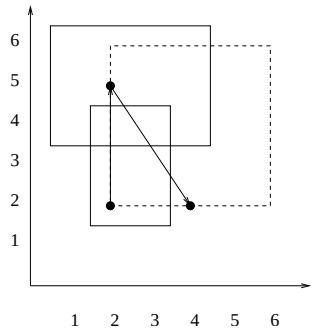


Fig. 8. Search for a feasible point

Thus we compute the lexicographically smallest point c' such that:

1. c' is lexicographically greater than or equal to c,

[6] Potential holes in the domains are reflected in internal constraints.

2. every element of c' is in the range of the corresponding element of $o.x$ (i.e., $\underline{o.x} \leq c' \leq \overline{o.x}$),
3. c' is not infeasible according to the current status of any internal geometrical constraint of $\mathcal{IC}_o$.

If no such c' exists, we remove from the shape variable of $o.sid$ the value s that was our current hypothesis for the shape of o (and fail if s was the only possible value for $o.sid$). Otherwise the minimum value of $o.x[d]$ is adjusted to the most significant element of c' (i.e., $c'[0]$ is the computed lower bound).

As we saw our sweep algorithm moves in increasing lexicographic order a point c from its lexicographically smallest potential feasible position to its lexicographically largest potential feasible position through all potential points. The algorithm uses the following two data structures:

– A data structure called the *sweep-point* status, which contains some information related to the current position c of the sweep-point:
  - All internal constraints from $\mathcal{IC}_o$ that can potential interact with the current position of the sweep-point c (i.e., the set of *active internal constraints*). This corresponds to all internal constraints $ictr_o \in \mathcal{IC}_o$ such that $\mathrm{LexInfeasible}(ictr_o, \mathbf{true}, d, k, o) \leq_{lex} c \leq_{lex} \mathrm{LexInfeasible}(ictr_o, \mathbf{false}, d, k, o)$.
  - A vector $n[0..k-1]$ that caches knowledge about already encountered sets of infeasible points while moving c from its first potential feasible position. The vector n is always element-wise greater than c and maintained as follows. Let $\mathrm{inf}, \mathrm{sup}$ denote the vectors $\mathrm{inf} = \mathrm{rot}(\underline{o.x}, d, k)$ and $\mathrm{sup} = \mathrm{rot}(\overline{o.x} + 1, d, k)$:
    * Initially, $n = \mathrm{sup}$.
    * Whenever a set of infeasible points f such that $c \in f$ is found, n is updated by taking the element-wise minimal value of n and the end coordinate of $\mathrm{rot}(f, d, k)$, indicating the fact that new candidate points can be found beyond that value.
    * Whenever we skip to the next candidate point, we reset the elements of n that were used to the corresponding values of sup.

  The following invariant holds for the vector n, and is used when advancing c to the next candidate point. Let i be the smallest j such that

$$n[j+1] = \mathrm{sup}[j+1] \wedge \cdots \wedge n[k-1] = \mathrm{sup}[k-1]$$

  and suppose c is known to be in some set of infeasible points. Then the next point, lexicographically greater than c and not yet known to be in any set of infeasible points, is:

$$(c[0], \ldots, c[i-1], n[i], \mathrm{inf}[i+1], \ldots, \mathrm{inf}[k-1])$$

– A data structure named the *event point series*, which holds the events to process, ordered in lexicographically increasing order. These events correspond to the lexicographically smallest point $\mathrm{LexInfeasible}(ictr_o, \mathbf{true}, d, k, o)$ associated with each internal constraint $ictr_o \in \mathcal{IC}_o$. These events are stored in a heap so that we can extract them in lexicographically increasing order.

Example 4. Figure 9 illustrates the k-dimensional lexicographic sweep algorithm in the context of $k = 2$. Parts (A) and (B) provide the variables of the problem (i.e., the abscissa and ordinate of each rectangle r_1, r_2, r_3, r_4 and r_5) as well as the non-overlapping constraint between the five previous rectangles. On Parts (D) to (L) we have represented the extreme possible feasible positions of each rectangle (i.e., rectangles r_1 to r_4): for instance the leftmost lower corner of rectangle r_1 can only be fixed at positions $(1, 2)$, $(1, 3)$, $(1, 4)$, $(2, 2)$, $(2, 3)$, $(2, 4)$, $(3, 2)$, $(3, 3)$, $(3, 4)$, $(4, 2)$, $(4, 3)$ and $(4, 4)$. Parts (C) to (L) of Figure 9 detail the different steps of the algorithm for adjusting the minimum value of the abscissa of rectangle r_5. Part (C) provides the internal constraints associated with the fact that we want to prune the coordinates of r_5: constraints ctr_1, ctr_2, ctr_3 and ctr_4 respectively correspond to the fact that rectangle r_5 should not overlap rectangles r_1, r_2, r_3 and r_4, while constraint ctr_5 represents the fact that the ordinate of r_5 should be different from 7. Each of these five internal constraints corresponds to an $\mathrm{outbox}([o_x, s_x], [o_y, s_y])$ constraint enforcing that the two conditions $x_5 \in [o_x, o_x + s_x - 1]$ and $y_5 \in [o_y, o_y + s_y - 1]$ are not both true. Part (D) represents the initialisation phase of the algorithm where we have inserted into the heap all five internal constraints with their respective lexicographically smallest feasible point (i.e., $(1, 1)$ for ctr_1, $(1, 3)$ for ctr_2, $(1, 7)$ for ctr_5, $(1, 8)$ for ctr_3 and $(3, 1)$ for ctr_4). Part (E) represents the first step of the sweep-point algorithm where we start the traversal of the placement space at point $c = (1, 1)$. We first transfer from the heap to the list of active internal constraints all internal constraints for which the first lexicographically smallest infeasible point if lexicographically greater than or equal to the current position of the sweep-point $c = (1, 1)$ (i.e., constraint $ctr_1 = \mathrm{outbox}([1, 2], [1, 2])$). We then search through the list of active constraints (represented on the figure by a box with the legend ACTRS on top of it) the first constraint for which $c = (1, 1)$ is infeasible. In fact, since ctr_1 is infeasible (represented on the figure by a box with the legend CONFLICT on top of it) we compute the feasible vector $f = (3, 3)$ that tells how to get the next feasible point in the different dimensions. Consequently the sweep-point moves to the next position $(1, 3)$ (see Part (F)) and the process is repeated until we finally find a feasible sweep-point for all internal constraints (i.e., point $(3, 8)$ in Part (L)). Observe that, when the lexicographically largest infeasible point associated with an active internal constraint is lexicographically strictly less than the position of the sweep-point, we remove that constraint from the list of active internal constraints. This is for instance the case in Part (I), where we remove constraint ctr_3 from the list of active internal constraints (i.e., since its lexicographically largest infeasible point $(2, 8)$ is lexicographically smaller than the position of the sweep-point $c = (3, 1)$).

Algorithms 7 through 12 implement this idea. The algorithms prune the bounds of each coordinate of every object wrt. its relevant internal constraints, iterating to fix-point. Given a point c and a list of active internal constraints $ACTRS$, Algorithm 7 looks for an internal constraint $ictr$ of $ACTRS$ such that the point c is infeasible for constraint $ictr$. If such a constraint can be found, Algorithm 7 returns the corresponding forbidden region f. Algorithm 8 filters all objects according to all external geometrical constraints where they are currently involved. It consists of a first phase that initialises for each geometrical constraint and for its objects a data structure describing this object versus that constraint. A second phase tries to prune all objects. In order to reach a fix-point, the process is started again until no pruning occurs anymore.

5.2 Handling Time

Given an object $o_i = \mathrm{object}(id, sid, o[], start, duration, end)$ of a $geost$ constraint, the sweep-point algorithm that we have introduced in the previous section can be easily

Fig. 9. Illustration of the sweep algorithm for adjusting the minimum value of the abscissa of rectangle r_5

```
PROCEDURE GetFR(d, k, o, c, ACTRS, increase) : {bool, region}
 1: if increase then
 2:    if ∃ictr ∈ ACTRS | (false, f) = IsFeasible(ictr, true, d, k, o, c) then
 3:       return (true, f)
 4:    else
 5:       return (false, f)
 6:    end if
 7: else
 8:    if ∃ictr ∈ ACTRS | (false, f) = IsFeasible(ictr, false, d, k, o, c) then
 9:       return (true, f)
10:    else
11:       return (false, f)
12:    end if
13: end if
```

Algorithm 7: Is a point c infeasible according to any currently active internal geometrical constraints (i.e., the internal constraints of $ACTRS$)? We currently try to prune the minimum ($increase = 1$) or the maximum ($increase = 0$) of $o.x[d]$.

```
PROCEDURE FilterCtrs(k, O, S, C) : bool
 1: nonfix ← true                                          // fix-point not yet reached
 2: while nonfix do
 3:    nonfix ← false                                      // assumes no filtering will be done
 4:    for all ectr ∈ C do
 5:       FRAME[ectr] ← InitFrameExternalConstraint(ectr, O, S)
 6:    end for
 7:    for all o ∈ O do
 8:       if ¬FilterObj(k, o, FRAME, S) then
 9:          return false                                   // no feasible origin
10:       else if o.x was pruned then
11:          nonfix ← true                                // has to saturate once again
12:       end if
13:    end for
14: end while
15: return true                                           // feasible origin
```

Algorithm 8: Main filtering algorithm associated with the $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{C})$ constraint, where k, $\mathcal{O}$, $\mathcal{S}$ and $\mathcal{C}$ respectively correspond to the number of dimensions, to the objects, to the shapes and to the external geometrical constraints.

PROCEDURE FilterObj$(k, o, FRAME, \mathcal{S})$: bool

1: $ICTRS \leftarrow \emptyset$ // build the list of internal constraints attached to o
2: **for** $d \leftarrow 0$ **to** $k-1$ **do**
3: $ICTRS \leftarrow ICTRS \bigcup$ possible outbox constraints corresponding to holes of $o.x[d]$
// holes correspond to adjacent forbidden values of $\mathrm{dom}(o.x[d])$
4: **end for**
5: **for all** external geometrical constraints $ectr$ involving o **do**
6: $ICTRS \leftarrow ICTRS \bigcup \mathrm{GenInternalCtrs}(ectr, o, \mathcal{O}, \mathcal{S}, FRAME[ectr])$
7: **end for**
8: **for** $d \leftarrow 0$ **to** $k-1$ **do**
9: **if** $\neg\mathrm{PruneMin}(o, d, k, ICTRS) \vee \neg\mathrm{PruneMax}(o, d, k, ICTRS)$ **then**
10: **return false** // no feasible origin
11: **end if**
12: **end for**
13: **return** $\mathbf{true}$ // feasible origin

Algorithm 9: Filtering all the k coordinates of a given object o according to all external geometrical constraints where o occurs; $FRAME[ectr]$ corresponds to a possible frame associated with an external constraint, while $\mathcal{S}$ is the set of shapes of the $geost$ constraint.

PROCEDURE NewPruneMin$(o, d, k, ICTRS)$: bool

1: $b \leftarrow \mathbf{true}$ // $b = \mathbf{true}$ while we have not failed
2: $c \leftarrow \underline{o.x}$ // initial position of the point
3: $n \leftarrow \overline{o.x} + 1$ // upper limits+1 in the different dimensions
4: $(infeasible, f) \leftarrow \mathrm{GetFR}(d, k, o, c, ICTRS, \mathbf{true})$
5: **while** $b \wedge infeasible$ **do**
6: $n \leftarrow \min(n, f.\max + 1)$
7: $(c, n, b) \leftarrow \mathrm{AdjustUp}(c, n, o, d, k)$
8: $(infeasible, f) \leftarrow \mathrm{GetFR}(d, k, o, c, ICTRS, \mathbf{true})$
9: **end while**
10: **if** b **then**
11: $\underline{o.x[d]} \leftarrow c[d]$
12: **end if**
13: **return** b

Algorithm 10: Adjusting the lower bound of the d^{th} coordinate of the origin of object o; $ICTRS$ is the set of internal constraints associated with object o.

PROCEDURE PruneMin$(o, d, k, ICTRS)$: bool

1: $b \leftarrow \mathbf{true}$ // $b = \mathbf{true}$ while we have not failed
2: $c \leftarrow \underline{o.x}$ // initial position of the point
3: $n \leftarrow \overline{o.x} + 1$ // upper limits+1 in the different dimensions

// insert within the heap all internal constraints of $ICTRS$
// and sort them on their smallest infeasible point m according to the pruning dimension d
4: $HEAP \leftarrow$ empty
5: **for all** $ictr \in ICTRS$ **do**
6: $(found, m) \leftarrow \text{LexInfeasible}(ictr, \mathbf{true}, d, k, o)$
7: **if** $found$ **then**
8: $\text{InsertInMinHeap}(HEAP, ictr, m)$ // insert only if at least one infeasible point
9: **end if**
10: **end for**

// transfer from the heap to the list of active internal constraints
11: $ACTRS \leftarrow$ empty // all internal constraints that can interfere with the position of c
12: **while** $\text{NonEmptyHeap}(HEAP) \wedge \text{SmallestElemHeap}(HEAP) \leq_k^d c$ **do**
13: $ACTRS \leftarrow ACTRS \bigcup \text{GetAndRemoveSmallestElemHeap}(HEAP)$
14: **end while**

// check if there is an active constraint for which c is infeasible
// if this is actually the case, f will contain the forbidden region that allows to jump
15: $(infeasible, f) \leftarrow \text{GetFR}(d, k, o, c, ACTRS, \mathbf{true})$
16: **while** $b \wedge infeasible$ **do**
17: $n \leftarrow \min(n, f.\max + 1)$ // updating the vector n according to f
18: $(c, n, b) \leftarrow \text{AdjustUp}(c, n, o, d, k)$ // update position of point c to check
19: remove from $ACTRS$ all internal constraints $ictr | c >_k^d \text{LexInfeasible}(ictr, \mathbf{false}, d, k, o)$
// and possibly transfer new internal constraints that interfere with the new position of c
20: **while** $\text{NonEmptyHeap}(HEAP) \wedge \text{SmallestElemHeap}(HEAP) \leq_k^d c$ **do**
21: $ACTRS \leftarrow ACTRS \bigcup \text{GetAndRemoveSmallestElemHeap}(HEAP)$
22: **end while**
23: $(infeasible, f) \leftarrow \text{GetFR}(d, k, o, c, ACTRS, \mathbf{true})$ // check again if c is infeasible
24: **end while**
25: **if** b **then**
26: $\underline{o.x}[d] \leftarrow c[d]$
27: **end if**
28: **return** b

Algorithm 11: Adjusting the lower bound of the d^{th} coordinate of the origin of object o; $ICTRS$ is the set of internal constraints associated with object o.

PROCEDURE $\mathrm{AdjustUp}(c, n, o, d, k) : (\mathrm{int}[], \mathrm{int}[], \mathrm{bool})$

1: **for** $j \leftarrow k-1$ **downto** 0 **do**
2: $j' \leftarrow (j+d) \mod k$ // rotation wrt. d, k
3: $c[j'] \leftarrow n[j']$ // use vector n to jump
4: $n[j'] \leftarrow \overline{o.x}[j'] + 1$ // reset component of n to maximum value
5: **if** $c[j'] \leq \overline{o.x}[j']$ **then**
6: **return** $(c, n, \mathbf{true})$ // candidate point found (since did not exceed upper limit)
7: **else**
8: $c[j'] \leftarrow \underline{o.x}[j']$ // since exhausted a dimension reset component of c
9: **end if**
10: **end for**
11: **return** $(c, n, \mathbf{false})$ // no candidate point found

Algorithm 12: Moving up to the next feasible point

adapted in order to handle the start in time $o_i.start$, duration in time $o_i.duration$ and end in time $o_i.end$. Beside maintaining bound consistency for the constraint $o_i.end = o_i.start + o_i.duration$, we add an extra *time* dimension to the geometric coordinates of object o_i: when we adjust the minimum or maximum value of a geometric coordinate of o_i or when we adjust the minimum value of $o_i.start$ (CASE 1) we set this new time coordinate to $o_i.start$. Otherwise, when we want to adjust the maximum value of $o_i.end$ (CASE 2), we set this extra time coordinate to $o_i.end$. Now, in the context of CASE 1, each time we need to compute the set of infeasible points for the coordinates of o_i according to a second object o_j we first compute the set of time-points $\mathcal{T}_{ij}$ for the start of o_i, so that if the start in time of o_i is fixed to a value of $\mathcal{T}_{ij}$, o_i and o_j overlap in time. The set of points $\mathcal{T}_{ij}$ corresponds to the points of interval $[\overline{o_j.start} - \underline{o_i.duration} + 1, \underline{o_j.end} - 1]$. If this interval is empty, then the set of infeasible points for the coordinates of o_i according to o_j is empty. In the context of CASE 2, each time we need to compute the set of infeasible points for the coordinates of o_i according to a second object o_j, we first compute the set of time-points for the end of o_i, so that if the end in time of o_i is fixed to one of these time-points, o_i and o_j overlap in time. As before if this set is empty, then the set of infeasible points for the coordinates of o_i according to o_j is empty.

PROCEDURE $\mathrm{PruneMax}(o, d, k, ICTRS) : \mathrm{bool}$

1: $b \leftarrow \mathbf{true}$ // $b = \mathbf{true}$ while we have not failed
2: $c \leftarrow \overline{o.x}$ // initial position of the point
3: $n \leftarrow \underline{o.x} - 1$ // lower limits-1 in the different dimensions

// insert within the heap all internal constraints of $ICTRS$
// and sort them on their largest infeasible point m according to the pruning dimension d
4: $HEAP \leftarrow$ empty
5: **for all** $ictr \in ICTRS$ **do**
6: $(found, m) \leftarrow \mathrm{LexInfeasible}(ictr, \mathbf{false}, d, k, o)$
7: **if** $found$ **then**
8: $\mathrm{InsertInMaxHeap}(HEAP, ictr, m)$ // insert only if at least one infeasible point
9: **end if**
10: **end for**
// transfer from the heap to the list of active internal constraints
11: $ACTRS \leftarrow$ empty // all internal constraints that can interfere with the position of c
12: **while** $\mathrm{NonEmptyHeap}(HEAP) \wedge \mathrm{LargestElemHeap}(HEAP) \geq_k^d c$ **do**
13: $ACTRS \leftarrow ACTRS \bigcup \mathrm{GetAndRemoveLargestElemHeap}(HEAP)$
14: **end while**
// check if there is an active constraint for which c is infeasible
// if this is actually the case, f will contain the forbidden region that allows to jump
15: $(infeasible, f) \leftarrow \mathrm{GetFR}(d, k, o, c, ACTRS, \mathbf{false})$
16: **while** $b \wedge infeasible$ **do**
17: $n \leftarrow \max(n, f.\min - 1)$ // updating the vector n according to f
18: $(c, n, b) \leftarrow \mathrm{AdjustDown}(c, n, o, d, k)$ // update position of point c to check
19: remove from $ACTRS$ all internal constraints $ictr | c <_k^d \mathrm{LexInfeasible}(ictr, \mathbf{true}, d, k, o)$
// and possibly transfer new internal constraints that interfere with the new position of c
20: **while** $\mathrm{NonEmptyHeap}(HEAP) \wedge \mathrm{LargestElemHeap}(HEAP) \geq_k^d c$ **do**
21: $ACTRS \leftarrow ACTRS \bigcup \mathrm{GetAndRemoveLargestElemHeap}(HEAP)$
22: **end while**
23: $(infeasible, f) \leftarrow \mathrm{GetFR}(d, k, o, c, ACTRS, \mathbf{false})$ // check again if c is infeasible
24: **end while**
25: **if** b **then**
26: $\overline{o.x[d]} \leftarrow c[d]$
27: **end if**
28: **return** b

Algorithm 13: Adjusting the upper bound of the d^{th} coordinate of the origin of object o; $ICTRS$ is the set of internal constraints associated with object o.

PROCEDURE $\mathrm{AdjustDown}(c, n, o, d, k) : (\mathrm{int}[], \mathrm{int}[], \mathrm{bool})$

1: **for** $j \leftarrow k-1$ **downto** 0 **do**
2: $\quad j' \leftarrow (j+d) \mod k$ // rotation wrt. d, k
3: $\quad c[j'] \leftarrow n[j']$ // use vector n to jump
4: $\quad n[j'] \leftarrow \underline{o.x}[j'] - 1$ // reset component of n to minimum value
5: $\quad$ **if** $c[j'] \geq \underline{o.x}[j']$ **then**
6: $\quad\quad$ **return** $(c, n, \mathbf{true})$ // candidate point found (since not under lower limit)
7: $\quad$ **else**
8: $\quad\quad c[j'] \leftarrow \overline{o.x}[j']$ // since exhausted a dimension reset component of c
9: $\quad$ **end if**
10: **end for**
11: **return** $(c, n, \mathbf{false})$ // no candidate point found

Algorithm 14: Moving down to the next feasible point

6 Support for Greedy Assignment within the $geost$ Kernel

6.1 Motivation and Functionality Description

Since, for performance reasons[7], the $geost$ kernel offers a mode where it tries to fix all objects during one single propagation step, we provide a way to specify a preferred order on how to fix all the objects[8] in one single propagation step.[9] This is achieved by:

- Fixing the objects according to the order they were passed to the $geost$ kernel.
- When considering one object, fixing its shape variable as well as its coordinates:
  - According to an order on these variables that can be explicitly specified.
  - A value to assign that can either be the smallest or the largest value, also specified by the user.

  This is encoded by a term that has exactly the same structure as the term associated to an object of $geost$. The only difference consists of the fact that a variable is replaced by an expression _,[10] $\min(I)$ (respectively, $\max(I)$), where I is a strictly positive integer. The meaning is that the corresponding variable should be fixed to its minimum (respectively maximum value) in the order I. We can in fact give a list of vectors $v_1, v_2, \ldots, v_p$ in order to specify how to fix objects $o_{1+p\cdot\alpha}, o_{2+p\cdot\alpha}, \ldots, o_{p+p\cdot\alpha}$. This is illustrated by Figure 10: for instance, Part (I) specifies that we alternatively (1) fix the shape variable of an object to its maximum value (i.e., by using $\max(1)$), fix the x-coordinate of an object to its its minimum value (i.e., by using $\min(2)$), fix the y-coordinate of an object to its its minimum value (i.e., by using $\min(3)$) and (2) fix the shape variable of an object to its maximum value (i.e., by using $\max(1)$), fix the x-coordinate of an object to its its maximum value (i.e., by using $\max(2)$), fix the y-coordinate of an object to its its maximum value (i.e., by using $\max(3)$). In the example associated with Part (I) we successively fix objects o_1, o_2, o_3, o_4, o_5, o_6 by alternatively using strategies (1) (i.e., $\mathrm{object}(\_, \max(1), x[\min(2), \min(3)])$ and (2) (i.e., vector $\mathrm{object}(\_, \max(1), x[\max(2), \max(3)])$.

From an implementation point of view the main modification[11] consists of modifying lines 8 and 9 of $\mathrm{FilterObj}$ (see Algorithm 9) in order to follow the ordering imposed by the list of vectors $v_1, v_2, \ldots, v_p$. This is detailed in the next section.

[7] Experiments have shown that this allows to reduce significantly, both the time and the memory consumption.

[8] This is only a preference that the kernel may not completely follow for implementation or performance reasons. For instance, as we will see in the next section, the current implementation fix always the shape variable first.

[9] This is in fact not incompatible from using specific heuristics: at each choice point of the search space the idea is to first try to fixes every object within one single propagation step and, if this leads to failure, propagate and use the specific heuristics written in Java or in Prolog depending on the constraint system you are using.

[10] The character _ denotes the fact that the corresponding attribute is irrelevant, since for instance, we know that it is always fixed.

[11] Beside fixing the shape variable.

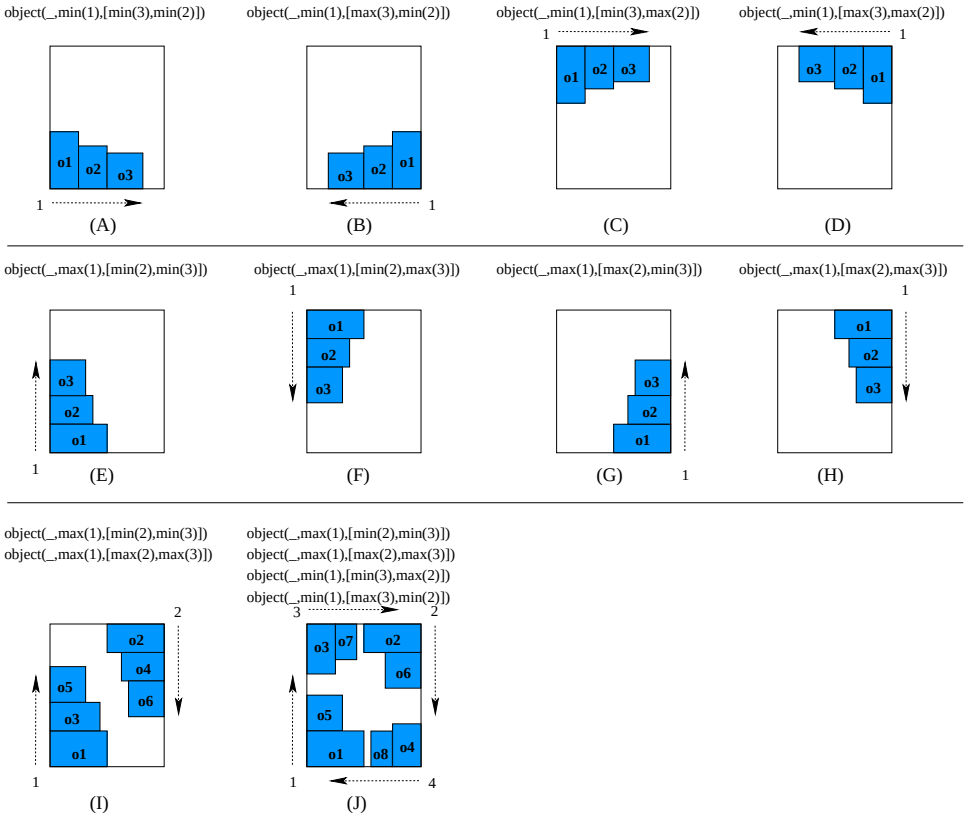


Fig. 10. Illustration of possible heuristics for fixing all objects within one single propagation step and their corresponding parameters in the context of two dimensions.

6.2 Implementation

The greedy algorithm for fixing an object o is controlled by a vector v of length $k+1$ such that:

- The shape variable $o.sid$ should be set to its minimum possible value if $v[0] < 0$, and to its maximum possible value otherwise.
- $\mathrm{abs}(v[1]) - 2$ is the most significant dimension (the one that varies the slowest) during the sweep. The values are tried in ascending order if $v[1] < 0$, and in descending order otherwise.
- $\mathrm{abs}(v[2]) - 2$ is the next most significant dimension, and its sign indicates the value order, and so on.

For example, a term `object(_,min(1),[max(3),min(4),max(2)])` is encoded as the vector $[-1, 4, 2, -3]$.

PROCEDURE FixAllObjs$(k, \mathcal{O}, \mathcal{S}, \mathcal{C}, v)$: bool
1: **for all** $ectr \in \mathcal{C}$ **do**
2: $FRAME[ectr] \leftarrow$ InitFrameExternalConstraint$(ectr, \mathcal{O}, \mathcal{S})$
3: **end for**
4: **for all** $o \in \mathcal{O}$ **do**
5: **if** $\neg$FixObj$(k, o, FRAME, \mathcal{S}, v)$ **then**
6: **return false**
7: **else**
8: **for all** external geometrical constraints $ectr$ involving o **do**
9: $FRAME[ectr] \leftarrow$ InitFrameExternalConstraint$(ectr, \mathcal{O}, \mathcal{S})$ // update the relative forbidden regions
10: **end for**
11: **end if**
12: **end for**
13: **return** `true`

Algorithm 15: Fixing all the objects, where k, $\mathcal{O}$, $\mathcal{S}$ and $\mathcal{C}$, v respectively correspond to the number of dimensions, to the objects, to the shapes, to the external geometrical constraints and to the controlling vector. Within the context of the greedy mode, this algorithm replaces Algorithm 8 (i.e., FilterCtrs) that is used in the standard propagation mode.

PROCEDURE FixObj$(k, o, FRAME, \mathcal{S}, v)$: bool

1: **if** $v[0] < 0$ **then**
2: $o.sid \leftarrow \underline{o.sid}$
3: **else**
4: $o.sid \leftarrow \overline{o.sid}$
5: **end if**
6: $ICTRS \leftarrow \emptyset$ // build the list of internal constraints attached to o
7: **for** $d \leftarrow 0$ **to** $k-1$ **do**
8: $ICTRS \leftarrow ICTRS \bigcup$ possible outbox constraints corresponding to holes of $o.x[d]$
// holes correspond to adjacent forbidden values of $\mathrm{dom}(o.x[d])$
9: **end for**
10: **for all** external geometrical constraints $ectr$ involving o **do**
11: $ICTRS \leftarrow ICTRS \bigcup \mathrm{GenInternalCtrs}(ectr, o, \mathcal{O}, \mathcal{S}, FRAME[ectr])$
12: **end for**
13: **return** PruneFix$(o, d, k, ICTRS, v[1..k])$ // we pass the vector $v[1..k]$ since we have to remove position 0 which corresponds to the shape id

Algorithm 16: Fixing all the k coordinates of a given object o according to all external geometrical constraints where o occurs; $FRAME[ectr]$ corresponds to a possible frame associated with an external constraint; $\mathcal{S}$ is the set of shapes of the $geost$ constraint; v is the controlling vector.

```
PROCEDURE PruneFix(o, k, ICTRS, v) : bool
 1: for d ← k − 1 downto 0 do
 2:    d′ ← abs(v[d]) − 2
 3:    if v[d] < 0 then
 4:       c[d′] ← o.x[d′]                          // initial position of the point
 5:       n[d′] ← ō.x[d′] + 1                      // initial next feasible pos
 6:    else
 7:       c[d′] ← ō.x[d′]                          // initial position of the point
 8:       n[d′] ← o.x[d′] − 1                      // initial next feasible pos
 9:    end if
10: end for
11: (infeasible, f) ← GetFR(abs(v[0]) − 2, k, o, c, ICTRS, true)
12: while infeasible do
13:    for d ← k − 1 downto 0 do
14:       d′ ← abs(v[d]) − 2
15:       if v[d] < 0 then
16:          n[d′] ← min(n[d′], f.max[d′] + 1)      // update next feasible pos wrt. f
17:       else
18:          n[d′] ← max(n[d′], f.min[d′] − 1)      // update next feasible pos wrt. f
19:       end if
20:    end for
21:    for d ← k − 1 downto 0 do
22:       d′ ← abs(v[d]) − 2
23:       c[d′] ← n[d′]                            // use vector n to jump
24:       if v[d] < 0 then
25:          n[d′] ← ō.x[d′] + 1                   // reset component of n to beyond limit
26:          if c[d′] < n[d′] then
27:             goto nextcand                      // new candidate point found
28:          else
29:             c[d′] ← o.x[d′]                   // since exhausted a dimension reset component of c
30:          end if
31:       else
32:          n[d′] ← o.x[d′] − 1                   // reset component of n to beyond limit
33:          if c[d′] > n[d′] then
34:             goto nextcand                      // new candidate point found
35:          else
36:             c[d′] ← ō.x[d′]                   // since exhausted a dimension reset component of c
37:          end if
38:       end if
39:    end for
40:    return false                                // no candidate point found
41:    label : nextcand
42:    (infeasible, f) ← GetFR(abs(v[0]) − 2, k, o, c, ICTRS, true)
43: end while
44: o.x ← c
45: return true
```

Algorithm 17: Fix completely all the coordinates of the origin of object o, by first starting to fix the $(\mathrm{abs}(v[0])-2)$-th coordinate of object o; $ICTRS$ is the set of internal constraints associated with object o; v is the controlling vector.

Acknowledgements

This research was conducted under European Union Sixth Framework Programme Contract FP6-034691 “Net-WMS”.

Part B: CHOCO documentation of *geost*

Armines

Geost Constraint Tutorial

Rida. S. Sadek

September 12, 2007

1 Introduction

The *geost* constraint is a global constraint that handle generically a variety of geometrical constraints.

The *geost(K, O, S, C)* constraint is given set of parameters which will define the environment of *geost*. The parameters are as follows:

K: The space dimension of the geometric objects to be handled.

O: The polymorphic (each object can have many shapes) K-dimensional objects.

S: The set of shapes that each object can have.

C: The set of geometrical constraints.

In the remaining of this tutorial we will explain by example how to implement and set problems using the *geost* constraint with choco solver.

2 Example and ways to implement it

Lets first describe a problem and then use it as an example. Consider we have 3 objects o_0, o_1, o_2 and we want to place them in a box B. Let the 3 objects be as shown in Figure 1. Given that the placement of the objects should be totally inside B this means that the domains of the origins of each object are as follows (we start from 0 this means that the placement space is from 0 to 9 on x and from 0 to 5 on y):

o_0: on x it is from 0 to 6 and on y it is from 0 to 4

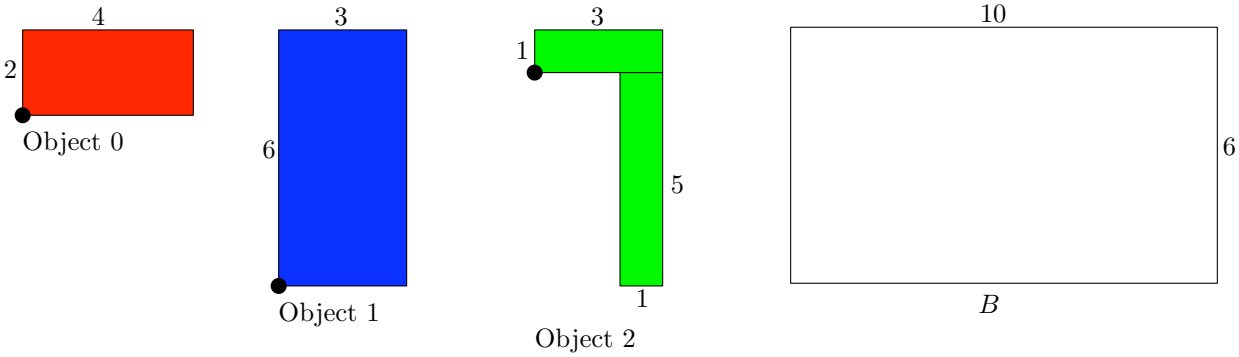


Figure 1: 3 objects in dimension 2 and the box B we want to place them in. The first 2 objects are regular rectangles and the third object is a union between 2 rectangles. This means that the first two each are made of one shifted box and the third by 2 shifted boxes

o_1: on x it is from 0 to 7 and on y it is from 0 to 0

o_2: on x it is from 0 to 7 and on y it is from 4 to 4

Please note that an Object can have many shape (polymorphism). However for the time being we haven't implemented this feature. The core of the constraint has been built taking care of this feature this is why the $ShapeId$ variable is an Integer Domain Variable ($IntDomainVar$) to be able to specify multiple shapes for the same object. We will see this shortly.

1. Set the Dimension:

To begin implementing this example we first need to specify the dimension K we are working in. This is done by assigning the dimension to a local variable that we will use later:

$int\ dim = 2;$

2. Create the Problem:

After that we create a choco problem:

$Problem\ pb = new\ Problem();$

3. Create the Objects:

Then we start by creating the objects and store them in an array or vector as such:

$Vector < Obj > obj = new\ Vector < Obj > ();$

Now we create the first object with objectId 0 and shapeId 0 with the corresponding domain variables for the origin.

$Obj\ o = new\ Obj(dim);$
$o.setObjectId(0);$

Now, as we have discussed above, to specify the $shapeId$ of the Object we give a $IntDomainVar$ however we give the inf value equal to the sup value since the polymorphism feature is not yet implemented.

$IntDomainVar\ shapeId = pb.makeEnumIntVar("sid", 0, 0);$
$IntDomainVar\ coords[] = new\ IntDomainVar[dim];$
$coords[0] = pb.makeEnumIntVar("x", 0, 6);$ //Domain of the x coordinate
$coords[1] = pb.makeEnumIntVar("y", 0, 4);$ //Domain of the y coordinate
$o.setCoordinates(coords);$
$o.setShapeId(shapeId);$

4. Create the temporal attributes:

Now before adding our object to the Vector obj we need to specify 3 more Integer Domain Variables which for the current implementation of $geost$ are not working, however we need to give them dummy values.

$o.setStart(pb.makeEnumIntVar("start", 1, 1));$
$o.setEnd(pb.makeEnumIntVar("end", 1, 1));$
$o.setDuration(pb.makeEnumIntVar("duration", 1, 1));$

5. Add the Object:

Now we are ready to add the object 0 to our obj Vector

$obj.add(o);$

6. Create the Shifted Boxes of the Object:

As you may have noticed we haven't specified the size of the rectangle but only the domain of its origin. This will come later when we specify the shifted box that correspond to the shape of this object. Now we do the same for the other 2 object o_1 and o_2 and add them to our obj vector.

To create the shapes and their shifted boxes we create the shifted boxes and associate them with the corresponding shapeId. This is done as follows, first we create a Vector called sb for example

$Vector < ShiftedBox > sb = new\ Vector < ShiftedBox > ();$

To create the shifted box for the shape 0 (that corresponds to object 0) we create 2 arrays one for the sizes of the box on each dimension and one to specify the offset of the box on each dimension.

$int[]\ sizes = \{4, 2\}$;

$int[]\ offset = \{0, 0\}$; //There is no offset since the origin of the object is the same as the origin of the shifted box.

7. Add the Shifted Boxes:

Now we add our shifted box to the *sb* Vector

$sb.add(new\ ShiftedBox(0,\ offset,\ sizes))$; //where the 0 in the creation of the shiftedBox object corresponding to the shapeId that this shiftedBox belongs to.

To create a shape which has more than one shifted box we just create a shiftedBox object for each box it has and assign the same shapeId to all of them taking care of the offset as well. for example to create the shape for Object 2 we do the following:

$int[]\ sizes1 = \{3, 1\}$;

$int[]\ offset1 = \{0, 0\}$

$int[]\ sizes2 = \{1, 5\}$;

$int[]\ offset2 = \{2, -4\}$;

$sb.add(new\ ShiftedBox(2,\ offset1,\ sizes1))$ //where the 2 in the creation of the shiftedBox object correspond to the shapeId that this shiftedBox belongs to.

$sb.add(new\ ShiftedBox(2,\ offset2,\ sizes2))$

8. Create the constraints:

Now to create the constraint we first create an array containing all the dimensions the constraint will be active in (in our example it is all dimensions) and lets name this array *ectrDim*. Then we create a list of objects that this constraint will apply to (in our example it is all objects). After that we add the constraint to a vector **ectr** that contains all the constraints we want to add. The code for these two steps is as follows:

$Vector < ExternalConstraint > ectr = new\ Vector < ExternalConstraint > ();;$

$int[]\ ectrDim = new\ int[dim];$

$for(i = 0;\ i < dim;\ i++)$

$\quad ectrDim[i] = i;$

$int[]\ objOfEctr = new\ int[obj.size()];$

$for(i = 0;\ i < obj.size();\ i++)$

$\quad objOfEctr[i] = obj.elementAt(i).getObjectId();$

All we need to do now is create the nonOverlapping constraint and add it to the **ectr** vector that holds all the constraints. this is done as follows:

$NonOverlapping\ n = new\ NonOverlapping(Constants.NON\_OVERLAPPING, ectrDim, objOfEctr);$

$ectr.add(n);$

Note that we can specify only a subset of dimensions where the constraint becomes active as well as we can specify a subset of objects that are constrained by the constraint. For example say we are working in dimension $k = 6$, meaning $Constants.DIM = 6$. And say we have 4 objects A, B, C and D with object ids 1, 2 , 3 and 4 respectiively . We want an NonOverlapping constraint for A and B but only in dimensions 1, 4 and 5 also we want an NonOverlapping constraint for C and D but only in dimensions 1 and 2. To do that we just add 2 constraints to the Setup object each constraint with the correct parameters, as follows:

$int[]\ ectrDim1 = new\ int[]\{1, 4, 5\};$ //Create the first list of dimensions

$int[]\ objOfEctr1 = new\ int[]\{1, 2\};$ //Create the first list of objects

$NonOverlapping\ n1 = new\ NonOverlapping(Constants.NON\_OVERLAPPING, ectrDim1, objOfEctr1);$

Now we create the second constraint:

$int[]\ ectrDim2 = new\ int[]\{1, 2\};$ //Create the first list of dimensions

$int[]\ objOfEctr1 = new\ int[]\{3, 4\};$ //Create the first list of objects

$NonOverlapping\ n2 = new\ NonOverlapping(Constants.NON\_OVERLAPPING, ectrDim2, objOfEctr2);$

Now we just add the constraints created to the textbfectr vector:

$ectr.add(n1)$;

$ectr.add(n2)$;

10. post the Geost Constraint:

We are almost done, we create the Array of Variables $vars$ to make choco happy (just take a look at the GeostTest.java example in the code) and then post the *geost* constraint to the choco problem as follows:

$pb.post(new\ Geost\_Constraint(vars, dim, obj, sb, ectr))$;

The Following is a Java example, this file can be found in the global package of the geost source code.
In this class I prepare 4 types of ways to use goest and give ccode to show how to do this.
The First way shows how to setup geost to read from a text file a problem.
The second way shows how to create a custom problem from class using java code.
The third way shows how to generate random problems and solve them. Please note that the power of the random generator is very limited
The fourth way shows how to post multiple geost constraints to the same choco problem.
Finally, I created a function called solve that is a bit generic. This is totally however I just wanted to seperate the solving related instructions from the example code.

```
package global;
import java.util.Vector;

import geometricPrim.Obj;
import geometricPrim.ShiftedBox;
import global.VRMLwriter;

import choco.Problem;
import choco.integer.IntDomainVar;
import externalConstraints.ExternalConstraint;
import externalConstraints.NonOverlapping;
import global.Constants;
import global.Geost_Constraint;
import global.InputParser;
import global.MyVarSelector;
import global.RandomProblemGenerator;
import global.SolutionTester;

/*
In the example I have implemented a function called solve just to make things consistent.
We can not use it of course and just say pb.solve().
What the local solve method does is that it takes care of  whether we want to solve for
the first solution only or for all solutions according to the mode variable.
Also it manages the writing of vrmlFiles for visualization and the test of solutions of
different geost constraints posted to the same choco problem.
*/


public class GeostTest {

	int dim;
	int mode;
	public static void main(String[] args) {

		//the parameters are dimension and runMode respectively,
		//see constructor for details
		GeostTest gt = new GeostTest(3, 1);
	}

	public GeostTest(int dim, int mode)
	{
		//The dim parameter is to specidy the dimension of the problem.
		//The mode parameter represents the run mode of the solver,
		//0 is to solve for all solution and 1 to solve for the first
		//solution
		this.dim = dim;
		this.mode = mode;
	}

	public boolean Use_GEOST_From_Text_File(String inputFileName)
	{

		Vector<Geost_Constraint> g = new Vector<Geost_Constraint>();
		//If the input is a text file similar to the input.txt
		//Create the InputParser and parse the input file
		InputParser parser = new InputParser(inputFileName, this.dim);
		try {
			parser.parse();
		} catch (Exception e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		//get the problem from the InputParser object
		Problem pb = parser.getProblem();

		//create a vector to hold in it all the external constraints
		//we want to add to geost
		Vector<ExternalConstraint> ectr = new Vector<ExternalConstraint>();

		//////////////Create the needed external constraints//////////////

		//first of all create a array of intergers containing all the
		//dimensions where the constraint will be active
		int[] ectrDim = new int[this.dim];
		for (int i = 0; i < this.dim; i++)
			ectrDim[i] = i;

		//Create an array of object ids representing all the objects that the
```

```
        //external constraint will be applied to
        int[] objOfEctr = new int[parser.getObjects().size()];
        for(int i = 0; i < parser.getObjects().size(); i++)
        {
                objOfEctr[i] = parser.getObjects().elementAt(i).getObjectId();
        }

        //Create the external constraint, in our case it is the
        //NonOverlapping constraint (it is the only one
        //implemented for now)
        NonOverlapping n = new
                NonOverlapping(Constants.NON_OVERLAPPING, ectrDim, objOfEctr);

        //add the created external constraint to the vector we created
        ectr.add(n);

        ////////Create the array of variables to make choco happy////////

        //vars will be stored as follows:
        //  object 1 coords(so k coordinates), sid, start, duration, end,
        //  object 2 coords(so k coordinates), sid, start, duration, end,
        //     and so on..
        //To retrieve the index of a certain variable, the formula is
        //(nb of the object in question = objId assuming objIds are
        //consecutive and start from 0) * (k + 4) + number of the variable
        //wanted the number of the variable wanted is decided
        //as follows: 0 ... k-1 (the coords), k (the sid), k+1 (start),
        //          k+2 (duration), k+3 (end)

        //Number of domain variables to represent the origin of all objects
        int originOfObjects = parser.getObjects().size() * this.dim;
        //each object has 4 other variables: shapeId, start, duration; end
        int otherVariables = parser.getObjects().size() * 4;
        IntDomainVar[] vars = new IntDomainVar[originOfObjects +
                                                        otherVariables];

        for(int i = 0; i < parser.getObjects().size(); i++)
        {
                for (int j = 0; j < this.dim; j++)
                {
                        vars[(i * (this.dim + 4)) + j] =
                                parser.getObjects().elementAt(i).getCoord(j);
                }
                vars[(i * (this.dim + 4)) + this.dim] =
                                parser.getObjects().elementAt(i).getShapeId();
                vars[(i * (this.dim + 4)) + this.dim + 1] =
                                parser.getObjects().elementAt(i).getStart();
                vars[(i * (this.dim + 4)) + this.dim + 2] =
                                parser.getObjects().elementAt(i).getDuration();
                vars[(i * (this.dim + 4)) + this.dim + 3] =
                                parser.getObjects().elementAt(i).getEnd();
        }

        //////////////Create the geost constraint//////////////
        Geost_Constraint geost = new Geost_Constraint
                                (vars, this.dim, parser.getObjects(),
                                parser.getShiftedBoxes(), ectr);

        //////////////Add the constraint to the choco problem//////
        pb.post(geost);
        g.add(geost);
        solve(pb, g);

        return true;
}

public boolean CustomProblem()
{
        Vector<Geost_Constraint> g = new Vector<Geost_Constraint>();

        int lengths [] = {5, 3, 2};
        int widths[] = {2 ,2 ,1};
        int heights[] = {1,1,1};

        int nbOfObj = 3;

        //create the choco problem
        Problem pb = new Problem();

        //Create Objects
        Vector<Obj> obj2 = new Vector<Obj>();

        for (int i = 0; i < nbOfObj; i++)
        {
                IntDomainVar shapeId = pb.makeEnumIntVar("sid", i, i);
                IntDomainVar coords[] = new IntDomainVar[this.dim];
                for(int j = 0; j < coords.length; j++)
                {
                        coords[j] = pb.makeEnumIntVar("x" + j, 0, 20);
                }
                Obj o2 = new Obj(this.dim);
```

```
            o2.setObjectId(i);
            o2.setCoordinates(coords);
            o2.setShapeId(shapeId);
            o2.setStart(pb.makeEnumIntVar("start", 1, 1));
            o2.setEnd(pb.makeEnumIntVar("end", 1, 1));
            o2.setDuration(pb.makeEnumIntVar("duration", 1, 1));
            obj2.add(o2);
        }

        //create shiftedboxes and add them to corresponding shapes
        Vector<ShiftedBox> sb2 = new Vector<ShiftedBox>();
        int h = 0;
        while(h< nbOfObj)
        {

            int[] l = {lengths[h], heights[h] ,widths[h]};
            int [] t ={0, 0, 0};


            sb2.add(new ShiftedBox(h,t,l));
            h++;
        }

        //Create the external constraints vecotr
        Vector<ExternalConstraint> ectr2 = new Vector<ExternalConstraint>();
        //create the list od dimensions for the external constraint
        int[] ectrDim2 = new int[this.dim];
        for (int d = 0; d < 3; d++)
            ectrDim2[d] = d;

        //create the list of object ids for the external constraint
        int[] objOfEctr2 = new int[nbOfObj];
        for(int d = 0; d < nbOfObj; d++)
        {
            objOfEctr2[d] = obj2.elementAt(d).getObjectId();
        }

        //create the external constraint of type non overlapping
        NonOverlapping n2 = new NonOverlapping
                    (Constants.NON_OVERLAPPING, ectrDim2, objOfEctr2);
        //add the external constraint to the vector
        ectr2.add(n2);

    ////////Create the array of variables to make choco happy////////

      //vars will be stored as follows:
      //  object 1 coords(so k coordinates), sid, start, duration, end,
      //  object 2 coords(so k coordinates), sid, start, duration, end,
      //     and so on..
      //To retrieve the index of a certain variable, the formula is
      //(nb of the object in question = objId assuming objIds are
      //consecutive and start from 0) * (k + 4) + number of the variable
       //wanted the number of the variable wanted is decided
      //as follows: 0 ... k-1 (the coords), k (the sid), k+1 (start),
      //           k+2 (duration), k+3 (end)

        //Number of domain variables to represent the origin of all objects
        int originOfObjects2 = nbOfObj * this.dim;
        //each object has 4 other variables: shapeId, start, duration; end
        int otherVariables2 = nbOfObj * 4;
        IntDomainVar[] vars2 = new IntDomainVar[originOfObjects2 +
                                                    otherVariables2];

        for(int i = 0; i < nbOfObj; i++)
        {
            for (int j = 0; j < this.dim; j++)
            {
                vars2[(i * (this.dim + 4)) + j] =
                        obj2.elementAt(i).getCoord(j);
            }
            vars2[(i * (this.dim + 4)) + this.dim] =
                    obj2.elementAt(i).getShapeId();
            vars2[(i * (this.dim + 4)) + this.dim + 1] =
                    obj2.elementAt(i).getStart();
            vars2[(i * (this.dim + 4)) + this.dim + 2] =
                    obj2.elementAt(i).getDuration();
            vars2[(i * (this.dim + 4)) + this.dim + 3] =
                    obj2.elementAt(i).getEnd();
        }

        //create the geost constraint object
        Geost_Constraint geost2 =
            new Geost_Constraint(vars2, this.dim, obj2, sb2, ectr2);
        //post the geost constraint to the choco problem
        pb.post(geost2);
        g.add(geost2);

        solve(pb, g);

        return true;
```

```
}

public void RandomProblemGeneration()
{
	Vector<Geost_Constraint> g = new Vector<Geost_Constraint>();

	//nb of objects, shapes, shifted boxes and maxLength respectively
	//The nb of Obj should be equal to nb Of shapes for NOW.
	//as For the number of the shifted Boxes it should be
	//greater or equal to the nb of Objects

	RandomProblemGenerator rp =
		new RandomProblemGenerator(this.dim, 7, 7, 7, 25);
	rp.generateProb();

	Problem pb = rp.getPb();

	Vector<ExternalConstraint> ectr = new Vector<ExternalConstraint>();
	int[] ectrDim = new int[this.dim];
	for (int i = 0; i < this.dim; i++)
		ectrDim[i] = i;

	int[] objOfEctr = new int[rp.getObjects().size()];
	for(int i = 0; i < rp.getObjects().size(); i++)
	{
		objOfEctr[i] = rp.getObjects().elementAt(i).getObjectId();
	}

	NonOverlapping n = new NonOverlapping
			(Constants.NON_OVERLAPPING, ectrDim, objOfEctr);
	ectr.add(n);

	//Number of domain variables to represent the origin of all objects
	int originOfObjects2 =
		rp.getObjects().size() * this.dim;
	//each object has 4 other variables: shapeId, start, duration; end
	int otherVariables2 = rp.getObjects().size() * 4;
	IntDomainVar[] vars3 =
		new IntDomainVar[originOfObjects2 + otherVariables2];

	for(int i = 0; i < rp.getObjects().size(); i++)
	{
		for (int j = 0; j < this.dim; j++)
		{
			vars3[(i * (this.dim + 4)) + j] =
				rp.getObjects().elementAt(i).getCoord(j);
		}
		vars3[(i * (this.dim + 4)) + this.dim] =
			rp.getObjects().elementAt(i).getShapeId();
		vars3[(i * (this.dim + 4)) + this.dim + 1] =
			rp.getObjects().elementAt(i).getStart();
		vars3[(i * (this.dim + 4)) + this.dim + 2] =
			rp.getObjects().elementAt(i).getDuration();
		vars3[(i * (this.dim + 4)) + this.dim + 3] =
			rp.getObjects().elementAt(i).getEnd();
	}

	Geost_Constraint geost3 =
		new Geost_Constraint(vars3, this.dim, rp.getObjects(),
					rp.getSBoxes(), ectr);
	pb.post(geost3);
	g.add(geost3);

	/*
	We could also use functions to write things to files
	in two formats. One to be used as input to the parser
	and one to be used for humans to read.
	geost3.getStp().printToFileInputFormat
		("PathToTheOutput_Input.txt");
	geost3.getStp().printToFileHumanFormat
		("PathToTheOutput_Human.txt");
	*/

	solve(pb, g);
}

public void MultipleGeostConstraintsInSameProblem()
{
	Vector<Geost_Constraint> g =
		new Vector<Geost_Constraint>();
	//This example is to show how to give to the same choco problem
	//2 different geost constraints.

	//////////////////THE FIRST PROBLEM//////////////////

	//If the input is a text file similar to the input.txt
	//Create the InputParser and parse the input file
	InputParser parser = new InputParser
		("/Users/ridasadek/Documents/geostInOutFiles/input3D.txt",
		 this.dim);
	try {
```

```
        parser.parse();
} catch (Exception e) {
        e.printStackTrace();
}

//get the problem from the InputParser object
Problem pb = parser.getProblem();

//create a vector to hold in it all the external constraints
//we want to add to geost
Vector<ExternalConstraint> ectr =
        new Vector<ExternalConstraint>();

//////////////Create the needed external constraints///////////

//first of all create a array of intergers containing all
//the dimensions where the constraint will be active
int[] ectrDim = new int[this.dim];
for (int i = 0; i < this.dim; i++)
        ectrDim[i] = i;

//Create an array of object ids representing all the objects
//that the external constraint will be applied to
int[] objOfEctr = new int[parser.getObjects().size()];
for(int i = 0; i < parser.getObjects().size(); i++)
{
        objOfEctr[i] = parser.getObjects().elementAt(i).getObjectId();
}

//Create the external constraint, in our case it is the
//NonOverlapping constraint (it is the only one implemented for now)
NonOverlapping n = new NonOverlapping
                (Constants.NON_OVERLAPPING, ectrDim, objOfEctr);

//add the created external constraint to the vector we created
ectr.add(n);

////////Create the array of variables to make choco happy//////

//See the above examples to understand how this
//array of variables is created
int originOfObjects =
        parser.getObjects().size() * this.dim;
int otherVariables =
        parser.getObjects().size() * 4;
IntDomainVar[] vars =
        new IntDomainVar[originOfObjects + otherVariables];

for(int i = 0; i < parser.getObjects().size(); i++)
{
        for (int j = 0; j < this.dim; j++)
        {
                vars[(i * (this.dim + 4)) + j] =
                        parser.getObjects().elementAt(i).getCoord(j);
        }
        vars[(i * (this.dim + 4)) + this.dim] =
                parser.getObjects().elementAt(i).getShapeId();
        vars[(i * (this.dim + 4)) + this.dim + 1] =
                parser.getObjects().elementAt(i).getStart();
        vars[(i * (this.dim + 4)) + this.dim + 2] =
                parser.getObjects().elementAt(i).getDuration();
        vars[(i * (this.dim + 4)) + this.dim + 3] =
                parser.getObjects().elementAt(i).getEnd();
}

////////////////Create the geost constraint//////////////////////
Geost_Constraint geost =
        new Geost_Constraint(vars, this.dim, parser.getObjects(),
                                        parser.getShiftedBoxes(), ectr);

////////////////Add the constraint to the choco problem//////////////
pb.post(geost);
g.add(geost);

//////////////////////////THE SECOND PROBLEM//////////////////

int lengths [] = {5, 3, 2};
int widths[] = {2 ,2 ,1};
int heights[] = {1,1,1};

int nbOfObj = 3;

//Create Objects
Vector<Obj> obj2 = new Vector<Obj>();

for (int i = 0; i < nbOfObj; i++)
{
        IntDomainVar shapeId = pb.makeEnumIntVar("sid", i, i);
        IntDomainVar coords[] = new IntDomainVar[this.dim];
        for(int j = 0; j < coords.length; j++)
        {
```

```
                coords[j] = pb.makeEnumIntVar("x" + j, 3, 20);
            }
            Obj o2 = new Obj(this.dim);
            o2.setObjectId(i);
            o2.setCoordinates(coords);
            o2.setShapeId(shapeId);
            o2.setStart(pb.makeEnumIntVar("start", 1, 1));
            o2.setEnd(pb.makeEnumIntVar("end", 1, 1));
            o2.setDuration(pb.makeEnumIntVar("duration", 1, 1));
            obj2.add(o2);
        }

        //create shiftedboxes and add them to corresponding shapes
        Vector<ShiftedBox> sb2 = new Vector<ShiftedBox>();
        int h = 0;
        while(h< nbOfObj)
        {

            int[] l = {lengths[h], heights[h] ,widths[h]};
            int [] t ={0, 0, 0};


            sb2.add(new ShiftedBox(h,t,l));
            h++;
        }

        Vector<ExternalConstraint> ectr2 =
            new Vector<ExternalConstraint>();
        int[] ectrDim2 = new int[this.dim];
        for (int d = 0; d < 3; d++)
            ectrDim2[d] = d;

        int[] objOfEctr2 = new int[nbOfObj];
        for(int d = 0; d < nbOfObj; d++)
        {
            objOfEctr2[d] = obj2.elementAt(d).getObjectId();
        }

        NonOverlapping n2 = new NonOverlapping
                (Constants.NON_OVERLAPPING, ectrDim2, objOfEctr2);
        ectr2.add(n2);

        //See the above examples to understand how this
        //array of variables is created
        int originOfObjects2 = nbOfObj * this.dim;
        int otherVariables2 = nbOfObj * 4;
        IntDomainVar[] vars2 =
            new IntDomainVar[originOfObjects2 + otherVariables2];

        for(int i = 0; i < nbOfObj; i++)
        {
            for (int j = 0; j < this.dim; j++)
            {
                vars2[(i * (this.dim + 4)) + j] =
                    obj2.elementAt(i).getCoord(j);
            }
            vars2[(i * (this.dim + 4)) + this.dim] =
                obj2.elementAt(i).getShapeId();
            vars2[(i * (this.dim + 4)) + this.dim + 1] =
                obj2.elementAt(i).getStart();
            vars2[(i * (this.dim + 4)) + this.dim + 2] =
                obj2.elementAt(i).getDuration();
            vars2[(i * (this.dim + 4)) + this.dim + 3] =
                obj2.elementAt(i).getEnd();
        }
        Geost_Constraint geost2 =
            new Geost_Constraint(vars2, this.dim, obj2, sb2, ectr2);
        pb.post(geost2);
        g.add(geost2);

        solve(pb, g);
    }

    private boolean solve(Problem pb, Vector<Geost_Constraint> g)
    {
    //If needed to test whether the solution is correct (for
        //non overlapping in all dimensons and all objects)
        Vector<SolutionTester> tester = new Vector<SolutionTester>();
        for(int i = 0; i< g.size(); i++)
        {
             SolutionTester s =
                new SolutionTester(g.elementAt(i).getStp(),
                                   g.elementAt(i).getCst());
             tester.add(s);
        }

        //Still need to modify the MyVarSelector to handle
        //all the added geost constraints variables.
        if (g.size() == 1)
            pb.getSolver().setVarSelector(new MyVarSelector
                                (g.elementAt(0).getStp(),
```

```
                                    g.elementAt(0).getCst()));

            if(this.mode == 0)
            {
                  //for all solutions
                  System.out.println("Getting all Solutions ...");
                  //solve
                  if (pb.solve() == Boolean.TRUE) {
                        do {
                              if(pb.isCompletelyInstantiated())
                              {
                                    for(int i = 0; i < g.size(); i++)
                                    {
                                          if(!tester.elementAt(i).testSolution())
                                          {
                                                System.err.println("Wrong Solution found");
                                                return false;
                                          }
                                          else
                                          {
                                                //Specify the  vrml output folder, meaning
                                                //the folder we will be writing
                                                //the vrml file to
                                                g.elementAt(i).getCst().setVRML_OUTPUT_FOLDER
                                                      ("PathToOutputVrmlFolder");

                                                //write the vrml file ".wrl" so that we can
                                                //visualize it later if we want to
                                                VRMLwriter.printVRML3D(g.elementAt(i).getStp(),
                                                      g.elementAt(i).getCst(),"solution"+i,
                                                      pb.getSolver().getNbSolutions());
                                          }
                                    }

                              }

                        } while(pb.nextSolution() == Boolean.TRUE);
                  }
            }
            else if(this.mode == 1)
            {
                  //for the first solution
                  System.out.println("Getting first Solution ...");
                  //solve
                  pb.solve();
                  for(int i = 0; i < g.size(); i++)
                  {
                        //Specify the  vrml output folder, meaning the
                        //folder we will be riting the vrml file to
                        g.elementAt(i).getCst().setVRML_OUTPUT_FOLDER
                              ("PathToOutputVrmlFolder");
                        //write the vrml file ".wrl" so that we package global;
                        //can visualize it later if we want to
                        VRMLwriter.printVRML3D(g.elementAt(i).getStp(),
                                    g.elementAt(i).getCst(),"solution", i);
                        //test the solution
                        if(!tester.elementAt(i).testSolution())
                        {
                              System.err.println("Wrong Solution foud");
                              return false;
                        }
                        else
                              System.out.println(tester.elementAt(i).testSolution());
                        //print the solution to a file easily read by humans
                        g.elementAt(i).getStp().printToFileHumanFormat
                              ("PathToFileOutput");
                  }

            }
            System.out.println("NbSol: " + pb.getSolver().getNbSolutions());
            return true;
      }

}
```

Part C: SICStus documentation of *geost*

SICS

```
ex([O1,Y1a,Y1b,Y1c,
    O2,Y2a,Y2b,Y2c,Y2d,
    O3,Y3a,Y3b,Y3c,Y3d,
    O4,Y4a,Y4b,Y4c],
   Synch) :-
        domain([Y1a,Y1b,Y1c,
                Y2a,Y2b,Y2c,Y2d,
                Y3a,Y3b,Y3c,Y3d,
                Y4a,Y4b,Y4c], 1, 5),
        O1 in 1..28,
        O2 in 1..26,
        O3 in 1..22,
        O4 in 1..25,
        disjoint2([t(1,1,5,1),    t(20,4,5,1),
                   t(1,1,4,1),    t(14,4,4,1),
                   t(1,2,3,1),    t(24,2,3,1),
                   t(1,2,2,1),    t(21,1,2,1),
                   t(1,3,1,1),    t(14,2,1,1),
                   t(O1,3,Y1a,1),
                   t(O1,3,Y1b,1),
                   t(O1,3,Y1c,1),
                   t(O2,5,Y2a,1),
                   t(O2,5,Y2b,1),
                   t(O2,5,Y2c,1),
                   t(O2,5,Y2d,1),
                   t(O3,9,Y3a,1),
                   t(O3,9,Y3b,1),
                   t(O3,9,Y3c,1),
                   t(O3,9,Y3d,1),
                   t(O4,6,Y4a,1),
                   t(O4,6,Y4b,1),
                   t(O4,6,Y4c,1)],
                  [synchronization(Synch)]).
```

The file `library('clpfd/examples/squares.pl')` contains an example where `disjoint2/2` is used for tiling squares.

`geost(+`*`Objects`*`,+`*`Shapes`*`)`
`geost(+`*`Objects`*`,+`*`Shapes`*`,+`*`Options`*`)`

constrains the location in space of non-overlapping multi-dimensional *Objects*, each of which taking a shape among a set of *Shapes*.

Each shape is defined as a finite set of *shifted boxes*, where each shifted box is described by a box in a k-dimensional space at the given offset with the given sizes. A shifted box is described by a ground term `sbox(`*`Sid`*`,`*`Offset`*`,`*`Size`*`)` where *Sid*, an integer, is the shape id; *Offset*, a list of k integers, denotes the offset of the shifted box from the origin of the object; and *Size*, a list of k integers greater than zero, denotes the size of the shifted box. Then, a *shape* is a collection of shifted boxes all sharing the same shape id. Note that the

shifted boxes associated with a given shape may or may not overlap. *Shapes* is thus the list of such `sbox/3` terms.

Each object is described by a term `object(`*`Oid`*`,`*`Sid`*`,`*`Origin`* where *Oid*, an integer, is the unique object id; *Sid*, an integer or domain variable, is the shape id; and *Origin*, a list of integers or domain variables, is the origin coordinate of the object. If *Sid* is nonground, the object is said to be *polymorphic*. The possible valies for *Sid* are the shape ids that occur in *Shapes*. *Objects* is thus the list of such `object/3` terms.

Options is a list of zero or more of the following, where *Boolean* must be `true` or `false` (`false` is the default):

`lex(`*`Boolean`*`)`
: If `true`, for any two objects *O1* and *O2* such that they have the same shape id and *O1* occurs before *O2* in *Objects*, the origin coordinate of *O1* is constrained to be lexicographically less than or equal to the origin coordinate of *O2*.

`cumulative(`*`Boolean`*`)`
: If `true`, redundant reasoning methods are enabled, based on projecting the objects onto each dimension.

`longest_hole(`*`Boolean`*`)`
: If `true`, the filtering algorithm computes and uses information about holes that can be tolerated without necessarily failing the constraint.

`parconflict(`*`Boolean`*`)`
: If `true`, redundant reasoning methods are enabled, based on computing the number of items that can be put in parallel in the different dimensions.

`visavis(`*`Boolean`*`)`
: If `true`, a redundant method is enabled that dynamically detect holes that will necessarily fail the constraint.

`corners(`*`Boolean`*`)`
: If `true`, a redundant method is enabled that reasons in terms on borders that impinge on the corners of objects. This method has not been shown to pay off experimentally.

`task_intervals(`*`Boolean`*`)`
: If `true`, a redundant reasoning method is enabled that detects overcrowded and undercrowded regions of the placement space. This method has not been shown to pay off experimentally.

`dynamic_programming(`*`Boolean`*`)`
: If `true`, a redundant reasoning method is enabled that solves a knapsack problem for each column of the projection of the objects onto each dimension. This method has pseudo-polynomial complexity but can be quite powerful.

`polymorphism(`*Boolean*`)`

If `true`, a reasoning method is enabled that is relevant in the context of polymorphic objects and no slack. The method detects parts of the placement space that cannot be filled and thus fails the constraint.

`fixall(`*Flag*`,`*Patterns*`)`

If given, *Flag* is an integer or domain variable in `0..1`. If *Flag* equals 1, either initially or by binding *Flag* during search, the constraint switches behavior into greedy assignment mode. The greedy assignment will either succeed and assign all shape ids and origin coordinates to values that satisfy the constraint, or merely fail. *Flag* is never bound by the constraint; its sole function is to control the behavior of the constraint.

Greedy assignment is done one object at a time, in the order of *Objects*. The assignment per object is controlled by *Patterns*, which should be a list of one or more pattern terms of the form `object(_,SidSpec,OriginSpec)`, where *SidSpec* is a term `min(`*I*`)` or `max(`*I*`)`, *OriginSpec* is a list of k such terms, and I is a unique integer between 1 and k+1.

The meaning of the pattern is as follows. The variable in the position of `min(1)` or `max(1)` is fixed first; the variable in the position of `min(2)` or `max(2)` is fixed second; and so on. `min(`*I*`)` means trying values in ascending order; `max(`*I*`)` means descending order.

If *Patterns* contains m pattern, then object 1 is fixed according to pattern 1, ..., object m is fixed according to pattern m, object m+1 is fixed according to pattern 1, and so on. For example, suppose that the following option is given:

```
fixall(F, [object(_,min(1),[min(3),max(2)]),
           object(_,max(1),[min(2),max(3)])])
```

Then, if the program binds `F` to 1, the constraint enters greedy assignment mode and endeavors to fix all objects as follows.

- For object 1, 3, ..., (a) the shape is fixed to the smallest possible value, (b) the Y coordinate is fixed to the largest possible value, (c) the X coordinate is fixed to the smallest possible value.
- For object 2, 4, ..., (a) the shape is fixed to the largest possible value, (b) the X coordinate is fixed to the smallest possible value, (c) the Y coordinate is fixed to the largest possible value.

Suppose that you have a placement problem where you are only interested in finding out whether a solution exists or not, and what that solution is. Then the search space can be reduced by *domination constraints*, i.e. constraints that rule out solutions that are dominated by some other solution, with a guarantee that not all solutions are ruled out. The following auxiliary predicates will post domination constraints that are valid for 2-dimensional placement problem modelled with `geost/[2,3]` where each object consists of a single rectangle,

and the origin coordinates are completely unrestrained within the problem's placement space.

Suppose in particular that you want to find the smallest rectangle in which a given set of rectangles can be packed without overlap. This typically leads to solving a series of subproblems with non-decreasing size of the placement space. In this scenario, a heavy computation can be factored out of the computation of domination constraints and be done once for the whole series. For this reason, we provide two prediactes: `geost_domination_data/3`, which performs a heavy, factorable computation, and `geost_domination_post/4`, which uses the output of the former and posts the actual domination constraints.

`geost_domination_data(+`*Sizes*`, [+`*MaxX*`,+`*MaxY*`], -`*Data*`)`
: *Sizes* is a list of pairs `[`*Length*`,`*Height*`]` of rectangles that will be used in a 2-dimensional placement problem, or in a series of such problems. *MaxX* and *MaxY* resp. are the maximum length resp. height of the placement space of the problem or series of problems. *Data* is unified with a term suitable for passing to `geost_domination_post/4`.

`geost_domination_post(+`*Origins*`, +`*Sizes*`, [+`*MaxX*`,+`*MaxY*`], +`*Data*`)`
: *Origins* is a list of pairs `[`*X*`,`*Y*`]` of origin coordinates of the objects of a 2-dimensional placement problem. *Sizes* is a list of pairs `[`*Length*`,`*Height*`]` of the corresponding rectangles, in the same order. *MaxX* and *MaxY* resp. are the length resp. height of the placement space of the problem. *Data* is a term passed from `geost_domination_data/3`. This predicate posts domination constraints, as explained above.

The following constraints express the fact that several vectors of domain variables are in ascending lexicographic order:

`lex_chain(+`*Vectors*`)`
`lex_chain(+`*Vectors*`,+`*Options*`)`
: where *Vectors* is a list of vectors (lists) of domain variables with finite bounds or integers. The constraint holds if *Vectors* are in ascending lexicographic order.

  Options is a list of zero or more of the following:

  `op(`*Op*`)`
  : If *Op* is the atom `#=<` (the default), the constraints holds if *Vectors* are in non-descending lexicographic order. If *Op* is the atom `#<`, the constraints holds if *Vectors* are in strictly ascending lexicographic order.

  `increasing`
  : This option imposes the additional constraint that each vector in *Vectors* be sorted in strictly ascending order.

  `among(`*Least*`,`*Most*`,`*Values*`)`
  : If given, *Least* and *Most* should be integers such that $0 \leq Least \leq Most$ and *Values* should be a list of distinct integers. This option imposes the additional constraint on each vector in *Vectors* that at least *Least* and at most *Most* elements belong to *Values*.

Part D: Exploitation of *geost*

KLS-Optim

1 Impacts and concrete utilisation

1.1. Academics

KLS OPTIM gives industrial constraint programming lectures to academics mainly for high engineering schools or in the last year of the LMD cycle. The objectives of the course are:

- Introduction of the constraint programming system Choco.
- Introduction of novel modelling techniques of combinatorial industrial problems.
- Modelling of scheduling and assignment problems in production.
- Modelling of packing problems using global constraints developed in the context of Net-WMS project.

Durations of lectures range from 2 hours to 3 hours.

1.2 Research

KLS OPTIM works with other research teams on combinatorial problems. All research and prototypes are developed with the constraint programming system Choco.

1.3 KLS OPTIM

KLS OPTIM uses the constraint programming system Choco to develop business components. KLS OPTIM and EMN teams put a lot of synergy in the Choco project. KLS OPTIM provides expertise, requirements and recommendations of developments of new constraints to enrich the Choco system. All the implementations are carried on by EMN. KLS OPTIM decides to put more supports by maintaining a release and building a library of tests to automate the test phase when Choco is enriched with new constraints or when new improvements are made available.

The figure below shows the architecture and the expected results of the Net-WMS project. This section deals with the palletizer business component. The solvers of the business components are developed completely in Java using the constraint programming Choco system. The important contribution of the Net-WMS is the global constraint Geost. The basic version is used by KLS OPTIM and some of the components are deployed in the industry.

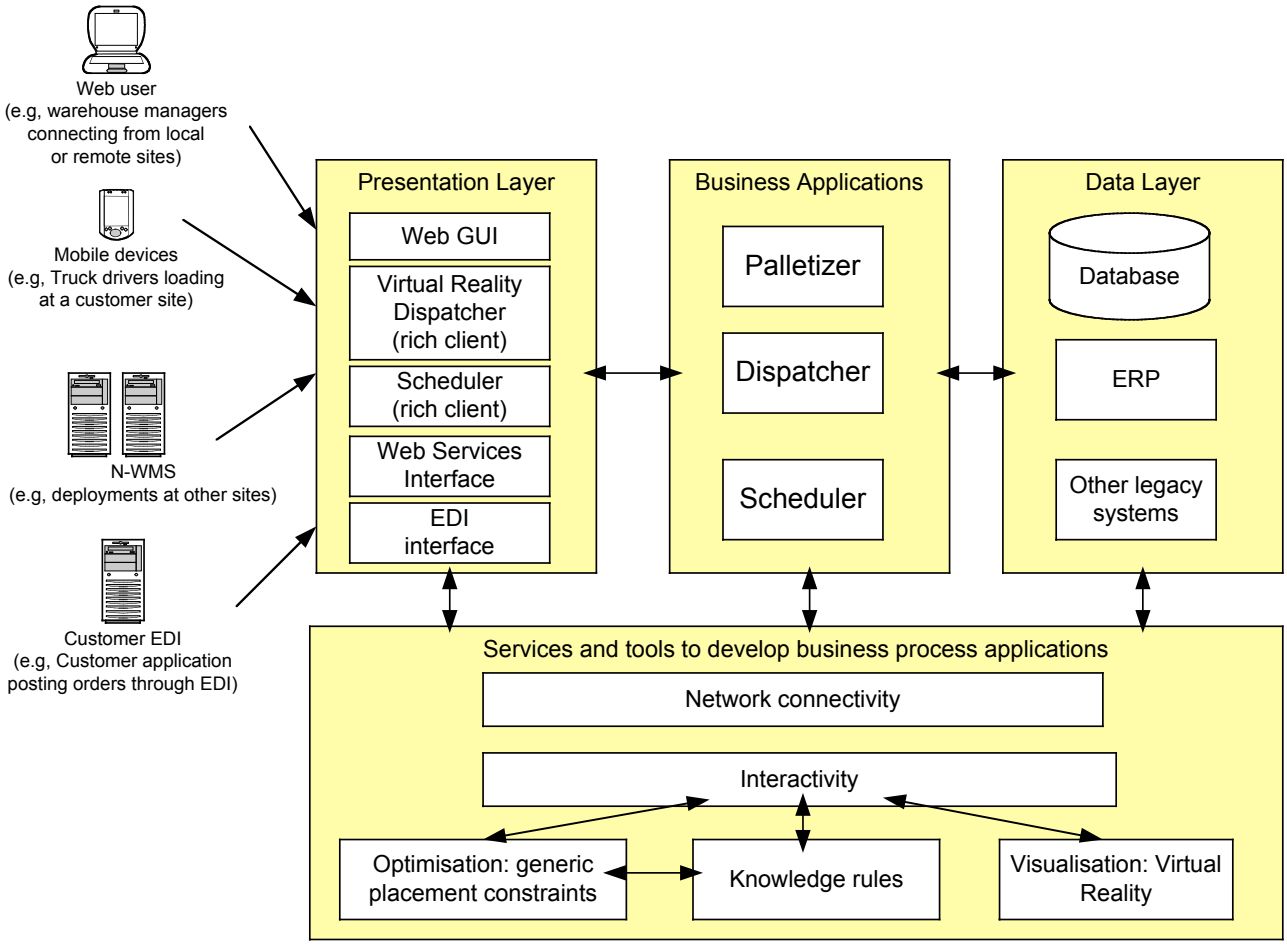


Figure 1: **Innovative Net-WMS J2EE architecture**

In the context of the Net-WMS project, KLS OPTIM is developing two business components

- Optim Pallet: The palletizer takes as input a set of cartons or conditionings, a set of parameters and directives. It produces a set of optimised pallets.
- Optim Truck: the system takes as input a set of pallets, a set of parameters and directives. It produces an optimal loading plan of pallets in trucks.

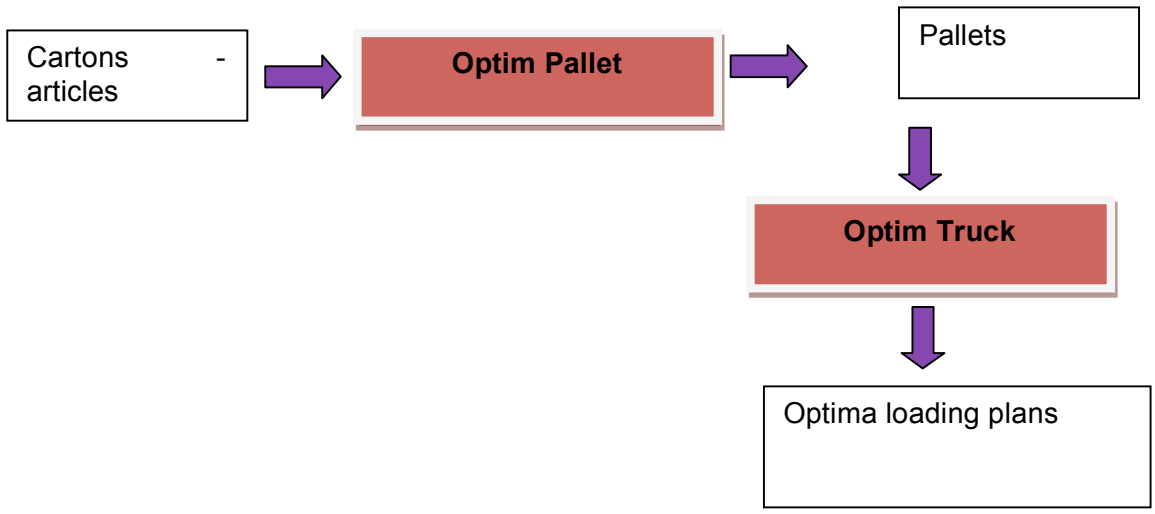


Figure 2: Business packing components.

The first versions take of the components takes as input Excel data and produce also Excel results.

The business packing (Optim Pallet & Optim Truck) components are structured into components:

- Packing container: the logic of packing. This is the main entry to drive the different components of the packing module.
- Packing solver: the optimisation; this components takes well defined inputs and produces results which are then processed by the packing container.
- Packing Player: This is 3D visualisation of containers and items of the container.

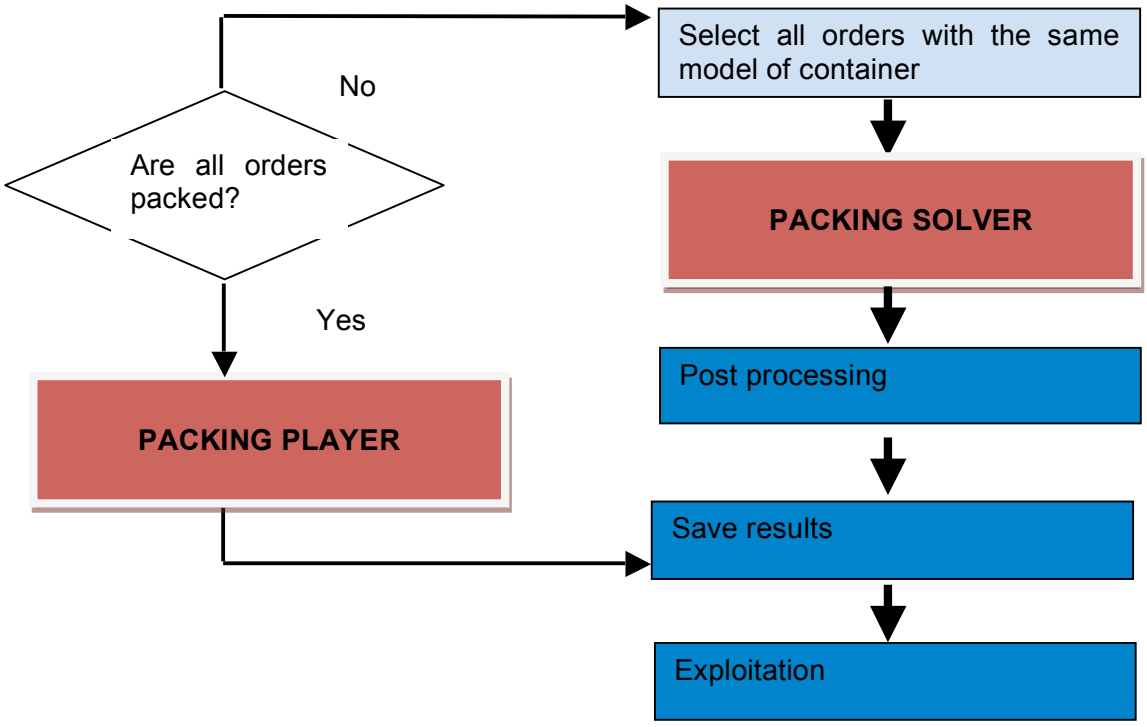


Figure 3: Multiple steps of the Packing Solver of KLS OPTIM

The coming versions will take in addition XML inputs and produce XML results.

A new version of the business components is planned for year 2 of the project. The business components will benefit from the new features which are mainly the polymorphism in Geost; which is the capability to choose between several shapes.

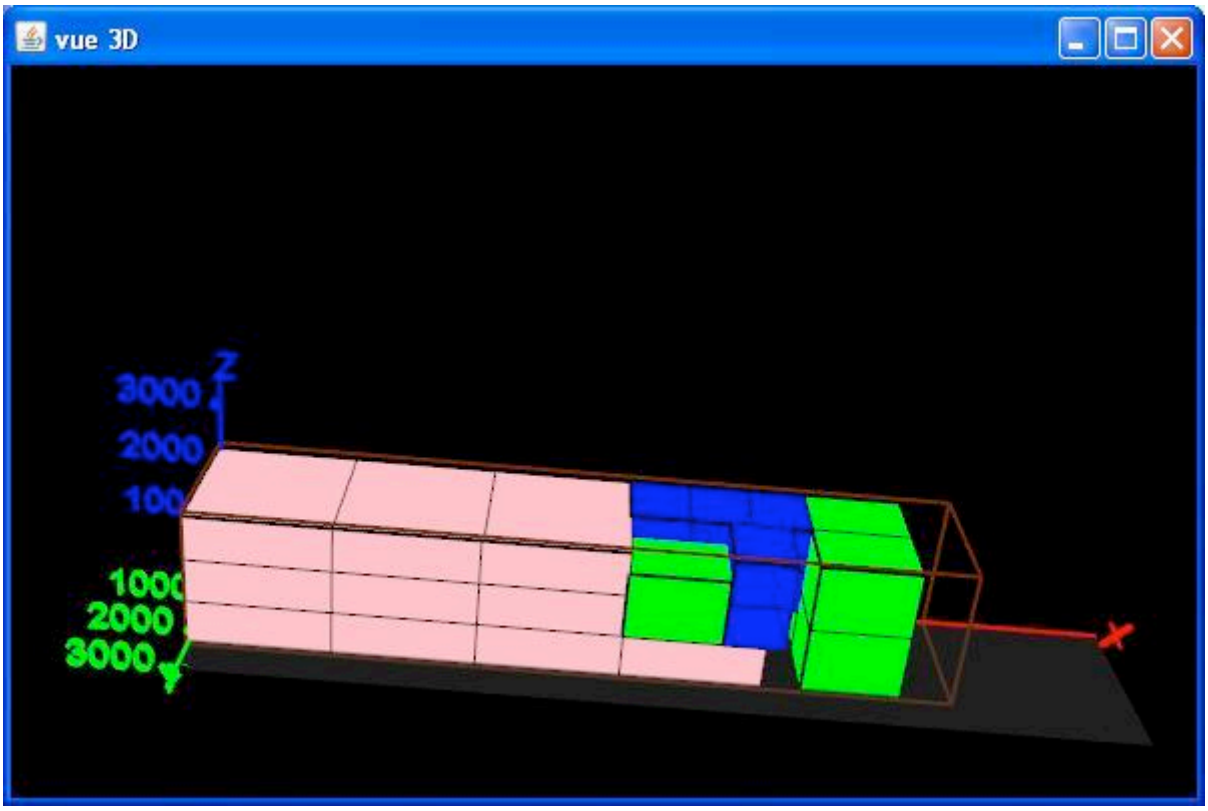


Figure 4: KLS OPTIM 3D Player

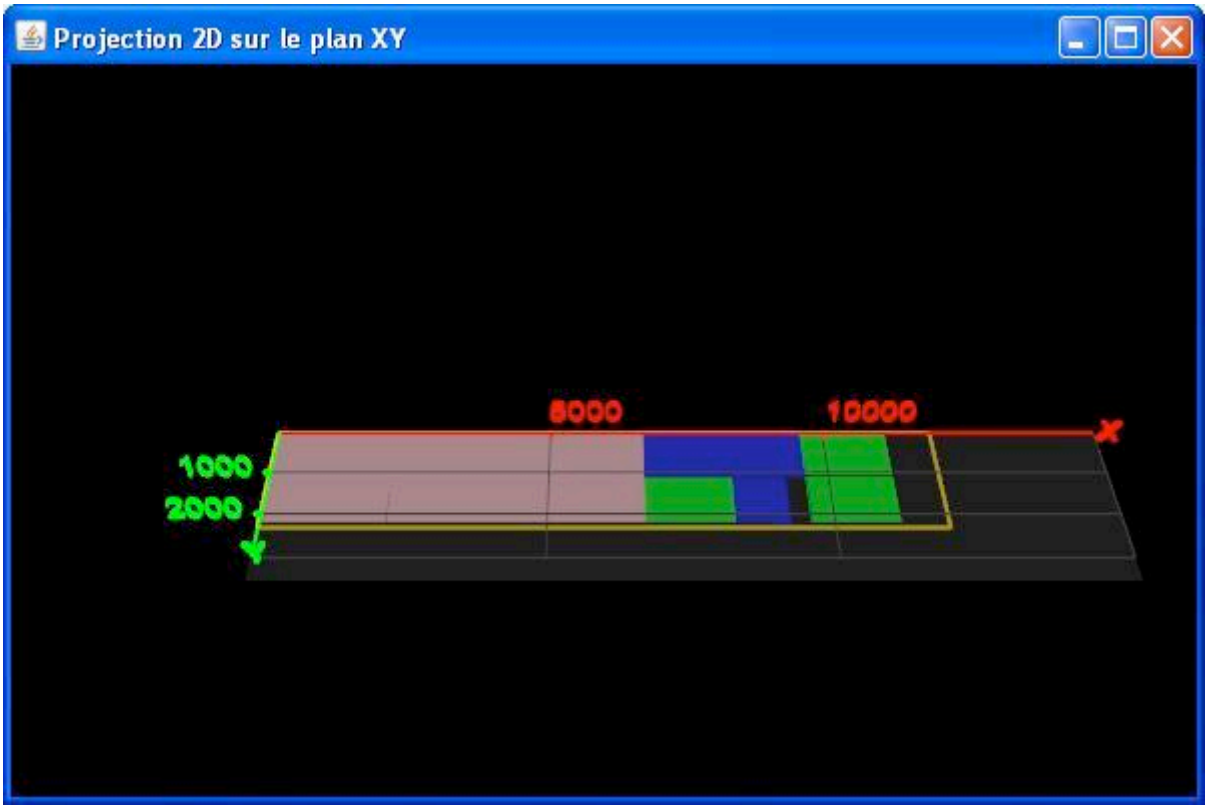


Figure 5: KLS OPTIM 3D Player – 2D projection

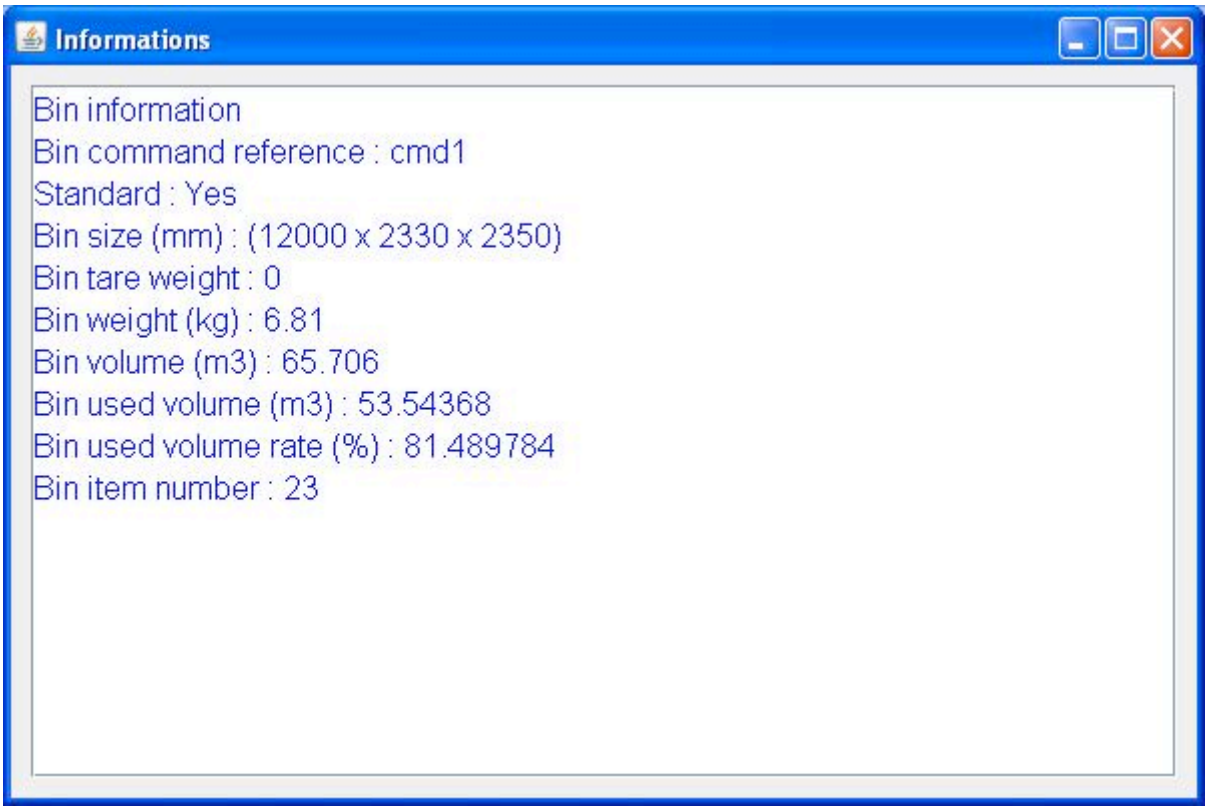


Figure 6: KLS OPTIM 3D Player – Bin information

1.4 End users

KLS OPTIM deployed the first version of the palletizer in an international distribution company; a client of KLS OPTIM which contributes to the requirement definitions and testing phase. In returns, the company had special conditions as defined in the Net-WMS project.

Significant improvements are measured:

- Packing around 100 conditionings (cartons) in less than 1 minute.
- Significant improvements for large orders (up to 20%).
- The total packing tasks takes less than 15 minutes for the system whilst it requires a full day for an operator.