From Rules to Constraint Programs with the Rules2CP Modelling Language

Francois Fages, Julien Martin

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

From Rules to Constraint Programs with the Rules2CP Modelling Language

François Fages — Julien Martin

N° 6495

April 2008

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--6495--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
PARIS - ROCQUENCOURT

From Rules to Constraint Programs with the Rules2CP Modelling Language

François Fages , Julien Martin

Thème SYM — Systèmes symboliques
Équipes-Projets Contraintes

Rapport de recherche n° 6495 — April 2008 — 30 pages

Abstract: In this paper, we show that the business rules knowledge representation paradigm, which is widely used in the industry, can be developped as a front-end modelling language for constraint programming. We present a general purpose rule-based modelling language, called Rules2CP, and describe its compilation to constraint programs over finite domains with reified and global constraints, using term rewriting and partial evaluation. We prove the confluence of these transformations and provide a complexity bound on the size of the generated programs. The expressiveness of Rules2CP is illustrated with a complete library for packing problems, called PKML, which, in addition to pure bin packing and bin design problems, can deal with common sense rules about weights, stability, as well as specific packing business rules which compile efficiently into constraints.

Key-words: Modelling Languages, Constraint Programming, Rules-based Programming, Combinatorial Optimization

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Des règles aux contraintes avec le langage de modélisation Rules2CP

Résumé : Dans cet article, nous montrons que le paradigme de représentation des connaissances par des règles métiers, qui est largement utilisé dans l'industrie, peut être développé en un langage de modélisation pour la programmation par contraintes. Nous présentons un langage de modélisation par règles à usage général, appelé Rules2CP, et décrivons sa compilation en des programmes avec contraintes sur les domaines finis avec réification et contraintes globales, procédant par réécriture de termes et évaluation partielle. Nous prouvons la confluence de ces transformations et donnons une borne de complexité sur la taille des programmes générés. L'expressivité de Rules2CP est illustrée par une bibliothèque complète pour les problèmes d'empaquetage, appelée PKML, qui, en plus des problèmes purs de remplissage et de conception de containers, peut traiter des règles de sens commun sur les poids, la stabilité, de même que des règles métiers d'empaquetage qui se compilent efficacement en contraintes.

Mots-clés : Langages de modélisation, programmation par contraintes, programmation par règles, optimisation combinatoire

1 Introduction

From a programming language standpoint, one striking feature of constraint programming is its declarativity for stating combinatorial problems, describing only the "what" and not the "how", and yet its efficiency for solving large size problem instances in many practical cases. From a non-expert user standpoint however, constraint programming is not as declarative as one would wish, and constraint programming systems are in fact very difficult to use by non-expert users outside the range of already treated examples. This well recognized difficulty has been presented as a main challenge for the constraint programming community, and has motivated the search for more declarative front-end problem modelling languages, such as most notably OPL [1] and Zinc [2, 3]. In these languages, a problem is modelled with variables, arrays, primitive constraints, set constructs, iterators and quantifiers. Such problem models can then be mapped to constraint programs, mixed integer linear programs [4], combinations of both [1], or local search programs [5] for solving them.

In the industry however, the business rules approach to knowledge representation has a wide audience because of the property of independence of the rules which can be introduced, checked, and modified independently of the others, and independently of any particular procedural interpretation by a rule engine [6]. This provides an attractive knowledge representation scheme for fastly evolving regulations and constraints, and for maintaining systems with up to date information.

In this article, we show that the business rules knowledge representation paradigm can be developped as a front-end modelling language for constraint programming. We present a general purpose rule-based modelling language for constraint programming, called Rules2CP. Rules2CP rules are not general condition-action rules, also called production rules in the expert system community, but restricted *logical rules*, with one head and no imperative actions, and where bounded quantifiers are used to represent complex conditions. They comply to the business rules manifesto [6], and in particular to the independence from a procedural interpretation by a rule engine. This is concretely demonstrated in Rules2CP by their compilation to constraint programs using a completely different representation. As a consequence, the rule language proposed in this paper comes with a simple semantics in classical first-order logic, instead of the default logics usually considered in the rule-based knowledge representation community [7, 8].

Furthermore, our aim at designing a knowledge modelling language for non-programmers led us to abandon recursion and data strutures such as arrays and lists, and retain only records (feature terms) and finite collections (enumerated lists) with quantifiers and aggregates as iterators. In the next section, we present the syntax of Rules2CP with its main predefined functions and predicates. We show how search strategies and heuristics can be specified in a declarative manner, and illustrate the main language constructs with simple examples of combinatorial and scheduling problems.

Then in Sec. 2, we formally describe the compilation of Rules2CP models into constraint programs over finite domains with reified constraints, using a term rewriting system and partial evaluation. We prove the confluence of these transformations which shows that the generated constraint program does not depend on the order of application of the rewritings. Furthermore, we provide a

complexity bound on the size of the generated constraint program which reflects the simplicity of the Rules2CP design choices.

Then in Sec. 4, we illustrate the expressive power and efficiency of this approach with a particular Rules2CP library, called the Packing Knowledge Modelling Library (PKML), developed in the EU project Net-WMS[1] for dealing with real-size non-pure bin packing problems coming from the automotive industry. In addition to pure bin packing and bin design problems, we show the capability of PKML to express heuristic knowledge and common sense rules about weights, equilibrium and stability constraints [9], as well as business rules taking into consideration industrial requirements and expertise. Furthermore, we show how a large subset of PKML rules can be directly compiled into the geometric global constraint `geost` [10] and its integrated rule language [11].

Finally in Sec. 5 we discuss the main features of Rules2CP by comparing them to other formalisms: business rules, OPL and Zinc modelling languages, constraint logic programming and term rewriting systems.

We conclude on the generality of this rule-based knowledge modelling language as a front-end for constraint programming.

2 The Rules2CP Language

2.1 Syntax

Rules2CP is a term rewriting rule language based on first-order logic with bounded quantification and aggregate operators. Its only data structures are integers, strings, enumerated lists and records. Because of the importance of naming objects in Rules2CP, the language includes a simple module system that prefixes names with module and package names, similarly to [12].

The syntax of Rules2CP is given in Table 2.1. An *ident* is a word beginning with a lower case letter or any word between 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 _. The set, denoted by $V(E)$, of *free variables* in an expression E is the set of variables occurring in E and not bound by a `forall, exists, let, map` or `aggregate` operator. The *size* of an expression or formula is the number of nodes in its tree representation.

In a Rules2CP file, the order of the statements is not relevant. Recursive definitions and multiple definitions of a same head symbol are forbidden. In a rule, L`-->`R, we assume $V(R) \subseteq V(L)$, whereas in a declaration, H`=`E, the introduced variables, in $V(E) \setminus V(H)$, represent the unknown variables of the problem.

An expression *expr* can be a *fol* formula considered as a 0/1 integer. This usual coercion between booleans and integers, called *reification*, provides a great expressiveness [13]. The grammar does distinguish however the logical formulas from other expressions. For instance, a goal cannot be any expression but a logical formula.

The aggregate operator cannot be defined in first-order logic and is a Rule2CP builtin. This operator iterates the application of a binary operator (given in the third argument), to copies of the expression given in the last argument, where

[1] `http://net-wms.ercim.org`

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
	|	`aggregate`*(variable,expr,logop,fol,fol)*	logical aggregate
expr	*::=*	*varint*	
	|	*fol*	reification
	|	*string*	
	|	`[` *enum* `]`	list
	|	`{`*ident* `=` *expr,...,ident*`=` *expr*`}`	record
	|	*name*	
	|	*name(expr,...,expr)*	function
	|	*expr op expr*	
	|	`aggregate`*(variable,expr,op,expr,expr)*	
	|	`map`*(variable,expr,expr)*	list mapping
enum	::=	*enum* `,` *enum*	enumeration
	|	*expr*	value
	|	*expr* `..` *expr*	interval of integers
varint	*::=*	*variable*	
	|	*integer*	
varbool	*::=*	*variable*	
	|	`0`	false
	|	`1`	true
op	*::=*	$+ \mid - \mid * \mid / \mid$ `min` | `max`	
relop	*::=*	$< \mid =< \mid = \mid$ `#` $\mid >= \mid >$	
logop	*::=*	`and` | `or` | `implies` | `equiv` | `xor`	
name	*::=*	*ident*	
	|	*name*`:`*ident*	module prefix

Table 1: Syntax of Rules2CP.

the variable in the first argument is replaced by the successive elements of the list given in the second argument. For instance, the product of the elements in a list is defined by `product(L)=aggregate(X,L,*,1,X)`, the maximum by `maximum(L)=aggregate(X,L,max,0,X)`, etc.

Lists of expressions can be formed by enumerating their elements, or intervals of values in the case of integers. For instance `[1,3..6,8]` represents the list `[1,3,4,5,6,8]`. Such lists are used to represent the domains of variables in *(var* `in` *list)* formula, and in the answers returned to Rules2CP goals.

The following expressions are predefined for accessing the components of lists and records:

- `length(`*list*`)` returns the length of the list (after expansion of the intervals), or 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.
- *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.

Furthermore, records have a default integer attribute `uid` which provides a unique identifier for each record. The predefined function

- `variables(`*expr*`)`

returns the list of variables contained in an expression. The predefined predicates

- *X* `in` *list*
- `domain(`*expr,min,max*`)`

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

2.2 Predicates for Search

Describing the search strategy in a modelling language is a challenging task as search is usually considered as inherently procedural, and thus contradictory to declarative modelling. This is however not our point of view in Rules2CP. Our approach to this question is to specify the *decision variables* and the *branching formulas* of the problem in a declarative manner, as well as the heuristics as *preference orderings* on variables and values.

Decision variables can be declared with the predefined predicate

- `labeling(`*expr*`)`

for enumerating the possible values of all the variables *contained* in an expression, that is occurring as attributes of a record, or recursively in a record referenced by attributes, in a list, or in a first-order formula. This labeling predicate thus provides an easy way to refer to the variables contained in an object or in a formula, without having to collect them explicitly in a list as is usually done in constraint programs. Moreover, *Branching formulas* can be declared in Rules2CP with the predicate

- `search(`*fol*`)`

This more original predicate specifies a search by branching on all the disjunctions and existential quantifications occurring in a first-order formula. Note that a similar approach to specifying search has been proposed for SAT in [14]. Here however, the only normalization is the elimination of negations in the formula by descending them to the constraints. The structure of the formula is kept as an *and-or* search tree where the disjunctions constitute the choice points[2].

In Rules2CP, the following *optimization predicates* specify optimization criteria independently of search:

- `minimize(`*expr*`)` for minimizing an expression
- `maximize(`*expr*`)` for maximizing an expression

with no restriction on their number of occurrences in a formula. This makes it possible to express multicriteria optimization problems and the search for Pareto optimal solutions according to the lexicographic ordering of the criteria as read from left to right.

2.3 Predicates for Heuristics

Adding the capability to express heuristic knowledge in Rules2CP is mandatory for efficiency. This is done with two predicates for specifying both static and dynamic variable choice as well as value choice heuristics. Dynamic criteria are standard in constraint programming systems, see for instance [15, 16]. The definition of the static criteria use the expressive power of Rules2CP.

The `variable_choice_heuristics` predicate takes a list of criteria for ordering the variables in search. 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 an unspecified order. Each criterion has one of the following forms:

- `greatest(`*expr*`)`, `greatest(`*expr, option*`)`,
- `smallest(`*expr*`)`, `smallest(`*expr, option*`)`,
- `any(`*expr*`)`, `any(`*expr, option*`)`,
- `is(`*expr*`)`, `is(`*expr, option*`)`,

[2]In order to avoid an exponential growth of the formulas, `equiv` and `xor` formulas are kept as constraints and are not treated as choice points.

where *expr* is an expression containing the symbol `^` which denotes, for a given variable, the left-hand side of the Rules2CP declaration that introduced a given variable. If the expression cannot be evaluated on a variable, the criterion is ignored. A *greatest* (resp. *smallest*) form selects a variable with greatest (resp. smallest) value for the expression. An `any` form selects a variable for which the expression applies independently of its value. An `is` form selects a variable if it is equal to the result of the expression. *option* is a *dynamic choice criterion* using the following keywords: `leftmost`, smallest lower bound `min`, greatest upper bound `max`, smallest domain `ff`, or most constrained `ffc`. The dynamic choice criteria are used in Rules2CP for ordering at run-time the variables which have the same static criteria. For instance, in a bin packing problem, the predicate

```
variable_choice_heuristics([greatest(volume(^)), smallest(uid(^), ff)])
```

specifies a lexicographic static ordering of the variables by decreasing volume of the object in which they have been declared, by increasing *uid* attribute of the object, and for those variables appearing in the same object (i.e. with the same uid), a dynamic ordering by increasing domain size (`ff`). The static criteria are used at compile-time to order the variables for labelings, while the dynamic criteria are used at run-time by the solver.

The `variable_choice_heuristics` predicate takes a list of criteria of the following forms:

- `up`, `up(`*`expr`*`)`,
- `down`, `down(`*`expr`*`)`,
- `step`, `step(`*`expr`*`)`,
- `enum`, `enum(`*`expr`*`)`,
- `bisect`, `bisect(`*`expr`*`)`,

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 (a criterion without expression always applies). The different criteria enumerate the values in, respectively, ascending order (the default), descending order, with binary choices (the default), with a multiple choice, or by dichotomy. For instance, in a bin packing problem, the predicate

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

specifies the enumeration in ascending order for the z coordinates, and by dichotomy for the x and y coordinates (and with the default strategy for the other variables).

The capabilities of dissociating the specifications of the variable and value heuristics, and of using static criteria about the objects in which the variables appear, are very powerful. It is worth noticing that this expressive power for the heuristics creates difficulties however for their compilation into constraint systems that mix both kinds of strategies in a single option list, and for which one cannot expresses different value choice heuristics for different variables [16]. For convenience in Rules2CP, the dynamic heuristic criteria can also be added to the labeling predicate as an optional argument.

2.4 Simple Examples

Example 1 *The N-queens problem can be modelled in Rules2CP with declarations for creating a list of records representing the position of each queen on the chess board, and with one rule for stating when a list of queens do not attack each other, another rule for stating the constraints of a problem of size N, and a goal for stating the size of the problem to solve:*

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

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

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

solve(N) -->
   let(B, board(N),
      domain(B,1,N) and
      safe(B) and
      labeling(B)).

? solve(4).
```

In such a simple example, there is no point in separating data from rules in different files but this is recommended in larger examples using `import` *statements.*

Example 2 *A disjunctive scheduling problem can be modelled as follows:*

```
t1 = {start=_, dur=1}. t2 = {start=_, dur=2}.
t3 = {start=_, dur=3}. t4 = {start=_, dur=4}.
t5 = {start=_, dur=2}. t6 = {start=_, dur=0}.
cost = start(t6).

precedences -->
prec(t1,t2) and prec(t2,t3) and
prec(t3,t6) and prec(t1,t4) and
prec(t4,t5) and prec(t5,t6).

disjunctives -->
disj(t2,t5) and disj(t4,t3).

prec(T1,T2) --> start(T1)+dur(T1) =< start(T2).
disj(T1,T2) --> prec(T1,T2) or prec(T2,T1).

? start(t1)>=0 and cost<20 and precedences and
  search(disjunctives) and minimize(cost).
```

The goal posts the precedence constraints, and develops a search tree for the disjunctive constraints without labeling variables. Note that the instantiation of the cost is usually required in CP minimization predicates and the labeling of the

cost expression is thus automatically added by the Rules2CP compiler according to the target language, as shown in the next section on compilation. The answer computed by the solver is translated back to Rules2CP with domain expressions for the variables. The goal

```
? start(t1)>=0 and cost<20 and precedences and
  disjunctives and search(disjunctives) and minimize(cost).
```

adds the disjunctive constraints for pruning, and develops a similar search tree. The goal

```
? start(t1)>=0 and cost<20 and precedences and
  disjunctives and search(disjunctives) and
  labeling(precedences) and minimize(cost).
```

adds the labeling of variables for getting ground solutions.

3 Compilation to Constraint Programs over Finite Domains with Reified Constraints

Rules2CP models compile to constraint satisfaction problems over finite domains with reified constraints by interpreting Rules2CP statements using a term rewriting system, i.e. with a rewriting process that rewrites subterms inside terms according to general term rewriting rules. The Rules2CP declarations and rules provide the term rewriting rules, while the Rules2CP goals provide the terms to rewrite. It is worth noticing that for user-interaction and debugging purpose at runtime, book-keeping information needs to be implemented in this transformation in order to maintain the dependency from CP variables back to Rules2CP statements [17]. Let us denote by $\rightarrow_{csp}$ the term rewriting relation of the compilation process.

3.1 Generic Rewrite Rules

The following term rewriting rules are associated to Rules2CP declarations and rules:

- $L \rightarrow_{csp} R$ for every rules of the form L `-->` R,
- $L \rightarrow_{csp} R$ for every declarations of the form L `=` R with $V(R) \subseteq V(L)$;
- $L\sigma \rightarrow_{csp} R\sigma\theta$ for every declarations of the form L `=` R with $V(R) \not\subseteq V(L)$ and every ground substitution σ of the variables in $V(L)$, where θ is a renaming substitution that gives unique names indexed by $L\sigma$ to the variables in $V(R) \setminus V(L)$.

In a Rules2CP rule, all the free variables of the right-hand side have to appear in the left-hand side. In a Rules2CP declaration, there can be free variables introduced in the right hand side and their scope is global. Hence these variables are given unique names (with substitution θ) which will be the same at each invocation of the object. These names are indexed by the left-hand side of the declaration statement which has to be ground in that case (substitution

σ). For example, the row variables in the records declared by `q(N)` in Example 1 are given a unique name indexed by the instances of the head[3] $q(i)$. These conventions provide a basic book-keeping mechanism for retrieving the Rules2CP variables introduced in declarations from their variable names. It is worth noting that in rules as in declarations, the variables in L may have several occurrences in R, and thus that subexpressions in the expression to rewrite can be duplicated by the rewriting process.

The ground arithmetic expressions are rewritten with the following evaluation rule:

- *expr* $\rightarrow_{csp}$ v if *expr* is a ground expression and v is its value,

This rule provides a *partial evaluation* mechanism for simplifying the arithmetic expressions as well as the boolean conditions. This is crucial to limiting the size of the generated program and eliminating at compile time the potential overhead due to the data structures used in Rules2CP.

The accessors to data structures are rewritten in the obvious way with the following rule schemas that impose that the lists in arguments are expansed first[4]:

- `[`i `..` j`]` $\rightarrow_{csp}$ `[`i`,` $i+1$`,...,`j`]` if i and j are integers and $i \leq j$
- `length([`e_1`,...,`e_N`])` $\rightarrow_{csp}$ N
- `nth(i,[`e_1`,...,`e_N`])` $\rightarrow_{csp}$ e_i
- `pos(`e`,[`e_1`,...,`e_N`])` $\rightarrow_{csp}$ i where e_i is the *first* occurrence of e in the list after rewriting,
- *attribute(R)* $\rightarrow_{csp}$ V if R is a record with value V for *attribute*.

The quantifiers, aggregate, map and let operators are binding operators which use a dummy variable X to denote place holders in an expression. They are rewritten under the condition that their first argument X is a *variable* and their second argument is an expansed list, by duplicating and substituting expressions as follows:

- `aggregate(`X`,[`e_1`,`$\cdots$`,`e_N`],`op`,`e`,`ϕ`)` $\rightarrow_{csp}$ $\phi[X/e_1]$ *op...op* $\phi[X/e_N]$ (e if $N = 0$)
- `forall(`X`,[`e_1`,`$\cdots$`,`e_N`],`ϕ`)` $\rightarrow_{csp}$ $\phi[X/e_1]$ `and` ... `and` $\phi[X/e_N]$ (`1` if $N = 0$)
- `exists(`X`,[`e_1`,`$\cdots$`,`e_N`],`ϕ`)` $\rightarrow_{csp}$ $\phi[X/e_1]$ `or` ... `or` $\phi[X/e_N]$ (`0` if $N = 0$)
- `map(`X`,[`e_1`,`$\cdots$`,`e_N`],`ϕ`)` $\rightarrow_{csp}$ `[`$\phi[X/e_1], ..., \phi[X/e_N]$`]`
- `let(`X`,`e`,`ϕ`)` $\rightarrow_{csp}$ $\phi[X/e]$

[3] In the N-Queens example, the unique names given to the row variables are of the form `Q_i_` as they appear in records declared by $q(i)$ and they are the first anonymous variables in the records (if the record contained other anonymous variables they would be named by `Q_i_2`, `Q_i_3`, etc.)

[4] The expansion rule for intervals in lists is given here for the sake of simplicity of the presentation. For efficiency reasons however, this expansion is not done in some built-in predicates which accept lists of intervals, like for instance `X in` *list*.

where $\phi[X/e]$ denotes the formula ϕ where each free occurrence of variable X in ϕ is replaced by expression e (after the usual renaming of the variables in ϕ in order to avoid name clashes with the free variables in e).

Negations are eliminated by descending them to the comparison operators, with the obvious duality rules for the logical connectives, such as for instance, the rewriting of the negation of `equiv` into `xor`. It is worth noting that these transformations do not increase the size of the formula.

3.2 Inlining Rewrite Rules for Target CP Builtins

The constraint builtins of the target language (including global constraints) are specified with specific *inlining rules*. Such rules are mandatory for the terms that are not defined by Rules2CP statements, as well as for the arithmetic and logical expressions that are not expanded with the generic rewrite rules described in the previous section. The result of an inlining rule is called a *terminal term*.

The free variables in declarations are translated into finite domain variables of the target language. Interestingly, the naming conventions for the free variables in declarations described in the previous section provide a book-keeping mechanism that establishes the correspondance between the target language variables and their declaration in Rules2CP. This is crucial to debugging purposes and user-interaction [17]. The correspondance between the target language constraints and Rules2CP rules can be implemented similarly by keeping track of the Rules2CP rules that generate the input constraints for the target language by inlining rules.

The examples of inlining rules given in this section concern the compilation of Rules2CP to SICStus-Prolog [16]. Basic constraints are thus rewritten with term rewriting rules such as the following ones:

- `domain(`E, M, N`)` $\rightarrow_{csp}$ `"domain(`L, M, N`)"` if M and N are integers and where L is the list of variables remaining in E after rewriting
- A `>` B $\rightarrow_{csp}$ `"‘`A `#> ‘`B`"`
- A `and` B $\rightarrow_{csp}$ `"‘`A `#/\ ‘`B`"`
- `lexicographic(`L`)` $\rightarrow_{csp}$ `"lex_chain(‘`L`)"`

where backquotes in strings indicate subexpressions to rewrite. Obviously, such inlining rules generate programs of linear size.

The inlining rules for Rules2CP search predicates are more complicated as they need to create the list of the variables contained in an expression, and to sort the constraints, search predicates and optimization criteria in conjunctions. For example, the inlining *rule schema* for single criterion optimization is the following:

- A `and minimize(`C`)` $\rightarrow_{csp}$ `"‘`B`,minimize((‘`D`,labeling([up],‘`L`)),‘`C`)"`

where L is the list of variables occurring in the cost expression C, D is the goal associated to the labeling and search expressions occurring in A with disjunctions replaced by choice points, and B is the translation of formula A without its labeling and search expressions. Note that the generated code by this inlining rule is again of linear size.

Example 3 *The compilation of the N-queens problem in Example 1 generates the following SICStus Prolog goal:*

```
?  domain([Q_1_,Q_2_,Q_3_,Q_4_],1,4),
   Q_1_#\=Q_2_, Q_1_#\=1+Q_2_, Q_1_#\= -1+Q_2_,
   Q_1_#\=Q_3_, Q_1_#\=2+Q_3_, Q_1_#\= -2+Q_3_,
   Q_1_#\=Q_4_, Q_1_#\=3+Q_4_, Q_1_#\= -3+Q_4_,
   Q_2_#\=Q_3_, Q_2_#\=1+Q_3_, Q_2_#\= -1+Q_3_,
   Q_2_#\=Q_4_, Q_2_#\=2+Q_4_, Q_2_#\= -2+Q_4_,
   Q_3_#\=Q_4_, Q_3_#\=1+Q_4_, Q_3_#\= -1+Q_4_,
   labeling([],[Q_1_,Q_2_,Q_3_,Q_4_]).
```

Note that the inequality constraints are properly posted on ordered pairs of queens and that the other pairs of queens generated by the universal quantifiers have been eliminated at compile time by partial evaluation.

Example 4 *The result of compiling the disjunctive scheduling problem in Example 2 is the following:*

```
T1_ #>= 0, T6_#< 20,
T1_+1 #=< T2_, T2_+2 #=< T3_, T3_+3 #=< T6_,
T1_+1 #=< T4_, T4_+4 #=< T5_, T5_+2 #=< T6_,
minimize((((T2_+2 #=< T5_;T5_+2 #=< T2_),
           (T4_+4 #=< T3_;T3_+3 #=< T4_)),
         labeling([up],[T6_])),T6_).
```

The `search` *predicate applied to a first-order formula has been transformed into an* and-or *search tree, keeping the nesting of disjuncts without normalization. This is crucial to maintaining a linear complexity for this transformation.*

3.3 Confluence, Termination and Complexity

By forbidding multiple definitions, and restricting heads to contain only distinct variables as arguments, the compilation rules can be shown to be confluent. This means that the rewriting rules can be applied in any order, and generate the same constraint program on a given input model.

Proposition 1 *For any* `Rules2CP` *model, the compilation term rewriting system* $\rightarrow_{csp}$ *is confluent.*

Proof 1 *Let us show that the term rewriting system* $\rightarrow_{csp}$ *is orthogonal, i.e. left-linear and non-overlapping, which entails confluence [18].*

First, the heads of the $\rightarrow_{csp}$ *rewrite rules associated to Rules2CP rules and declarations are formed with one symbol and distinct variables as arguments, hence these rules are left-linear. Furthermore, multiple definitions of a head symbol are not allowed, and the renaming of free variables in declarations is deterministic, hence these rules are non-overlapping and constitute an orthogonal term rewriting system.*

Second, all the other $\rightarrow_{csp}$ *rules for predefined predicates and for inlined builtins are non-overlapping, since the symbol they rewrite can be rewritten with only one rewrite rule, and in only one way. This is enforced both in the predefined predicates dealing with lists, by imposing that their list arguments are expansed before rewriting, and in several inlining rules by imposing the expansion of the arguments first. Furthermore, the rules for builtins are also left-linear. This is clear in all cases except for the rules associated to binding operators,*

since the binding variable X appears in the expression e. However such a binding variable X denotes substitution occurrences in e and no pattern matching is done on X. In particular, no rewriting rule applies if X is not a variable. Hence the associated $\rightarrow_{csp}$ rule is left-linear w.r.t. pattern matching. Therefore the term rewriting system $\rightarrow_{csp}$ is orthogonal.

It is worth noticing that the preceding proof does not assume termination[5]. The property of confluence of $\rightarrow_{csp}$ compilation rules would thus hold as well for Rules2CP with recursive statements. By forbidding recursion however, it is intuitively clear that the compilation term rewriting system $\rightarrow_{csp}$ terminates. Without loss of generality, let us assume that Rules2CP models contains only one goal *solve* defined by a rule.

Definition 1 *Given a Rule2CP model M, let the* definition rank $\rho(s)$ *of a symbol s be defined inductively by:*

- $\rho(s) = 0$ *if s is not the head symbol of a declaration or rule in M,*
- $\rho(s) = n + 1$ *if s is the head symbol of a declaration or rule in M, and n is the greatest definition rank of the symbols in the right hand side of its declaration or rule.*

The definition rank of M is the maximum definition rank of the symbols in M.

Proposition 2 *For any* `Rules2CP` *model, the term rewriting system $\rightarrow_{csp}$ is Noetherian.*

Proof 2 *Each $\rightarrow_{csp}$ rewrite rule associated to Rules2CP declarations and rules strictly decreases the definition rank of the symbol it rewrites, and the other $\rightarrow_{csp}$ rules do not increase the ranks. As the multiset extension of a well-founded ordering is well-founded [21], this entails that the $\rightarrow_{csp}$ term rewriting system is Noetherian [19].*

Termination proofs by multiset path ordering imply primitive recursive derivation lengths [22]. Having forbidden recursion in Rules2CP statements however, a better complexity bound on the size of the generated program can be obtained:

Definition 2 *Given a Rule2CP model M, let the* aggregate rank $\alpha(s)$ *of a symbol s be defined inductively by:*

- $\alpha(s) = 0$ *if s is not the head symbol of a declaration or rule in M,*
- $\alpha(s) = max\{n + \alpha(s') \mid L = R \in M$, *$s$ is the head symbol of L and R contains a nesting of n aggregate operators or quantifiers on an expression containing symbol $s'\}$.*

The aggregate rank of M is the maximum aggregate rank of the symbols in M.

The size of the constraint program generated from a Rules2CP model can be bounded according to the maximum length of the lists in the model. For the case where the model contains list constructions of the form $[M..N]$, where M and N can be the result of arbitrary arithmetic calculations, we express the complexity bound as a function of the maximum length of the lists developed in the rewriting process:

[5] When termination is assumed, the non-overlapping condition, or more generally the confluence of critical pairs [19, 20], suffices to prove confluence without left-linearity.

Theorem 1 *For any* `Rules2CP` *model M with a goal of size one, the size of the generated program is in $O(l^a * b^r)$, where l is the maximum length of the lists developed in M (or at least 1), a is the aggregate rank of M, b is the maximum size of the declaration and rule bodies in M, and r is the definition rank of M.*

Proof 3 *The proof is by induction on a.*

In the base case, $a = 0$, there is no aggregate operator in M, and the size of the generated program is linearly bounded by r duplications of rule bodies, i.e. is in $O(b^r)$.

In the induction case, $a > 0$, let us first consider the size of the program generated without rewriting the outermost occurrences of aggregate and quantifier operators. By induction, this size is in $O(l^{a-1} * b^r)$. Now, this generated program can be duplicated at most l times by the outermost aggregate operators, hence the total size is in $O(l^a * b^r)$ under this strategy. Since by confluence Prop. 1, the generated program is independent of the strategy, the size of the generated program is thus in $O(l^a * b^r)$ under any strategy.

In the N-queens problem of Example 1 the aggregate rank is 2. The theorem thus tells us that the size of the generated program for a board of size l is indeed in $O(l^2)$.

4 The Packing Knowledge Modelling Library PKML

In this section, we illustrate the expressive power of Rules2CP with the definition of a Packing Knowledge Modelling Library (PKML) that is developed within the Net-WMS project for dealing with real size non-pure bin packing problems of the automotive and logistic industries.

4.1 Shapes and Objects

PKML refers to shapes in K-dimensional space with integer coordinates in $\mathbb{Z}^K$. A *point* in this space is represented by the list of its K integer coordinates `[i1,...,iK]`. These coordinates may be variables or fixed integer values.

In PKML, a *shape* is a *rigid assembly of boxes*. A *box* is an orthotope in $\mathbb{Z}^K$, and is represented in PKML by a record containing a *size* attribute which gives the list of the lengths of the box in each dimension. A shape is represented by a record containing an attribute *boxes* which gives the list of boxes composing the shape, and an attribute *positions* which gives the list of their positions in the assembly (i.e. a list of lists of coordinates). Some other attributes may be added for virtual reality representations, weights, etc.

The following declarations define respectively the volume of a box, a shape composed of a single box, the size of a shape (i.e. assembly of boxes) in a given dimension, and the volume of a shape in given dimensions (assuming no overlap in the assembly):

```
volume_box(B) = product(size(B)).

box(L) = { boxes = [ {size = L} ], positions = [ map(_,L,0) ] }.

size(S, D) = aggregate(I, [1..length(boxes(S))], max, 0,
                       nth(D,nth(I,positions(S))) +
```

```
                        nth(D,size(nth(I,boxes(S)))).

volume_assembly(S, Dims) = aggregate(B, boxes(S), +, 0, volume_box(B)).
```

It is worth noting that if the sizes of the boxes composing the shapes are known, the size and volume expressions evaluate into fixed integer values, whereas if the sizes are unknown, the expressions will evaluate to terms containing variables.

A PKML *object* such as a bin or an item, is a record containing an attribute `shapes` giving a list of *alternative shapes* for the object, an *origin* point, and some optional attributes such as weight, virtual reality representations or others. The alternative shapes of an object may be used to represent the different shapes obtained by rotating a basic shape around the different dimension axes, or for expressing the choice between different object shapes in a configuration problem. We do not distinguish between items and bins features, as bins at one level can become items at another level, like for instance in a multilevel bin packing problem for packing items into cartons, cartons in pallets, and pallets into trucks. The origin of an object in one dimension, its end in one dimension and its volume by cases on the alternative shapes (using reifcation), plus some obvious rules for weights, are predefined in PKML as follows:

```
origin(O, D) = nth(D, origin(O)).
end(O, D) = origin(O, D) + aggregate(S, shapes(O), +, 0,
                              (shape(O)=pos(S,shapes(O)))*size(S, D)).
volume(O, Dims) = aggregate(S, shapes(O), +, 0,
                    (shape(O)=pos(S,shapes(O)))*volume_assembly(S,Dims)).
volume(O) = volume(O, [1..length(origin(O))]).
lighter(O1, O2) --> weight(O1) =< weight(O2).
heavier(O1, O2) --> weight(O1) >= weight(O2).
```

4.2 Placement Relations

PKML uses Allen's interval relations [23] in one dimension, and the topological relations of the Region Connection Calculus [24] in higher-dimensions, to express placement constraints. These relations are predefined in the libraries given in Appendices A and B respectively[6]. They are used in PKML to define packing rules for pure bin packing and pure bin design problems, symmetry breaking strategies, as well as specific packing business rules for non pure problems taking into account other common sense rules and industrial requirements and expertise.

The part of the PKML library dealing with pure *bin packing problems* is defined as follows:

```
non_overlapping(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,
```

[6]For the sake of simplicity of the presentation, the region connection relations between box assemblies are defined, using the `size(S,D)` function, between the least boxes containing the assemblies. A better handling of assemblies involves formulas with quantifiers on the boxes of the assembly.

```
      exists(B, Bins,
         contains_touch_rcc(B,I,Dims))).

bin_packing(Items, Bins, Dims) -->
   containmentAE(Items, Bins, Dims) and
   non_overlapping(Items, Dims) and
   labeling(Items).
```

The rules define respectively the non-overlapping of a list of items in a list of dimensions, the containment of all items in bins, and pure bin packing problems. Pure *bin design problems* are defined similarly with a containment rule in some bin of all items:

```
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
   labeling(Items) and
   minimize(volume(Bin)).
```

Example 5 *Let us consider the following simple pure bin packing problem*

```
s1 = box([5,4,4]).
s2 = box([5,4,2]).
s3 = box([4,4,2]).
o1 = object(s1, [0,0,0]).
o2 = object(s2, [_,_,_]).
o3 = object(s3, [_,_,_]).
dimensions = [1,2,3].
bins = [o1].
items = [o2,o3].

? bin_packing(items, bins, dimensions) and
  variable_choice_heuristics([greatest(volume(^)), is(z(^))]).
```

On this example, the compiler described in the previous section generates the following SICStus-Prolog program:

```
:- use_module(library(clpfd)).

solve([O3_3,O3_,O3_2,O2_3,O2_,O2_2]) :-
   0#=<O2_, O2_+4#=<5,
   0#=<O2_2, O2_2+4#=<4,
   0#=<O2_3, O2_3+2#=<4,
   0#=<O3_, O3_+5#=<5,
   0#=<O3_2, O3_2+4#=<4,
   0#=<O3_3, O3_3+2#=<4,
   O3_+5#=<O2_#\/O2_+4#=<O3_#\/
   (O3_2+4#=<O2_2#\/O2_2+4#=<O3_2#\/
   (O3_3+2#=<O2_3#\/O2_3+2#=<O3_3)),
   labeling([],[O3_3,O3_,O3_2,O2_3,O2_,O2_2]).
```

```
?- ?- solve(L).
L = [0,0,0,2,0,0] ? ;
L = [0,0,0,2,1,0] ? ;
L = [2,0,0,0,0,0] ? ;
L = [2,0,0,0,1,0] ? ;
no
```

4.3 Packing Business Rules

Packing business rules are defined in Rules2CP to take into account further common sense or industrial requirements that are beyond the scope of pure bin packing problems [9]. For instance, the following rules about weights

```
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 on_top(O1, O2))
         implies
         lighter(O1,O2))).

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

express particular constraints on the weights of the items in an admissible packing. The following ones express constraints on the size of objects in a stack.

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

The complete PKML library including common sense rules dealing with the weight of objects and the surface contact of stacked items, is given in Appendix C. With these rules, Theorem 1 shows that PKML models containing lists of at most l elements generate constraint programs of size $O(l^4)$ in presence of both alternative shapes and assemblies of boxes, $O(l^3)$ in presence of only one of them, and $O(l^2)$ in presence of single box shapes only.

4.4 Packing Business Patterns

Business patterns can be used in PKML to express knowledge about some predefined (partial) solutions to packing problems. Such patterns are used in the industry, for instance for filling pallets, or trucks, with maximum stability according to some predefined solutions. Stability conditions can also be expressed with non-guillotine or non-visibility constraints [9], but packing patterns provide a pragmatic and complementary approach to these important requirements.

In PKML, packing patterns can be defined as records containing a list of item shapes given with the coordinates of their origin, and bounds on their weight:

```
pat1={shapes=[s1,...,sN], origins=[p1,...,pN], weight_max=[m1,...,mN]}.

pattern(Items, Bin, Patterns)
-->
exists(P, Patterns,
 forall(S, shapes(P),
 let(J, pos(S,shapes(P),
  exists(I, Items,
   S = shapes(I) and
   weight(I) =< nth(J,weight_max(P)) and
   origin(I) = nth(J,origins))))))).
```

The packing pattern rule places items in a bin according to some pattern taken from a list of patterns. This rule can be used in packing problems by first applying a pattern, and second completing the packing with the general `bin_packing` rule as follows:

```
? search(pattern(items, bin, patterns)) and
  bin_packing(items,[bin],[1 .. d]).
```

4.5 Compilation with the Global Constraint `geost`

The constraint `geost` [10] is a generic global constraint for higher-dimensional placement problems which is now parameterized by an integrated rule language [11]. A subset of PKML rules can be directly transformed into `geost` rules providing a very high level of pruning and remarkable efficiency.

The concerned subset of PKML is restricted to objects and shapes records, and to *linear* arithmetic expressions, i.e. linear combinations of domain variables. This excludes for instance the volume function used for bin design problems. With these restrictions, `geost` rules can be compiled into k-indexicals, i.e. functions that compute *forbidden sets* of object points represented as collections of k-dimensional boxes composed by unions and intersections [10].

The compilation of a PKML model into a constraint satisfaction problem using `geost`, mainly consists in the following steps:

1. extracting the definitions of objects and shapes from PKML statements in order to provide them to the `geost` constraint,

2. extracting the declarations and rules that refer to objects and shapes and satisfy the linearity condition, for providing them to the `geost` constraint,

3. compiling the PKML goals into the `geost` constraint plus the remaining constraint programming code for the rules and search predicates that are not accepted by the integrated rule language of `geost`, as described in the previous section.

This basic compilation scheme can be refined by adding extra dimensions, for instance for handling multiple bins packing problems by adding an extra dimension for bin assignment where each item has size one, or for handling scheduling aspects by adding an extra dimension for time, etc. [11].

5 Related Work

5.1 Comparison with Business Rules

Rules2CP is an attempt to use the business rules knowledge representation paradigm for constraint programming. Business rules are very popular in the industry because they provide a declarative mean for expressing expertise knowledge. Business rules should describe independent pieces of knowledge, and should be independent from a particular procedural interpretation by a rule engine [6]. Rules2CP realizes this aim in the context of combinatorial optimization problems, by tranforming business rules into efficient programs using completely different representations. Rules2CP rules are not general condition-action rules, also called production rules in the expert system community, but *logical rules* with only one head and no imperative actions. Bounded quantifiers are used to represent complex conditions. Such conditions can also be expressed in many production rules systems, but here they are used at compile-time to setup a constraint satisfaction problem, instead of at run-time to match patterns in a database of facts. As a rule-based modelling language, Rules2CP complies to the principles of the business rules manifesto [6].

5.2 Comparison with OPL and Zinc

Rules2CP differs from OPL [1] and Zinc [2, 3] modelling languages in several aspects among which: the restriction in Rules2CP to simple data structures of records and enumerated lists, the absence of recursion, the declarative specification of heuristics as preference orderings, and the absence of program annotations. This trade-off for ease of use was motivated by our search for a declarative modelling language with no complicated programming constructs. We have shown that the declarations and rules of Rules2CP allow the user to give names to data and knowledge rules without complicated variable scopes. A simple module system is used in Rules2CP to avoid name clashes.

The simplicity of these design choices is reflected in the obtention of a complexity bound on the size of the constraint programs generated from Rules2CP models (Theorem 1). Moreover, the partial evaluation mechanism used in the rewriting process eliminates at compile-time the overhead due to the simplicity of our data and control structures.

Interestingly, we have shown that complex search strategies can be expressed *declaratively* in Rules2CP, by specifying *decision variables* and *branching formulas*, as well as both static and dynamic choice heuristics as *preference orderings* on variables and values. These specifications are more declarative than what is

achieved in OPL for programming search, and use all the power of the language to define heuristic criteria.

On the other hand, we have not considered the compilation of Rules2CP to other solvers such as local search, or mixed integer linear programs, as has been done for OPL and Zinc systems.

5.3 Comparison with Constraint Logic Programming

As a modelling language, Rules2CP is a constraint logic programming language, but not in the formal sense of the CLP scheme of Jaffar and Lassez [25]. Rules2CP models can be compiled to CLP(FD) programs in a straightforward way by translating Rules2CP rules into Prolog clauses, and by keeping the $\rightarrow_{csp}$ rewriting for the remaining expressions. Note that the converse translation of Prolog programs into Rules2CP models is not possible (apart from an arithmetic encoding) because of the absence of recursion and of general list constructors in Rules2CP.

Furthermore, free variables are not allowed in the right hand side of Rules2CP rules. Instead of the local scope mechanism used for the free variables in CLP rules, a global scope mechanism in used for the free variables in Rules2CP declarations. This global scope mechanism has no counterpart in the CLP scheme which makes it often necessary to pass the list of all variables as arguments to CLP predicates[7].

5.4 Comparison with Term Rewriting Systems Tools

The compilation of Rules2CP models to constraint programs is defined and implemented by a term rewriting system. The properties of confluence and termination of this process have been shown using term rewriting theory.

There are several term rewriting system tools available that could be directly used for the implementation of the Rules2CP compiler. For instance, in the context of target constraint solvers in Java, such as e.g. Choco, and for Java programming environments in which Rules2CP data structures may be defined by Java objects, the term rewriting system TOM [26] provides a pattern matching compiler for programming term transformations defined by rules. This would make of TOM an ideal system for implementing a Rules2CP compiler to Java, through a direct translation of $\rightarrow_{csp}$ rules into TOM rules.

6 Conclusion

The Rules2CP language is a rule-based modelling language for constraint programming. It has been designed to allow non-programmers express common sense rules and industrial requirements about combinatorial optimization problems with business rules (using appropriate editors). In compliance to the business rules manifesto [6], Rules2CP rules are declarative, independent from each other, and not necessarily executed by a rule engine.

[7]For that reason, global variables are introduced as extra logical features in many CLP systems.

We have shown that Rules2CP models can be compiled to constraint programs using term rewriting and partial evaluation. We have shown the confluence of these transformations and provided a bound on the size of the generated program. The obtention of such a complexity result reflects the simplicity of our design choices for Rules2CP, such as the absence of recursion and of general list constructor for instance.

The expressivity of Rules2CP has been illustrated with a complete library for packing problems, called PKML, which, in addition to pure bin packing and bin design problems, can deal with extra constraints about weights, oversizes, equilibrium constraints, and specific packing business rules. Furthermore, a substantial part of PKML rules can be very efficiently compiled within the geometric global constraint `geost` [11].

Search strategies can also be specified declaratively in Rules2CP, as well as both static and dynamic heuristics defined as preference orderings on variables and values. This method for specifying heuristics is very expressive, and revealed a weakness in the constraint programming systems that cannot express different value choice heuristics for different variables ordered by a variable choice heuristics.

Acknowledgements.

This work is supported by the European FP7 Strep project Net-WMS. We would like to thank especially Mats Carlsson, Magnus Ågren, Nicolas Beldiceanu, and Abder Aggoun from KLS-Optim, and our partners at PSA Peugeot Citroën, for the numerous discussions we had together on rules. The first author is also grateful to the previous collaboration he had on this topic in the RNTL project Manifico with Christian de Sainte Marie and Xavier Ceugniet from ILOG and Claude Kirchner and Pierre-Etienne Moreau from CNRS LORIA.

References

[1] Van Hentenryck, P.: The OPL Optimization programming Language. MIT Press (1999)

[2] Rafeh, R., de la Banda, M.G., Marriott, K., Wallace, M.: From Zinc to design model. In: Proceedings of PADL'07, Springer-Verlag (2007) 215–229

[3] de la Banda, M.G., Marriott, K., Rafeh, R., Wallace, M.: The modelling language Zinc. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming CP'06), Springer-Verlag (2006) 700–705

[4] Puchinger, J., Stuckey, P.J., Wallace, M., Brand, S.: From high-level model to branch-and-price solution in g12. In: Proceedings of CPAIOR'08. Lecture Notes in Computer Science, Paris, France, Springer-Verlag (2008)

[5] Van Hentenryck, P., Michel, L.: Constraint-based Local Search. MIT Press (2005)

[6] Group, B.R.: The business rules manifesto (2003) Business Rules Group `http://www.businessrulesgroup.org/brmanifesto.htm`.

[7] Vianu, V.: Rule-based languages. Annals of Mathematics and Artificial Intelligence **19** (1997) 215 – 259

[8] Fages, F.: Consistency of Clark's completion and existence of stable models. Methods of Logic in Computer Science **1** (1994) 51–60

[9] Carpenter, H., Dowsland, W.: Practical consideration of the pallet loading problem. Journal of the Operations Research Society **36** (1985) 489–497

[10] Beldiceanu, N., Carlsson, M., Poder, E., Sadek, R., Truchet, C.: A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In Bessière, C., ed.: Proc. CP'2007. Volume 4741 of LNCS., Springer (2007) 180–194 Also available as SICS Technical Report T2007:08, `http://www.sics.se/libindex.html`.

[11] Beldiceanu, N., Carlsson, M., Martin, J.: A geometric constraint over k-dimensional objects and shapes subject to business rules. SICS Technical Report T2008:04, Swedish Institute of Computer Science (2008)

[12] Haemmerlé, R., Fages, F.: Modules for Prolog revisited. In: Proceedings of International Conference on Logic Programming ICLP 2006. Number 4079 in Lecture Notes in Computer Science, Springer-Verlag (2006) 41–55

[13] Van Hentenryck, P.: Constraint satisfaction in Logic Programming. MIT Press (1989)

[14] Huang, J., Darwiche, A.: The language of search. Journal of Artificial Intelligence Research **29** (2007) 191–219

[15] Apt, K., Wallace, M.: Constraint Logic Programming using Eclipse. Cambridge University Press (2006)

[16] Carlsson, M., et al.: SICStus Prolog User's Manual. Swedish Institute of Computer Science. Release 4 edn. (2007) ISBN 91-630-3648-7.

[17] Fages, F., Soliman, S., Coolen, R.: CLPGUI: a generic graphical user interface for constraint logic programming. Journal of Constraints, Special Issue on User-Interaction in Constraint Satisfaction **9** (2004) 241–262

[18] Rosen, B.: Tree-manipulating systems and Church-Rosser theorems. Journal of the ACM **20** (1973) 160–187

[19] Terese: Term Rewriting Systems. Volume 55 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2003)

[20] Haemmerlé, R., Fages, F.: Abstract critical pairs and confluence of arbitrary binary relations. In: Proceedings of th 18th International Conference on Rewriting Techniques and Applications, RTA'07. Number 4533 in Lecture Notes in Computer Science, Springer-Verlag (2007) 214–228

[21] Manna, Z.: Lectures on the Logic of Computer Programming. Number 0031 in CBMS-NSF regional conference series in applied mathematics. SIAM (1980)

[22] Hofbauer, D.: Termination proofs by multiset path orderings imply primitive recursive derivation lengths. Theoretical Computer Science **105** (1992) 129–140

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

[24] Randell, D., Cui, Z., Cohn, A.: A spatial logic based on regions and connection. In Nebel, B., Rich, C., Swartout, W.R., eds.: Proc. of 2nd International Conference on Knowledge Representation and reasoning KR'92, Morgan Kaufmann (1992) 165–176

[25] Jaffar, J., Lassez, J.L.: Constraint logic programming. In: Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany, ACM (1987) 111–119

[26] Balland, E., Brauner, P., Kopetz, R., Moreau, P.E., Reilles, A.: Tom: Piggybacking rewriting on java. In: Proceedings of th 18th International Conference on Rewriting Techniques and Applications, RTA'07. Number 4533 in Lecture Notes in Computer Science, Springer-Verlag (2007)

7 Appendix A: Allen's Interval Relations Library

In one dimension, the library of Allen's interval relations [23] between objects is predefined in Rules2CP in the following 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.

8 Appendix B: Region Connection Calculus Library

In higher-dimensions, the library of topological relations of the Region Connection Calculus [24] 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.

```
import(allen2).

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.

9 Appendix C: PKML Library

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

```
import(rcp).
import(rcc8).
import(pkml_surface).
import(pkml_weight).

volume_box(B) = product(size(B)).

box(L) = { boxes = [ {size = L} ], positions = [ map(_,L,0) ] }.

size(S, D) = aggregate(I, [1..length(boxes(S))], max, 0,
```

```
                              nth(D,nth(I,positions(S))) +
                              nth(D,size(nth(I,boxes(S))))).

volume_assembly(S, Dims) =
   aggregate(B, boxes(S), +, 0, volume_box(B)).

object(S, L) = {shapes=[S], shape=1, origin=L}.

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

domain_shape(O) --> shape(O) in [1..length(shapes(O))].

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) + aggregate(S, shapes(O), +, 0,
                            (shape(O)=pos(S,shapes(O)))*size(S, D)).

volume(O, Dims) =
   aggregate(S, shapes(O), +, 0,
             (shape(O)=pos(S,shapes(O)))*volume_assembly(S,Dims)).

volume(O) = volume(O, [1..length(origin(O))]).

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(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(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
   labeling(Items) and
   minimize(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:

```
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 on_top(O1, O2))
         implies
         lighter(O1,O2))).

weight_balancing(Items, Bin, D, Ratio) -->
 let(L,sum(map(Il,Items,weight(Il)*(end(Il,D)=<(end(Bin,D)/2)))),
  let(R,sum(map(Ir,Items,weight(Ir)*(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:

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

10 Appendix D: Small Example with Weights

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

```
import(pkml).

psa_bin_packing(Items, Bin, Dims) -->
        gravity(Items) and
        weight_stacking(Items) and
        weight_balancing(Items, Bin, 1, 20) and
   stack_oversize(Items, 10) and
        variable_choice_heuristics([greatest(weight(^)),
                                    greatest(volume(^,Dims)),
                                    smallest(uid(^)),
                is(z(^))
                ]) and
        bin_packing(Items,[Bin],Dims).

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

o1 = object(s1, [0,0,0]).
o2 = object(s4, [_,_,_], 413).
o3 = object(s5, [_,_,_], 463).
o4 = object(s5, [_,_,_], 842).
o5 = object(s3, [_,_,_], 422).
o6 = object(s4, [_,_,_], 266).
o7 = object(s4, [_,_,_], 321).
o8 = object(s2, [_,_,_], 670).
o9 = object(s6, [_,_,_], 440).
o10 = object(s7, [_,_,_], 325).

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

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

The generated code in SICStus-Prolog on this small example is the following:

```
:- use_module(library(clpfd)).

solve([O3_3,O3_,O3_2,O2_3,O2_,O2_2]) :-
R_359#<=>O2_+224#=<1203/2,
R_360#<=>O3_+224#=<1203/2,
R_361#<=>O2_#>=1203/2,
R_362#<=>O3_#>=1203/2,
R_363#<=>O2_+224#=<1203/2,
```

```
R_364#<=>O3_+224#=<1203/2,
R_365#<=>O2_#>=1203/2,
R_366#<=>O3_#>=1203/2,
O2_3#=0#\/O2_+224#>O3_#/\O3_+224#>O2_#/\
(O2_2+224#>O3_2#/\O3_2+224#>O2_2)#/\O3_3+74#=O2_3,
O3_3#=0#\/O3_+224#>O2_#/\O2_+224#>O3_#/\
(O3_2+224#>O2_2#/\O2_2+224#>O3_2)#/\O2_3+111#=O3_3,
O3_+224#=<O2_#\/O2_+224#=<O3_#\/
(O3_2+224#=<O2_2#\/O2_2+224#=<O3_2)#\/O2_3+111#\=O3_3,
100*max(413*R_359+(463*R_360+0),413*R_361+(463*R_362+0))#=<
120*min(413*R_363+(463*R_364+0),413*R_365+(463*R_366+0)),
O2_+224#=<O3_#\/O3_+224#=<O2_#\/
(O2_2+224#=<O3_2#\/O3_2+224#=<O2_2)#\/
max(max(O2_,O3_)-min(O2_,O3_),
        max(O2_+224,O3_+224)-min(O2_+224,O3_+224))#=<10#/\
max(max(O2_2,O3_2)-min(O2_2,O3_2),
        max(O2_2+224,O3_2+224)-min(O2_2+224,O3_2+224))#=<10,
O3_+224#=<O2_#\/O2_+224#=<O3_#\/
(O3_2+224#=<O2_2#\/O2_2+224#=<O3_2)#\/
max(max(O3_,O2_)-min(O3_,O2_),
        max(O3_+224,O2_+224)-min(O3_+224,O2_+224))#=<10#/\
max(max(O3_2,O2_2)-min(O3_2,O2_2),
        max(O3_2+224,O2_2+224)-min(O3_2+224,O2_2+224))#=<10,
0#=<O2_,
O2_+224#=<1203,
0#=<O2_2,
O2_2+224#=<235,
0#=<O2_3,
O2_3+111#=<239,
0#=<O3_,
O3_+224#=<1203,
0#=<O3_2,
O3_2+224#=<235,
0#=<O3_3,
O3_3+74#=<239,
O3_+224#=<O2_#\/O2_+224#=<O3_#\/
(O3_2+224#=<O2_2#\/O2_2+224#=<O3_2#\/
(O3_3+74#=<O2_3#\/O2_3+111#=<O3_3)),
labeling([],[O3_3,O3_,O3_2,O2_3,O2_,O2_2]).
```

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399