From Rules to Constraint Programs with the Rules2CP Modelling Language

François Fages and Julien Martin

Projet Contraintes, INRIA Rocquencourt,
BP105, 78153 Le Chesnay Cedex, France
http://contraintes.inria.fr

Abstract. In this paper, we present a rule-based modelling language for constraint programming, called Rules2CP. Unlike other modelling languages, Rules2CP adopts a single knowledge representation paradigm based on rules without recursion, and a restricted set of data structures based on records and enumerated lists given with iterators. We show that this is sufficient to model constraint satisfaction problems, together with search strategies where search trees are expressed by logical formulae, and heuristic choice criteria are defined by preference orderings on variables and formulae. We describe the compilation of Rules2CP statements to constraint programs over finite domains, by a term rewriting system 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 first with simple examples, and then 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. The performances of both the compiler and the generated code are evaluated on Korf's benchmarks of optimal rectangle packing problems.

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 an application expert 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-programmers 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 for instance OPL [1,2], Zinc [3,4] and Essence [5].

In industry, the business rules approach to knowledge representation has a wide audience because of the declarativity and granularity of rules which can be introduced, checked, and modified one by one, and independently of any particular procedural interpretation by a rule engine [6]. This provides an attractive knowledge representation scheme for quickly evolving requirements, and for maintaining systems with up to date information. In this article, we show that such a rule-based knowledge representation paradigm can be developed as a front-end modelling language for constraint programming. We present a general purpose rule-based modelling language for constraint programming, called Rules2CP. Unlike multi-headed condition-action rules, also called production rules, Rules2CP rules are restricted to *logical rules*, with one head and no imperative actions, and where bounded quantifiers are used to represent complex conditions. Such rules comply to the principle of independence from a procedural interpretation by a rule engine [6], which is concretely demonstrated in Rules2CP by their compilation to constraint programs using a completely different representation.

Unlike the other modelling languages proposed for constraint programming, Rules2CP adopts a restricted set of data structures based on records and enumerated lists, given with iterators. We show that this is sufficient to express constraint satisfaction problems, together with search strategies where the search tree is expressed by logical formulae, and complex heuristic choice criteria are defined as preference orderings on variables and formulae.

The next section presents the Rules2CP language and shows how search strategies and heuristics can be specified in a declarative manner. Sec. 2 describes the compilation of Rules2CP models into constraint programs over finite domains with reified constraints, by term rewriting 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, and provide a complexity bound on the size of the generated program.

Sec. 4 illustrates the expressive power 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 of the automotive industry. The performances of both the compiler and the generated code are evaluated in this section on Korf's benchmarks of optimal rectangle packing [7].

Finally, Sec. 5 compares Rules2CP with related work on OPL, Zinc and Essence modelling languages, business rules, constraint logic programming and term rewriting systems. We conclude on the simplicity and efficiency of Rules2CP and on some of its current limitations.

2 The Rules2CP Language

2.1 Introductory Examples

Rules2CP is an untyped language for modelling constraint satisfaction problems over finite domains using rules and declarations with records and enumerated lists as data structures. Let us first look at some simple examples.

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

Example 1. The classical N-queens problem, i.e. placing N queens on a chess-board of size $N \times N$ such that the queens are not on the same raw, column or diagonal, can be modelled in Rules2CP with two declarations (`q` and `board`), for creating a list of records representing the positions of the queens on the chess board, one rule `safe` for defining when the queens do not attack each other (using the global constraint `all_different` below), another rule `solve` for defining the constraints and the search strategy, and one goal for solving a problem of a given size:

```
q(I) = {row = _, column = I}.
board(N) = map(I, [1..N], q(I)).
safe(B) --> all_different(B) and
            forall(Q, B, forall(R, B,
             let(I, column(Q), let(J, column(R),
              I<J implies 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
                dynamic(variable_ordering([least(domain_size(row(^)))])
                        and labeling(B))).
? solve(4).
```

The search is specified in the `solve` rule by the `labeling` predicate for enumerating the variables contained in `B` with a *dynamic* variable ordering heuristics by least domain size (first-fail heuristics).

Example 2. A disjunctive scheduling problem, such as the classical bridge problem [1], consists in finding the earliest start dates for a set of tasks given with their durations, under constraints of precedence and mutual exclusion between tasks. Such problems can be modelled in Rules2CP with records for tasks, and rules for precedence and disjunctive constraints, as follows:

```
t1 = {start=_, duration=2}. t2 = {start=_, duration=5}.
t3 = {start=_, duration=4}. t4 = {start=_, duration=3}.
t5 = {start=_, duration=1}.
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)
disjunctives --> disj(t2,t3) and disj(t2,t4) and disj(t3,t4).
? domain([t1,t2,t3,t4,t5], 0, 20) and precedences and
  conjunct_ordering([greatest(duration(A)+duration(B) if ^ is disj(A,B))])
  and minimize(disjunctives, start(t5)).
```

The goal posts the domain and precedence constraints, specifies a heuristic criterion for ordering the disjunctive constraints by decreasing durations of tasks, and defines the search strategy by a logical formula (`disjunctives`) composed of a conjunction of disjunctive constraints, and a minimization criterion (the starting date of task `t6`). It is worth noting that this model does not use variable labeling. In a computed optimal solution, the non-critical tasks will have a flexible starting date.

The ordering criterion is about the duration attributes of the tasks involved in the `disj` rules, and does not actually depend on the variables. This strategy corresponds to the ordering used implicitly in the classical bridge problem benchmark. By adding a criterion for selecting the disjunctive with highest difference of durations in case of equality, as follows

```
conjunct_ordering([greatest(duration(A)+duration(B) if ^ is disj(A,B)),
                   greatest(abs(duration(A)-duration(B)) if ^ is disj(A,B))]).
```

the performances are slightly improved in the bridge problem.

2.2 Syntax

Let an *ident* be a word beginning with a lower case letter or any word between quotes, a *name* be an identifier possibly prefixed by other identifiers for module and package names, and a *variable* be a word beginning with either an upper case letter or the underscore character. The syntax of Rules2CP statements is given by the following grammar:

```
statement ::= import name. | head = expr. | head --> fol. | ? fol.
name      ::= ident | name:ident
head      ::= ident | ident(var,...,var)
fol       ::= varbool | name | name(expr,...,expr) | expr relop expr
            | fol logop fol | not fol | forall(var,expr,fol) | exists(var,expr,fol)
            | foldl(var,expr,logop,expr,expr) | foldr(var,expr,logop,expr,expr)
            | let(var,expr,fol) | search(fol) | dynamic(fol)
expr      ::= varint | fol | string | [ enum ] | {ident = expr,...,ident= expr}
            | name | name(expr,...,expr) | expr op expr |
            | foldl(var,expr,op,expr,expr) | foldr(var,expr,op,expr,expr)
            | map(var,expr,expr)
enum      ::= enum , enum | expr | expr .. expr
varint    ::= var | integer
varbool   ::= var | 0 | 1
op        ::= + | - | * | / | min | max | log | exp
relop     ::= < | =< | = | # | >= | >
logop     ::= and | or | implies | equiv | xor
```

A statement is either a module import, a declaration, a rule or a goal. In order to avoid name clashes in declaration and rule heads, the language includes a simple module system that prefixes names with module and package names, similarly to [8]. A head is formed with an *ident* with distinct variables as arguments. Recursive definitions, as well as multiple definitions of a same head symbol, are forbidden in declarations and rules, and each name must be defined before its use. Apart from this, the order of the statements in a Rules2CP file is not relevant.

The set $V(E)$ of *free variables* in an expression E is the set of variables occurring in E and not bound by a `forall, exists, let, foldl, foldr` or `map` operator. 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 unknown variables of the problem.

The only data structures are integers, strings, enumerated lists and records. Lists are formed without a binary list constructor, by enumerating all 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: `length(`*list*`)`, `nth(`*integer,list*`)`, `pos(`*element,list*`)` and *attribute*`(`*record*`)` are predefined for accessing the components of lists and records. Furthermore, records have a default integer attribute `uid` which provides them with a unique identifier.

The predefined function `variables(`*expr*`)` returns the list of variables contained in an expression. The predefined predicate X `in` *list* constrains the variable X to take integer values in a list of integer values. `domain(`*expr,min,max*`))` is predefined to set the domain of all variables occurring in *expr*.

A *fol* formula can be considered as a 0/1 integer expression. This usual coercion between booleans and integers, called *reification*, provides a great expressivity [9]. The (left and right) fold operators cannot be defined in first-order logic and are Rule2CP builtins. These operators iterate the application of a binary operator on a list of arguments. For instance, the product of the elements in a list is defined by `product(L)=foldr(X,L,*,1,X)`. Furthermore, a *fol* formula can be evaluated dynamically instead of statically by prefixing the formula with the predicate `dynamic`.

2.3 Search Predicates

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 issue is to specify the *decision variables* and the *branching formulas* of the problem in a declarative manner, and then heuristics as *preference orderings* on variables and formulae.

In Rules2CP, the labeling of decision variables can be specified 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 (see Example 1). The *branching formulas* are declared similarly with the predicate `search(`*fol*`)` for specifying a search procedure by branching on all disjunctions and existential quantifications occurring in a first-order formula (see Example 2). Note that without the search predicate, the formula in argument would be treated as a constraint by reification. A similar approach to specifying search has been proposed for SAT in [10]. 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.

The predefined *optimisation predicates*, `minimize(`*fol,expr*`)` for searching a fol and minimizing an expression, and `maximize(`*fol,expr*`)`, can be imbricated. This makes it possible to express multicriteria optimisation problems, and the search for Pareto optimal solutions according to the lexicographic ordering of the criteria as read from left to right.

2.4 Heuristics as Ordering Criteria

Adding the capability to express heuristic knowledge is mandatory for efficiency. This is done in Rules2CP with predefined predicates for specifying both static and dynamic choice criteria on variables and values for `labeling`, and on conjunctive and disjunctive formulae for `search`. Dynamic criteria for ordering variables and values in `labeling` are standard in constraint programming systems, see for instance [11,12]. In Rules2CP, they are defined more generally using the expressive power of the language for specifying various criteria depending on static or dynamic expression values.

The `variable_ordering` predicates take a list of criteria for ordering the variables in subsequent `labeling` predicate. 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. The criteria have the following forms: `greatest(`*expr*`)`, `least(`*expr*`)`, `any(`*expr*`)` or `is(`*expr*`)`. The expression *expr* in a criterion contains the symbol `^` for denoting, for a given variable, the left-hand side of the Rules2CP declaration that introduced the variable. If the expression cannot be evaluated on a variable, the criterion is ignored. 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. For instance, in a 3-dimensional bin packing problem, the predicate `variable_ordering([greatest(volume(^)), least(uid(^))])` specifies a lexicographic static ordering of the variables by decreasing volume of the object in which they have been declared, and by increasing *uid* attribute of the object (for grouping the variables belonging to a same object).

The `value_ordering` predicate takes similarly a list of criteria of the forms: `up`, `up(`*expr*`)`, for enumerating values in ascending order for the variables matching the expression, or `down`, `step` for binary choices, `enum` for multiple choices, `bisect` for dichotomy choices. A criterion applies to a variable if it matches the expression. For instance, in a bin packing problem with x, y, z coordinates, the predicate `value_ordering([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.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 to the constraint programming systems that mix variable and value choice strategies in a single option list, and for which one cannot express different value choice heuristics for the different variables in a same labeling predicate [12]. In these cases, the compiler generates a labeling program.

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 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.

3 Compilation to Constraint Programs over Finite Domains with Reified Constraints

Rules2CP models can be compiled 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. Let the *size* of an expression or formula be the number of nodes in its tree representation, and let us denote by $\rightarrow$ the term rewriting rules of the compilation process. These rules are composed of generic rewrite rules and code generation rules.

3.1 Generic Rewrite Rules

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

$L \rightarrow R$ for every rule of the form L `-->` R (where $V(R) \subseteq V(L)$)
$L\sigma \rightarrow R\sigma\theta$ for every declaration of the form L `=` R 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 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 declaration. 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 instance of the head $q(i)$. These conventions provide a basic book-keeping mechanism for retrieving the Rules2CP variables introduced in declarations from their variable names. This is necessary to implement the heuristic criteria, as well as for debugging and user-interaction purposes [13].

The ground arithmetic expressions are rewritten with the rule

expr $\rightarrow$ 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 with the following rule schemas that impose that the lists in arguments are expanded first:

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

The quantifiers, `foldr`, `foldl`, `map` and `let` operators are binding operators which use a dummy variable to denote place holders in an expression. They are rewritten under the condition that their first argument is a *variable* and their second argument is an expanded list:

`foldr(`X`,[`e_1`,`$\cdots$`,`e_N`],`op`,`e`,`ϕ`)` $\rightarrow$ $\phi[X/e_1]$ op (... op $\phi[X/e_N]$) (e if $N = 0$)
`forall(`X`,[`e_1`,`$\cdots$`,`e_N`],`ϕ`)` $\rightarrow$ $\phi[X/e_1]$ `and` ... `and` $\phi[X/e_N]$ (`1` if $N = 0$)
`exists(`X`,[`e_1`,`$\cdots$`,`e_N`],`ϕ`)` $\rightarrow$ $\phi[X/e_1]$ `or` ... `or` $\phi[X/e_N]$ (`0` if $N = 0$)
`map(`X`,[`e_1`,`$\cdots$`,`e_N`],`ϕ`)` $\rightarrow$ `[`$\phi[X/e_1]$, ..., $\phi[X/e_N]$`]`
`let(`X`,`e`,`ϕ`)` $\rightarrow$ $\phi[X/e]$

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 variables and comparison operators, with the obvious duality rules for the logical connectives, such as for instance, replacing the negation of `and` (resp. `equiv`) into `or` (resp. `xor`) etc. It is worth noting that these transformations do not increase the size of the formula.

3.2 Code Generation Rules

After the application of the previous generic rewrite rules, the actual transformation of a Rules2CP model to a constraint program of some target language, is specified with *code generation rules.* Such rules are needed for the terms that are not defined by Rules2CP statements, e.g. builtin constraints, as well as for the arithmetic and logical expressions that are not expanded with the generic rewrite rules described in the previous section. The free variables in declarations are translated into finite domain variables of the target language, with the basic book-keeping mechanism provided by the naming conventions.

The examples of code generation rules given in this section concern the compilation of Rules2CP to SICStus-Prolog [12]. Basic constraints are thus rewritten with term rewriting rules such as the following ones, where backquotes in strings indicate subexpressions to rewrite:

A `>` $B \rightarrow$ `"‘`A `#>` `‘`B`"`
A `and` $B \rightarrow$ `"‘`A `#/\` `‘`B`"`
`lexicographic(`L`)` $\rightarrow$ `"lex_chain(‘`L`)"`
`domain(`E, M, N`)` $\rightarrow$ `"domain(`L, M, N`)"` if M and N are integers and where L is the list of variables remaining in E after rewriting
`minimize(`F, C`)` $\rightarrow$ `"minimize((search(‘`F`),labeling([up],‘`L`)),‘`C`)"` where L is the list of variables occurring in the cost expression C.

Obviously, such code generation rules generate programs of linear size. In addition to this static expansion of Rules2CP goals in a constraint program goal, clauses are also generated for rules and declarations in order to interpret the expressions under `dynamic` with the Rules2CP interpreter, which is not be described for lack of space.

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

```
:- use_module(library(clpfd)).
:- use_module(r2cp).
...
solve([Q_1,Q_2,Q_3,Q_4]) :-
  rcp_var(from(q(1),0,1), Q_1), rcp_var(from(q(2),0,1), Q_2),...
  domain([Q_1,Q_2,Q_3,Q_4], 1, 4),
  all_different([Q_1,Q_2,Q_3,Q_4]),
  Q_1#\=1+Q_2, Q_1#\= -1+Q_2, Q_1#\=2+Q_3, Q_1#\= -2+Q_3, Q_1#\=3+Q_4,
  Q_1#\= -3+Q_4, Q_2#\=1+Q_3, Q_2#\= -1+Q_3, Q_2#\=2+Q_4,
  Q_2#\= -2+Q_4, Q_3#\=1+Q_4, Q_3#\= -1+Q_4,
  rcp_variable_ordering([least(var_order_criterion(1,[]))]),
  rcp_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. As the search heuristics is dynamic, the Rules2CP interpreter is included in the generated program to interpret the dynamic variable ordering heuristics using the labeling predicate of the Rules2CP interpreter. In this case, the program is equivalent to SICStus Prolog `labeling` with the first-fail option but the method is general.

Example 4. The disjunctive scheduling problem in Example 2 is compiled in a constraint program which does not use the Rules2CP interpreter:

```
solve([T1,T2,T4,T3,T5]) :-
  domain([T1,T2,T3,T4,T5], 0, 20),
  T1+2#=<T2, T2+5#=<T5, T1+2#=<T3, T3+4#=<T5, T1+2#=<T3, T3+4#=<T5,
  minimize((((T2+5#=<T3;T3+4#=<T2),(T2+5#=<T4;T4+3#=<T2),
            (T3+4#=<T4;T4+3#=<T3)),labeling([up],[T5])), T5).
```

In the `minimize` predicate, the disjunctive formulae in the *and-or* search tree have been reordered according to the heuristics by decreasing sums of the task durations. The labeling of the variables contained in the cost function is added by the compiler.

3.3 Confluence, Termination and Complexity

By having forbidden multiple definitions, and restricted the heads to contain only distinct variables as arguments, one can show :

Proposition 1. *For any* `Rules2CP` *model, the compilation term rewriting system* $\rightarrow$ *is confluent.*

This means that the rewriting rules can be applied in any order, and generate the same constraint program on a given input model. The proof in [14] shows that the term rewriting system $\rightarrow$ is orthogonal, i.e. left-linear and non-overlapping, which entails confluence [15] without termination assumption. By forbidding recursion however, termination clearly holds:

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 $\rightarrow$ is Noetherian.*

Furthermore, a complexity bound on the size of the generated program can be obtained.

Definition 2. *Given a Rule2CP model M, let the* fold 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 fold operators or quantifiers on an expression containing symbol s'*$\}$.

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

Proposition 3. *For any* `Rules2CP` *model M, the size of the generated program is in $O(l^a * b^r)$, where l is the maximum length of the lists in M (or at least 1), a is the fold 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. The proof is by induction on a. In the base case, $a = 0$, there is no fold 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 fold 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 fold 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, since the fold rank is 2, the proposition 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) developed in the Net-WMS project for dealing with real size non-pure bin packing problems in logistics and automotive industry. A large subset of PKML rules restricted to linear constraints has been shown in [16] to be compilable with indexical constraints in the geometrical kernel of the global constraint `geost` [17] for higher-dimensional placement problems. Here we define PKML as a library of Rules2CP declarations and rules.

4.1 Shapes and Objects

PKML refers to shapes 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 one *size* attribute giving the list of the lengths of the box in each dimension. A shape is represented by a record containing one attribute *boxes* for the list of boxes composing the shape, and one attribute *positions* for the list of their positions in the assembly (i.e. a list of lists of coordinates). For instance:

```
point1 = [x1,...,xK].
box1 = {size = [l1,...,lk]}.
shape1 = { boxes=[b1,...,bM], positions=[p1,...,pM]}
object1 = {shapes=[s1,...,sN], shape=_, origin=[x1,...,xK]}
```

A PKML *object*, such as a bin or an item, is a record containing one attribute `shapes` for the list of its *alternative shapes*, one *origin* point, and some optional attributes such as weight, virtual reality representations or others. The alternative shapes of an object may be the discrete rotations of a basic shape, or different object shapes in a configuration problem. We do not distinguish between items and bins features, since 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 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) = foldr(I, [1..length(boxes(S))], max, 0,
                        nth(D,nth(I,positions(S))) +
                        nth(D,size(nth(I,boxes(S))))).
volume_assembly(S, Dims) = foldr(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 evaluate to terms containing variables. These terms are used in PKML to define with reified constraints the end in one dimension and the volume of an object with alternative shapes, as follows:

```
origin(O, D) = nth(D, origin(O)).
end(O, D) = origin(O, D) + foldr(S, shapes(O), +, 0,
                            (shape(O)=pos(S,shapes(O)))*size(S, D)).
volume(O, Dims) = foldr(S, shapes(O), +, 0,
                   (shape(O)=pos(S,shapes(O)))*volume_assembly(S,Dims)).
```

4.2 Placement Relations

PKML uses Allen's interval relations [18] in one dimension, and the topological relations of the Region Connection Calculus [19] in higher-dimensions, to express placement constraints. These relations are predefined in libraries [14]. 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, 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 declaration for the volume of a bin, and 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
    minimize(labeling(Items), volume(Bin)).
```

Example 5. On the following simple shape pure bin packing problem

```
import(lib:pkml:pkml).
s1 = box([5,4,4]). s2 = box([4,4,2]). s3 = box([5,4,2]).
o1=object(s1,[0,0,0]). o2=object(s2,[_,_,_]). o3=object(s3,[_,_,_]).
```

```
dimensions = [1,2,3]. bins = [o1]. items = [o2, o3].
? variable_ordering([greatest(volume(^, dimensions)), is(z(^))]) and
  bin_packing(items, bins, dimensions).
```

the compiler generates the following SICStus-Prolog goal where the coordinate variables are statically ordered for labeling:

```
solve([O2,O2_2,O2_3,O3,O3_2,O3_3]) :-
  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,
  O2+4#=<O3#\/O3+5#=<O2#\/(O2_2+4#=<O3_2#\/O3_2+4#=<O2_2
          #\/(O2_3+2#=<O3_3#\/O3_3+2#=<O2_3)),
  labeling([], [O3_3,O3,O3_2,O2_3,O2,O2_2]).
```

4.3 Packing 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 [20]. 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 weight constraints in an admissible packing.

The complete PKML library including common sense rules dealing with the weight of objects and the surface contact of stacked items, is given in [14]. With these rules, Proposition 3 entails:

Proposition 4. *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.*

Business patterns can also 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 be expressed with non-guillotine or non-visibility constraints [20], however 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.

4.4 Performance Evaluation

We report here the performances of the Rules2CP compiler and of the generated constraint program, on Korf's benchmarks of optimal rectangle packing problems [7]. These problems consists in finding the smallest rectangle containing n squares of sizes $S_i = i$ for $1 \leq i \leq n$. In [21] Simonis and O'Sullivan proposed a constraint program implemented in SICStus Prolog which improved best known runtimes up to a factor of 300.

Their search strategy decomposes the optimisation problem in two subproblems. First, the different non symmetric bounding rectangle candidates are enumerated in ascending order of areas. Then, for each bounding rectangle candidate, the N squares packing satisfaction problem is solved with a search strategy based on interval splitting, working together with the disjoint2 and cumulative global constraints. The strategy places the N squares ordered by decreasing sizes. It first splits the domain of x coordinates into intervals, before fixing these coordinates by dichotomy. The process is then repeated for the y coordinates.

Table 1. Optimal Rectangle Packing programs runtimes in seconds

N	R2CP compilation	Rules2CP	Original
18	0.266	13	6
19	0.310	11	5
20	0.320	20	10
21	0.342	76	36
22	0.369	364	197
23	0.404	2076	1150
24	0.443	5230	1847
25	0.509	52909	17807

Table 1 compares the computation time in seconds obtained in Rules2CP with their original program in SICStus-Prolog . The SICStus-Prolog program generated from the Rules2CP model with dynamic search explores exactly the same search space and is slower by a factor less than 3, due to the interpretation overhead for the dynamic predicates. In all these examples, the compilation times are below one second.

5 Related Work

5.1 Comparison with OPL, Zinc and Essence

Rules2CP differs from OPL [1], Zinc [3,4] and Essence [5] modelling languages in several aspects, among which: the naming of rules, the restriction 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 scope. A simple module system is used in Rules2CP to avoid name clashes. The simplicity of these design choices is reflected in the obtainment of a complexity bound on the size of the constraint programs generated from Rules2CP models (Prop. 3). 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 use all the power of the language to define heuristic criteria. This is currently not expressible in Zinc and Essence, and can be achieved in OPL in aless declarative manner by programming. 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.

5.2 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 [22]. Rules2CP models can be compiled to CLP(FD) programs of potentially exponential size. 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.

5.3 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 optimisation 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.

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 [23] 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 $\rightarrow$ rules into TOM pattern matching expressions.

6 Conclusion

The Rules2CP language is a rule-based modelling language for constraint programming. It has been designed to allow application experts express knowledge, common sense and industrial requirements about combinatorial optimisation problems with rules (using appropriate editors). Rules2CP rules are declarative and can be easily introduced, checked and modified one by one, independently from their particular interpretation by a rule engine.

Search trees can also be specified declaratively in Rules2CP with logical formulae, and search heuristics can be defined as preference orderings on variables, values, conjunctive and disjunctive formulae, using pattern matching on rule names. This is in contrast with other modelling languages for which search strategies still need be programmed. We have shown that search strategies for scheduling can be easily expressed in Rules2CP in this manner, as well as the search strategies of Simonis and O'Sullivan [21] for solving Korf's optimal rectangle packing problem [7], with a constant overhead factor in the generated code.

The PKML library dedicated to bin packing and bin design problems used in these experiments, can deal in addition with extra requirements about weights, oversizes, equilibrium constraints, and specific packing business rules. Furthermore, a large subset of PKML has been shown in [16] to be efficiently compilable with indexicals within the geometrical kernel of the global constraint `geost`.

The transformation of Rules2CP models into constraint programs has been described here by a term rewriting system with partial evaluation. The confluence of these transformations has been shown, together with a complexity bound on the size of the generated program. The obtainment 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. This complexity bound shows however a potential exponential blow-up in the size of the generated constraints. In such cases, the dynamical expansion strategy can be used.

As for future work, several issues have not been discussed in this paper. Rules2CP is currently untyped. One difficulty in typing Rules2CP models lies in the coercions between expressions and formulae used in reification and involving a subtyping relation between booleans and integers [24]. More experiments are also needed to evaluate the module system of Rules2CP and its capability to develop libraries of models that can be reused in a hierarchy of models and for special purpose applications. Finally, the specification of search strategies in Rules2CP needs be explored more systematically, and could also be evaluated with adaptive strategies in which the dynamic criteria depend on execution profiling criteria.

Acknowledgements. This work is supported by the European FP7 Strep project Net-WMS. We gratefully acknowledge Helmut Simonis for providing us with his program, and all the partners of this project, as well as Thierry Martinez and Sylvain Soliman, for numerous discussions on this topic.

References

1. Van Hentenryck, P.: The OPL Optimization programming Language. MIT Press, Cambridge (1999)
2. Hentenryck, P.V., Perron, L., Puget, J.F.: Search and strategies in opl. ACM Transactions on Compututational Logic 1, 285–320 (2000)
3. Rafeh, R., de la Banda, M.G., Marriott, K., Wallace, M.: From Zinc to design model. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 215–229. Springer, Heidelberg (2006)
4. de la Banda, M.G., Marriott, K., Rafeh, R., Wallace, M.: The modelling language Zinc. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 700–705. Springer, Heidelberg (2006)
5. Frisch, A.M., Harvey, W., Jefferson, C., Martinez-Hernandez, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints 13, 268–306 (2008)
6. Group, B.R.: The business rules manifesto Business Rules Group (2003), `http://www.businessrulesgroup.org/brmanifesto.htm`
7. Korf, R.E.: Optimal rectangle packing: New results. In: ICAPS, pp. 142–149 (2004)
8. Haemmerlé, R., Fages, F.: Modules for prolog revisited. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 41–55. Springer, Heidelberg (2006)
9. Van Hentenryck, P.: Constraint satisfaction in Logic Programming. MIT Press, Cambridge (1989)
10. Huang, J., Darwiche, A.: The language of search. Journal of Artificial Intelligence Research 29, 191–219 (2007)
11. Apt, K., Wallace, M.: Constraint Logic Programming using Eclipse. Cambridge University Press, Cambridge (2006)
12. Carlsson, M., et al.: SICStus Prolog User's Manual. Swedish Institute of Computer Science, 4th edn. (2007), ISBN 91-630-3648-7

13. 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, 241–262 (2004)
14. Fages, F., Martin, J.: From rules to constraint programs with the Rules2CP modelling language. INRIA Research Report RR-6495, Institut National de Recherche en Informatique (2008)
15. Rosen, B.: Tree-manipulating systems and Church-Rosser theorems. Journal of the ACM 20, 160–187 (1973)
16. Carlsson, M., Beldiceanu, N., Martin, J.: A geometric constraint over k-dimensional objects and shapes subject to business rules. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 220–234. Springer, Heidelberg (2008)
17. 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.) CP 2007. LNCS, vol. 4741, pp. 180–194. Springer, Heidelberg (2007); SICS Technical Report T2007:08, `http://www.sics.se/libindex.html`
18. Allen, J.: Time and time again: The many ways to represent time. International Journal of Intelligent System 6 (1991)
19. 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 1992, pp. 165–176. Morgan Kaufmann, San Francisco (1992)
20. Carpenter, H., Dowsland, W.: Practical consideration of the pallet loading problem. Journal of the Operations Research Society 36, 489–497 (1985)
21. Simonis, H., O'Sullivan, B.: Using global constraints for rectangle packing. In: Proceedings of the first Workshop on Bin Packing and Placement Constraints BPPC 2008, associated to CPAIOR 2008 (2008)
22. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: Proceedings of the 14th ACM Symposium on Principles of Programming Languages, pp. 111–119. ACM, Munich (1987)
23. Balland, E., Brauner, P., Kopetz, R., Moreau, P.E., Reilles, A.: Tom: Piggybacking rewriting on java. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 36–47. Springer, Heidelberg (2007)
24. Fages, F., Coquery, E.: Typing constraint logic programs. Journal of Theory and Practice of Logic Programming 1, 751–777 (2001)