

# A Geometric Constraint over $k$ -Dimensional Objects and Shapes Subject to Business Rules

Mats Carlsson  
SICS, P.O. Box 1263, SE-164 29 Kista, Sweden  
Mats.Carlsson@sics.se

Nicolas Beldiceanu  
École des Mines de Nantes, LINA UMR CNRS 6241, FR-44307 Nantes, France  
Nicolas.Beldiceanu@emn.fr

Julien Martin  
INRIA Rocquencourt, BP 105, FR-78153 Le Chesnay Cedex, France  
Julien.Martin@inria.fr

SICS Technical Report T2008:04  
ISSN: 1100-3154  
ISRN: SICS-T-2008/04-SE

**Abstract:** This report presents a global constraint that enforces rules written in a language based on arithmetic and first-order logic to hold among a set of objects. In a first step, the rules are rewritten to Quantifier-Free Presburger Arithmetic (QFPA) formulas. Secondly, such formulas are compiled to generators of  $k$ -dimensional forbidden sets. Such generators are a generalization of the indexicals of cc(FD). Finally, the forbidden sets generated by such indexicals are aggregated by a sweep-based algorithm and used for filtering.

The business rules allow to express a great variety of packing and placement constraints, while admitting efficient and effective filtering of the domain variables of the  $k$ -dimensional object, without the need to use spatial data structures. The constraint was used to directly encode the packing knowledge of a major car manufacturer and tested on a set of real packing problems under these rules, as well as on a packing-unpacking problem.

**Keywords:** Global Constraint, Geometric Constraint, Rule, Sweep, Quantifier-Free Presburger Arithmetic.

April 3, 2008



# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>The Rule Language: Syntax and Features</b>	<b>4</b>
<b>3</b>	<b>QFPA Core Fragment</b>	<b>7</b>
3.1	Rewriting into QFPA . . . . .	7
<b>4</b>	<b>Compiling to an Efficient Run-Time Representation</b>	<b>11</b>
4.1	Necessary Conditions . . . . .	14
4.2	Pruning Rules . . . . .	14
4.3	$k$ -Indexicals . . . . .	15
4.4	Compilation . . . . .	15
4.5	Filtering Algorithm . . . . .	18
<b>5</b>	<b>Polymorphism</b>	<b>18</b>
<b>6</b>	<b>Experimental Results</b>	<b>20</b>
<b>7</b>	<b>Discussion</b>	<b>21</b>
<b>8</b>	<b>Conclusion</b>	<b>23</b>
<b>A</b>	<b>Prolog Syntax</b>	<b>25</b>
<b>B</b>	<b>Region Connection Calculus Rules</b>	<b>26</b>
B.1	Rules for RCC-8 Relations between Two Shifted Boxes . . . . .	27
B.2	Rules for RCC-8 Relations between Two Objects . . . . .	28
<b>C</b>	<b>A Real-Life Problem Instance</b>	<b>29</b>
<b>D</b>	<b>A Packing-Unpacking Problem</b>	<b>32</b>

# 1 Introduction

This report extends a global constraint  $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{R})$  for handling the location in space of  $k$ -dimensional objects  $\mathcal{O}$  ( $k \in \mathbb{N}^+$ ), each of which taking a shape among a set of shapes  $\mathcal{S}$ , subject to rules  $\mathcal{R}$  in a language based on arithmetic and first-order logic

In order to model directly a lot of side constraints, which always show up in the context of real-life applications, many global constraints have traditionally been extended with extra options or arguments. This is why, in a closely related area, the *diffn* constraint of CHIP provides, beside non-overlapping, a variety of other geometrical constraints (in fact more than 10 side constraints). This was also the case for the *cycle* and *tree* constraints [1, 2] where, beside a basic graph partitioning constraint, a variety of useful side constraints were also provided. Even if this makes sense when one wants to efficiently solve specific real-life applications, this proliferation of arguments and options has two major drawbacks:

- Having a lot of ad-hoc side constraints is too specific and can sometimes be quite frustrating since it does not allow to express a small variant of an existing side constraint.
- Designing a filtering algorithm for each side constraint independently is not enough and managing the interaction of several side constraints becomes more and more challenging as the number and variety of side constraints increase.

The approach presented in this report addresses these two issues in the following way:

- Firstly, having a rule language for expressing side constraints is obviously more flexible than having a large set of predefined side constraints.
- Secondly, as we will see later on, our filtering algorithms allow to directly take into account the interaction between all rules.

The *geost* constraint can also be seen as a natural target constraint of the PKML modeling language [3], being developed by our colleagues in the “Net-WMS” project. In  $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{R})$ , 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*  $s \in \mathcal{S}$  is an entity defined by its shape id  $s.sid$ , shift offset  $s.t[d]$ ,  $1 \leq d \leq k$ , and sizes  $s.l[d]$  (where  $s.l[d] > 0$  and  $1 \leq d \leq k$ ). All attributes of a shifted box are integer values. A *shape* is a collection of shifted boxes all sharing the same shape id.<sup>1</sup>

Each object  $o \in \mathcal{O}$  is an entity defined by its unique object id  $o.oid$  (an integer), shape id  $o.sid$  (an integer if the object has a fixed shape, or a domain variable for *polymorphic* objects, which have alternative shapes), and origin  $o.x[d]$ ,  $1 \leq d \leq k$  (integers, or domain variables that do not occur anywhere else in the constraint).<sup>2</sup> Objects

<sup>1</sup>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.

<sup>2</sup>A *domain variable*  $v$  is a variable ranging over a finite set of integers denoted by  $\text{dom}(v)$ ;  $\underline{v}$  and  $\bar{v}$  denote respectively the minimum and maximum possible values for  $v$ .

and shifted boxes may also have additional, integer (but see also Section 7) attributes, such as weight, customer, or fragility, used by the rules.

Each rule in  $\mathcal{R}$  is a first-order logical formula over the attributes of objects and shifted boxes. From the point of view of domain filtering, the main contribution of this report is that multi-dimensional forbidden sets can be automatically derived from such formulas and used by the sweep-based algorithm of *geost* [4].<sup>3</sup> This contrasts with the previous version of *geost*, where an ad-hoc algorithm computing the multi-dimensional forbidden sets had to be worked out for each side constraint.  $\mathcal{R}$  may also contain macros, providing abbreviations for expressions occurring in formulas or in other macros.

**The rule language.** The language that makes up the rules to be enforced by the *geost* constraint is based on first-order logic with arithmetic, as well as several features including macros, bounded quantifiers, folding and aggregation operators. We will show how all but a core fragment of the language can be eliminated by equivalence-preserving rewriting. The remaining fragment is a subset of Quantifier-Free Presburger Arithmetic (QFPA), which has a very simple semantics and, as we also will show, is amenable to efficient compilation.

Constraint satisfaction problems using quantified formulas (QCSP) have for instance been studied by Benedetti et al. [5], mostly in the context of modeling games. QCSP does not provide disjunction but actively uses quantifiers in the evaluation, whereas we eliminate all quantifiers in the process of rewriting to QFPA.

**Example 1** *This running example will be used to illustrate the way we compile rules to code used by the sweep-based algorithm [4] for filtering the nonground attributes of each object. Suppose that we have five objects  $o_1, o_2, o_3, o_4$  and  $o_5$  such that:*

- $o_1, o_2$  and  $o_4$  correspond to fixed rectangles of respective size  $3 \times 1, 1 \times 1$  and  $3 \times 1$ .
- The coordinates of  $o_3$  are fixed but not its shape variable  $s_3$ , which can take values 3 or 4 (i.e., we can choose among two shapes for object  $o_3$ ). We will denote by  $\ell_{31}$  resp.  $\ell_{32}$  the length resp. height of  $o_3$ .
- The coordinates of the non-fixed square  $o_5$  of size  $2 \times 2$  correspond to the two variables  $x_{51} \in [1, 9]$  and  $x_{52} \in [1, 6]$ .
- $o_2, o_4$  and  $o_5$  have the additional attribute *type* with value 1 whereas  $o_1$  and  $o_3$  have *type* with value 2.
- Two rules must be obeyed:
  - All objects should be mutually non-overlapping (see Fig. 11).

---

<sup>3</sup>The sweep-based algorithm performs recursive traversals of the placement space for each coordinate increasing as well as decreasing lexicographic order and skips unfeasible points that are located in a multi-dimensional forbidden set.

- If the type attribute of two objects both equal 1, the two objects should not meet (see Fig. 11 again).<sup>4</sup>

The full details and geost encoding of the example are shown in Fig. 1; for an explanation of the notation, see Section 2 and Table 4.

□

**Declarative semantics.** As usual, the semantics is given in terms of ground objects. The constraint  $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{R})$  holds if and only if the conjunction of the logical formulas in  $\mathcal{R}$  is true.

**Implementation overview.** Fig. 2 provides the overall architecture of the implementation. When the  $geost$  constraint is posted, the given business rules are translated, first into QFPA, then into generators of  $k$ -dimensional forbidden sets. Such generators,  $k$ -indexicals, are a generalization of the indexicals of cc(FD) [6]. Each time the constraint wakes up, the sweep-based algorithm [4] generates forbidden sets for a specific object  $o$  by invoking the relevant  $k$ -indexicals, then looks for points that are not contained in any forbidden set in order to prune the nonground attributes of  $o$ .

**Report outline.** In Section 2, we present the rule language, its abstract syntax and its features. In Section 3, we present the QFPA core fragment of the language, its declarative semantics, and how the rule language is rewritten into QFPA. In Section 4, we describe (1) how a QFPA formula is compiled to generators of  $k$ -dimensional forbidden sets, and (2) how the forbidden sets generated by such generators are aggregated by a sweep-based algorithm and used for filtering. In Section 5, we extend the filtering to accommodate polymorphic objects. Before concluding, in Section 7, we mention a number of issues that we are currently working on. In the Appendix, we show the Prolog representation of the various language elements that we actually use in the implementation. The Appendix also shows how the Region Connection Calculus may be expressed in our language, as well as rules encoding a problem instance provided by a major car manufacturer and rules encoding a packing-unpacking problem.

The syntax descriptions are kept abstract, with inductive definitions of legal terms instead of BNF grammars of legal sentences. The inductive definitions do use BNF-like notation.

## 2 The Rule Language: Syntax and Features

Fig. 3 shows the inductive definition of the rule language. A *macro* is simply a shorthand device: during a rewriting phase, whenever an expression matching the left-hand side of a macro is encountered, it is replaced by the corresponding right-hand side. A *fol* is a first-order logic formula that must hold for the constraint to be true. A *term* is a *variable*, an *integer*, an *identifier*, or a *compound term*. A *compound term* consists of a *functor* (an identifier) and one or more arguments (terms). A term is *ground* if it is

---

<sup>4</sup>Two rectangles meet also if their corners meet.

```

example(S3, X51, X52) :-
  % PROBLEM VARIABLES
  S3 in 3..4, X51 in 1..9, X52 in 1..6,
  geost(% OBJECTS TO PLACE
    [object(oid-1, sid-1,x-[ 1, 2],type-2),
     object(oid-2, sid-2,x-[ 3, 3],type-1),
     object(oid-3,sid-S3,x-[ 2, 5],type-2),
     object(oid-4, sid-1,x-[ 3, 7],type-1),
     object(oid-5, sid-5,x-[X51,X52],type-1)],
  % SHAPES THAT CAN BE ASSIGNED TO OBJECTS
  [sbox(sid-1,t-[0,0],l-[3,1]),
   sbox(sid-2,t-[0,0],l-[1,1]),
   sbox(sid-3,t-[0,0],l-[1,2]),
   sbox(sid-4,t-[0,0],l-[2,1]),
   sbox(sid-5,t-[0,0],l-[2,2])],
  [% MACROS DEFINING FUNCTIONS (DERIVED ATTRIBUTES)
   (origin(O1,S1,D) ---> O1^x(D)+S1^t(D)),
   (end(O1,S1,D) ---> O1^x(D)+S1^t(D)+S1^l(D)),
  % MACROS DEFINING PAIRWISE TOPOLOGICAL RELATIONS
   (overlap_sboxes(Dims, O1, S1, O2, S2) --->
     forall(D, Dims,
       end(O1,S1,D) #> origin(O2,S2,D) #/\
       end(O2,S2,D) #> origin(O1,S1,D))),
   (meet_sboxes(Dims, O1, S1, O2, S2) --->
     forall(D, Dims,
       end(O1,S1,D) #>= origin(O2,S2,D) #/\
       end(O2,S2,D) #>= origin(O1,S1,D) #/\
     exists(D, Dims,
       end(O1,S1,D) #= origin(O2,S2,D) #\
       end(O2,S2,D) #= origin(O1,S1,D))),
  % MACROS DEFINING N-ARY CONSTRAINTS
   (all_not_overlap_sboxes(Dims,OIDs) --->
     forall(O1,objects(OIDs),
       forall(S1,sboxes([O1^sid]),
         forall(O2,objects(OIDs),
           O1^oid #< O2^oid #=>
           forall(S2,sboxes([O2^sid]),
             #\ overlap_sboxes(Dims,O1,S1,O2,S2))))),
   (all_type1_not_meet_sboxes(Dims,OIDs) --->
     forall(O1,objects(OIDs),
       forall(S1,sboxes([O1^sid]),
         forall(O2,objects(OIDs),
           O1^oid #< O2^oid #/\ O1^type#=1 #/\ O2^type#=1 #=>
           forall(S2,sboxes([O2^sid]),
             #\ meet_sboxes(Dims,O1,S1,O2,S2))))),
  % BUSINESS RULES
  all_not_overlap_sboxes([1,2],[1,2,3,4,5]),
  all_type1_not_meet_sboxes([1,2],[1,2,3,4,5])).

```

Figure 1: Running example encoded with *geost*.

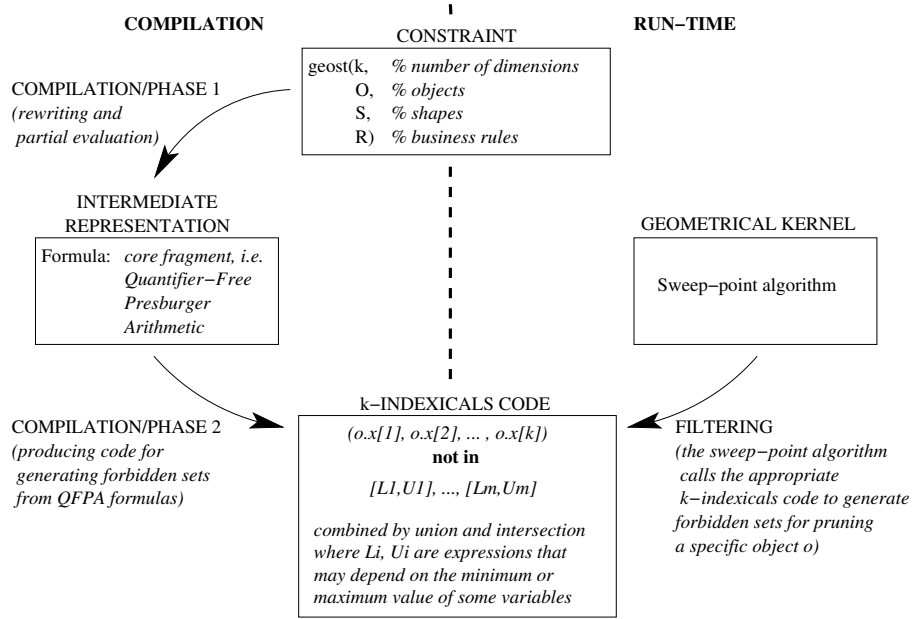


Figure 2: Overall architecture of the implementation.

free of variables. An *entity* denotes an object resp. a shifted box, the exact structure of which is left unspecified, but a possible Prolog representation is shown in Appendix A. An *attref* is a reference to an attribute of an entity.

*Bounded existential* resp. *universal quantifiers* are provided. They are meaningful if the quantified variable occurs in the quantified *fol*. They are treated by expansion to a disjunction resp. a conjunction of instances of that *fol* where each element of the *collection* is substituted for the quantified variable. For example, formulas (1) and (2) below are equivalent:

$$\forall(x, [0, 1, 2], p(x)) \quad (1)$$

$$p(0) \wedge p(1) \wedge p(2) \quad (2)$$

In the context of our application, quantified variables typically vary over a collection of dimensions, objects, or shifted boxes. `objects(S)` is a shorthand for the collection of objects with object id in  $S$ . Similarly, `sboxes(S)` is a shorthand for a collection of shifted boxes.

A *cardinality formula* specifies a variable quantified over a list of terms, a lower and an upper bound, and a *fol* template mentioning the quantified variable. The formula is true if and only if the number of true instances of the *fol* template is within the given bounds. Cardinality formulas [7] are treated by expansion to  $\neg$ ,  $\wedge$  and  $\vee$  connectives [8]. For example, formulas (3) and (4) below are equivalent:

$$\#(y, [o_1, o_2, o_3], 2, 3, y.type > 5) \quad (3)$$

$$\bigvee \left( \begin{array}{l} o_1.type > 5 \wedge o_2.type > 5 \\ o_1.type > 5 \wedge o_3.type > 5 \\ o_2.type > 5 \wedge o_3.type > 5 \end{array} \right) \quad (4)$$

*Arithmetic expressions* and *comparisons* are over the rational numbers. The rationale for this is that business rules often involve fractions of measures like weight or volume, and such fractions are more convenient to express with a notation for rational division than in a purely integer setting.

A *folding operator* allows to express e.g. the sum of some attribute over a set of objects. The operator specifies a variable quantified over a list of terms, a binary operator, an identity element, and a template mentioning the quantified variable. The identity element is needed for the empty list case. For example, formulas (5) and (6) below are equivalent:

$$@ (y, [o_1, o_2, o_3], +, 0, y.weight) \quad (5)$$

$$o_1.weight + o_2.weight + o_3.weight \quad (6)$$

### 3 QFPA Core Fragment

In this section, we show how a formula  $p$  in the rule language is rewritten by a series of equivalence-preserving transformations into a *qfpa*, i.e. a formula of the core fragment of the language shown in Fig. 4. In fact, the fragment coincides with Quantifier-Free Presburger Arithmetic (QFPA), although QFPA is usually described with a less restrictive syntax. The declarative semantics of a *qfpa* is the natural one.

QFPA is widely used in symbolic verification, and there has been much work on deciding whether a given QFPA formula is satisfiable [9]. Many methods based on integer programming techniques [10] rely on having the formula on disjunctive normal form. However, for constraint programming purposes, we are interested in necessary conditions that can be used for filtering domain variables, and we are not aware on any such work on QFPA.

#### 3.1 Rewriting into QFPA

We now show the details of rewriting the formula given as the *geost* parameter  $\mathcal{R}$  in the following eight steps into a *qfpa*  $\hat{\mathcal{R}}$ . Fig. 5 shows the details of some of these steps as tables. The cell in the column entitled **condition**, if nonempty, mentions the condition under which the rewrite is done. We will later show how  $\hat{\mathcal{R}}$  is translated to generators of forbidden sets.

<i>sentence</i>	::=	<i>macro</i>   <i>fol</i>	
<i>macro</i>	::=	<i>head</i> $\implies$ <i>body</i>	
<i>head</i>	::=	<i>term</i>	{ to be substituted by a <i>body</i> }
<i>body</i>	::=	<i>term</i>	{ to substitute for a <i>head</i> }
<i>fol</i>	::=	$\neg fol$	{ negation }
		$fol \wedge fol$	{ conjunction }
		$fol \vee fol$	{ disjunction }
		$fol \implies fol$	{ implication }
		$fol \Leftrightarrow fol$	{ equivalence }
		$\exists(var, collection, fol)$	{ existential quantification }
		$\forall(var, collection, fol)$	{ universal quantification }
		$\#(var, collection, integer, integer, fol)$	{ cardinality }
		<b>true</b>	
		<b>false</b>	
		<i>expr</i> <i>relop</i> <i>expr</i>	{ arith. comparison over $\mathbb{Q}$ }
		<i>head</i>	{ macro application }
<i>expr</i>	::=	<i>expr</i> + <i>expr</i>	
		<i>expr</i> - <i>expr</i>	
		min( <i>expr</i> , <i>expr</i> )	
		max( <i>expr</i> , <i>expr</i> )	
		<i>expr</i> $\times$ <i>groundexpr</i>	
		<i>groundexpr</i> $\times$ <i>expr</i>	
		<i>expr</i> / <i>groundexpr</i>	
		<i>attref</i>	
		<i>integer</i>	
		@( <i>var</i> , <i>collection</i> , <i>fop</i> , <i>expr</i> , <i>expr</i> )	{ folding }
		<i>variable</i>	{ quantified variable }
		<i>head</i>	{ macro application }
<i>groundexpr</i>	::=	<i>expr</i>	{ where <i>expr</i> is ground }
<i>attref</i>	::=	<i>entity.attr</i>	
<i>attr</i>	::=	<i>term</i>	{ attribute name }
		<i>variable</i>	{ quantified variable }
<i>relop</i>	::=	<   =   >   $\neq$   $\leq$   $\geq$	
<i>fop</i>	::=	+   min   max	
<i>collection</i>	::=	<i>list</i>	
		objects( <i>list</i> )	{ list of oids }
		sboxes( <i>list</i> )	{ list of sids }
<i>list</i>	::=	[]   [ <i>term</i>   <i>list</i> ]	

Figure 3: The rule language

$qfpa$	$::=$	$qfpa \wedge qfpa$	{ conjunction }
		$qfpa \vee qfpa$	{ disjunction }
		$\sum_i integer_i \cdot attref_i \geq integer$	{ base case }

Figure 4: Core fragment of the language. An *attref* corresponds to a nonground attribute of an object or an attribute of a shifted box of a polymorphic object.

**Macro expansion and constant folding.** The implication and equivalence connectives, bounded quantifiers, and cardinality and folding operators are eliminated. Ground integer expressions are replaced by their values. Object and shifted box collections are expanded.

**Elimination of negation.** Using DeMorgan's laws and negating relevant *relops*.

**Normalization of arithmetic.** Arithmetic relations are normalized to one of the forms  $expr \geq 0$  or  $expr > 0$ .

**Elimination of  $\times$ ,  $/$  and  $-$ .** Any occurrence of these operators in arithmetic expressions is eliminated. At the same time, all operands are associated with a rational coefficient ( $c$  in the table). The elimination is made possible by the fact that in multiplication, at least one factor must be ground and is simply multiplied into the coefficient. Similarly, in division, the coefficient is simply divided by the divisor, which must be ground. After this step, an arithmetic expression is:

- a rational number  $c$ , denoted  $c \cdot 1$ , or
- an *attref*  $r$  with a rational coefficient  $c$ , denoted  $c \cdot r$ , or
- two arithmetic expressions combined with  $+$ ,  $\min$  or  $\max$ .

**Moving  $+$  inside  $\min$  and  $\max$ .** Any expression with  $\min$  or  $\max$  occurring inside  $+$  are rewritten by using the commutative and distributive laws (7) so that the  $+$  is moved inside the other operator.

$$\begin{aligned}
 a + b &= b + a \\
 a + \min(b, c) &= \min(a + b, a + c) \\
 a + \max(b, c) &= \max(a + b, a + c)
 \end{aligned} \tag{7}$$

**Elimination of  $\min$  and  $\max$ .** Any  $\min$  or  $\max$  operators occurring in arithmetic relations are eliminated, replacing such relations by new relations combined by  $\wedge$  or  $\vee$ . After this step, an arithmetic expression is a linear combination of *attrefs* with rational coefficients, plus an optional constant.

**Elimination of rational numbers.** Any arithmetic relation  $r$ , which can now only be of the form  $e > 0$  or  $e \geq 0$ , is normalized into the form  $e'' \geq c''$  where  $e'$  and  $c'$  are intermediate expressions in:

line	$p$	$R_1(p)$	condition
1	$p$	$R_1(q)$	$q = \text{macro}(p)$
2	$\neg p$	$\neg R_1(p)$	
3	$p \Rightarrow q$	$R_1(q \vee \neg p)$	
4	$p \Leftrightarrow q$	$R_1((p \Rightarrow q) \wedge (q \Rightarrow p))$	
5	$\exists(x, [y_1, \dots, y_n], p)$	$R_1(p_{x/y_1} \vee \dots \vee p_{x/y_n})$	
6	$\forall(x, [y_1, \dots, y_n], p)$	$R_1(p_{x/y_1} \wedge \dots \wedge p_{x/y_n})$	
7	$@(x, [y_1, \dots, y_n], \circ, z, p)$	$R_1(p_{x/y_1} \circ \dots \circ p_{x/y_n} \circ z)$	
8	$\#(x, [], l, u, p)$	true	$l \leq 0 \leq u$
9	$\#(x, [], l, u, p)$	false	$l > 0 \vee 0 > u$
10	$\#(x, [y_1, \dots, y_n], l, u, p)$	$R_1 \left( \begin{array}{l} (p_{x/y_1} \wedge \#(x, [y_2, \dots, y_n], l-1, u-1, p) \vee \\ (\neg p_{x/y_1} \wedge \#(x, [y_2, \dots, y_n], l, u, p)) \end{array} \right)$	$n > 0$
11	$expr$	$i$	$i = \text{ieval}(p)$
12	$\text{objects}([o_1, \dots, o_n])$	objects with the given oids	
13	$\text{sboxes}([s_1, \dots, s_n])$	sboxes with the given sids	

$p$	$R_3(p)$
$x < y$	$y - x > 0$
$x > y$	$x - y > 0$
$x \leq y$	$y - x \geq 0$
$x \geq y$	$x - y \geq 0$
$x = y$	$x - y \geq 0 \wedge y - x \geq 0$
$x \neq y$	$x - y > 0 \vee y - x > 0$

$p$	$R_4(p, c)$	condition
$\min(x, y)$	$\min(R_4(x, c), R_4(y, c))$	$c > 0$
$\min(x, y)$	$\max(R_4(x, c), R_4(y, c))$	$c < 0$
$\max(x, y)$	$\max(R_4(x, c), R_4(y, c))$	$c > 0$
$\max(x, y)$	$\min(R_4(x, c), R_4(y, c))$	$c < 0$
$x + y$	$R_4(x, c) + R_4(y, c)$	
$x - y$	$R_4(x, c) + R_4(y, -c)$	
$x \times y$	$R_4(x, c \times v)$	$v = \text{reval}(y)$
$x \times y$	$R_4(y, c \times v)$	$v = \text{reval}(x)$
$x/y$	$R_4(x, c/v)$	$v = \text{reval}(y)$
$x$	$(c \times x) \cdot 1$	$x$ integer
$x$	$c \cdot x$	$x$ attref

$p$	$R_6(p)$
$\max(x, y) > 0$	$x > 0 \vee y > 0$
$\min(x, y) > 0$	$x > 0 \wedge y > 0$
$\max(x, y) \geq 0$	$x \geq 0 \vee y \geq 0$
$\min(x, y) \geq 0$	$x \geq 0 \wedge y \geq 0$

Figure 5: **Top.** Rewrite phase 1, of a formula  $p$  into a formula  $R_1(p)$ , eliminates macros (line 1), implication (line 3), equivalence (line 4), bounded quantifiers (line 5-6), folding operators (line 7), cardinality operators (line 8-10), ground attribute references (line 11), and entity collections (line 12-13). If a compound term does not match any line 1-13, its arguments are rewritten recursively.  $p_{x/y}$  denotes the term  $p$  with  $y$  substituted for  $x$ .  $\text{macro}(p)$  denotes the macro expansion of the formula  $p$ .  $\text{ieval}(p)$  denotes the integer value of the ground expression  $p$ . **Bottom left.** Rewrite phase 3, of a formula  $p$  into a formula  $R_3(p)$ , normalizes comparison operators into either  $\geq$  or  $>$ . **Bottom center.** Rewrite phase 4, of a formula  $p$  into a formula  $R_4(p, 1)$ , eliminates the  $-$ ,  $\times$  and  $/$  operators, and assigns a coefficient  $c$  to each operand of the rewritten formula.  $\text{reval}(y)$  denotes the rational value of the ground expression  $y$ . **Bottom right.** Rewrite phase 6, of a formula  $p$  into a formula  $R_6(p)$ , eliminates min and max.

- Let  $e'$  be the linear combination obtained by multiplying  $e$  by the least common multiplier of the denominators of the coefficients of  $e$ . Recall that those coefficients are rational numbers. Thus, the coefficients of  $e'$  are integers.
- Let  $c'$  be 1 if  $r$  is of the form  $e > 0$ , or 0 if  $r$  is of the form  $e \geq 0$ .
- If  $e'$  contains a constant term  $c$ , then  $e'' = e' - c$  and  $c'' = c' - c$ . Otherwise,  $e'' = e'$  and  $c'' = c'$ .

**Simplification.** Any entailed or disentailed arithmetic comparison is replaced by the appropriate Boolean constant (`true` or `false`). Any  $\wedge$  or  $\vee$  expression containing one of these constants is simplified using partial evaluation.

**Example 2** Returning to our running example, we show in Figs. 6-7 how the initial business rules are successively rewritten into a qfpa. The example shows that the rewrite process essentially amounts to partial evaluation. The resulting qfpa  $\hat{\mathcal{R}}$  is a conjunction of six subformulas corresponding respectively to:

- From the business rule `all_not_overlap_sboxes`, conditions to prevent  $o_5$  from overlapping  $o_1, o_2, o_3$  and  $o_4$ .
- From the business rule `all_type1_not_meet_sboxes`, conditions to prevent  $o_5$  from meeting  $o_2$  and  $o_4$ .

□

## 4 Compiling to an Efficient Run-Time Representation

It is straightforward to obtain necessary conditions for *qfpas* as well as pruning rules operating on one variable at a time. Based on such conditions and pruning rules, we will show how to construct generators of  $k$ -dimensional forbidden sets. We call such generators *k-indexicals*, for they are generalization of the indexicals of `cc(FD)` [6]. Finally, we show how the forbidden sets generated by such indexicals are aggregated by the sweep-based algorithm [4] and used for filtering.

Indexicals were first introduced for the language `cc(FD)` [6] and later used in the context of `CLP(FD)` [11, 12], `AKL` [13] and finite set constraints [14]. They have proven a powerful and efficient way of implementing constraint propagation. A key feature of an indexical is that it is a function of the current domains of the variables on which it depends. Thus, indexicals also capture the propagation from variables to variables that occurs as variables are pruned. In the cited implementations, an indexical is a procedure that computes the feasible set of values for a variable. We generalize this notion to generating a forbidden set of  $k$ -dimensional points for an object, and so  $k$ -indexicals captures the propagation from objects to objects that occurs as object attributes are pruned.



$$\begin{array}{c}
\bigwedge \left( \begin{array}{c}
\bigvee \left( \begin{array}{c} x_{51} \geq 4 \\ -1 \cdot x_{51} \geq 1 \\ x_{52} \geq 3 \\ -1 \cdot x_{52} \geq 0 \end{array} \right) \\
\bigvee \left( \begin{array}{c} x_{51} \geq 4 \\ -1 \cdot x_{51} \geq -1 \\ x_{52} \geq 4 \\ -1 \cdot x_{52} \geq -1 \end{array} \right) \\
\bigvee \left( \begin{array}{c} -1 \cdot \ell_{31} \geq -1 \\ -1 \cdot \ell_{32} \geq -2 \\ -1 \cdot \ell_{31} + x_{51} \geq 2 \\ -1 \cdot x_{51} \geq 0 \\ -1 \cdot \ell_{32} + x_{52} \geq 5 \\ -1 \cdot x_{52} \geq -3 \end{array} \right) \\
\bigvee \left( \begin{array}{c} x_{51} \geq 6 \\ -1 \cdot x_{51} \geq -1 \\ x_{52} \geq 8 \\ -1 \cdot x_{52} \geq -5 \end{array} \right) \\
\bigvee \left( \begin{array}{c} x_{51} \geq 5 \\ -1 \cdot x_{51} \geq 0 \\ x_{52} \geq 5 \\ -1 \cdot x_{52} \geq 0 \end{array} \right) \\
\bigvee \left( \begin{array}{c} \bigwedge \left( \begin{array}{c} \bigvee \left( \begin{array}{c} -1 \cdot x_{51} \geq -3 \\ x_{51} \geq 5 \end{array} \right) \\ \bigvee \left( \begin{array}{c} x_{51} \geq 2 \\ -1 \cdot x_{51} \geq 0 \end{array} \right) \\ \bigvee \left( \begin{array}{c} -1 \cdot x_{52} \geq -3 \\ x_{52} \geq 5 \end{array} \right) \\ \bigvee \left( \begin{array}{c} x_{52} \geq 2 \\ -1 \cdot x_{52} \geq 0 \end{array} \right) \end{array} \right) \\
\bigvee \left( \begin{array}{c} x_{51} \geq 7 \\ -1 \cdot x_{51} \geq 0 \\ x_{52} \geq 9 \\ -1 \cdot x_{52} \geq -4 \end{array} \right) \\
\bigvee \left( \begin{array}{c} \bigwedge \left( \begin{array}{c} \bigvee \left( \begin{array}{c} -1 \cdot x_{51} \geq -5 \\ x_{51} \geq 7 \end{array} \right) \\ \bigvee \left( \begin{array}{c} x_{51} \geq 2 \\ -1 \cdot x_{51} \geq 0 \end{array} \right) \\ \bigvee \left( \begin{array}{c} -1 \cdot x_{52} \geq -7 \\ x_{52} \geq 9 \end{array} \right) \\ \bigvee \left( \begin{array}{c} x_{52} \geq 6 \\ -1 \cdot x_{52} \geq -4 \end{array} \right) \end{array} \right) \end{array} \right)
\end{array} \right)
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}$$

Figure 7: Running example formula after elimination of rational numbers (left) and simplification (right), resulting in a QFPA formula  $\hat{\mathcal{R}}$ .

## 4.1 Necessary Conditions

For a formula  $R$  denoting a linear combination of variables, let  $MAX(R)$  denote the expression that replaces every *attref*  $x$  in  $R$  by  $\bar{x}$  if  $x$  occurs with a positive coefficient, and by  $\underline{x}$  otherwise. Thus,  $MAX(R)$  is a formula that computes an upper bound of  $R$  wrt. the current domains.

We will ignore the degenerate cases where  $\hat{\mathcal{R}}$  is *true* resp. *false*, in which case *geost* merely succeeds resp. fails. For the normal *qfpa* cases, we obtain the necessary conditions shown in Table 1.

<i>qfpa</i> $t$	necessary condition $N(t)$
$\sum_i c_i \cdot x_i \geq r$	$MAX(\sum_i c_i \cdot x_i) \geq r$
$p \vee q$	$N(p) \vee N(q)$
$p \wedge q$	$N(p) \wedge N(q)$

Table 1: Necessary condition  $N(t)$  for *qfpa*  $t$

## 4.2 Pruning Rules

For the base case  $\sum_i c_i \cdot x_i \geq r$ , we have the well-known pruning rules (8), which provide sharp bounds; see e.g. [15] for details.

$$\forall j \begin{cases} x_j \geq \lceil \frac{r - MAX(\sum_{i \neq j} c_i \cdot x_i)}{c_j} \rceil, & \text{if } c_j > 0 \\ x_j \leq \lfloor \frac{-r + MAX(\sum_{i \neq j} c_i \cdot x_i)}{-c_j} \rfloor, & \text{otherwise} \end{cases} \quad (8)$$

Consider now a disjunction  $p \vee q$  of two base cases and a variable  $x_j$  occurring in at least one disjunct.

- If  $x_j$  occurs in  $p$  but not in  $q$ , rule (8) is only valid for  $p$  if the necessary condition for  $q$  does not hold.
- Similarly if  $x_j$  occurs in  $q$  but not in  $p$ .
- If  $x_j$  occurs in both  $p$  and  $q$ , we can use rule (8) for both  $p$  and  $q$  and conclude that  $x_j$  must be in the union of the two feasible intervals.

Finally, consider a conjunction  $p \wedge q$ , i.e. both  $p$  and  $q$  must hold. If  $x_j$  occurs in both  $p$  and  $q$ , we can use rule (8) for both  $p$  and  $q$  and conclude that  $x_j$  must be in the intersection of the two feasible intervals.

**Example 3** *Returning to our running example, consider the fragment  $x_{51} \geq 4 \vee x_{52} \geq 3$  of the *qfpa*, which comes from a rule preventing  $o_5$  from overlapping  $o_1$ . Suppose that we want to prune  $x_{52}$ . Then we can combine the necessary condition for  $x_{51} \geq 4$  with rule (8) for  $x_{52} \geq 3$  into the conditional pruning rule:*

$$\max(x_{51}) < 4 \Rightarrow x_{52} \geq 3$$

However, as we will show in the next section, instead of using such conditional pruning rules, we unify necessary conditions and pruning rules into multi-dimensional forbidden sets and aggregate them per object. For the above fragment, the two-dimensional forbidden set for  $o_5$  is  $([1, 3], [1, 2])$ , denoting the fact that  $(x_{51}, x_{52})$  should be distinct from all the pairs  $(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2)$ .  $\square$

### 4.3 $k$ -Indexicals

Recall that the set of rules given in  $\mathcal{R}$  has been rewritten into a *qffa*  $\hat{\mathcal{R}}$ . Consider this formula, or some subformula  $\hat{\mathcal{R}}_i$  of it if  $\hat{\mathcal{R}}$  is a conjunction (see Section 4.4). The idea is to compile this subformula, for each object  $o$  mentioned by it, into a  $k$ -indexical for  $\hat{\mathcal{R}}_i$  and  $o$ . The forbidden sets that it generates can then be aggregated and used by the sweep-point kernel [4] to prune the nonground attributes of  $o$ . Let us introduce some notation to make this idea clear.

**Definition 1** A forbidden set for a *qffa*  $r$  and object  $o$  is a set<sup>5</sup> of  $k$ -dimensional points such that, if  $o$  is placed at any of these points,  $r$  is disentailed.

**Definition 2** A  $k$ -indexical for a *qffa*  $r$  and an object  $o$  is a procedure that functions as a generator of forbidden sets for  $r$  and  $o$ . It is of the form  $o.x \notin \text{ibody}$  where *ibody* is defined in Fig. 8. The  $k$ -indexical depends on object  $o'$  if *ibody* mentions  $o'$ .

$k$ -indexicals are described by the inductive definition shown in Fig. 8. They are built up from generators of  $k$ -dimensional half-planes, combined by union and intersection operations.

### 4.4 Compilation

The *qffa*  $\hat{\mathcal{R}}$ , normally<sup>6</sup> a conjunction  $\hat{r}_1 \wedge \dots \wedge \hat{r}_n$ , is compiled to  $k$ -indexicals by the following steps:

1. Partition the conjuncts of  $\hat{\mathcal{R}}$  into equivalence classes  $\hat{\mathcal{R}}_1, \dots, \hat{\mathcal{R}}_m$  such that for all  $1 \leq i < j \leq n$ ,  $\hat{r}_i$  and  $\hat{r}_j$  are in the same equivalence class if and only if they mention<sup>7</sup> the same set of objects of  $\mathcal{O}$ .
2. For each equivalence class  $\hat{\mathcal{R}}_i$  and object  $o \in \mathcal{O}$  mentioned by  $\hat{\mathcal{R}}_i$ , map  $\hat{\mathcal{R}}_i$  (as a conjunction) into a  $k$ -indexical for  $o$ , of the form  $o.x \notin F_o(\hat{\mathcal{R}}_i)$ , according to Table 2.

The mapping closely follows the pruning rules (8), except now we want to obtain a forbidden set instead of a feasible interval. Row 5 of Table 2 corresponds to the case where  $r$  does not mention  $o$ , in which case all points are forbidden for  $o$  if  $r$  is disentailed, and no points are forbidden for  $o$  otherwise.

<sup>5</sup>A forbidden set is not explicitly represented as a set of points, but rather by a set of boxes, as is the case in the earlier implementation [4].

<sup>6</sup>Since it comes from the conjunction of business rules stated in the last argument of *geost*.

<sup>7</sup>A formula *mentions* an object  $o$  if it refers to a nonground attribute of  $o$ .

$k$ -indexical	$::=$	$object.x \notin ibody$	
$ibody$	$::=$	$ibody \cap ibody$   $ibody \cup ibody$   $\{p \in \mathbb{Z}^k \mid p[d] < \lceil \frac{integer - \sum ubterm}{usi} \rceil\}$   $\{p \in \mathbb{Z}^k \mid p[d] > \lfloor \frac{integer + \sum ubterm}{usi} \rfloor\}$   <b>if</b> $\sum ubterm < r$ <b>then</b> $\mathbb{Z}^k$ <b>else</b> $\emptyset$	
$ubterm$	$::=$	$usi \cdot \overline{attref}$   $-usi \cdot \underline{attref}$   $integer$	
$d$	$::=$	$integer$	{ denoting a dimension }
$usi$	$::=$	$integer$	{ $> 0$ }

Figure 8:  $k$ -indexicals

$r$	$F_o(r)$	condition
$p \vee q$	$F_o(p) \cap F_o(q)$	
$p \wedge q$	$F_o(p) \cup F_o(q)$	
$\sum_i c_i \cdot x_i \geq r$	$\{p \in \mathbb{Z}^k \mid p[d] < \lceil \frac{r - MAX(\sum_{i \neq j} c_i \cdot x_i)}{c_j} \rceil\}$	$x_j = o.x[d], c_j > 0$
$\sum_i c_i \cdot x_i \geq r$	$\{p \in \mathbb{Z}^k \mid p[d] > \lfloor \frac{-r + MAX(\sum_{i \neq j} c_i \cdot x_i)}{-c_j} \rfloor\}$	$x_j = o.x[d], c_j < 0$
$\sum_i c_i \cdot x_i \geq r$	<b>if</b> $MAX(\sum_i c_i \cdot x_i) < r$ <b>then</b> $\mathbb{Z}^k$ <b>else</b> $\emptyset$	$o.x[d] \notin \{x_i\}$

Table 2: Mapping a *qfpa*  $r$  to a generator of forbidden sets,  $F_o(r)$ , for the object  $o$ . We assume here that  $o$  is not polymorphic.

The rationale for aggregating the conjuncts into equivalence classes, as opposed to mapping one conjunct at a time, is the opportunity to increase the granularity of the indexicals and to merge subformulas coming from different business rules. This opens the scope for future work on global simplification of formulas, and increases the amount of subexpressions that can be shared within a  $k$ -indexical.

It is well known that indexicals can be efficiently compiled and executed by a virtual machine [11, 12]. In our context, we predict that there will be a large amount of common subterms in the  $k$ -indexicals, and so common subexpression elimination will be quite important. Therefore, a register-based virtual machine would seem an appropriate choice.

It is worth noting that the forbidden sets generated by our compiler do not necessarily include all infeasible points. Consider e.g. the *qfpa*:

$$o.x[1] + o.x[2] \geq 3 \wedge o.x[1] + o.x[2] \leq 3$$

with  $o.x[1] \in [0, 3]$ ,  $o.x[2] \in [0, 3]$ , which we compile to:

$$o.x \notin \bigcup \left( \begin{array}{l} ([0, 2 - \overline{o.x[2]}], [0, 2 - \overline{o.x[1]}]) \\ ([4 - \underline{o.x[2]}, 3], [4 - \underline{o.x[1]}, 3]) \end{array} \right)$$

So with the initial domains, the forbidden set would be empty, whereas a forbidden set that includes all points such that  $o.x[1] + o.x[2] \neq 3$  could easily be computed. However, such a set would require a number of boxes that depends on the domain sizes, whereas our compiler has no such dependency. This example illustrates a trade-off between space complexity and pruning effectiveness.

**Example 4** *Returning to our running example, we obtained a qfpa which was a conjunction of six subformulas (see Fig. 7). They are partitioned into two equivalence classes: one for the single conjunct that mentions both  $o_3$  and  $o_5$ , mapped to  $k$ -indexicals (9) and (10) below; and one for the five conjuncts that only mention  $o_5$  (because  $o_1$ ,  $o_2$  and  $o_4$  are ground), mapped to  $k$ -indexical (11) below. The three  $k$ -indexicals reflect the following business rules:*

1.  $o_3$  must not take a shape that will cause it to overlap  $o_5$ . Note that this  $k$ -indexical propagates from  $o_5$  to the shape id of  $o_3$ . Pruning of shape ids of polymorphic objects is discussed in Section 5. Initially, no forbidden boxes are generated.

$$s_3 \notin \bigcap \left( \begin{array}{l} \{i \in \text{dom}(s_3) \mid s_3 = i \Rightarrow \ell_{31} > \overline{x_{51}} - 2\} \\ \{i \in \text{dom}(s_3) \mid s_3 = i \Rightarrow \ell_{32} > \overline{x_{52}} - 5\} \\ \text{if } \underline{x_{52}} > 3 \text{ then } \mathbb{Z} \text{ else } \emptyset \end{array} \right) \quad (9)$$

2.  $o_5$  must not overlap  $o_3$ . Note that this  $k$ -indexical propagates from  $o_3$  to  $o_5$ . Initially, it will generate the forbidden box shown in Fig. 9 (top left).

$$o_5.x \notin ([1, (\underline{\ell}_{31} + 1)], [4, (\underline{\ell}_{32} + 4)]) \quad (10)$$

3.  $o_5$  must not overlap  $o_1$ ,  $o_2$  nor  $o_4$ , nor meet  $o_2$  nor  $o_4$ . This  $k$ -indexical will generate the forbidden boxes shown in Fig. 9 (top right).

$$o_5.x \notin \bigcup \left( \begin{array}{c} \left( \begin{array}{c} ([1, 3], [1, 2]) \\ ([2, 3], [2, 3]) \\ ([2, 5], [6, 6]) \\ ([1, 4], [1, 4]) \\ ([4, 4], [1, 6]) \\ ([1, 1], [1, 6]) \\ ([1, 9], [4, 4]) \\ ([1, 9], [1, 1]) \end{array} \right) \\ \cap \\ \left( \begin{array}{c} ([1, 6], [5, 6]) \\ ([1, 9], [5, 5]) \\ ([6, 6], [1, 6]) \\ ([1, 1], [1, 6]) \end{array} \right) \end{array} \right) \quad (11)$$

□

#### 4.5 Filtering Algorithm

We now give a sketch of a filtering algorithm for  $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{R})$ . Let  $I(o)$  denote the set of  $k$ -indexicals for object  $o \in \mathcal{O}$  wrt. the given rules  $\mathcal{R}$ , let  $eval(i)$  denote the evaluation of  $k$ -indexical  $i$  wrt. the current domains, let  $sweep(o, F)$  denote the application of the sweep-based algorithm to the object  $o$  wrt. the forbidden set  $F$ , which prunes the minimum and maximum values of the origin coordinates of  $o$ . Our proposed Algorithm 1 is a straightforward propagation loop.

**Example 5** Returning to our running example, suppose now that the sweep-point kernel wants to adjust the lower bound of  $x_{51}$ . Fig. 9 (bottom) traces the steps performed by the algorithm when it walks from a lexicographically smallest position to the first feasible position of  $o_5$ . The result is that the lower bound of  $x_{51}$  is adjusted to 5. □

## 5 Polymorphism

We say that an object  $o$  is *polymorphic* if its shape id is nonground. This feature could for example be used to model a crate that can be rotated 90 degrees around some axis, in which case each rotated position would correspond to a distinct shape.

In the context of configuration problems, polymorphism can also be used to model the fact that we have to select for an abstract object a possible concrete object that realizes a given function, e.g. selecting a table among different possible table models.

Polymorphism is not a semantic issue, as the declarative semantics is defined in terms of ground objects. But it is an issue for the operational semantics, i.e. for filtering. We now describe how a small extension to the implementation allows to deal with polymorphic objects.

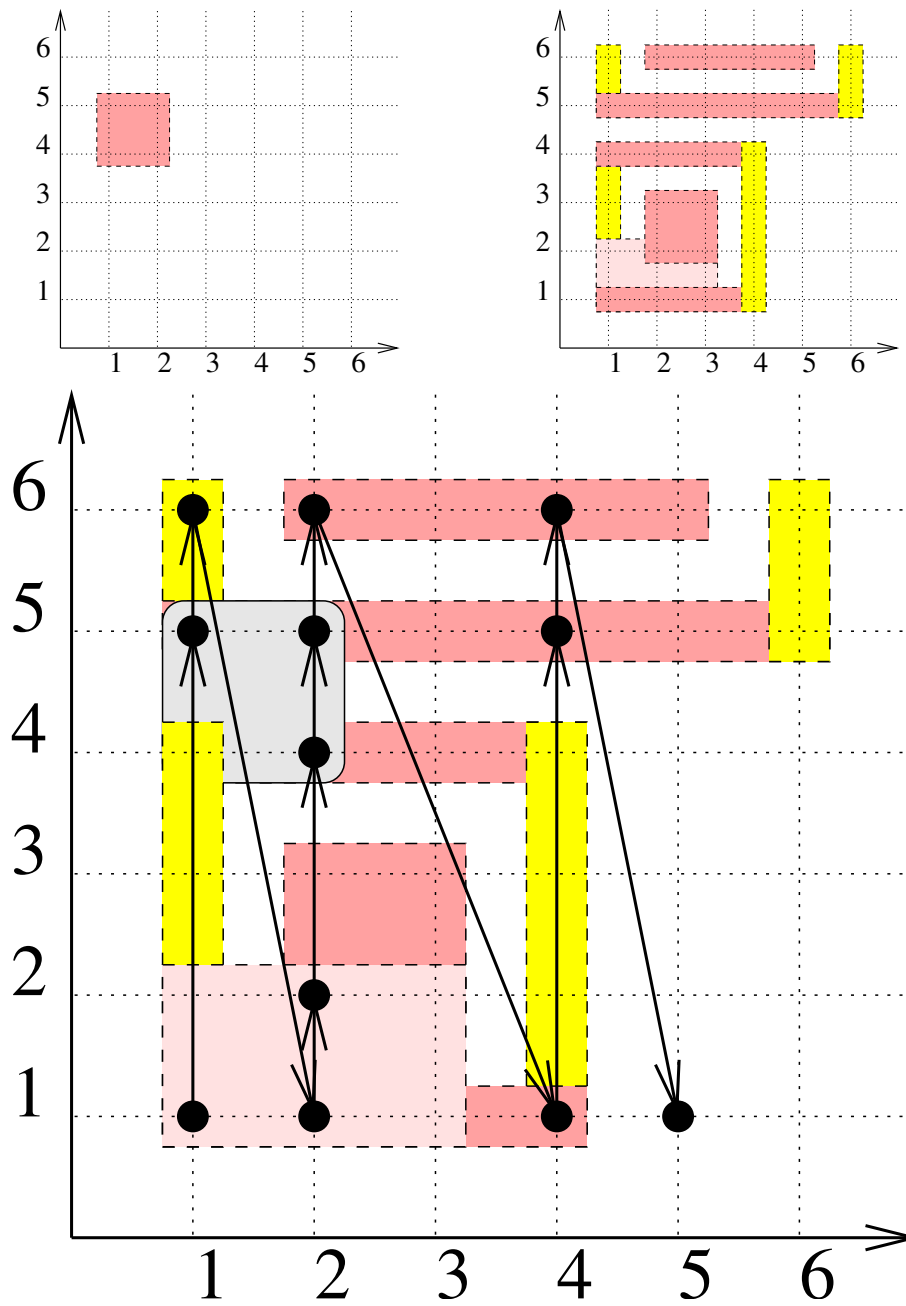


Figure 9: Running example: forbidden boxes generated by (10) (top left) and by (11) (top right). Sequence of candidate positions explored by the sweep-based algorithm in order to reach the feasible position (5, 1) (bottom). The only purpose of using different colors and shadows of grey is to show the borders of the forbidden boxes.

**PROCEDURE** Filter( $\mathcal{O}, I$ )

```

1:  $Q \leftarrow \mathcal{O}$ 
2: while  $Q \neq \emptyset$  do
3:    $o \leftarrow$  some element from  $Q$ 
4:    $Q \leftarrow Q \setminus \{o\}$ 
5:    $F \leftarrow \bigcup \{\text{eval}(i) \mid i \in I(o)\}$ 
6:   if  $\neg \text{sweep}(o, F)$  then
7:     return fail
8:   else if a coordinate of  $o$  was pruned then
9:      $Q \leftarrow Q \cup \{o' \mid I(o') \text{ depends on } o\}$ 
10:  end if
11: end while
12: if all objects in  $\mathcal{O}$  are ground then
13:  return succeed
14: else
15:  return suspend
16: end if

```

**Algorithm 1:** Sketch of a filtering algorithm for  $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{R})$ 

**Polymorphic shifted boxes.** With polymorphic objects, the expanded sentences of the rule language will mention attributes of shifted boxes, where the values of those attributes depend on the shape id. To deal with this complication, we introduce for polymorphic objects  $o$  a virtual  $pbox[j]$  attribute, which stands for the  $j^{th}$  shifted box that has the same shape id as  $o$ . Thus a  $pbox$  attribute behaves like a shifted box but with nonground attributes that have evaluable lower and upper bounds, which is precisely what is needed in order to use the necessary conditions (Table 1) and pruning rules (8). Phase 1 of the rewrite process introduces  $pboxes$  when it encounters an expression  $sboxes([o.sid])$  and  $o$  is polymorphic. Assuming that each possible shape of  $o$  consists of the same number,  $n$ , of shifted boxes, the expression is rewritten to  $[o.pbox[1], \dots, o.pbox[n]]$ . Thus the requirement that  $n$  be fixed is a restriction of the approach.

**Propagating to  $o.sid$ .** We take the approach of treating variable  $o.sid$  as the  $(k+1)^{th}$  dimension, where the sweep-based algorithm treats the  $(k+1)^{th}$  dimension as an *assignment dimension* — it seeks a witness for each value in the domain. For the compilation, all we have to change is to make the indexicals generate forbidden sets in  $\mathbb{Z}^{k+1}$  instead of  $\mathbb{Z}^k$ , and to add two more types of generators of forbidden sets. Table 3 shows the updated table of generators of forbidden sets. Its rows 5 and 6 generate forbidden sets for the assignment dimension  $k+1$ , i.e. for  $o.sid$ .

## 6 Experimental Results

The *geost* constraint, including the rewriting, compilation, and sweep-based algorithms, have been implemented in Prolog using the global constraint programming API of

<b>r</b>	<b><math>F_o(r)</math></b>	<b>condition</b>
$p \vee q$	$F_o(p) \cap F_o(q)$	
$p \wedge q$	$F_o(p) \cup F_o(q)$	
$\sum_i c_i \cdot x_i \geq r$	$\{p \in \mathbb{Z}^{k+1} \mid p[d] < \lceil \frac{r - \text{MAX}(\sum_{i \neq j} c_i \cdot x_i)}{c_j} \rceil\}$	$x_j = o.x[d], c_j > 0$
$\sum_i c_i \cdot x_i \geq r$	$\{p \in \mathbb{Z}^{k+1} \mid p[d] > \lfloor \frac{-r + \text{MAX}(\sum_{i \neq j} c_i \cdot x_i)}{-c_j} \rfloor\}$	$x_j = o.x[d], c_j < 0$
$\sum_i c_i \cdot x_i \geq r$	$\{p \in \mathbb{Z}^{k+1} \mid o.sid = p[k+1] \Rightarrow x_j < \lceil \frac{r - \text{MAX}(\sum_{i \neq j} c_i \cdot x_i)}{c_j} \rceil\}$	$x_j = o.pbox[-]., c_j > 0$
$\sum_i c_i \cdot x_i \geq r$	$\{p \in \mathbb{Z}^{k+1} \mid o.sid = p[k+1] \Rightarrow x_j > \lfloor \frac{-r + \text{MAX}(\sum_{i \neq j} c_i \cdot x_i)}{-c_j} \rfloor\}$	$x_j = o.pbox[-]., c_j < 0$
$\sum_i c_i \cdot x_i \geq r$	<b>if</b> $\text{MAX}(\sum_i c_i \cdot x_i) < r$ <b>then</b> $\mathbb{Z}^{k+1}$ <b>else</b> $\emptyset$	$o \notin \{x_i\}$

Table 3: Mapping a *qffa*  $r$  to a generator of forbidden sets,  $F_o(r)$ , for the object  $o$ , which may be polymorphic.

SICStus Prolog 4 [16], compiled with `gcc -O2` version 4.0.2 on a 3GHz Pentium IV with 1MB of cache.

In order to get a first assessment of the scalability of the approach, we ran a benchmark suite consisting of 84 bin packing problems. In each benchmark instance, a number  $n$  of containers of varying sizes up to  $600 \times 1200 \times 350$  needs to be packed in seven bins of size  $800 \times 1200 \times 1500$ , subject to the constraints:

- No objects overlap.
- Each object is either on the floor or resting on some other object.
- For any two objects in a pile, the overhang can be at most 10 units.

The search was performed by labeling the coordinates of one object at a time. For each instance, we measured two space and one time quantity: (1) the amount of memory in use after posting the constraint, (2) the extra amount of memory in use just after finding the first solution with all choicepoints still open, and (3) time spent posting the constraint and finding the first solution. We report the memory in use in the Prolog stacks after garbage collection.

Fig. 10 summarizes the result. We find that the time and space complexity, static as well as dynamic, is  $O(n^2)$ . The coefficient of the  $n^2$  term is rather high, but when we implement the sweep-based algorithm and all management of forbidden boxes in C, like in the previous version of *geost*, we expect this coefficient to decrease sharply.

## 7 Discussion

**Generality.** Our restriction that object attributes (except shape id and origin) must be ground is somewhat artificial, and we plan to lift it. The rewritten QFPA formulas would simply have more variables per object, and the sweep-based algorithm would

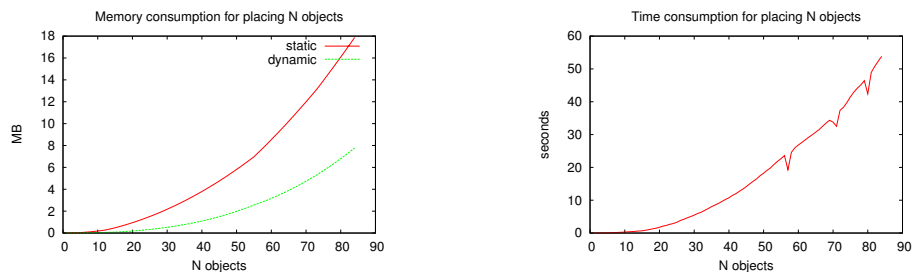


Figure 10: Memory and time consumption for placing  $n$  containers to be placed in seven bins without overlap. The **static** curve is the memory in use just after posting the constraint. The **dynamic** curve is the extra memory in use just after finding the first solution.

deal not with a  $k$ - or  $k + 1$ -dimensional *placement* space, but with an  $m$ -dimensional *solution* space, where  $m$  is the number of possibly nonground attributes per object. In particular, in order to deal with objects whose length in some dimension is a domain variable that occurs in some other constraint, the length and possibly the endpoint would have to be expressed as nonground object attributes. Similarly, to treat the time dimension, we would add three nonground object attributes *start*, *duration*, and *completion*, as in [4], to be included in the solution space.

**Built-in rules.** Non-overlapping constraints are laws of nature and are likely to be present in any packing problem. Similarly, lexicographic ordering constraints are a well-known symmetry breaking device, and are expected to be crucial in problems involving several objects of the same shape. Previously in the project, we have worked out a wealth of powerful, special methods for handling these two constraints. We plan to come up with a software architecture where the general rule mechanism coexists with these special methods. Since both the general and the special methods are based on objects, shifted boxes and the sweep-point kernel, this should present no problem in principle, as long as the methods agree on the set of attributes to use.

**Theoretical properties.** It has been shown [3, Proposition 1-2] that the PKML/Rules2CP rewriting system is confluent and Noetherian (i.e., terminating). Since our rule language is essentially a subset of Rules2CP, the results apply to *geost* rules as well. A size bound on programs generated from Rules2CP is also known [3, Proposition 3] and applies to *geost* provided that min, max and cardinality is not used in the rules, since these operators can cause an exponential (for min and max) resp. quadratic (for cardinality) [8] blow-up. Consequently, one can certainly construct pathological cases where the rewrite phases and/or runtime representation require huge amounts of memory. Even if, at this time, this has not really been a problem for the

instances and rules we have experimented with<sup>8</sup>, one way to manage the complexity of the rewrite phases is to apply simplifying rewrites, e.g. Phase 8, as eagerly as possibly. Another way could be to memoize patterns that have already been rewritten. Finally, common subexpression elimination will mitigate this problem.

## 8 Conclusion

We have presented a global constraint that enforces rules written in a language based on arithmetic and first-order logic to hold among a set of objects. By rewriting the rules to QFPA formulas, we have shown how to compile them to  $k$ -indexicals. Finally, we have shown how the forbidden sets generated by such indexicals can be aggregated by a sweep-based algorithm and used for filtering. Initial experiments support the feasibility of the approach. The approach combines an expressive logic-based rule modeling language for stating business rules with a generic geometrical algorithm for effective and efficient filtering.

## Acknowledgements

This research was conducted under European Union Sixth Framework Programme Contract FP6-034691 “Net-WMS”. The second author was also partly supported by ANR (CANAR/06-BLAN-0383-02). The data and placement rules used in Section C were kindly provided by PSA Peugeot Citroën.

## References

- [1] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.
- [2] N. Beldiceanu, P. Flener, and X. Lorca. Combining tree partitioning, precedence, and incomparability constraints. *Constraints Journal*, 2008. To appear, available at <http://www.springerlink.com/content/08p0h427w7n22681/>.
- [3] F. Fages and J. Martin. From rules to constraint programs with the Rules2CP modelling language. Technical Report, INRIA Rocquencourt, 2008.
- [4] N. Beldiceanu, M. Carlsson, E. Poder, R. Sadek, and C. Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic  $k$ -dimensional objects. In C. Bessière, editor, *Proc. CP'2007*, volume 4741 of *LNCS*, pages 180–194. Springer, 2007. Also available as SICS Technical Report T2007:08, <http://www.sics.se/libindex.html>.

---

<sup>8</sup>They involved at most 100 objects.

- [5] M. Benedetti, A. Lallouet, and J. Vautard. QCSP made practical by virtue of restricted quantification. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 38–43, 2007.
- [6] P. Van Hentenryck, V. Saraswat, and Y. Deville. Constraint processing in cc(FD). unpublished manuscript, Computer Science Department, Brown University, 1991.
- [7] P. Van Hentenryck and Y. Deville. The *cardinality* operator: a new logical connective in constraint logic programming. In *Int. Conf. on Logic Programming (ICLP'91)*. MIT Press, 1991.
- [8] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–25, 2006.
- [9] V. Ganesh, S. Berezin, and D.L. Hill. Deciding presburger arithmetic by model checking and comparisons with other methods. In *Proc. FMCAD'02*, volume 2517 of *LNCS*, pages 171–186. Springer, 2002.
- [10] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.
- [11] P. Codognot and D. Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.
- [12] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Programming Languages: Implementations, Logics, and Programming*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997.
- [13] B. Carlson. *Compiling and Executing Finite Domain Constraints*. PhD thesis, Uppsala University, 1995.
- [14] G. Tack, C. Schulte, and G. Smolka. Generating propagators for finite set constraints. In F. Benhamou, editor, *Proc. CP'2006*, volume 4204 of *LNCS*, pages 575–589. Springer, 2006.
- [15] W. Harvey and P. J. Stuckey. Constraint representation for propagation. In M. Maher and J.-F. Puget, editors, *Proc. CP'98*, volume 1520 of *LNCS*, pages 235–249. Springer, 1998.
- [16] M. Carlsson et al. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, release 4 edition, 2007. ISBN 91-630-3648-7.
- [17] D. A. Randell, Z. Cui, and A. G. Cohn. A spatial logic based on regions and connection. In B. Nebel, C. Rich, and W. R. Swartout, editors, *Proc. of 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, pages 165–176. Morgan Kaufmann, 1992.

## A Prolog Syntax

Table 4 shows the Prolog syntax of the various operators, objects, shifted boxes and attributes.

<b>abstract</b>	<b>Prolog</b>
.	$\hat{\quad}$
$\neg$	$\#\backslash$
$\wedge$	$\#\wedge\backslash$
$\vee$	$\#\backslash/$
$\Rightarrow$	$\#=>$
$\Leftrightarrow$	$\#<=>$
$<$	$\#<$
$=$	$\#=$
$>$	$\#>$
$\leq$	$\#=<$
$\geq$	$\#>=$
$\neq$	$\#\backslash=$
$\forall$	forall
$\exists$	exists
$\#$	card
$@$	fold
$\Longrightarrow$	$--->$
$x[D]$	$x(D)$
$t[D]$	$t(D)$
$l[D]$	$l(D)$
object	<code>object(oid-OID,sid-SID,x-X,Atts)</code>
shifted box	<code>sbox(sid-SID,t-T,l-L,Atts)</code>

Table 4: Abstract syntax vs. Prolog syntax. Atts stands for possible additional attributes of the form Name-Value.

## B Region Connection Calculus Rules

Region Connection Calculus (RCC-8, [17]) provides eight topological relations (i.e., *disjoint*, *meet*, *overlap*, *equal*, *covers*, *coveredby*, *contains*, *inside*) between two ground objects such that any two ground objects are in one and exactly one of these topological relations. Fig. 11 illustrates the meaning of each topological relation. In this section, we provide the corresponding rules in our language for these binary relations.

For objects consisting of multiple shifted boxes, the relations can be interpreted in more than one way. We therefore present two sets of rules: first, unambiguous rules between two shifted boxes, and then one version of rules between objects.

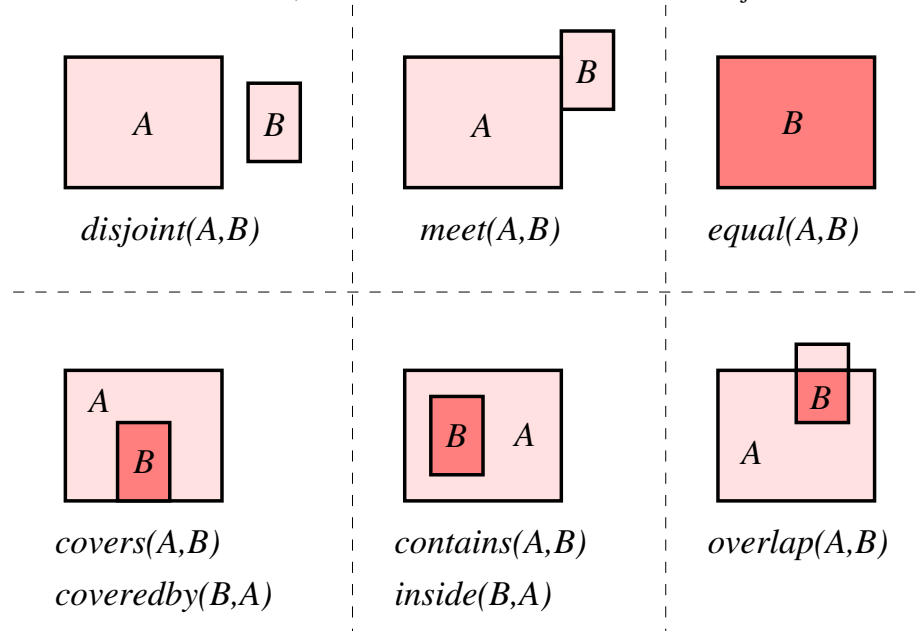


Figure 11: The eight topological relations of RCC-8

## B.1 Rules for RCC-8 Relations between Two Shifted Boxes

```
origin(O1,S1,D) ---> % origin for object O1, sbox S1, dim D
    O1^x(D)+S1^t(D).

end(O1,S1,D) ---> % end for object O1, sbox S1, dim D
    O1^x(D)+S1^t(D)+S1^l(D).

contains_sboxes(Dims, O1, S1, O2, S2) --->
    forall(D, Dims , origin(O1,S1,D) #< origin(O2,S2,D) #/\
        end(O2,S2,D) #< end(O1,S1,D)).

coveredby_sboxes(Dims, O1, S1, O2, S2) --->
    forall(D, Dims , origin(O2,S2,D) #=< origin(O1,S1,D) #/\
        end(O1,S1,D) #=< origin(O2,S2,D)
    ) #/\
    exists(D, Dims , origin(O2,S2,D) #= origin(O1,S1,D) #\
        end(O1,S1,D) #= end(O2,S2,D)).

covers_sboxes(Dims, O1, S1, O2, S2) --->
    forall(D, Dims , origin(O1,S1,D) #=< origin(O2,S2,D) #/\
        end(O2,S2,D) #=< end(O1,S1,D)
    ) #/\
    exists(D, Dims , origin(O1,S1,D) #= origin(O2,S2,D) #\
        end(O1,S1,D) #= end(O2,S2,D)).

disjoint_sboxes(Dims, O1, S1, O2, S2) --->
    exists(D, Dims , origin(O1,S1,D) #> end(O2,S2,D) #\
        origin(O2,S2,D) #> end(O1,S1,D)).

inside_sboxes(Dims, O1, S1, O2, S2) --->
    forall(D, Dims , origin(O2,S2,D) #< origin(O1,S1,D) #/\
        end(O1,S1,D) #< end(O2,S2,D)).

equal_sboxes(Dims, O1, S1, O2, S2) --->
    forall(D, Dims , origin(O1,S1,D) #= origin(O2,S2,D) #/\
        end(O1,S1,D) #= end(O2,S2,D)).

overlap_sboxes(Dims, O1, S1, O2, S2) --->
    forall(D, Dims , end(O1,S1,D) #> origin(O2,S2,D) #/\
        end(O2,S2,D) #> origin(O1,S1,D)).

meet_sboxes(Dims, O1, S1, O2, S2) --->
    forall(D, Dims,
        end(O1,S1,D) #>= origin(O2,S2,D) #/\
        end(O2,S2,D) #>= origin(O1,S1,D)) #/\
    exists(D, Dims,
        end(O1,S1,D) #= origin(O2,S2,D) #\
        end(O2,S2,D) #= 27origin(O1,S1,D)).
```

## B.2 Rules for RCC-8 Relations between Two Objects

```
contains_objects(Dims, O1, O2) --->
  forall(S1, sboxes([O1^sid]),
    exists(S2, sboxes([O2^sid]),
      contains_sboxes(Dims, O1, S1, O2, S2))).

coveredby_objects(Dims, O1, O2) --->
  forall(S1, sboxes([O1^sid]),
    exists(S2, sboxes([O2^sid]),
      coveredby_sboxes(Dims, O1, S1, O2, S2))).

covers_objects(Dims, O1, O2) --->
  forall(S2, sboxes([O2^sid]),
    exists(S1, sboxes([O1^sid]),
      covers_sboxes(Dims, O1, S1, O2, S2))).

disjoint_objects(Dims, O1, O2) --->
  forall(S1, sboxes([O1^sid]),
    forall(S2, sboxes([O2^sid]),
      disjoint_sboxes(Dims, O1, S1, O2, S2))).

inside_objects(Dims, O1, O2) --->
  forall(S1, sboxes([O1^sid]),
    exists(S2, sboxes([O2^sid]),
      inside_sboxes(Dims, O1, S1, O2, S2))).

equal_objects(Dims, O1, O2) --->
  forall(S1, sboxes([O1^sid]),
    exists(S2, sboxes([O2^sid]),
      equal_sboxes(Dims, O1, S1, O2, S2))).

overlap_objects(Dims, O1, O2) --->
  forall(S1, sboxes([O1^sid]),
    exists(S2, sboxes([O2^sid]),
      overlap_sboxes(Dims, O1, S1, O2, S2))).

meet_objects(Dims, O1, O2) --->
  forall(S1, sboxes([O1^sid]),
    exists(S2, sboxes([O2^sid]),
      meet_sboxes(Dims, O1, S1, O2, S2))).
```

## C A Real-Life Problem Instance

This section contains a number of examples of rules encoding a problem instance provided by a major car manufacturer, involving a  $1203 \times 235 \times 239$  container (with *oid* 0) and 9 objects (with *oid* 1-9) with an extra *weight* attribute, subject to the following rules:

**inside** Each object is placed inside the container.

**gravity** Each object is either on the floor or resting on some other object.

**non\_overlap** The objects do not pairwise overlap.

**stack\_weight** A heavier object cannot be piled on top of a lighter one.

**stack\_oversize** For any two objects in a pile, the overhang can be at most 10 units.

The following rule was not used, for it leads to an over-constrained problem.

**wedging** All four faces of a box in the horizontal dimensions must lean against a container wall or against some other box.

Our Prolog implementation solved this problem instance in 1 CPU second and about 1 megabyte of memory. The rules generated 90 *k*-indexicals with a total of 50140 virtual instructions. During the search, the sweep-point kernel was applied 731 times.

### General macros.

```
origin(O1,S1,D) ---> % origin of object O1, sbox S1, dim D
    O1^x(D)+S1^t(D).

end(O1,S1,D) ---> % end of object O1, sbox S1, dim D
    O1^x(D)+S1^t(D)+S1^l(D).

soverlap(O1,O2,S1,S2,D) ---> % sboxes overlap in dim D
    end(O1,S1,D) #> origin(O2,S2,D) #/\
    end(O2,S2,D) #> origin(O1,S1,D).

oversize(O1,O2,S1,S2,D) ---> % overhang between two sboxes
    % in dim D
    max(max(origin(O1,S1,D),origin(O2,S2,D)) -
        min(origin(O1,S1,D),origin(O2,S2,D)),
        max(end(O1,S1,D), end(O2,S2,D)) -
        min(end(O1,S1,D), end(O2,S2,D))).
```

**Inside rule: O1 is non-strictly inside O2 in all dimensions.**

```
inside(O1, O2) --->
  forall(S1, sboxes([O1^sid]),
    exists(S2, sboxes([O2^sid]),
      origin(O2,S2,1) #=< origin(O1,S1,1) #/\
      end(O1,S1,1) #=< end(O2,S2,1) #/\
      origin(O2,S2,2) #=< origin(O1,S1,2) #/\
      end(O1,S1,2) #=< end(O2,S2,2) #/\
      origin(O2,S2,3) #=< origin(O1,S1,3) #/\
      end(O1,S1,3) #=< end(O2,S2,3))).
```

**Non-overlap rule: for some dimension, O1 does not overlap O2.**

```
non_overlap(O1, O2) --->
  O1^oid #< O2^oid #=>
  forall(S1, sboxes([O1^sid]),
    forall(S2, sboxes([O2^sid]),
      #\ soverlap(O1,O2,S1,S2,1) #\ /
      #\ soverlap(O1,O2,S1,S2,2) #\ /
      #\ soverlap(O1,O2,S1,S2,3))).
```

**Gravity rule: O1 is either on the floor or sitting on some other object.**

```
gravity(O1, Os) --->
  forall(S1, sboxes([O1^sid]),
    (origin(O1,S1,3) #= 0 #\ /
    exists(O2,Os,
      O1^oid#\=O2^oid #/\
      exists(S2, sboxes([O2^sid]),
        soverlap(O1,O2,S1,S2,1) #/\
        soverlap(O1,O2,S1,S2,2) #/\
        origin(O1,S1,3) #= end(O2,S2,3)))))).
```

**Stacking rule: O1 heavier than O2 ⇒ O1 not piled above O2.**

```
stack_weight(O1, O2) --->
  O1^weight #> O2^weight #=>
  forall(S1, sboxes([O1^sid]),
    forall(S2, sboxes([O2^sid]),
      origin(O1,S1,3) #>= end(O2,S2,3) #=>
      #\ soverlap(O1,O2,S1,S2,1) #\ /
      #\ soverlap(O1,O2,S1,S2,2))).
```

**Overhang rule: for any two objects in a pile, the overhang can be at most 10.**

```
stack_oversize(O1, O2) --->
  O1^oid#\=O2^oid #=>
  forall(S1, sboxes([O1^sid]),
    forall(S2, sboxes([O2^sid]),
      (soverlap(O1,O2,S1,S2,1) #/\
        soverlap(O1,O2,S1,S2,2)) #=>
      (oversize(O1,O2,S1,S2,1) #=< 10 #/\
        oversize(O1,O2,S1,S2,2) #=< 10))).
```

**Wedging rule: all four faces of O1 in dimension X and Y must lean against the container or against some other box.**

```
wedged(O1,S1,Oc,Sc,Os,D) --->
  (origin(O1,S1,D) #= origin(Oc,Sc,D) #\ /
  exists(O2,Os,
    O1^oid#\=O2^oid #/\
    exists(S2, sboxes([O2^sid]),
      origin(O1,S1,D) #= end(O2,S2,D)))) #/\
  (end(O1,S1,D) #= end(Oc,Sc,D) #\ /
  exists(O2,Os,
    O1^oid#\=O2^oid #/\
    exists(S2, sboxes([O2^sid]),
      end(O1,S1,D) #= origin(O2,S2,D)))).
wedging(O1,Oc,Os) --->
  exists(Sc, sboxes([Oc^sid]),
    forall(S1, sboxes([O1^sid]),
      wedged(O1,S1,Oc,Sc,Os,1) #/\
      wedged(O1,S1,Oc,Sc,Os,2))).
```

**Business rules: putting all the rules together.**

```
forall(Box1,objects([1,2,3,4,5,6,7,8,9]),
  forall(Container,[objects([0])],inside(Box1,Container)) #/\
  gravity(Box1,objects([1,2,3,4,5,6,7,8,9])) #/\
  forall(Box2,objects([1,2,3,4,5,6,7,8,9]),
    non_overlap(Box1,Box2) #/\
    stack_weight(Box1,Box2) #/\
    stack_oversize(Box1,Box2))).
```

## D A Packing-Unpacking Problem

This section introduces a packing-unpacking problem that takes the space as well as the time dimensions into account. We have to pack (and unpack) a set of 48 rectangles into a bin. Each rectangle is present within the bin during a given time interval and the right hand side of the bin can be used for inserting and deleting rectangles. Beside the fact that, for each time point  $p$ , all rectangles that are present in the bin at instant  $p$  should not overlap, we also have a visibility constraint, which states that, when a rectangle enters (or leaves) the bin, there should not be any obstacle between the final (initial) position of the rectangle and the right hand side of the bin (we assume that the rectangle performs a direct translation).

The example illustrates how a packing plan can be obtained for such a packing-unpacking problem from a solution to a *geost* constraint problem. The example uses problem dimensions 1-2 for space and 3-5 for time, denoting respectively the virtual attributes *start*, *duration*, and *completion*. We now introduce the visibility constraint.

**Definition 3** Given a list *OIDs* of identifiers of objects of the *geost* constraint and an observation place, specified by a dimension *Dim* (an integer between 1 and  $k$ ) and a direction *Dir* (0 or 1), the *visible(OIDs, Dim, Dir)* constraint holds if, for all objects  $o$  mentioned in *OIDs*, at least one surface of each shifted box associated with  $o$  is entirely visible from the specified observation place  $\langle Dim, Dir \rangle$  at time  $o.start$ <sup>9</sup> as well as at time  $o.completion - 1$ .<sup>10</sup>

**Definition 4** Consider two distinct objects  $o$  and  $o'$  of the *visible(OIDs, Dim, Dir)* constraint (i.e.,  $o, o' \in OIDs$ ) as well as an observation place defined by the pair  $\langle Dim, Dir \rangle$ . The object  $o$  is masked by the object  $o'$  according to the observation place  $\langle Dim, Dir \rangle$  if there exist two shifted boxes  $s$  and  $s'$  respectively associated with  $o$  and  $o'$  such that conditions **A**, **B**, **C** and **D** all hold:

- A**  $o.duration > 0 \wedge o'.duration > 0 \wedge o.completion > o'.start \wedge o'.completion > o.start$  (i.e., the time intervals associated with  $o$  and  $o'$  intersect).
- B**  $s$  and  $s'$  intersect in all dimensions but *Dim* (i.e.,  $s$  and  $s'$  are in *vis-à-vis*).
- C**  $s$  precedes  $s'$  in dimension *Dim* and *Dir* = 1, or  $s'$  precedes  $s$  in dimension *Dim* and *Dir* = 0.
- D** At least one of the two instants respectively corresponding to the start time of  $o$  and to the completion time of  $o$  is located within interval  $[o'.start, o'.completion]$ .

Our Prolog implementation solved this problem instance in 7.5 CPU second and about 2.2 megabytes of memory. The rules generated 1744 *k*-indexicals with a total

---

<sup>9</sup>We assume that all objects for which the start time equals  $o.start$  are transparent. This makes sense since: (1) within the context of pick-up delivery problems all objects loaded (resp. unloaded) at the same place are equivalent; (2) by enforcing the start time to be distinct (for instance by using an *alldifferent* constraint on the start variables) one can impose the objects to be opaque.

<sup>10</sup>Again, we assume that all objects for which the completion time equals  $o.completion$  are transparent.

of 65456 virtual instructions. During the search the sweep-point kernel was applied 4502 times. The result is shown in Fig. 12. The four parts of the figure respectively correspond to the successive states of the bin (i.e., we have four time intervals):

**top** Initially, rectangles 1 to 16 enter the bin.

**bottom left** Later on, rectangles 17 to 32 enter the bin. They are placed into the bin in order not to block according to the right hand side of the bin, rectangles 1 to 16 which have to leave earlier.

**bottom center** Rectangles 1 to 16 leave the container and are replaced by rectangles 33 to 48. Again they are placed in order not to block the exit of rectangles 17 to 32.

**bottom right** After the exit of rectangles 17 to 32, rectangles 33 to 48 are the only rectangles left in the bin.

We now give the encoding of the problem.

### Shorthands and invariants for space and time.

```
% end of a rectangle in dimension 1 or 2
origin(O, S, D) ---> O^x(D)+S^t(D).

% end of a rectangle in dimension 1 or 2
end(O, S, D) ---> O^x(D)+S^t(D)+S^l(D)).

% start time (use dimension 3)
start(O) ---> O^x(3).

% duration (use dimension 4)
duration(O) ---> O^x(4).

% completion time (use dimension 5)
completion(O) ---> O^x(5).

% time attribute invariant: Start+Duration=Completion
start_dur_complete(OIDs) --->
  forall(O, objects(OIDs),
    start(O)+duration(O) #= completion(O)).
```

### Non-overlapping constraints considering both space and time.

```
overlap(O, S, Oi, Si, D) --->
  end(O, S, D) #> origin(Oi, Si, D) #/\
  end(Oi, Si, D) #> origin(O, S, D)).

non_overlap(OIDs) --->
  forall(O1, objects(OIDs),
    forall(S1, sboxes([O1^sid]),
      forall(O2, objects(OIDs),
        O1^oid #< O2^oid #=>
          forall(S2, sboxes([O2^sid]),
            #\ (overlap(O1, S1, O2, S2, 1) #/\
              overlap(O1, S1, O2, S2, 2) #/\
              completion(O1) #> start(O2) #/\
              completion(O2) #> start(O1)))))).
```

### Visibility rules.

```
visible(OIDs, Dim, Dir) --->
  #\ exists(O, objects(OIDs), masked(OIDs, O, Dim, Dir)).

masked(OIDs, O, Dim, Dir) --->
  exists(Oi, objects(OIDs),
    Oi^oid #\= O^oid #/\ masked_by(O, Oi, Dim, Dir)).

masked_by(O, Oi, Dim, Dir) --->
  exists(S, sboxes([O^sid]),
    exists(Si, sboxes([Oi^sid]),
      duration(O) #> 0 #/\
      duration(Oi) #> 0 #/\
      completion(O) #> start(Oi) #/\
      completion(Oi) #> start(O) #/\
      forall(D, [1,2], D #\= Dim #=> overlap(O, S, Oi, Si, D)) #/\
      (Dir #= 0 #=> origin(O, S, Dim) #>= end(Oi, Si, Dim)) #/\
      (Dir #= 1 #=> origin(Oi, Si, Dim) #>= end(O, S, Dim)) #/\
      (start(O) #> start(Oi) #\ / completion(O) #< completion(Oi)))).
```

### Business rules: putting all the rules together.

```
start_dur_complete(AllOIDs),
non_overlap(AllOIDs),
visible(AllOIDs, 1, 0).
```



Figure 12: Solution to the packing-unpacking problem