Rules2CP and PKML User's Manual

François Fages, Julien Martin
INRIA Paris-Rocquencourt, France

February 5, 2010

Contents

Chapter 1

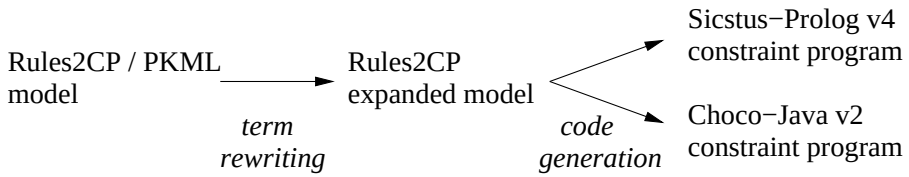
Getting Started

1.1 Overview

Rules2CP is a general purpose rule-based modeling language for constraint programming [4]. It aims at making constraint programming technology easier to use by non-programmers, by modeling combinatorial optimization problems with logical rules and elementary data structures, and by allowing the building of Rules2CP libraries for specific problems.

The Packing Knowledge Modeling Language PKML is such a library developed in the framework of the Net-WMS European project for higher-dimensional bin packing problems taking into account extra placement constraints and specific industrial requirements.

The Rules2CP compiler transforms Rules2CP statements into constraint programs in different target systems, currently Sicstus-Prolog and Choco-Java. More precisely, the Rules2CP compiler implements the following transformations:



This manual describes the syntax and built-in predicates of the Rules2CP modeling language, and the predefined predicates of the PKML library.

Rules2CP is an open-source software distributed under the GPL license and available at `http://contraintes.inria.fr/rules2cp`.

1.2 Running Rules2CP

The Rules2CP v1 compiler has two different target languages: Sicstus Prolog 4.0.4 [3] and Choco Java 2.0.0.3 [7]. Rules2CP (and PKML) source file names are suffixed by `rcp`.

Unpack the archive to *installation-path* and add *installation-path*`/rules2cp/compiler/bin/` to your PATH environment variable. Add *installation-path* to your CLASSPATH environment variable.

- The commands to execute in the installation directory to build the Rules2CP compiler are for:
  - SICStus-Prolog : `rcp.make` *r2cp-version*
    (produces `r2cp-`*r2cp-version* and set `rcpdir` default value to *installation-path*)
  - Choco-Java : `rcpchoco.make` *r2cp-version*
    (produces `r2cpchoco-`*r2cp-version* and set `rcpdir` default value to *installation-path*)
- For compiling a Rules2CP/PKML model `file.rcp` to a:
  - SICStus-Prolog program: use command
    `r2cp file.rcp [-rcppath paths] [-rcpdir path]`
    to produce the program `file.pl`
  - Choco-Java program: use command
    `r2cpchoco file.rcp [-rcppath paths]`
    to produce the program `file.java`

    the option `-rcppath` defines a list of paths separated by "`:`" where to find Rules2CP source files. Alternatively, instead of passing the value as an argument of `r2cp`, set the `RCPPATH` environment variable.

    the option `-rcpdir` defines the Rules2cp *installation-path*. Alternatively, instead of passing the value as an argument of `r2cp`, set the `RCPDIR` environment variable.

Chapter 2

Rules2CP Syntax

Rules2CP manipulates the following lexical entities:

- an *ident* is a word beginning with a lower case letter, or any word between simple quotes.
- a *name* is an identifier that can be prefixed by other identifiers for module and package names.
- a *variable* is a word beginning with either an upper case letter or the underscore character _.
- a *string* is a sequence of characters between double quotes.
- a *comment* is a sequence of characters beginning with `%` and ending with the end of line. All comments are ignored.

The syntax of Rules2CP is given by the following grammar.

statement	*::=*	`import` *name*`.`	module import
	|	*head* `=` *expr*`.`	declaration
	|	*head* `-->` *fol*`.`	rule
	|	`?` *fol*`.`	goal
head	*::=*	*ident*	
	|	*ident(variable,...,variable)*	
fol	*::=*	*varbool*	boolean
	|	*expr relop expr*	comparison
	|	*expr* `in` *expr*	domain
	|	*name*	
	|	*name(expr,...,expr)*	relation
	|	`not` *fol*	negation
	|	*fol logop fol*	logical operator
	|	`forall`*(variable,expr,fol)*	universal quantifier
	|	`exists`*(variable,expr,fol)*	existential quantifier
	|	`let`*(variable,expr,fol)*	variable binding
	|	`foldr`*(variable,expr,logop,expr,expr)*	logical fold right
	|	`foldl`*(variable,expr,logop,expr,expr)*	logical fold left
expr	*::=*	*varint*	
	|	*fol*	reification
	|	*string*	
	|	`[` *enum* `]`	list
	|	`{`*name* `=` *expr,...,name*`=` *expr*`}`	record
	|	*name*	
	|	*name(expr,...,expr)*	function
	|	*expr op expr*	
	|	`foldr`*(variable,expr,op,expr,expr)*	fold left
	|	`foldl`*(variable,expr,op,expr,expr)*	fold right
	|	`map`*(variable,expr,expr)*	list mapping
enum	::=	*enum* `,` *enum*	enumeration
	|	*expr*	value
	|	*expr* `..` *expr*	interval of integers
varint	*::=*	*variable*	
	|	*integer* | `min_integer` | `max_integer`	integers
varbool	*::=*	*variable*	
	|	`0`	false
	|	`1`	true
op	*::=*	`+` | `-` | `*` | `/` | `min` | `max` | `abs` | `log` | `exp`	arithmetic
relop	*::=*	`<` | `=<` | `=` | `#` | `>=` | `>`	arithmetic comparisons
logop	*::=*	`and` | `or` | `implies` | `equiv` | `xor`	logical connectives
name	*::=*	*ident*	
	|	*name*`:`*ident*	module prefix

Chapter 3

Data structures with their Built-in Functions and Predicates

The only data structures are booleans, integers, finite domain variables, strings, enumerated lists and records.

3.1 Booleans

The Boolean constants `true` and `false` are represented by the integers `1` and `0` respectively.

The usual Boolean operations are described in the syntax table 2 under the item *logop*.

3.2 Integers

The integer constants are noted as usual, e.g. `-2`, `0`, `42`...

The arithmetic operations and the usual ordering relations on integers are described in the syntax table under the items *op* and *relop* respectively.

- `min_integer`
  - represents the least integer.
- `max_integer`
  - represents the greatest integer.

3.3 Finite Domain Variables

A Rules2CP variable represents either a parameter of a function or predicate, or an unknown integer or boolean, called a finite domain (FD) variable.

A FD variable can be given an initial domain as a list of integers or intervals, with the following built-in predicates:

- *X* `in` *list*
  - constrains the variable *X* to take integer values in a list of integer values.
- `domain(`*expr, min, max*`)`
  - constrains the list of variables occurring in the expression *expr* to take integer values between *min* and *max*.

The arithmetic operators described above can be used with FD variables to create arithmetic constraints. Furthermore, the following built-in constraints are available:

- `all_different(`*list*`)`
  - where the argument is a list of FD variables or integers. The constraint holds if the elements of the list are all different.
- `lexicographic(`*list*`)`
  - where the argument is a list of lists of FD variables or integers. The constraint holds if the lists are in ascending or equal lexicographic order.
- `lexicographic_strict(`*list*`)`
  - where the argument is a list of lists of FD variables or integers. The constraint holds if the lists are in strictly ascending lexicographic order.
- `non_overlapping(`*list of PKML objects, list of dimensions*`)`
  - constrains a list of PKML objects to non-overlap in a given list of dimensions (see Section 11.3). The FD variables of this constraint are the coordinates of the objects. This built-in constraint uses the global constraint `geost` of the target systems [2].
- `non_overlapping(`*list of PKML objects, list of dimensions, fd variable, list of patterns*`)`
  - Defined as `non_overlapping/`*2* and additionally gives control on a greedy assignment of PKML objects' variables.

    The *fd variable* Flag is a 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 a *list of patterns* 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 i is fixed according to pattern i modulo m.

For example, suppose that the following option is given:
[`object`(min(1), [min(3), max(2)]), `object`(max(1), [min(2), max(3)])]
Then, if the program binds Flag 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.

The following built-in functions return information on the domain of the FD variables. These functions cannot be evaluated statically but can be called under the scope of the `dynamic` predicate.

- `domain_min(`X`)`

  returns the lower bound of the domain of X.

- `domain_max(`X`)`

  returns the upper bound of the domain of X.

- `domain_size(`X`)`

  returns the size of the domain of variable X.

3.4 Strings

A string is a sequence of characters between double quotes.

3.5 Lists

Lists are formed by enumerating all their elements between brackets. For instance `[1, 3, 4, 5, 6, 8]` is a list of integers which can also be written as a list of intervals as `[1,3..6,8]`. There is no binary list constructor.

The following built-in functions are predefined on lists:

- `length(`*list*`)`

  returns the length of the list (after expansion of the intervals). It is an error if the argument is not a list.

- `nth(`*integer,list*`)`

  returns the element of the list in the position (counting from 1) indicated by the first argument, or an error if the second argument is not a list containing the first argument.

- `pos(`*element,list*`)`

  returns the first position of an element occurring in a list as an integer (counting from 1), or returns an error if the element does not belong to the list.

- `variables(`*expr*`)`

  returns the list of finite domain variables contained in an expression, i.e. occurring as attributes of a record, or recursively in a record referenced by attributes, in a list, or in a first-order formula.

Furthermore, the following functions on lists of integers are predefined as follows (in library `lib/common/rcp.rcp`):

- `sum(L) = foldr(X, L, +, 0, X).`

  Returns the sum of integers contained in the list `L`.

- `product(L) = foldr(X, L, *, 1, X).`

  Returns the product of integers contained in the list `L`.

- `maximum(L) = foldr(X, L, max, min_integer, X).`

Returns the greatest integer contained in the list `L`.

- `minimum(L) = foldr(X, L, min, max_integer, X).`

  Returns the least integer contained in the list `L`.

3.6 Records

Records are constructed by enumerating their attribute names and values between braces with expressions of the form `{`ident `=` expr,...,ident`=` expr`}`.

For instance `{start=_, duration=2}` is a record representing a task with a variable start point and a fixed duration.

All records have an implicit integer attribute `uid`. This attribute provides a unique identifier for each record.

The attribute value of a record is accessed with the following built-in function:

- *attribute*`(`*record*`)`

  returns the expression associated to an attribute name of a record, or returns an error if the argument is not a record or does not have this attribute.

Chapter 4

User-defined Functions and Predicates

4.1 Functions defined by Declarations

New *functions* can be defined with declarations of the form

- *head* `=` *expr*.
  - defines the head as a shorthand for the right-hand side expression.

In a declaration, a variable occurring in the body expression and not in the head is a finite domain variable representing an unknown of the problem.

4.2 Predicates defined by Rules

New *predicates* can be defined with rules of the form

- *head* `-->` *fol*.
  - defines the head as a shorthand for the right-hand side formula.

In a rule, the variables in the body formula are assumed to appear in the head. A rule thus cannot introduce finite domain variables.

Chapter 5

Combinators

Rules2CP does not allow recursion in declarations and rules. Built-in combinators are thus available to define various iterations. Combinators cannot be defined in first-order logic and are thus Rule2CP built-ins.

The first-argument of a combinator is a variable X used to denote place holders in an expression. The second argument is an expression or a list representing the unique or successive values of X in the expression formed according to the following arguments.

5.1 Let

- `let(`X`, `e`, `ϕ`) = `$\phi[X/e]$

  substitutes X `for` e `in` ϕ

5.2 Map

- `map(`X`,[`e_1`, `$\cdots$`, `e_N`],`ϕ`) = [`$\phi[X/e_1]$`, `$\cdots$`, `$\phi[X/e_N]$`]`

where $\phi[X/e]$ denotes the expression ϕ where the free occurrences of X have been replaced by e.

5.3 Forall, Exists

- `forall(`X`, [`e_1`, `$\cdots$`, `e_N`], `ϕ`) = `$\phi[X/e_1] \wedge \cdots \wedge \phi[X/e_N]$

- `exists(`X`, [`e_1`, `$\cdots$`, `e_N`], `ϕ`) = `$\phi[X/e_1] \vee \cdots \vee \phi[X/e_N]$

5.4 Fold right, Fold left

- `foldr(`X`, [`e_1`, ⋯, `e_N`], `op`, `e`, `ϕ`) =` $\phi[X/e_1]$ op $(...$ op $(\phi[X/e_N]$ op $e))$

  iteratively combines the first element of the list with the result of the combination of the tail of the list

- `foldl(`X`, [`e_1`, ⋯, `e_N`], `op`, `e`, `ϕ`) =` $((e$ op $\phi[X/e_1])$ op $...)$ op $\phi[X/e_N]$

  iteratively combines the result of the combination of the first elements of the list with the last element of the list

The most general combinators are the left and right fold combinators. The `let`, `forall` and `exists` combinators are defined for convenience but are equivalent to the following folds:

```
let(X, E, F) = foldr(X, [E], and, 1, F)
forall(X, L, E) = foldr(X, L , and, 1, F)
exists(X, L , E) = foldr(X, L , or, 0, F)
```

Chapter 6

Search

In Rules2CP, decision variables and branching formulae of the problem are specified in a declarative manner, as well as heuristics as preference orderings.

6.1 Enumeration of Variables

- `labeling(`*expr*`)`

  specifies the enumeration of the possible values of all the variables *contained* in an expression.

6.2 And/Or Search Trees

- `search(`*fol*`)`

  specifies a *branching formula*, i.e. an and/or search tree explored by branching on all the disjunctions, implications and existential quantifications occurring in the formula.

The negations in the formula are eliminated by descending them to the constraints. In order to avoid an exponential growth of the formulae, `equiv` and `xor` formulae are kept as constraints and are not treated as choice points.

6.3 Optimization

- `minimize(`*fol,expr*`), maximize(`*fol,expr*`)`

  specifies a *branching formula* (like `search`) together with an optmization criterion given as an expression.

Chapter 7

Heuristics

Heuristics for guiding the search are stated in Rules2CP as preference orderings on choice points and branches.

Two pairs of predicates are predefined for specifying choice criteria between variables and values for `labeling`, and between conjunctive and disjunctive formulae for `search`.

The variables and values (resp. conjunctive and disjunctive formulae) occuring in a `labeling` (resp. `search`) are ordered according to the last encountered heuristics statement in the model.

7.1 Variable and Value Choice Heuristics for Labeling

- `variable_ordering([`*ident*`(`*expr*`), ···, `*ident*`(`*expr*`)])`

  ident is an identifier among the following :

  - `greatest` for selecting variables in descending order of *expr* value,
  - `least` for selecting variables in ascending order of *expr* value,
  - `any` for selecting variables for which the *expr* applies, independently of its value,
  - `is` for selecting a variable if it is equal to the *expr* value.

  The expression *expr* in a criterion contains the symbol `^` for denoting, for any variable, the left-hand side of the Rules4CP declaration that introduced that variable. If the expression cannot be evaluated on a given variable, the criterion is ignored.

  The list of criteria is used for ordering the variables in the next `labeling` predicate appearing in the model (see Example 10.1). The variables are sorted according to the first criterion when it applies, then the second, etc. The variables for which no criterion applies are considered at the end for labeling in the syntactic order.

- `value_ordering([`*ident*`(`*expr*`), ···, `*ident*`(`*expr*`)])`

  ident is an identifier among the following :

  - `up` for enumerating values in ascending order of *expr* value,
  - `down` for enumerating values in descending order of *expr* value,
  - `step` for binary choices,
  - `enum` for multiple choices,
  - `bisect` for dichotomy choices,

  and where *expr* is an expression containing the symbol `^` which denotes the left-hand side of the Rules 2CP declaration that introduces a given variable.

  A criterion applies to a variable if it matches the expression *expr*.

7.2 Conjunct and Disjunct Choice Heuristics for Search

In search trees defined by logical formulae, the criteria for `conjunct_ordering` and `disjunct_ordering` heuristics are defined similarly by pattern matching on the rule heads that introduce conjunctive and disjunctive formulae under the `search` predicate. This is illustrated in Example 10.2 with conditional expressions of the form `if ^ is` ϕ; where `^` denotes the conjunct or disjunct candidate for matching ϕ, and ϕ denotes either a rule head or directly a formula. The conjuncts or disjuncts for which no criterion applies are considered last, in the syntactic order.

- `conjunct_ordering([`*ident*`(`*expr*`), ···, `*ident*`(`*expr*`)])`

  ident is identifier among the following :

  - `greatest` for selecting conjuncts in descending order of *expr* value,
  - `least` for selecting conjuncts in ascending order of *expr* value,

  A criterion applies to a conjunct if it matches the expression *name*`(`$expr_1$`,···,`$expr_n$`)`.

- `disjunct_ordering([`*ident*`(`*expr*`), ···, `*ident*`(`*expr*`)])`

  ident is an identifier among the following :

  - `greatest` for selecting disjuncts in descending order of *expr* value,
  - `least` for selecting disjuncts in ascending order of *expr* value,

where *expr* is a conditional expression of the form :

expr `if ^ is` *name*`(`*expr*$_1$`, ⋯, `*expr*$_n$`)`

where the symbol `^` denotes a Rules2CP predicate of arity n.
A criterion applies to a disjunct if it matches the expression *name*`(`*expr*$_1$`,⋯, `*expr*$_n$`)`.

Chapter 8

Interpreter

It happens that combinatorial problems can be decomposed and that the definition of one sub-problem depends on the value or bounds of the FD variables of another component (see example 12.2). As the values of FD variables are unknown at compile-time, the term expansion cannot take place due to a lack of instantiation and would produce an error in the compiler. This is precisely when we need to use the Rules2CP interpreter with the `dynamic` predicate.

- `dynamic(`fol`)`

  prevents the compiler from expanding term fol, postpones its evaluation to run time in interpreted mode.

The only Rules2CP expressions that evaluate differently at run time than at compile time are the FD variable domain built-ins. The `dynamic` predicate should thus be used to evaluate the Rules2CP expressions that depend on the bounds (or value) of an FD variable.

The `dynamic` predicate can also be used to limit the size of the generated code.

Note that in version 1.0 of the Rules2CP compiler, the interpreter (i.e. the `dynamic` predicate) is implemented in SICStus Prolog but is not available in Choco Java.

Chapter 9

Error Messages

- `... not parsed as ...`

  Syntax error, the statement is ill-formed.

- `... is unknown.`

  Unknown function or predicate.

- `... should had been reduced to ...`

  Type error, the expression is not of the expected type.

- `unbound variable ...`

  Variable instanciation error.

- `... is not accepted by ... language`

  Function or predicate not implemented for the target language.

Chapter 10

Simple Rules2CP Examples

10.1 NQueens

This is a standard combinatorial puzzle introduced by Bezzel in 1848, for putting N queens on a chessboard of size $N \times N$ such that they do not attack each other, i.e. they are not placed on the same row, column or diagonal.

```
q(I) = {row = _, column = I}.

board(N) = map(I, [1..N], q(I)).

safe(L) -->
        all_different(L) and
        forall(Q, L,
          forall(R, L,
            let(I, column(Q),
            let(J, column(R),
            I < J implies
            row(Q) # J - I + row(R) and
            row(Q) # I - J + row(R))))).

queens_constraints(B, N) -->
          domain(B, 1, N) and safe(B).

queens_search(B) -->
          variable_ordering([least(domain_size(row(^)))]) and
          labeling(B).

? let(N, 4, let(B, board(N),
    queens_constraints(B, N) and dynamic(queens_search(B)))).
```

10.2 Disjunctive Scheduling

Scheduling problems are optimization problems in which we are interested in computing an ordering of the tasks that minimizes the start date of the last task. Each task is given a duration and a time window for its start date. There are precedence constraints between tasks. Furthermore, disjunctive scheduling problems include mutual exclusion constraints (for tasks sharing a same ressource) which make the disjunctive scheduling problem NP-hard in general.

```
t1 = {start=_, duration=2}.
t2 = {start=_, duration=5}.
t3 = {start=_, duration=4}.
t4 = {start=_, duration=3}.
t5 = {start=_, duration=1}.

cost = start(t5).

prec(T1, T2) -->
        start(T1) + duration(T1) =< start(T2).

disj(T1, T2) -->
        prec(T1, T2) or prec(T2, T1).

precedences -->
        prec(t1, t2) and prec(t2, t5) and prec(t1, t3) and
        prec(t3, t5) and prec(t1, t3) and prec(t3, t5).

disjunctives -->
        disj(t2, t4) and  disj(t3, t4) and disj(t2, t3).

? domain([t1, t2, t3, t4, t5], 0, 20) and precedences and
  conjunct_ordering([greatest(duration(A) + duration(B)) if ^ is disj(A,B)]) and
  disjunct_ordering([greatest(duration(A)) if ^ is prec(A, B)]) and
  minimize(disjunctives, cost).
```

10.3 Bridge Problem

This is the classical disjunctive scheduling problem for the construction of a bridge [8].

```
import('rules2cp/lib/common/rcp').

first = {start=_, duration=0}.

a1 = {start=_, duration=4}.
a2 = {start=_, duration=2}.
```

```
a3 = {start=_, duration=2}.
a4 = {start=_, duration=2}.
a5 = {start=_, duration=2}.
a6 = {start=_, duration=5}.

p1 = {start=_, duration=20}.
p2 = {start=_, duration=13}.

ue = {start=_, duration=10}.

s1 = {start=_, duration=8}.
s2 = {start=_, duration=4}.
s3 = {start=_, duration=4}.
s4 = {start=_, duration=4}.
s5 = {start=_, duration=4}.
s6 = {start=_, duration=10}.

b1 = {start=_, duration=1}.
b2 = {start=_, duration=1}.
b3 = {start=_, duration=1}.
b4 = {start=_, duration=1}.
b5 = {start=_, duration=1}.
b6 = {start=_, duration=1}.

ab1 = {start=_, duration=1}.
ab2 = {start=_, duration=1}.
ab3 = {start=_, duration=1}.
ab4 = {start=_, duration=1}.
ab5 = {start=_, duration=1}.
ab6 = {start=_, duration=1}.

m1 = {start=_, duration=16}.
m2 = {start=_, duration=8}.
m3 = {start=_, duration=8}.
m4 = {start=_, duration=8}.
m5 = {start=_, duration=8}.
m6 = {start=_, duration=20}.

l1 = {start=_, duration=2}.

t1 = {start=_, duration=12}.
t2 = {start=_, duration=12}.
t3 = {start=_, duration=12}.
t4 = {start=_, duration=12}.
t5 = {start=_, duration=12}.
```

```
ua = {start=_, duration=10}.
v1 = {start=_, duration=15}.
v2 = {start=_, duration=10}.
k1 = {start=_, duration=0}.
k2 = {start=_, duration=0}.

last = {start=_, duration=0}.

cost = start(last).

end(Task) = start(Task) + duration(Task).

maxDuration = sum(map(T, tasks, duration(T))).

tasks = [first, a1, a2, a3, a4, a5, a6, p1, p2, ue, s1, s2, s3, s4, s5, s6,
         b1, b2, b3, b4, b5, b6, ab1, ab2, ab3, ab4, ab5, ab6,
         m1, m2, m3, m4, m5, m6,
         l1, t1, t2, t3, t4, t5, ua, v1, v2, k1, k2, last].

precedences_list = [[first, a1], [first, a2], [first, a3], [first, a4],
                    [first, a5], [first, a6], [first, ue],
                    [a1, s1], [a2, s2], [a5, s5], [a6, s6], [a3, p1], [a4, p2],
                    [p1, s3], [p2, s4], [p1, k1], [p2, k1],
                    [s1, b1], [s2, b2], [s3, b3], [s4, b4], [s5, b5], [s6, b6],
                    [b1, ab1], [b2, ab2], [b3, ab3], [b4, ab4], [b5, ab5], [b6, ab6],
                    [ab1, m1], [ab2, m2], [ab3, m3], [ab4, m4], [ab5, m5], [ab6, m6],
                    [m1, t1], [m2, t1], [m2, t2], [m3, t2], [m3, t3], [m4, t3], [m4, t4], [
                        m5, t4],
                    [m5, t5], [m6, t5], [m1, k2], [m2, k2], [m3, k2], [m4, k2], [m5, k2], [
                        m6, k2],
                    [l1, t1], [l1, t2], [l1, t3], [l1, t4], [l1, t5],
                    [t1, v1], [t5, v2], [t2, last], [t3, last], [t4, last],
                    [v1, last], [v2, last], [ua, last], [k1, last], [k2, last]].

resource_crane = [l1, t1, t2, t3, t4, t5].
resource_bricklaying = [m1, m2, m3, m4, m5, m6].
resource_schal = [s1, s2, s3, s4, s5, s6].
resource_excavator = [a1, a2, a3, a4, a5, a6].
resource_ram = [p1, p2].
resource_pump = [b1, b2, b3, b4, b5, b6].
resource_caterpillar = [v1, v2].

resources = [resource_crane,
             resource_bricklaying,
             resource_schal,
             resource_excavator,
```

```
            resource_ram,
            resource_pump,
            resource_caterpillar].

max_nf_list = [[first, l1, 30], [a1, s1, 3], [a2, s2, 3],
               [a5, s5, 3], [a6, s6, 3], [p1, s3, 3], [p2, s4, 3]].
min_sf_list = [[ua, m1, 2], [ua, m2, 2], [ua, m3, 2], [ua, m4, 2],
               [ua, m5, 2], [ua, m6, 2]].
max_ef_list = [[s1, b1, 4], [s2, b2, 4], [s3, b3, 4], [s4, b4, 4],
               [s5, b5, 4], [s6, b6, 4]].
min_nf_list = [[first, l1, 30]].
min_af_list = [[ue, s1, 6], [ue, s2, 6], [ue, s3, 6], [ue, s4, 6],
               [ue, s5, 6], [ue, s6, 6]].

%% rules
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
max_nf(T1, T2, N) -->
        start(T2) =< end(T1) + N.

min_nf(T1, T2, N) -->
        start(T2) >= start(T1) + duration(T1) + N.

max_ef(T1, T2, N) -->
        end(T2) =< end(T1) + N.

min_af(T1, T2, N) -->
        start(T2) >= start(T1) + N.

min_sf(T1, T2, N) -->
        end(T2) =< start(T1) + N.

distances -->
        forall(T, max_nf_list, max_nf(nth(1, T), nth(2, T), nth(3, T))) and
        forall(T, min_sf_list, min_sf(nth(1, T), nth(2, T), nth(3, T))) and
        forall(T, max_ef_list, max_ef(nth(1, T), nth(2, T), nth(3, T))) and
        forall(T, min_nf_list, min_nf(nth(1, T), nth(2, T), nth(3, T))) and
        forall(T, min_af_list, min_af(nth(1, T), nth(2, T), nth(3, T))).

tasks_domain -->
        domain(tasks, 0, maxDuration).

prec(T1,  T2) -->
        end(T1) =< start(T2).

precedences -->
        forall(TaskPair,  precedences_list,
```

```
          prec(nth(1, TaskPair), nth(2, TaskPair))).

disj(T1, T2) -->
        prec(T1, T2) or prec(T2, T1).

disj_pairs(Tasks) -->
          forall(T1, Tasks,
            forall(T2, Tasks,
              uid(T1) < uid(T2) implies disj(T1, T2))).

disjunctives -->
        forall(Tasks, resources, disj_pairs(Tasks)).

minimize_completion_time -->
        conjunct_ordering([greatest(duration(A) + duration(B) if ^ is disj(A, B))]) and
        % disjunct_ordering([greatest(duration(A) if ^ is prec(A, B))]) and
        minimize(disjunctives, cost).

? tasks_domain and precedences and distances and disjunctives and minimize_completion_time.

% Cream
%% 271
%% 131
%% 129
%% 125
%% 121
%% 117
%% 109
%% 105
%% 103
%% 0.380 seconds
%% 167 bactracks

% Rules2CP
%% 271
%% 131
%% 129
%% 125
%% 121
%% 117
%% 109
%% 105
%% 103
%% 0.150 seconds
%% 367 bactracks
```

Chapter 11

The Packing Knowledge Modelling Language PKML

The Packing Knowledge Modelling Language (PKML) developed in the EU FP6 Net-WMS project is defined as a Rules2CP library. This makes PKML easily extensible with new features and customizable for particular needs.

The PKML package includes a library for dealing with Allen's interval relations in one dimension [1], another library for dealing with region connection calulus relations in an arbitrary number of dimensions [5], and a proper PKML library which defines higher-dimensional objects with alternative shapes, each shape being a rigid assembly of boxes.

Rules and strategies for solving pure bin packing problems as well as packing problems with specific requirements on the shape and weights of items to pack, are predefined in the PKML library.

11.1 Allen's Interval Relations in One Dimension

In one dimension, the library of Allen's interval relations between objects is predefined in Rules2CP in the following file `allen.rcp`:

```
% Copyright 2008 INRIA, F. Fages & J. Martin
% License GPL
%
% file allen.rcp
%

precedes(A, B, D) =
        end(A, D) < origin(B, D).

meets(A, B, D) =
        end(A, D) = origin(B, D).
```

```
overlaps(A, B, D) =
            origin(A, D) < origin(B, D) and
            end(A, D) < end(B, D) and
            origin(B, D) < end(A, D).

contains(A, B, D) =
        origin(A, D) < origin(B, D) and
        end(B, D) < end(A, D).

starts(A, B, D) =
        origin(A, D) = origin(B, D) and
        end(A, D) < end(B, D).

finishes(A, B, D) =
        origin(B, D) < origin(A, D) and
        end(A, D) = end(B, D).

equals(A, B, D) =
        origin(A, D) = origin(B, D) and
        end(A, D) = end(B, D).

started_by(A, B, D) =
        origin(A, D) = origin(B, D) and
        end(B, D) < end(A, D).

finished_by(A, B, D) =
        origin(B, D) > origin(A, D) and
        end(A, D) = end(B, D).

during(A, B, D) =
        origin(B, D) < origin(A, D) and
        end(A, D) < end(B, D).

overlapped_by(A, B, D) =
        origin(B, D) < origin(A, D) and
        origin(A, D) < end(B, D) and
        end(A, D) > end(B, D).

met_by(A, B, D) =
        end(B, D) = origin(A, D).

preceded_by(A, B, D) =
        end(B, D) < origin(A, D).

contains_touch(A, B, D) =
```

```
        origin(A, D) =< origin(B, D) and
        end(B, D) =< end(A, D).

overlaps_sym(A, B, D) =
        end(A, D) > origin(B, D) and
        end(B, D) > origin(A, D).
```

The predicate `contains_touch` and `overlaps_sym` have been added to Allen's relations. These relations can be defined by disjunctions of standard Allen's relations but their direct definition by conjunctions of inequalities is added here for efficiency reasons.

11.2 Region Connection Calculus in Higher-Dimensions

In higher-dimensions, the library of topological relations of the Region Connection Calculus [5] is predefined in Rules2CP between objects. For the sake of simplicity of the following file `rcc8.rcp`, the assemblies of boxes are treated as the least box containing the assembly, using the `size(S,D)` function.

```
% Copyright 2008 INRIA, F. Fages & J. Martin
% License GPL
%
% file rcc.rcp
%

import('rules2cp/lib/pkml/allen').

disjoint(O1, O2, Ds) =
        exists(D, Ds,
                precedes(O1, O2, D) or
                preceded_by(O1, O2, D)).

meet(O1, O2, Ds) =
        forall(D, Ds,
                not precedes(O1, O2, D) and
                not preceded_by(O1, O2, D)) and
        exists(D, Ds,
                meets(O1, O2, D) or
                met_by(O1, O2, D)).

equal(O1, O2, Ds) =
        forall(D, Ds, equals(O1, O2, D)).

covers(O1, O2, Ds) =
             forall(D, Ds,
                       started_by(O1, O2, D) or
                       contains(O1, O2, D) or
```

```
                    finished_by(O1, O2, D)) and
            exists(D, Ds, not contains(O1, O2, D)).

covered_by(O1, O2, Ds) =
        forall(D, Ds,
                starts(O1, O2, D) or
                during(O1, O2, D) or
                finishes(O1, O2, D)) and
        exists(D, Ds, not during(O1, O2, D)).

contains_rcc(O1, O2, Ds) =
        forall(D, Ds, contains(O1, O2, D)).

inside(O1, O2, Ds) =
        forall(D, Ds, during(O1, O2, D)).

overlap(O1, O2, Ds) =
        forall(D, Ds, overlaps_sym(O1, O2, D)).

contains_touch_rcc(O1, O2, Ds) =
        forall(D, Ds, contains_touch(O1, O2, D)).
```

The rule `contains_touch_rcc` has been added to the standard region calculus connection relations for convenience and efficiency reasons similar to the extension done to Allen's relations.

11.3 PKML Library

The PKML library is defined in Rules2CP by the following file `lib/pkml/pkml.rcp`:

```
% Copyright 2008 INRIA, F. Fages & J. Martin
% License GPL
%
% file pkml.rcp
%

import('rules2cp/lib/common/rcp').
import('rules2cp/lib/pkml/rcc').
import('rules2cp/lib/pkml/pkml_surface').
import('rules2cp/lib/pkml/pkml_weight').
%import('rules2cp/lib/pkml/pkml_gui').

%% Boxes
%
% boxes given with their size in each dimension
% b = {size = [s1,...,sk]}
```

```
make_box(L) = {size = L}.

box_volume(B) = product(size(B)).

%% Shifted Boxes
%
% shifted boxes given with their box and their offset
% b = {box = b, offset = [o1,..,ok]}

make_sbox(B, O) = {box = B, offset = O}.

sbox_size(SB, D) =  nth(D, size(box(SB))).

sbox_offset(SB, D) =  nth(D, offset(SB)).

sbox_end(SB, D) =  sbox_offset(SB, D) + sbox_size(SB, D).

%% Shapes
%
% shapes as assemblies of boxes given with their positions
% shape = {sboxes=[sb1,...,sbm]}
%
% - make_shape_box constructor for a single box
% - size of a shape in one dimension as the maximum size of its assembly
% - shape_volume is the volume of an assembly shape
%   (overapproximation if sboxes overlap)

make_shape(SBs) = {sboxes = SBs}.

make_shape_box(L) = make_shape([make_sbox(make_box(L), map(_, L, 0))]).

shape_volume(S) = sum(map(SB, sboxes(S), box_volume(box(SB)))).

shape_origin(S, D) = minimum(map(SB, sboxes(S), sbox_offset(SB, D))).

shape_end(S, D) = maximum(map(SB, sboxes(S), sbox_end(SB, D))).

shape_size(S, D) = shape_end(S, D) - shape_origin(S, D).

%% Objects
%
% objects with alternative shapes
% object = {shapes=[s1,...,sN], shape_index=_, origin=[X1,...,Xk]}
%
% - object constructors with a single shape
```

```
% - shape domain of an object
% - origin of an object
% - x,y,z coordinates of an object
% - end of an object with alternative shapes
% - volume of an object as the volume of its shape

make_object(SL, OL) = {shapes=SL, shape_index=S, origin=OL}.

make_object(SL, OL, S, W) = {shapes=SL, shape_index=S, origin=OL, weight=W}.

make_object_shape(S, L) = {shapes=[S], shape_index=1, origin=L}.

make_object_shape(S, L, W) = {shapes=[S], shape_index=1, origin=L, weight=W}.

object_shape_domain(O) = shape_index(O) in [1 .. length(shapes(O))].

object_shape_domains(Items) = forall(I, Items, object_shape_domain(I)).

origin(O, D) = nth(D, origin(O)).

x(O) = origin(O, 1).

y(O) = origin(O, 2).

z(O) = origin(O, 3).

end(O, D) = origin(O, D) +
            sum(map(S, shapes(O),
                (shape_index(O) = pos(S, shapes(O))) * shape_end(S, D))).

size(O, D) = sum(map(S, shapes(O),
                (shape_index(O) = pos(S, shapes(O))) * shape_size(S, D))).

volume(O) = sum(map(S, shapes(O),
                (shape_index(O) = pos(S, shapes(O))) * shape_volume(S))).

distance(O1, O2, D) = max(0, max(origin(O1, D), origin(O2, D))
                           - min(end(O1, D), end(O2, D))).

%%
% Rules for pure bin packing problems
non_overlapping_binary(Items, Dims) =
        forall(O1, Items,
                forall(O2, Items,
                        uid(O1) < uid(O2) implies
                        not overlap(O1, O2, Dims))).
```

```
containmentAE(Items, Bins, Dims) =
        forall(I, Items,
          exists(B, Bins,
            contains_touch_rcc(B, I, Dims))).

bin_packing_binary(Items, Bins, Dims) =
        containmentAE(Items, Bins, Dims) and
        non_overlapping_binary(Items, Dims) and
        labeling(Items).

bin_packing(Items, Bins, Dims) =
        containmentAE(Items, Bins, Dims) and
        non_overlapping(Items, Dims) and
        labeling(Items).

%%
% Rules for pure bin design problems
containmentEA(Items, Bins, Dims) =
        exists(B, Bins, forall(I, Items, contains_touch_rcc(B,I,Dims))).

bin_design(Bin, Items, Dims) =
        containmentEA(Items, [Bin], Dims) and
        non_overlapping(Items, Dims) and
        minimize(labeling(Items), volume(Bin)).
```

These rules allow us to express pure bin packing and pure bin design problems.

The file `pkml_weight.rcp` defines some additional common sense rules of packing taking into account the weight of items:

```
% Copyright 2008 INRIA, F. Fages & J. Martin
% License GPL
%
% file pkml_weight.rcp
%

lighter(O1, O2) =
        weight(O1) =< weight(O2).

heavier(O1, O2) =
        weight(O1) >= weight(O2).

gravity(Items) =
        forall(O1, Items,
          origin(O1, 3) = 0 or
          exists(O2, Items, uid(O1) # uid(O2) and on_top(O1, O2))).
```

```
weight_stacking(Items) =
        forall(O1,  Items,
                forall(O2, Items,
                        (uid(O1) # uid(O2) and above(O1, O2))
                        implies
                        lighter(O1, O2))).

weight_balancing(Items, Bin, D, Ratio) =
 let(L, sum( map(Il, Items, Il:weight * (end(Il, D) =< (end(Bin, D) / 2)))),
  let(R, sum( map(Ir, Items, Ir:weight * (origin(Ir, D) >= (end(Bin, D) / 2)))),
         100 * max(L, R) =< (100 + Ratio) * min(L, R))).
```

The file `pkml_surface.rcp` defines some additional rules for taking into account the surface of contact between stacked items:

```
% Copyright 2008 INRIA, F. Fages & J. Martin
% License GPL
%
% file pkml_surface.rcp
%

above(O1, O2) =
        overlap(O1, O2, [1, 2]) and
        preceded_by(O1, O2, 3) or met_by(O1, O2, 3).

on_top(O1, O2) =
        overlap(O1, O2, [1, 2]) and
        met_by(O1, O2, 3).

oversize(O1, O2, D) =
        max(  max( origin(O1, D), origin(O2, D))
            - min( origin(O1, D), origin(O2, D)),
              max( end(O1, D), end(O2, D))
            - min( end(O1, D), end(O2, D))).

stack_oversize(Items, Length) =
        forall(O1, Items,
                forall(O2, Items,
                        (overlap(O1, O2, [1,2]) and O1:uid # O2:uid)
                        implies
                        forall(D, [1,2], oversize(O1, O2, D) =< Length))).
```

Chapter 12

PKML Examples

12.1 Bin Packing

A small PKML example involving packing business rules taking into account the weight of objects and coming from the automotive industry at Peugeot Citroën PSA, is defined in the following file `psa.rcp`:

```
% Copyright 2008 INRIA, J. Martin & F. Fages
% License GPL
%
% file psa.rcp
%

import('rules2cp/lib/pkml/pkml').

s1 = make_shape_box([1203, 235, 239]).
s2 = make_shape_box([224, 224, 222]).
s3 = make_shape_box([224, 224, 148]).
s4 = make_shape_box([224, 224, 111]).
s5 = make_shape_box([224, 224, 74]).
s6 = make_shape_box([155, 224, 222]).
s7 = make_shape_box([112, 224, 148]).

o1 = make_object_shape(s1, [0, 0, 0]).
o2 = make_object_shape(s4, [_, _, _], 413).
o3 = make_object_shape(s5, [_, _, _], 463).
o4 = make_object_shape(s5, [_, _, _], 842).
o5 = make_object_shape(s3, [_, _, _], 422).
o6 = make_object_shape(s4, [_, _, _], 266).
o7 = make_object_shape(s4, [_, _, _], 321).
o8 = make_object_shape(s2, [_, _, _], 670).
o9 = make_object_shape(s6, [_, _, _], 440).
```

```
o10 = make_object_shape(s7, [_, _, _], 325).

s11 = make_shape_box([_, _, _]).
s41 = make_shape_box([224, 111, 224]).
s51 = make_shape_box([224, 74, 224]).

o11 = make_object_shape(s11, [0,0,0]).
o41 = {shapes=[s4, s41], shape_index=_, origin=[_, _, _], weight=413}.
o51 = {shapes=[s5, s51], shape_index=_, origin=[_, _, _], weight=463}.

bin = o1.
items = [o2, o3, o4, o5, o6, o7, o8, o9, o10].
dimensions = [1, 2, 3].

w(O) = size(O, 1).
h(O) = size(O, 2).
l(O) = size(O, 3).

geost_greedyflag = _.

items_domain(Items, Bin) -->
        forall(I, Items,
        domain(x(I), 0, w(Bin) - w(I)) and
        domain(y(I), 0, h(Bin) - h(I)) and
        domain(z(I), 0, l(Bin) - l(I))).

psa_bin_packing(Bin, Items, Dims) -->
        items_domain(Items, Bin) and
        gravity(Items) and
        weight_stacking(Items) and
        weight_balancing(Items, Bin, 1, 20) and
        stack_oversize(Items, 10) and

        non_overlapping(Items, Dims, geost_greedyflag,
          [object(min(1), [min(4),min(3),min(2)]),
           object(min(1), [max(4),min(3),min(2)]),
           object(min(1), [max(4),max(3),min(2)]),
           object(min(1), [min(4),max(3),min(2)])]) and

        variable_ordering([greatest(weight(^)),
                           greatest(volume(^)),
                           is(z(^))]) and

        value_ordering([up(z(^)),
                        bisect(x(^)),
                        bisect(y(^))]) and
```

```
        bin_packing(Items, [Bin], Dims).

? psa_bin_packing(bin, items, dimensions).
```

12.2 Optimal Rectangle Packing

This example is a transcription in PKML of the constraint program used by Simonis and O'Sullivan to solve optimal rectangle packing problems [6]. The search strategy is modeled using the `dynamic` predicate.

In this model, the `disjoint2` and `cumulative` predicates are undocumented constraints used for the sake of comparison with the original program in SICStus Prolog. These constraints are subsumed in Rules2CP by the `geost` predicate.

```
% Rectangle Packing model (based on [Simonis & O'Sullivan, CPAIOR'08])
%
% Find the smallest rectangle (bin) containing
% N squares (items) of sizes 1*1, 2*2, ..., N*N.
%

import('rules2cp/lib/pkml/pkml').

% Data structures & handy macro defs
%
make_object_shape_area(S, OL, A) = {shapes=[S], shape_index=1, origin=OL, area=A}.

bin = make_object_shape_area(make_shape_box([_, _]), [1, 1], _).

items(N) = map(S, squares(N), item(S)).

item(S) = {shapes = [S], shape_index = 1, origin = [_,_]}.

items_area(Items) = sum(map(I, Items, volume(I))).

square(S) = make_shape_box([S, S]).

squares(N) = map(Size, reverse_list([2 .. N]), square(Size)).

w(O) = size(O, 1).
h(O) = size(O, 2).
l(O) = size(O, 3).

xs(Items) = map(I, Items, x(I)).
ys(Items) = map(I, Items, y(I)).
```

```
reverse_list(L) = foldl(X, L, flip(cons), [], X).


% Search strategy definition
%

% interval splitting
%
interval_predicate(E, List) =
        let(X, nth(1, List),
        let(L, nth(2, List),
        let(XDLBL, domain_min(X) + L,
          X =< XDLBL or (X > XDLBL and E)))).

interval_split(X, Min, Max, L) =
        foldl(Cut, [1..((Max - Min) / (L + 1)) + 1], interval_predicate, true, [X, L]).

% dychotomic splitting
%
dichotomy_predicate(List, X) =
        let(XDM, (domain_min(X) + domain_max(X)) / 2,
          X =< XDM or X > XDM).

dichotomic_split(X) =
        let(L, domain_max(X) - domain_min(X) + 1,
          foldl(Cut, [1..log(2, L)], dichotomy_predicate, true, X)).

state_items_domain(Items, W, H) =
        forall(It, Items,
          let(X, x(It),
          let(Y, y(It),
          let(S, w(It),
            domain(X, 1, W - S + 1) and
            domain(Y, 1, H - S + 1) )))) and
            lower_quadrant(Items, W, H).

lower_quadrant(Items, W, H) =
        let(FstIt, nth(1, Items),
        let(X, x(FstIt),
        let(Y, y(FstIt),
        let(S, w(FstIt),
          domain(X, 1, (W - S + 2) / 2) and
          domain(Y, 1, (H + 1) / 2) )))).

state_items_constraints(Items, W, H) =
        let(Xs, map(It, Items, x(It)),
```

```
        let(Ys, map(It, Items, y(It)),
        let(Ss, map(It, Items, w(It)),
          disjoint2(Xs, Ys, Ss) and
          cumulative(Xs, Ss, Ss, H) and
          cumulative(Ys, Ss, Ss, W) ))).

state_items_search(Items, W, H) =
        let(XSs, map(It, Items, {coord = x(It), siz = w(It)}),
        let(YSs, map(It, Items, {coord = y(It), siz = h(It)}),
        let(Min, 1,
        let(MaxX, W + 1,
        let(MaxY, H + 1,
          dynamic(
           search(
            forall(XS, XSs, siz(XS) > 6 implies
              interval_split(coord(XS), Min, MaxX, max(1, (siz(XS)*3)/10))) and
            forall(XS, XSs,
              dichotomic_split(coord(XS))) and
            forall(YS, YSs,
              interval_split(coord(YS), Min, MaxY, max(1, (siz(YS)*3)/10))) and
            forall(YS, YSs,
              dichotomic_split(coord(YS)))
            ))))))).

% Items subproblem definition
%
solve_items_subproblem(Items, W, H) =
        state_items_domain(Items, W, H) and
        state_items_constraints(Items, W, H) and
        state_items_search(Items, W, H).

state_bin_domain(W, H, A, L, U, N) =
        domain([W, H], N, L) and
        domain([A], L, U).

state_bin_constraints(W, H, A, N) =
        let(K, (W + 1)/2,
          A = W * H and
          W =< H and
          (W >= 2 * N - 1 or H >= (N * N + N - (K - 1) * (K - 1) - (K - 1)) / 2)).

state_bin_search =
        variable_ordering([is(area(^)), is(w(^)), is(h(^))]) and
        labeling(bin).

% Bin subproblem definition
```

```
%
solve_bin_subproblem(W, H, A, L, U, N) =
        state_bin_domain(W, H, A, L, U, N) and
        state_bin_constraints(W, H, A, N) and
        state_bin_search.

% Query
%
% N : problem instance size (N = card(Items))
% Items : item records {box(K, K) | forall K in 1..N}
% Width : bin width (length in the first dimension (x))
% Height : bin height (length in the second dimension (y))
% Area : bin area
% LB : optimal solution lower bound
% UB : optimal solution upper bound
 ? let(N, 22,
   let(Items, items(N),
   let(Width, size(bin, 1),
   let(Height, size(bin, 2),
   let(Area, area(bin),
   let(LB, items_area(Items) + 1,
   let(UB, LB + 200,

   solve_bin_subproblem(Width, Height, Area, LB, UB, N) and
   solve_items_subproblem(Items, Width, Height) )))))))).
```

12.3 Bin Packing with Polymorphic Shapes

This small example illustrates the use of polymorphic shapes to represent the possible rotations of an object. In this example the object `o2` (as well as `o3`) can be rotated by 90 degrees in the x-y plane. This is represented by using alternative shapes `s2` and `s3` for this object.

```
% Copyright 2008 INRIA, J. Martin & F. Fages
% License GPL
%
% file psa.rcp
%

import('rules2cp/lib/pkml/pkml').

s1 = make_shape_box([5, 4, 4]).
s2 = make_shape_box([4, 5, 2]).
s3 = make_shape_box([5, 4, 2]).
```

```
o1 = make_object_shape(s1, [0, 0, 0]).
o2 = make_object([s2, s3], [_, _, _], _, 10).
o3 = make_object([s2, s3], [_, _, _], _, 11).

dimensions = [1, 2, 3].
bin = o1.
items = [o2, o3].

w(O) = size(O, 1).
h(O) = size(O, 2).
l(O) = size(O, 3).

items_domain(Items, Bin) -->
        forall(I, Items,
        domain(x(I), 0, w(Bin)) and
        domain(y(I), 0, h(Bin)) and
        domain(z(I), 0, l(Bin))).

? items_domain(items, bin) and
  object_shape_domains(items) and
  variable_ordering([greatest(weight(^)), is(shape_index(^)), is(z(^))]) and
  bin_packing(items, [bin], dimensions).
```

Bibliography

[1] J. Allen. Time and time again: The many ways to represent time. *International Journal of Intelligent System*, 6(4), 1991.

[2] 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`.

[3] M. Carlsson et al. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, release 4 edition, 2007. ISBN 91-630-3648-7.

[4] François Fages and Julien Martin. From rules to constraint programs with the Rules2CP modelling language. In *Recent Advances in Constraints, Revised Selected Papers of the 13th Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP'08*, volume 5655 of *Lecture Notes in Artificial Intelligence*, pages 66–83. Springer-Verlag, 2009.

[5] 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 Knowledge Representation and reasoning KR'92*, pages 165–176. Morgan Kaufmann, 1992.

[6] Helmut Simonis and Barry O'Sullivan. Using global constraints for rectangle packing. In *Proceedings of the first Workshop on Bin Packing and Placement Constraints BPPC'08, associated to CPAIOR'08*, May 2008.

[7] Choco Team. CHOCO web page. `http//www.emn.fr/x-info/choco-solver/doku.php`.

[8] Pascal Van Hentenryck. *Constraint satisfaction in Logic Programming*. MIT Press, 1989.

Index