Net-WMS

SPECIFIC TARGETED RESEARCH OR INNOVATION PROJECT

Networked Businesses

D6.1: Specification of the knowledge modeller

Due date of deliverable: 15 January 2008

Actual submission date: 31 March 2008

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

Organisation name of lead contractor for this deliverable: KLS OPTIM

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

COVER AND CONTROL PAGE OF DOCUMENT	
Project Acronym:	Net-WMS
Project Full Name:	Towards integrating Virtual Reality and optimisation techniques in a new generation of Networked businesses in Warehouse Management Systems under constraints
Document id:	D6.1
Document name:	Specification of the knowledge modeller
Document type (PU, INT, RE, CO)	PU
Version:	Final
Submission date:	31 March 2008
Authors: Organisation: Email:	

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

ABSTRACT :
This deliverable defines the Net-WMS packing knowledge modeller: first the architecture of packing components, then the rule-based Packing Knowledge Modelling Language PKML, its compilation to constraint programs over finite domains, and its very efficient integration into the geometric constraint *geost*. We take the opportunity of this deliverable to produce a glossary of packing terminology.

KEYWORD LIST :
knowledge modelling, rule-based languages, constraint programming, business rules, bin packing, global constraints

MODIFICATION CONTROL			
Version	Date	Status	Author
0	24-02-2008	Draft	F. Fages
1	20-03-2008	Reviewed version	A. Aggoun
2	27-03-2008	Final version	F. Fages

Deliverable manager

- Abder Aggoun, KLS OPTIM

List of Contributors

- Abder Aggoun, KLS OPTIM
- Nicolas Beldiceanu, EMN
- Mats Carlsson, SICS
- Camille Chigot, CEA
- François Fages, INRIA
- Philippe Gravez, CEA
- Julien Martin, INRIA

List of Evaluators

- Nicolas Beldiceanu, EMN
- Olivier Gourguechon, PSA

Contents

Chapter 1

Introduction to the Net-WMS Knowledge Modeller

The scientific challenge addressed in the project is the integration of Constraint Programming (CP) and a Rule Based (RB) language in order to achieve an expressive and efficient knowledge modelling language for stating knowledge in Warehouse Management Systems (WMS) as packing rules (c.f. Figure 1.1). Rules are expressed by experts to define constraints over objects to pack in containers (e.g. a pallet, maritime box). In order to ease the understanding of the approach chosen in the Net-WMS project we will detail step by step the key points and concepts developed in the project. When necessary, concepts are illustrated using knowledge examples taken from packing problems.

CP is a programming language used by experts in optimisation to encode models to solve combinatorial problems. It is not dedicated for non-experts in optimisation and less for experts in WMS.

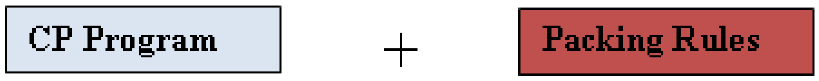


Figure 1.1: Integration of CP and RB

In a rule based language, rules are independent from each other. Rules can be checked and modified independently and then introduced in the knowledge database one by one. The modelling language for expressing packing rules must respect these features and concepts of RB languages. The investigated approach is to allow the user to express the expertise in terms of rules which are then transformed into CP Program as highlighted in Figure 1.2.

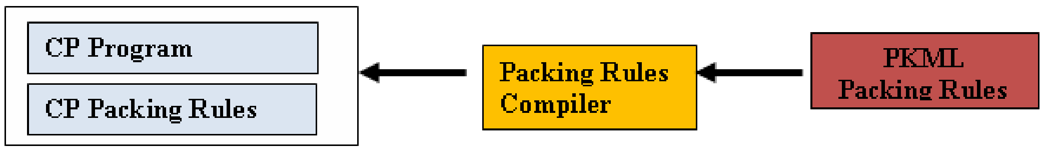


Figure 1.2: Transformation of rules into CP programs

The aim of the project is to design a knowledge modelling language for non-programmers to express packing rules to be compiled into CP programs. **The language is called PKML and stands for Packing Knowledge Modelling Library**. The main objective is to focus first on packing problems. However, concepts chosen in the specifications make PKML open to other domains than packing problems. It is a major advantage to integrate optimisation and especially CP techniques in other domains.

So far in the Net-WMS project, CP systems (SICStus Prolog, Choco-Java) have been enriched with a new global constraint called Geost [c.f. deliverable D4.1]. It allows expressing geometrical constraints for non-overlapping objects in k dimensions. For example in pallet packing, it is used to model non-overlapping between boxes (3D) on a pallet. The availability of this constraint has contributed a lot to compiling packing rules into efficient CP programs.

In the test cases [c.f. deliverable D2.2] we have identified a set of classes of packing constraints. For example, if the type attribute of two objects both equal 1 then the two objects should not meet (c.f. Figure 1.3).

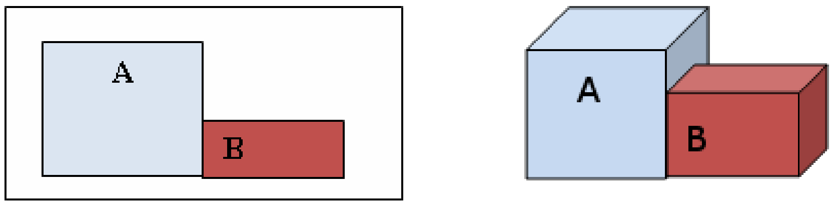


Figure 1.3: meet(A,B) constraint.

Constraints like meet(A,B), covers(A,B) etc. are classified as side constraints. Global constraints have traditionally been extended with new arguments to handle each such class of side constraints From the implementation point of view, each side constraint will require its specific handling algorithm. This approach will lead to a poor way of expressing constraints and combining them. This approach will not contribute efficiently to the compilation of packing rules into CP programs. Contrary to this, the approach chosen in Net-WMS is the development of a generic rule language in the geost constraint. With this generic rule language, a wide range of side constraints may be expressed. A CP system extended by the Geost constraint with the Geost rule language can be seen as a natural target language for the transformation of the packing rules into CP programs. The approach proposed is that the integrated rule language features are derived from features of the packing rule language. It can be seen as a subset of the packing rule language (see Figure 1.4).

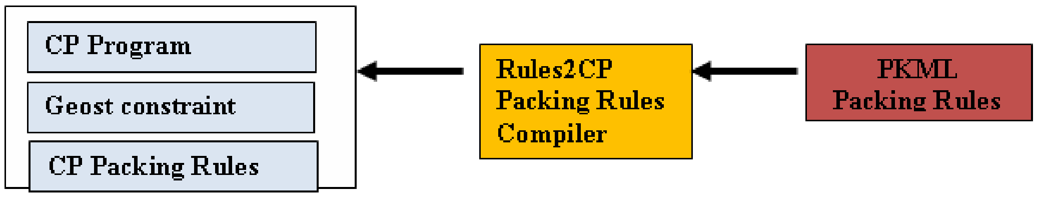


Figure 1.4: Transformation of rules into CP programs

The approach chosen is motivated by the fact that it will ease the transformation of packing rules into an efficient CP program. In fact, **the compilation of rules will use robust and well known rewriting techniques** which are described below in the report. **The system which transforms PKML rules into CP programs is called Rules2CP** (from Rules to Constraint Programming).

Another advantage of this approach is that the CP system is enriched by two developments, the Geost constraint and the integrated rule language, both of which are major contributions. CP developers will have the possibility to use the Geost rule language to develop efficient packing applications independently from the packing rule language.

PKML is a language containing data structures, a limited number of types, arithmetic operators and logical expressions with quantifiers "forall", "exists", etc. Experts in packing having a good background in computer science will be able to use the PKML language directly. As the target of the project is end-users that are non-experts in computer-science however, dedicated graphical interfaces for editing PKML rules will be developed, for instance with spreadsheets or standard business rules editors (Figure 1.5).

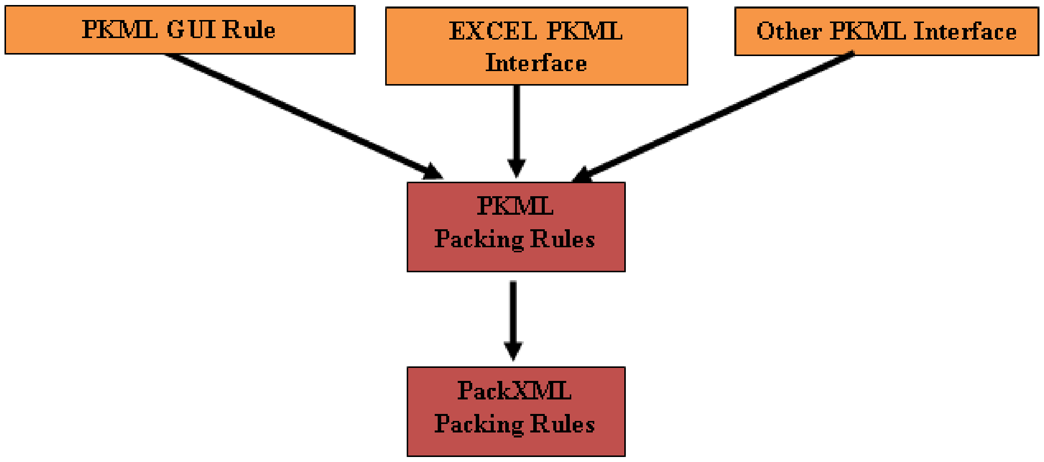


Figure 1.5: Non-expert users' interfaces

The graphical interface rule (GUI Rule) for PKML is a guided editor for expressing packing rules; it allows the packing expert to build rules in a GUI driven fashion using wizard menus based on PKML entities.

In order to ease the editing of the expertise, each rule is mapped into XML structures. Therefore, the user can load an existing rule, do changes and then save it. As explained above rules are independent from each other. The XML format will ease the integration into business applications and complex architectures. The result is a specification called PackXML for packing problems. This is a major contribution of Net-WMS for contributing to standards in logistics.

The report is organised in several and independent sections:

Chapter 2 contains a glossary of words and explanations to ease the understanding of the report.

Chapter 3 introduces the general architecture;

Chapter 4 details the PKML language as an instance of the general rule-based modelling language Rules2CP;

Chapter 5 describes some additional features of PKML for virtual reality components;

Chapter 6 is dedicated to the subset of PKML integrated in the geometric global constraint Geost.

Chapter 2

Glossary of Packing Terminology

The aim of the section is to give a definition of key words as used in logistics in order to ease the understanding of the technology developed within Net-WMS. A set of technical words used in logistics are well defined in the encyclopaedia `http://en.wikipedia.org`. When possible we will use definitions from this encyclopaedia. When necessary some comments are added to ease the understanding in the context of the Net-WMS project.

Pallet: Normally found as a wooden platform without sides, which is used for stacking a number of packages on in preparation for loading and distribution to their destination. It is common to find that the pallets have space underneath so that they can be lifted by mechanical equipment such as forklift trucks.

Remark: Pallets are commonly used in distribution warehouses. In car manufacturing, one can find several types of pallets. Therefore we distinguish non standard and standard pallets like ISO Pallets, North American Pallets and European pallets. Pallet users want pallets to easily pass through buildings, stack and fit in racks, forklifts, pallet jacks and automated warehouses. To avoid shipping air, pallets should also pack tightly inside containers and vans.

Container: A trailer body that can easily be detached from a truck chassis for loading onto a vessel, a rail car, or stacked in a container depot. A container may be varied in length and width. Containers may be ventilated, refrigerated, insulated, vehicle rack, flat rack, open top, bulk liquid or equipped with interior devices.

Packaging is the science, art and technology of enclosing or protecting products for distribution, storage, sale, and use. Packaging also refers to the process of design, evaluation, and production of packages. Package labelling or labelling is any written, electronic, or graphic communications on the packaging or on a separate but associated label. Packaging may be looked at as several different types. For example a transport package or distribution package is the package form used to ship, store, and handle the product or inner packages. Some identify a consumer package as one which is directed toward a consumer or household.

Remark: It is sometimes convenient to categorize packages by layer or function: ”primary”, ”secondary”, etc.

Primary packaging is the material that first envelops the product and holds it. This usually is the smallest unit of distribution or use and is the package which is in direct contact with the contents.

Secondary packaging is outside the primary packaging perhaps used to group primary packages together.

Tertiary packaging is used for bulk handling, warehouse and transport shipping. The most common form is a palletized unit load that packs tightly into containers.

PKML is the knowledge modelling language developed in Net-WMS to express packing business rules and problems. PKML models are compiled into constraint programs and in the Geost constraint using the Rules2CP transformation scheme.

A **unit load** combines packages or items into a single ”unit” of a few thousand kilograms that can be moved easily with simple equipment (e.g. pallet jack). A unit load packs tightly into warehouse racks, containers, trucks, and railcars, yet can be easily broken apart at a distribution point, usually a distribution centre, retail store, etc. Most consumer and industrial products move through the supply chain in unitized or unit load form for at least part of their distribution cycle. Unit loads make handling, storage, and distribution more efficient. They help reduce handling costs and damage by reducing individual handling. A typical unit load might consist of corrugated fibreboard boxes stacked on a pallet and stabilized with stretch wraps or other materials.

Remark: Unit loads, pallets and containers are the most commonly used packing units in Net-WMS.

A **box** describes a variety of containers and receptacles. When no specific shape is described, a typical rectangular box may be expected. Nevertheless, a box may have a horizontal cross section that is square, elongated, round or oval; sloped or domed top surfaces, or non-vertical sides. Whatever its shape or purpose or the material of which it is fashioned, it is the direct descendant of the chest, one of the most ancient articles of domestic furniture. Its uses are innumerable, and the name, preceded by a qualifying adjective, has been given too many objects of artistic or antiquarian interest. Objects are often placed inside boxes, for a variety of reasons.

Remark: Boxes are key units in packing in Net-WMS. Complex objects are modelled as assemblies of other objects where their corresponding shapes are approximated by boxes.

Packing problems are one area where mathematics meets puzzles. Many of these problems stem from real-life packing problems. In a packing problem, you are given: one or more (usually two- or three-dimensional) containers; several goods, some or all of which must be packed into this container.

Usually the packing must be without gaps or overlaps, but in some packing problems the overlapping (of goods with each other and/or with the boundary of the container) is allowed but should be minimised. In others, gaps are allowed, but overlaps are not (usually the total area of gaps has to be minimised).

A **puzzle** is a problem that challenges ingenuity. In a basic puzzle you piece together objects in a logical way in order to come up with the desired shape, picture or solution. Puzzles are often contrived as a form of entertainment, but they can also stem from serious mathematical or logistical problems. In such cases, their successful resolution can be a significant contribution to mathematical research (refer to bin packing problems).

Remark: Packing problems are essential for logistical problems and especially for transport. In optimisation many variants of packing problems are treated as bin packing problems; a very active research area.

Business rules describe and control the structure, operation and strategy of an organisation. Business rules are separated from the application code and mainly the engine. Business rules can be changed independently. Modern engine business rules combine the power of rule-based programming and object-oriented programming.

Geost constraint is a new global constraint developed in the context of the Net-WMS project. It allows expressing geometrical constraints for non-overlapping objects in k dimensions. For example, in pallet packing it can be used to model non-overlapping boxes (3D) in a pallet.

Geost rules are additional rules which are compiled and handled as part of the Geost constraint which is extended with a limited macro language to define such additional rules on the non-overlapping objects.

Packing rules are rules describing packing expertise. They are compiled in a target CP language.

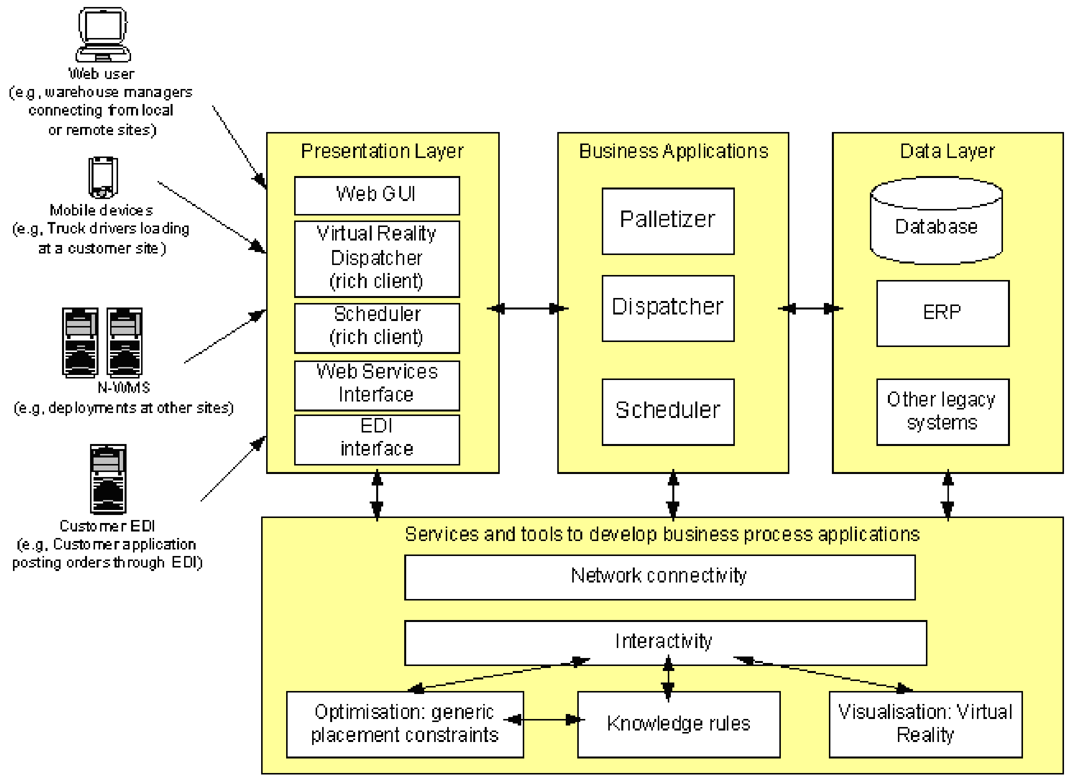
PackXML is the XML Schema to express packing rules.

Rules2CP is the general rule-based knowledge modelling language which transforms rules into constraint programs and in which the dedicated library PKML is written.

Chapter 3

Architecture of Packing Components

The general architecture proposed in Net-WMS is shown in Figure 3.1. In this section we will highlight the packing modules. Concepts of the knowledge modeller and especially of the packing knowledge modelling language are detailed in the other sections.



Web user
(e.g, warehouse managers connecting from local or remote sites)
Mobile devices
(e.g. Truck drivers loading at a customer site)
N-WMS
(e.g. deployments at other sites)
Customer EDI
(e.g. Customer application posting orders through EDI)
Presentation Layer
Web GUI
Virtual Reality Dispatcher (rich client)
Scheduler (rich client)
Web Services Interface
EDI interface
Business Applications
Palletizer
Dispatcher
Scheduler
Data Layer
Database
ERP
Other legacy systems
Services and tools to develop business process applications
Network connectivity
Interactivity
Optimisation: generic placement constraints
Knowledge rules
Visualisation: Virtual Reality

Figure 3.1: Innovative Net-WMS J2EE architecture

Data layer: The data layer will include the systems own database, integration with ERP databases or other legacy/standard systems.

Business layer: The business layer is a middleware set of components that interact among themselves. The components encapsulate the optimisation knowledge and complexity of palletising, dispatching and scheduling.

Services and tools to develop business process applications: The services and tools described illustrate the foundations of the three business applications proposed (Palletizer, Dispatcher and Scheduler), and could alternatively be seen also as middleware Business Applications.

The word “Palletizer” as described in Figure 3.1 is too restrictive in packing. We will use instead words like *business packing component* or *packing module*; which is a complete process of packing including different functionalities: creating and editing knowledge rules, solving optimisation packing problems, visualisation of packing results. In order to highlight the exploitation of the modelling language, it is necessary to detail the different components of a business packing component. By zooming in Figure 3.1, the module Knowledge rules corresponds to Figure 1.4 and Figure 1.5.

3.1 Exploitation

Figure 3.2 shows the exploitation architecture of the packing module. Orders or nomenclatures to pack are handled in the WMS module. The packing module is called several times to perform the packing activities. The manager selects orders and launches the packing solver.

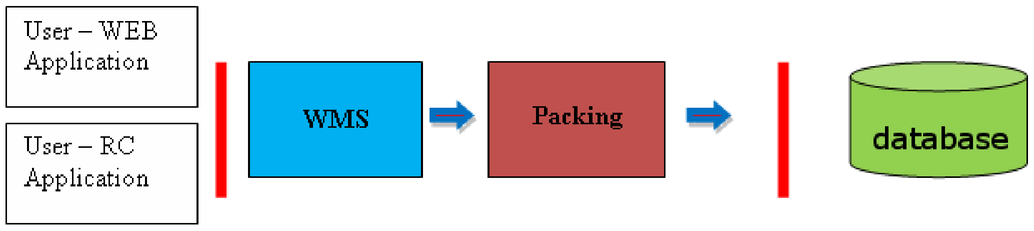


Figure 3.2: General architecture exploitation

The business packing component is structured into components:

Packing container: the logic of packing. This is the main entry to drive the different components of the packing module.

Packing solver: the optimisation; this components takes well defined inputs and produces results which are then processed by the packing container.

Packing Player: 3D visualisation of containers and items of the containers. The packing player is expected to run in Applet in a web application.

Packing Designer: this module allows the user to design packing modules. The user can edit a configuration, fix items and launch the solver to complete packing (see Figure 3.3).

3.2 Player

The player is a plug-in which must comply with the following requirements summarized in Figure 3.4:

- Case 1: the player is installed as rich client application
  - Input: XML files or Data Base

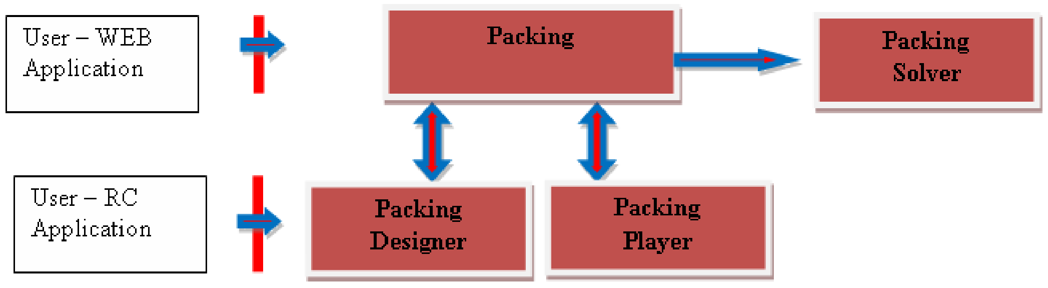


Figure 3.3: General architecture of Business packing component

- Visualisation of the loads (bins):
  - ∗ Bin data (the container)
  - ∗ Items (content)
  - ∗ All specific information for bins and items are displayed provided as properties

- Case 2: the player is called from a web application for end users
  - Input: Data Base
  - Visualisation of the loads (bins)

- Case 3: the player is called from a web application by the packing solver. The user is performing simulation.
  - Input: XML files or Data base
  - Visualisation of the loads

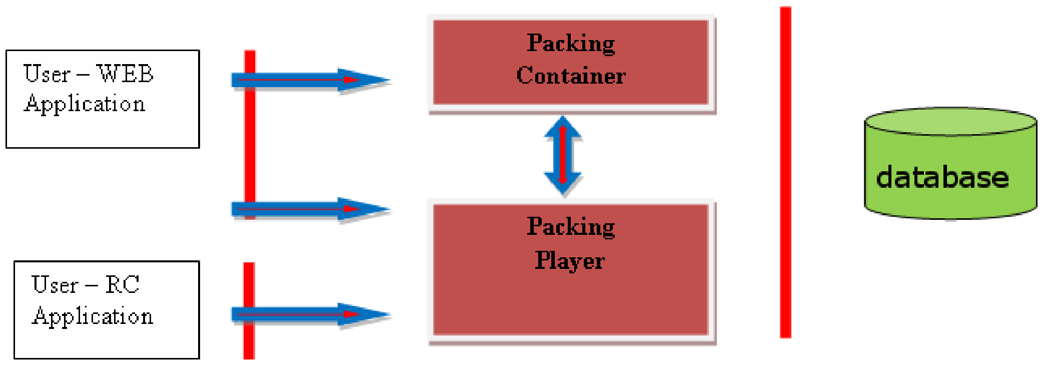


Figure 3.4: General architecture of Packing player component

3.3 Design

Figure 3.5 depicts the general architecture of the plug-in components packing solver.

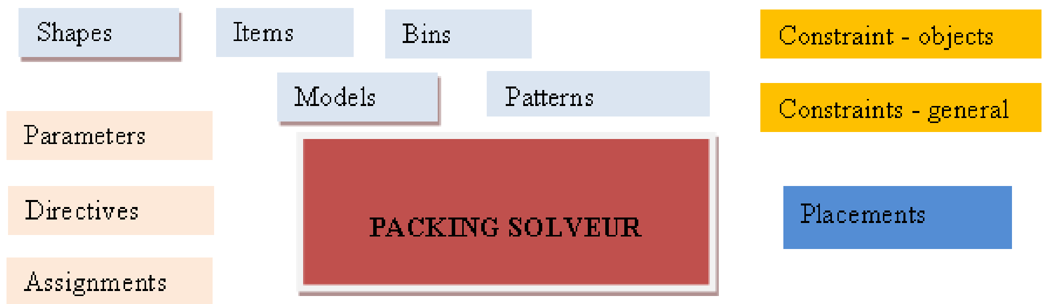


Figure 3.5: General architecture of the plug-in components Packing Solver

3.4 Packing solver

We distinguish two classes of solvers:

1. the business packing solver
2. and the optimisation packing solver.

The business packing solver can be seen as a companion of a warehouse management system. It is either integrated in the application as a standalone application or as an integrated module in a WMS. For its exploitation, we need to distinguish three phases:

1. it reads the data
2. it solves the packing problem by calling one or several time the optimisation packing solver.
3. it produces packing results.

The optimisation packing solver is referred here as PKML solver (Figure 3.6).

The Net-WMS expects to provide two possible implementations of the PKML solver. A first implementation in constraint logic programming based on SICStus Prolog, used for rapid prototyping and for accessing already implemented powerful global constraints. A second implementation in a Java platform based on Choco, for easier integration with the other Net-WMS components and for their industrial deployment.

If the PKML solver is designed as a black box then its exploitation will be limited to well-known scenarios. Therefore, a glass box is suited to cope with various and complex scenarios in today's logistics. This is mainly motivated by the following cases:

- The expression of PKML strategies is limited, dedicated strategies are suited to come up with quality solution.
- The existing constraints in PKML Solver are not sufficient to handle the packing problem of the user. Logistics evolve and we may need extra constraints.

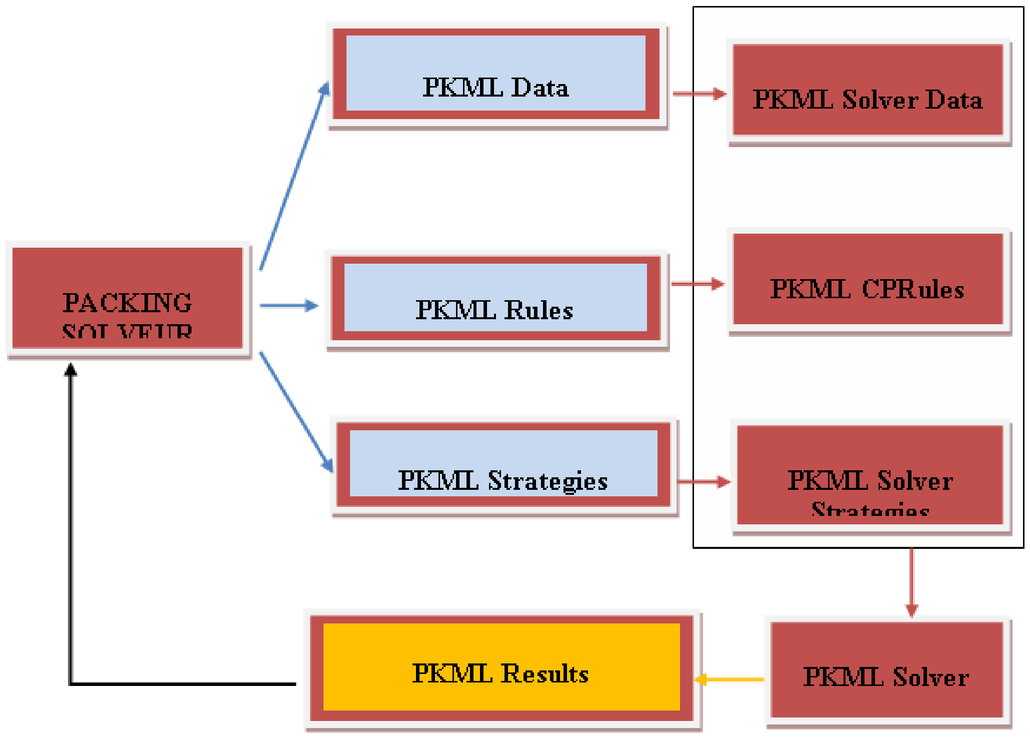


Figure 3.6: General architecture of the interaction between the packing solver and PKML solver

KLS OPTIM is mainly interested by the result of transforming PKML Rules into PKML CPRules. The result is a Java class which can be easily integrated in its business packing solver.

Remark: As explained above, the optimisation solver can be called several hundred times to solve parts of the problems. The business packing module is responsible of the management of the optimisation solver. When solving packing problems with nomenclatures (carton in pallets, pallets in containers, etc.), rules are changing from one instance to another one.

Scenario 1: The packing solver generates a complete PKML instance (data, rules, strategies). The optimisation solver is called with the right parameters and computes the best solution which is then processed by the business packing (saving in database, etc.).

Scenario 2: The packing solver generates a partial PKML instance (data, rules) but not the strategies. The compiler transforms the PKML rules into a CP module which is then integrated for example in the KLS OPTIM PKML Solver; which in turn solves the problem and processes the obtained results.

Scenario 3: The packing solver generates a partial PKML instance (partial data, rules) and not the strategies. By partial data, the business packing solver allows the PKML solver to share its objects (bins, objects to pack). The PKML compiles the rules and produces a CP module which is then integrated for example in the KLS OPTIM PKML Solver; which in turn solves the problem and processes the obtained results. For example, this is a typical case when designing new packing configuration to create patterns.

This is a typical packing in distribution platform. The objective is to call the packing solver thousands of times with a complete transparency to the user, see Figures 3.7 and 3.8.

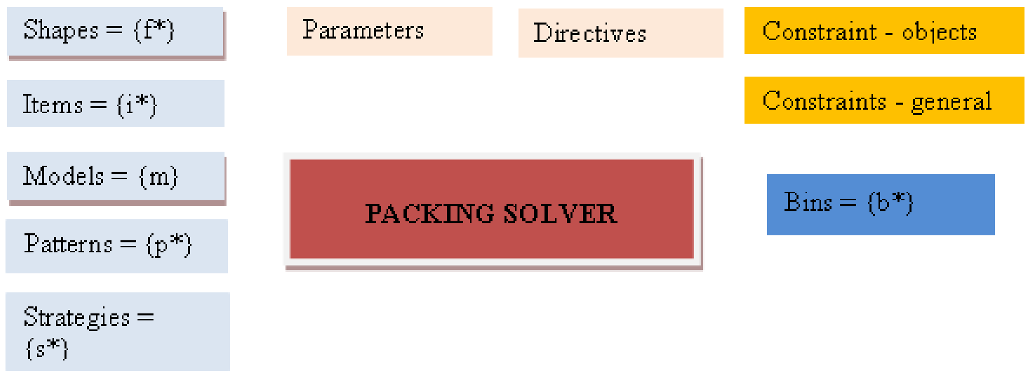


Figure 3.7: A single step of the Packing Solver

3.5 Java implementation

The first prototype of the modelling language will be developed in SICStus Prolog. The rewriting engine of Rules2CP requires reification techniques to compile rules into a target language. For "PKML CPRules" the result could be one of the following forms:

1. A Java program to compile;
2. An XML file to read and to transform into Java objects (instances of Java classes);
3. Instances of Java classes (compilation on the fly).

One of the potential tools to use to compile rules is TOM (http://tom.loria.fr).

- Tom is a software environment for defining transformations in Java.
- Tom is an extension of Java designed to manipulate tree structures and XML documents.
- Tom is compatible with Java: any Java program is a correct Tom program.
- Data are represented using an efficient object oriented tree based data-structure.
- Java built-ins (int, char, String, etc) can be used.
- Tom provides pattern matching facilities to inspect objects and retrieve values.
- A powerful strategy language can be used to control transformations.
- Tom is used in several companies to implement transformations of programs and queries.

The software is available as an Eclipse plug-in and released under the GPL General Public License[1], and the BSD license[2]. Tom is a potential candidate for the implementation of the engine of Rules2CP in Java.

[1]`http://www.gnu.org/copyleft/gpl.html`
[2]`http://www.opensource.org/licenses/bsd-license.php`

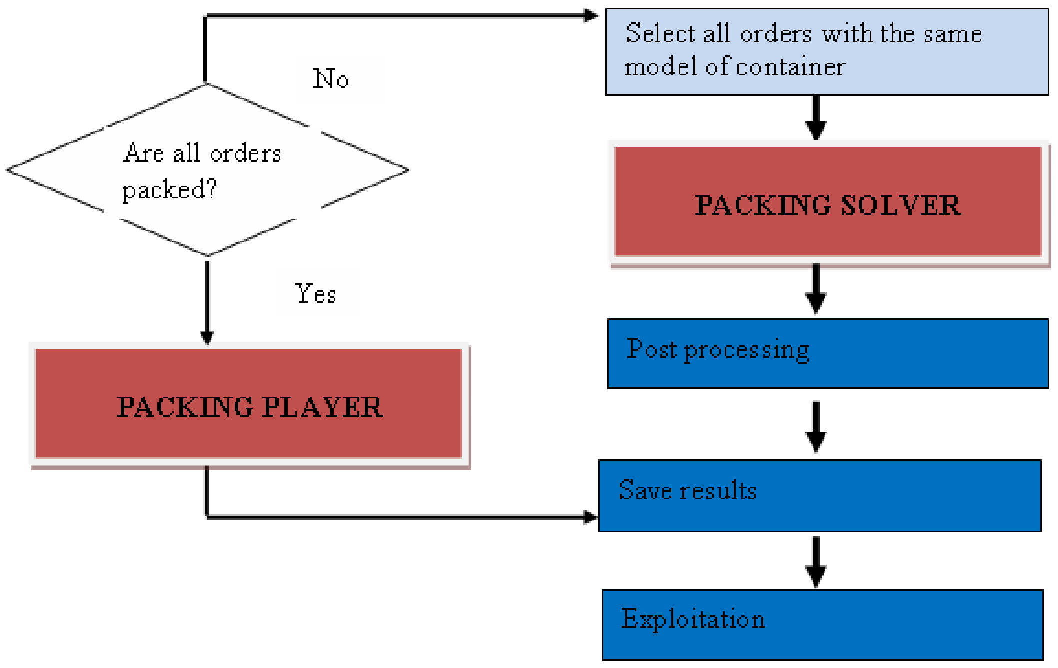


Figure 3.8: Multiple steps of the Packing Solver

Chapter 4

The Packing Knowledge Modelling Language PKML

4.1 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, a constraint program is not as declarative as one would wish, and constraint programming languages are in fact very difficult to use by non-expert users outside the range of already treated example problems. 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 [33] and Zinc [27, 12]. In these languages, a problem is modelled with variables, arrays, primitive constraints, set constructs, iterators and quantifiers. Such a problem model can then be mapped to a constraint program, a mixed integer linear program [25], a combination of both [33], or a local search program [34] for solving it. Despite the undeniable progress towards more declarativity and independence from the solving techniques achieved by these front-end modelling languages, one cannot nevertheless say that this is sufficient to make constraint programming easy to use by non-experts in the industry.

In the industry, 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 [17]. This provides an attractive knowledge representation scheme in the industry for fastly evolving regulations and constraints, and for maintaining systems with up to date information.

In this chapter, we argue that the business rules knowledge representation paradigm can be developped as a front-end modelling language for constraint programming. We present the PKML modelling language for packing problems, as a library of a general purpose rule-based modelling language for constraint programming, called Rules2CP. PKML and Rules2CP rules are not general condition-action rules, also called production rules in the expert system community, but 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 [17], and in particular to the independence from procedural interpretation which is concretely demonstrated in Rules2CP by their compilation to constraint programs using a completely different representation, instead of their execution by a rule engine. As a consequence, the rule language proposed in this chapter comes with a simple semantics in classical first-order logic, instead of the default logics usually considered in the rule-based knowledge representation community [35, 14].

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 feature terms (records) and finite collections (enumerated lists) with quantifiers and aggregates as iterators. In the next section, we present the syntax of Rules2CP, its predefined functions and predefined predicates for specifying search strategies in a declarative manner, and illustrate the main language constructs with simple examples of combinatorial and scheduling problems.

Then in Sec. 4.2, we present how Rules2CP models compile into constraint programs over finite domains with reified constraints, using a term rewriting system and partial evaluation. We show the confluence of these transformations and provide a complexity bound on the size of the generated constraint program.

Then in Sec. 4.4, we illustrate the expressive power and efficiency of this approach with a particular Rules2CP library, called the Packing Knowledge Modelling Library (PKML), developed 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 packing business rules taking into consideration, for instance, stacking constraints w.r.t. the weight of objects, their difference of size, and other common sense requirements or industrial expertise. Furthermore, we show how a large subset of PKML rules can be directly compiled into the geometric global constraint `geost` [4] and its integrated rule language which is essentially a subset of PKML, as shown in Chapter 6.

Finally in Sec. 4.5 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 approach for rule-based knowledge modelling as a front-end to constraint programming.

4.2 The Rules2CP Language

4.2.1 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 [18].

The syntax of Rules2CP is given in Table 4.2.1. 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 a 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 not allowed. In a rule, L-->R, we assume $V(R) \subseteq V(L)$, whereas in a declaration, H=E, the variables in $V(E) \setminus V(H)$ represent the

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
	|	`{`*name* = *expr,...,name*= *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	::=	+ | − | ∗ | / | `min` | `max`	
relop	::=	< | =< | = | /= | >= | >	
logop	::=	`and` | `or` | `implies` | `equiv` | `xor`	
name	::=	*ident*	
	|	*name* `:` *ident*	module prefix

Table 4.1: Syntax of Rules2CP.

unknowns of the problem.

An expression *expr* can be a *fol* formula considered as a $0/1$ integer. This coercion between booleans and integers is called *reification* and provides a great expressiveness. 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 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).`

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 both 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, 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*).

4.2.2 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. *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 [22]. 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.

Optimization predicates are also defined as usual:

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

but 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.

Adding the capability to express heuristic search knowledge in Rules2CP is mandatory for efficiency. This is done by adding options to the `labeling` predicate for specifying the variable choice heuristics as well as the value choice heuristics. Options that are standard in constraint programming systems like [2, 9], have been adapted to the Rules2CP labeling predicate, namely for the variable choice heuristics: `leftmost`, smallest lower bound `min`, greatest upper bound `max`, smallest domain `ff`, most constrained `ffc` options ; and `step, enum, bisect` for the value choice heurisitcs. This is a first step toward a more general modelling of heuristic knowledge using all the power of Rules2CP rules for defining heuristic choice functions for the `search` predicate as well.

4.2.3 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([bisect,up],B)).
? solve(8).
```

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

The solve *rule 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 rule*

```
solve2 --> 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 rule

```
solve3 --> 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. □

4.3 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. The term rewriting relation of the compilation process is denoted by $\rightarrow_{csp}$. It is worth noticing that for user-interaction at runtime and debugging purposes, bookkeeping information needs to be implemented in this transformation in order to maintain the dependency from CP variables back to Rules2CP statements [15]. This is described in the following.

4.3.1 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 and 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 variables in 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 is supposed 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 $q(i)$ of the head[1]. 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[2]:

- `[`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 $N = 0$)

[1]In this example, the unique names given to the row variables are `Q_i_1` as they are the first anonymous variables in the records, which is simplified into `Q_i` as they are the only variables in the records.

[2]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*.

- `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`)` $\rightarrow_{csp}$ $\phi[X/e]$

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` is rewritten with `xor`. It is worth noting that these transformations do not increase the *size* of the formula.

4.3.2 Inlining 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 tanslated 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. The correspondance between the target language constraints and Rules2CP rules can be implemented similarly by keeping track of the Rules2CP rules that generate constraints of the target language by inlining rules.

The examples of inlining rules given in this section are for the compilation of Rules2CP to SICStus-Prolog [9]. Basic constraints are 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 $\rightarrow_{csp}$ `"`$\grave{}A$ `#>` $\grave{}B$`"`
- A `and` B $\rightarrow_{csp}$ `"`$\grave{}A$ `#/\` $\grave{}B$`"`
- `lexicographic(`L`)` $\rightarrow_{csp}$ `"lex_chain(`$\grave{}L$`)"`

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

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:*

```
1 #=< Q_1 #/\ Q_1 #=< 4 #/\
1 #=< Q_2 #/\ Q_2 #=< 4 #/\
1 #=< Q_3 #/\ Q_3 #=< 4 #/\
1 #=< Q_4 #/\ Q_4 #=< 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([bisect,up],[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.* □

4.3.3 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 [29].*

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 trivially 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 for quantifiers and aggregation, 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. Therefore the associated $\rightarrow_{csp}$ rule is left-linear w.r.t. pattern matching. The term rewriting system $\rightarrow_{csp}$ is thus orthogonal. □

It is worth noticing that the preceding proof does not assume termination[3]. 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 there is 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 its right hand side.*

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 [24], this entails that the $\rightarrow_{csp}$ term rewriting system is Noetherian [31].* □

Termination proofs by multiset path ordering imply primitive recursive derivation lengths [21]. 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, the* aggregate rank $\alpha(s)$ *of a symbol s is defined inductively by:*

[3]When termination is assumed, the non-overlapping condition, or more generally the confluence of critical pairs [31], suffices to prove confluence without left-linearity.

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

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

There is no complexity bound on the size of the generated program if the model contains list constructions of the form $[M..N]$, where M and N can be the result of arbitrary arithmetic calculations. Apart from this case however, the size of the generated program can be bounded as follows:

Proposition 3 *For any* `Rules2CP` *model M with a goal of size one, and containing no list constructor of the form* `[varint..varint]`*, the size of the generated program is less than or equal to $l^a * b^r$, where l is the maximum length of the lists occurring in M, 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 bounded by r duplications of rule bodies, i.e. by 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 bounded by $l^{a-1} * b^r$. Now, the generated program can be duplicated at most l times by the outermost aggregate operators, hence the total size is bounded by $l^a * b^r$ under this strategy. By confluence Prop. 1, the generated program is independent of the strategy, hence the size of the generated program is bounded by $l^a * b^r$ under any strategy. □

Corollary 1 *The time complexity for compiling a Rules2CP model is in $O(l^a * b^r)$.*

In the N-queens problem with a board of fixed size l, like in Example 1 when replacing the variable size board declaration by an explicit list `[q(1),...,q(l)]`, the aggregate rank is 2. The previous proposition tells us that the size of the generated program is in $O(l^2)$.

4.4 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.

4.4.1 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.

PKML shapes are records containing a shape attribute, plus possibly other attributes for virtual reality representation, weight, etc. The possible values for the shape attribute, and the complementary attributes, are the following ones:

```
s1={shape=box, size=[size1,size2,...,sizeK]}.
s2={shape=polytope, points=[p1,p2,...,pN]}.
s3={shape=assembly, objects=[o1,o2,...,oN])}.
s4={shape=alternative, shapes=[s1,s2,...,sN])}.
```

The first form specifies orthotopes in $\mathbb{Z}^K$, i.e. rectangular boxes with their sizes given in the different dimensions. The second form specifies a K-dimensional convex polytope defined as the convex hull of a given list of N points. Note that a box is a particular case of convex polytope. The third form specifies a rigid assembly of objects, i.e. shapes given relative position, as detailed in next section. The fourth form gives a shape name to a set of alternative shapes. This is used for instance to represent the different shapes obtained by rotating a shape around the different dimension axes, or according to a finite set of angles, or in a configuration problem for expressing the choice between different objects. By considering assemblies of convex polytopes, one can easily see that any polytope in $\mathbb{Z}^K$ can be represented as a PKML shape.

Shapes come with some predefined functions and relations in PKML. For example, for box shapes, the followings declarations with their size in a given dimension and their volume:

```
size(S,D)=nth(D,size(S)).
volume(S,Dim)= aggregate(D,Dim,*,1,size(S,D)).
```

A PKML *object*, such as a bin or an item, is a record containing a shape name attribute `sid`, an *origin* point, plus some optional attributes such as weight, virtual reality representations or others.

```
o1={sid=name, origin=[x1,...,xK]}
o2={sid=name, origin=[y1,...,yK], weight=30, ...}
```

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, and its end if it has a box shape, plus some obvious rules for weights, are predefined in PKML by:

```
origin(O,D) = nth(D, origin(O)).
length(S,D) = nth(D, size(S)).
end(O,D) = origin(O, D) + length(sid(O), D).
lighter(O1, O2) --> weight(O1) < weight(O2).
heavier(O1, O2) --> weight(O1) > weight(O2).
```

4.4.2 Placement Relations

PKML uses Allen's interval relations [1] in one dimension, and the topological relations of the Region Connection Calculus [28] in higher-dimensions, to express placement constraints. These relations are predefined in the libraries given in Appendices A and B respectively. They are used in PKML to define packing rules for pure bin packing and pure bin design problems, as well as specific packing business rules for non pure problems taking into account other common sense requirements or industrial expertise.

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

```
non_overlapping(Items, Dims) -->
   forall(O1, Items,
      forall(O2, Items,
         iid(O1) < iid(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).

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

volume(O, Dims) =
   product( map(D, Dims, size(O, D)) ).

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)).
```

The rules define respectively the non-overlapping of a list of items in a list of dimensions, the containment of all items in bins, pure bin packing problems, and then for bin design, the volume of a bin, the containment in some bin of all items, and pure bin design problems. 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.

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

```
s1 = {shape=box, size=[5,4,4]}.
s2 = {shape=box, size=[5,4,2]}.
s3 = {shape=box, size=[4,4,2]}.
o1 = {oid=1, sid=s1, origin=[0,0,0]}.
o2 = {oid=2, sid=s2, origin=[_,_,_]}.
o3 = {oid=3, sid=s3, origin=[_,_,_]}.
dimensions = [1,2,3].
bins = [o1].
items = [o2,o3].
? binpacking(items, bins, dimensions).
```

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

```
0#=<O2,
O2+5#=<5,
0#=<O2_2,
O2_2+4#=<4,
0#=<O2_3,
O2_3+2#=<4,
0#=<O3,
O3+4#=<5,
0#=<O3_2,
O3_2+4#=<4,
0#=<O3_3,
O3_3+2#=<4,
O2+5#=<O3#\/O3+4#=<O2#\/
   (O2_2+4#=<O3_2#\/O3_2+4#=<O2_2#\/
      (O2_3+2#=<O3_3#\/O3_3+2#=<O2_3)),
labeling([bisect,up],[O2,O2_2,O2_3,O3,O3_2,O3_3]).
```

□

4.4.3 Packing Business Rules and Patterns

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. For instance the following rules about weights

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

weight_stacking(Items) -->
   forall(O1,  Items,
      forall(O2, Items,
         (iid(O1) # iid(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. Such packing business rules can be simply added to PKML models. These rules about weights are predefined (see Appendix C).

Furthermore, *business patterns* can be used in PKML to express knowledge about some predefined 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. In PKML, packing patterns are records containing a list of item shapes given with the coordinates of their origin:

```
pat1={sids=[s1,...,sN], origins=[p1,...,pN]}.

pattern(Items, Bin, Patterns)
-->
```

```
exists(P, Patterns,
 forall(S, sids(P), let(J, pos(S,sids(P),
  exists(I, Items,
   S=sid(I) 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 trying to apply a pattern, and completing the packing with the general `bin_packing` rule as follows:

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

4.4.4 Compilation to the Global Constraint `geost`

The constraint `geost` [4] is a generic global constraint for higher-dimensional placement problems which is now parameterized by a rule language described in Chapter 6. A subset of PKML rules can be directly transformed into `geost` rules providing a very high level of pruning and remarkable efficiency.

This subset of PKML uses records for objects and shapes only, and is retricted to linear arithmetic expressions, i.e. linear combinations of domain variables excluding for instance the previous volume function used in 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.

The compilation of a PKML model into a constraint satisfaction problem with the `geost` with rules global constraint, mainly consists in:

- extracting the definitions of objects and shapes from PKML statements in order to provide them to the `geost` constraint,
- sorting the declarations and rules that refer to objects and shapes and satisfy the linearity condition, for providing them to the `geost` constraint,
- compiling the PKML goals into the `geost` constraint with integrated rules plus additional constraint programming code, as described in the previous section, for the other rules and search predicates that are not accepted by the integrated rule language of `geost`.

This 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. as described in Chapter 6.

4.5 Related Work

4.5.1 Comparison to 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, such as by a rule engine [17]. Rules2CP realizes this aim in the context of combinatorial optimization problems, by tranforming business rules into efficient programs using a completely different representation. 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 thus complies to the business rules manifesto [17].

4.5.2 Comparison to OPL and Zinc

Rules2CP differs from OPL [33] and Zinc [27, 12] modelling languages in several aspects among which: the restriction in Rules2CP to simple data structures of records and enumerated lists, the absence of recursion and the absence of solver specific 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 name data and knowledge rules without complicated variable scopes. A simple module system is used in Rules2CP to avoid name clashes.

Furthermore, we have shown that non trivial search strategies can be declaratively expressed in Rules2CP, by specifying decision variables and branching formulas. The generated constraint programs are reasonably efficient, thanks to the compilation scheme which uses partial evaluation in the rewriting process to eliminate the overhead due to the simplicity of data and control structures. Interestingly, the generated program may be more efficient than constraint programs written by non-expert users, when the compiler uses the global constraints of the target language as illustrated for packing problems with the PKML library and the `geost` constraint.

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.

4.5.3 Comparison to 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 [23]. 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) due to the absence of recursion and of general list constructors in Rules2CP.

Unlike 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 where it is often necessary to pass the list of all variables as arguments to CLP predicates[4]. On the other hand, free variables are not allowed in the right hand side of Rules2CP rules.

[4]For that reason, global variables have also been introduced as extra logical features in many CLP systems.

4.5.4 Comparison to Term Rewriting Systems Tools and Compilation to Constraint Solvers in Java

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 used directly for the implementation of the Rules2CP compiler. For instance, in the context of target constraint solvers in Java, such as Choco, and for Java programming environments in which Rules2CP data structures may be defined by Java objects, the term rewriting system TOM [3] 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.

4.6 Conclusion

The Rules2CP language is a rule-based modelling language for constraint programming. It has been designed to allow non-programmers express industrial requirements about combinatorial optimization problems with business rules (using appropriate editors). In compliance to the business rules manifesto [17], Rules2CP rules are declarative, independent from each other, and not necessarily executed by a rule engine. 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 nevertheless 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`, as described in Chapter 6.

Search strategies can also be specified declaratively in Rules2CP, together with some predefined heuristics. We are however currently exploring the full power of Rules2CP for modelling heuristic knowledge with more flexibility than what we have achieved so far.

4.7 Appendix A: Allen's Interval Relations Library

The library of Allen's interval relations [1] is predefined in Rules2CP by the following file `allen.rcp`:

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

size(S, D) = nth(D, size(S)).

end(O, D) = origin(O, D) + size(shape(O), D).


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 with standard Allen's relations but their direct definition with non strict inequalities is added here for efficiency reasons.

4.8 Appendix B: Region Connection Calculus Library

The library of topological relations of the Region Connection Calculus [28] is predefined in Rules2CP in higher-dimensions by the following file `rcc8.rcp`:

```
import(allen).

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

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

equal(O1, O2, Ds) -->
        forall(D, Ds, equals(O1, O2, D)).

covers(O1, O2, Ds) -->
             forall(D, Ds,
                     started_by(O1, O2, D) or
                     contains(O1, O2, D) or
                     finished_by(O1, O2, D)) and
             exists(D, Ds, not contains(O1, O2, D)).

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

contains_rcc(O1, O2, Ds) -->
        forall(D, Ds,
                contains(O1, O2, D)).

inside(O1, O2, Ds) -->
        forall(D, Ds,
                during(O1, O2, D)).
```

```
overlap(O1, O2, Ds) -->
        forall(D, Ds,
                overlaps_sym(O1, O2, D)).

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

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

4.9 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).

non_overlapping(Items, Dims) -->
        forall(O1, Items,
                forall(O2, Items,
                        iid(O1) < iid(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).

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

volume(O, Dims) =
        product( map(D, Dims, size(O, D)) ).

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, iid(O1) # iid(O2) and on_top(O1, O2))).

weight_stacking(Items) -->
        forall(O1,  Items,
                forall(O2, Items,
                        (iid(O1) # iid(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))).
```

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

4.10 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 Citron 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
   bin_packing(Items,[Bin],Dims).

s1 = {sid=1, size=[1203, 235, 239]}.
s2 = {sid=2, size=[224, 224, 222]}.
s3 = {sid=3, size=[224, 224, 148]}.
s4 = {sid=4, size=[224, 224, 111]}.
s5 = {sid=5, size=[224, 224, 74]}.
s6 = {sid=6, size=[155, 224, 222]}.
s7 = {sid=7, size=[112, 224, 148]}.

i1 = {iid=1, shape=s1, origin=[0,0,0]}.
i2 = {iid=2, shape=s4, origin=[_,_,_], weight=413}.
i3 = {iid=3, shape=s5, origin=[_,_,_], weight=463}.
i4 = {iid=4, shape=s5, origin=[_,_,_], weight=842}.
i5 = {iid=5, shape=s3, origin=[_,_,_], weight=422}.
i6 = {iid=6, shape=s4, origin=[_,_,_], weight=266}.
i7 = {iid=7, shape=s4, origin=[_,_,_], weight=321}.
i8 = {iid=8, shape=s2, origin=[_,_,_], weight=670}.
i9 = {iid=9, shape=s6, origin=[_,_,_], weight=440}.
i10 = {iid=10, shape=s7, origin=[_,_,_], weight=325}.

bin = i1.
items = [i2,i3].
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([I2,I2_2,I2_3,I3,I3_2,I3_3]) :-
R_54#<=>I2+224#=<1203/2,
R_55#<=>I3+224#=<1203/2,
R_56#<=>I2+224#>=1203/2,
R_57#<=>I3+224#>=1203/2,
R_58#<=>I2+224#=<1203/2,
R_59#<=>I3+224#=<1203/2,
R_60#<=>I2+224#>=1203/2,
R_61#<=>I3+224#>=1203/2,

I2_3#=0#\/I2+224#>I3#/\I3+224#>I2#/\
   (I2_2+224#>I3_2#/\I3_2+224#>I2_2)#/\I3_3+74#=I2_3,
I3_3#=0#\/I3+224#>I2#/\I2+224#>I3#/\
    (I3_2+224#>I2_2#/\I2_2+224#>I3_2)#/\I2_3+111#=I3_3,
I3 +224#=< I2 #\/ I2 +224#=< I3 #\/
          ( I3 _2 +224#=< I2 _2 #\/ I2 _2 +224#=< I3 _2 ) #\/ I2 _3 +111#\=I3 _3 ,

100*max(413*R_54+(463*R_55+0),413*R_56+(463*R_57+0))#=<
       120*min(413*R_58+(463*R_59+0),413*R_60+(463*R_61+0)),
I2 +224#=< I3 #\/ I3 +224#=< I2 #\/( I2 _2 +224#=< I3 _2 #\/ I3 _2 +224#=< I2 _2 )#\/
max(max(I2 , I3)−min(I2 , I3),
          max(I2 +224, I3 +224) − min(I2 +224, I3 +224))# <10#/\
max(max(I2 _2 , I3 _2)−min(I2 _2 , I3 _2),
          max(I2 _2 +224, I3 _2 +224) − min(I2 _2 +224, I3 _2 +224))# <10,

I3 +224#=< I2 #\/ I2 +224#=< I3 #\/
( I3 _2 +224#=< I2 _2 #\/ I2 _2 +224#=< I3 _2 )#\/
max(max(I3 , I2)−min(I3 , I2),
          max(I3 +224, I2 +224) − min(I3 +224, I2 +224))# <10#/\
max(max(I3 _2 , I2 _2)−min(I3 _2 , I2 _2),
          max(I3 _2 +224, I2 _2 +224) − min(I3 _2 +224, I2 _2 +224))# <10,

0#=<I2 ,
I2 +224#=<1203,
0#=<I2 _2 ,
I2 _2 +224#=<235,
0#=<I2 _3 ,
I2 _3 +111#=<239,
0#=<I3 ,
I3 +224#=<1203,
0#=<I3 _2 ,
I3 _2 +224#=<235,
0#=<I3 _3 ,
I3 _3 +74#=<239,

I2 +224#=< I3 #\/ I3 +224#=< I2 #\/( I2 _2 +224#=< I3 _2 #\/ I3 _2 +224#=< I2 _2 #\/
( I2 _3 +111#=< I3 _3 #\/ I3 _3 +74#=< I2 _3 )),

labeling([bisect ,up],[I2 , I2 _2 , I2 _3 , I3 , I3 _2 , I3 _3 ]).
```

Chapter 5

Extra Features of PKML for Virtual Reality

The Virtual Reality components intend to find solutions to the packing problem that are not devisable with the optimisation solvers. Thus, it will need parameters that are geometrically more accurate to be added in the definition of objects and shapes. This involves two parameters in the objects and one in the shapes.

5.1 Object Parameters

5.1.1 vr_shape

In graphical representation and physical simulation, the Virtual Reality module processes the exact geometry of the handled objects and not only their enveloping boxes. For that purpose, a vr_shape attribute is added to the PKML Objects.

This attribute is a string consisting in XML code that represents the accurate geometry of the considered object:

```
<GraphicalRepresentation>
 <ObjRepresentation Name="knob_freeze_9_C0_T_3DXMLREFID_9">
  <Faces Triangles="0 2 1 3 2 0 4 2 3 5 2 4 6 2 5 6 7 2 7 8 2 2 8 9 ..."/>
  <Vertex>
   <Positions>-0.017092 -0.007080 0.017366 -0.018500 ......</Positions>
  </Vertex>
 </ObjRepresentation>
</GraphicalRepresentation>
```

Where:

- Name is the name of the object.
- Positions represent the position of all vertices;
  order: vertex0.x vertex0.y vertex0.z vertex1.x ...
- Triangles represent the faces of the object;
  order: triangle0(index of vertex 0, index of vertex 1, index of vertex 2)
  triangle1 ...;
  for example, 0 2 1 3 2 0 means:

  - triangle 0 is made of vertex 0, 2 and 1,
  - triangle 1 is made of vertex 3, 2 and 0.

This is actually a simplified version of the ".obj" format: `http://www.royriggs.com/obj.html`.

5.1.2 vr_displacement

Due to the fact that the Virtual Reality component consider object positions as real numbers rather than integers, an attribute vr_displacement that represents the real position of an object in space is added to the PKML description of this object. It consists in a string of 7 real numbers specifying a position (x, y, z) and a rotation (in the form of a quaternion). This parameter will need to be updated after each use of the solvers.

5.2 Shape Parameters

5.2.1 vr_rotation

Since an object may be associated with multiple alternative PKML shapes corresponding to the unique real shape of this object but with different rotations, it is required to explicitly indicate these rotations so as to simplify the updating of an object position after it has been processed by the solvers. For this purpose, we define a vr_rotation attribute consisting in a string that specifies the quaternion transforming the original into this alternative shape.

Example:

```
s1 = {shape=box, size=[1,2,3], vr_rotation= "0 0 0 1" }.
s2 = {shape=box, size=[1,3,2], vr_rotation= "0.707 0 0 0.707"}.
s3 = {shape=box, size=[3,2,1], vr_rotation= "0 0.707 0 0.707"}.
s4 = {shape=alternative, shapes=[ s1, s2, s3]}.
o1 = {sid=s4, origin=[_,_,_], vr_displacement = "0 0 0 0 0 0 1"}
```

In the Virtual Reality context, we do not allow an object to be made of alternative shapes that do not represent the same shape up to a rotation. This is allowed in PKML and `geost` for handling for instance configuration problems where the different shapes correspond to different design choices.

5.3 Additional parameters for physical simulations

It is foreseen that physical simulation may require additional parameters (possibly dry friction parameters, deformation parameters, ...) that cannot be completely defined at this stage. Thanks to the extensibility of PKML records, these will be added later without particular difficulties. They will be specified in subsequent releases of the current document.

Chapter 6

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

6.1 Introduction

This chapter extends a global constraint $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{R})$ for handling the location in space of k-dimensional objects $\mathcal{O}$ ($k \in \mathbb{N}^+$), each of which taking a shape among a set of shapes $\mathcal{S}$, subject to rules $\mathcal{R}$ in a language which is essentially a subset of PKML.

In order to model directly a lot of side constraints, which always show up in the context of real-life applications, many global constraints have traditionally been extended with extra options or arguments. This is why, in a closely related area, the *diffn* constraint of CHIP provides, beside non-overlapping, a variety of other geometrical constraints (in fact more than 10 side constraints). This was also the case for the *cycle* and *tree* constraints [5, 6] where, beside a basic graph partitioning constraint, a variety of useful side constraints were also provided. Even if this makes sense when one wants to efficiently solve specific real-life applications, this proliferation of arguments and options has two major drawbacks:

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

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

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

The *geost* constraint can thus be seen as a natural target constraint of the `PKML` modeling language. In $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{R})$, each shape from $\mathcal{S}$ is defined as a finite set of shifted boxes, where each shifted box is described by a box in a k-dimensional space at the given offset with the given sizes. More precisely a *shifted box* $s \in \mathcal{S}$ is an entity defined by its shape id $s.sid$, shift offset $s.t[d]$, $1 \leq d \leq k$, and sizes $s.l[d]$ $(s.l[d] > 0, 1 \leq d \leq k)$. All attributes of a shifted box are integer values. Then, a *shape* is a collection of shifted boxes all sharing the same shape id.[1]

Each object $o \in \mathcal{O}$ is an entity defined by its unique object id $o.oid$ (an integer), shape id $o.sid$ (an integer if the object has a fixed shape, or a domain variable for *polymorphic* objects, which have alternative shapes), and origin $o.x[d]$, $1 \leq d \leq k$ (integers, or domain variables that do not occur anywhere else in the constraint).[2] Objects and shifted boxes may also have additional, integer (but see also Section 6.7) attributes, such as weight, customer, or fragility, used by the rules.

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

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

Constraint satisfaction problems using quantified formulas (QCSP) have been, for instance, studied by Benedetti et al. [7], mostly in the context of modeling games. QCSP does not provide disjunction but actively uses quantifiers in the evaluation, whereas we eliminate all quantifiers in the process of rewriting to QFPA.

Example 6 *This running example will be used to illustrate the way we compile rules to code used by the sweep-based algorithm [4] for filtering the nonground attributes of each object. Suppose that we have five objects o_1, o_2, o_3, o_4 and o_5 such that:*

- *o_1, o_2 and o_4 correspond to fixed rectangles of respective size* 3×1, 1×1 *and* 3×1.
- *The coordinates of o_3 are fixed but not its shape variable s_3, which can take values* 3 *or* 4 *(i.e., we can choose among two shapes for object o_3). We will denote by ℓ_{31} resp. ℓ_{32} the length resp. height of o_3.*
- *The coordinates of the non-fixed square o_5 of size* 2×2 *correspond to the two variables* $x_{51} \in [1, 9]$ *and* $x_{52} \in [1, 6]$.

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

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

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

- *o_2, o_4 and o_5 have the additional attribute $type$ with value 1 whereas o_1 and o_3 have $type$ with value 2.*
- *Two rules must be obeyed:*
  - *All objects should be mutually non-overlapping (see Fig. 6.11).*
  - *If the $type$ attribute of two objects both equal 1, the two objects should not meet (see Fig. 6.11 again).*[4]

The full details and geost *encoding of the example are shown in Fig. 6.1; for an explanation of the notation, see Section 6.2 and Table 6.4.*

□

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

Implementation overview. Fig. 6.2 provides the overall architecture of the implementation. When the $geost$ constraint is posted, the given business rules are translated, first into QFPA, then into generators of k-dimensional forbidden sets. Such generators, *k-indexicals*, are a generalization of the indexicals of cc(FD) [32]. Each time the constraint wakes up, the sweep-based algorithm [4] generates forbidden sets for a specific object o by invoking the relevant k-indexicals, then looks for points that are not contained in any forbidden set in order to prune the nonground attributes of o.

Chapter outline. In Section 6.2, we present the rule language, its abstract syntax and its features. In Section 6.3, we present the QFPA core fragment of the language, its declarative semantics, and how the rule language is rewritten into QFPA. In Section 6.4, we describe (1) how a QFPA formula is compiled to generators of k-dimensional forbidden sets, and (2) how the forbidden sets generated by such generators are aggregated by a sweep-based algorithm and used for filtering. In Section 6.5, we extend the filtering to accommodate polymorphic objects. In Section 6.7, we conclude and mention a number of issues that we are currently working on. In the Appendix, we show the Prolog representation of the various language elements that we actually use in the implementation. The Appendix also shows how the Region Connection Calculus may be expressed in our language, as well as rules encoding a problem instance provided by a major car manufacturer and rules encoding a packing-unpacking problem.

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

6.2 The Rule Language: Syntax and Features

Fig. 6.3 shows the inductive definition of the rule language. A *macro* is simply a shorthand device: during a rewriting phase, whenever an expression matching the left-hand side of a macro is encountered, it is replaced by the corresponding right-hand side. A *fol* is a first-order logic

[4] Two rectangles meet also if their corners meet.

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

Figure 6.1: Running example encoded with *geost*.

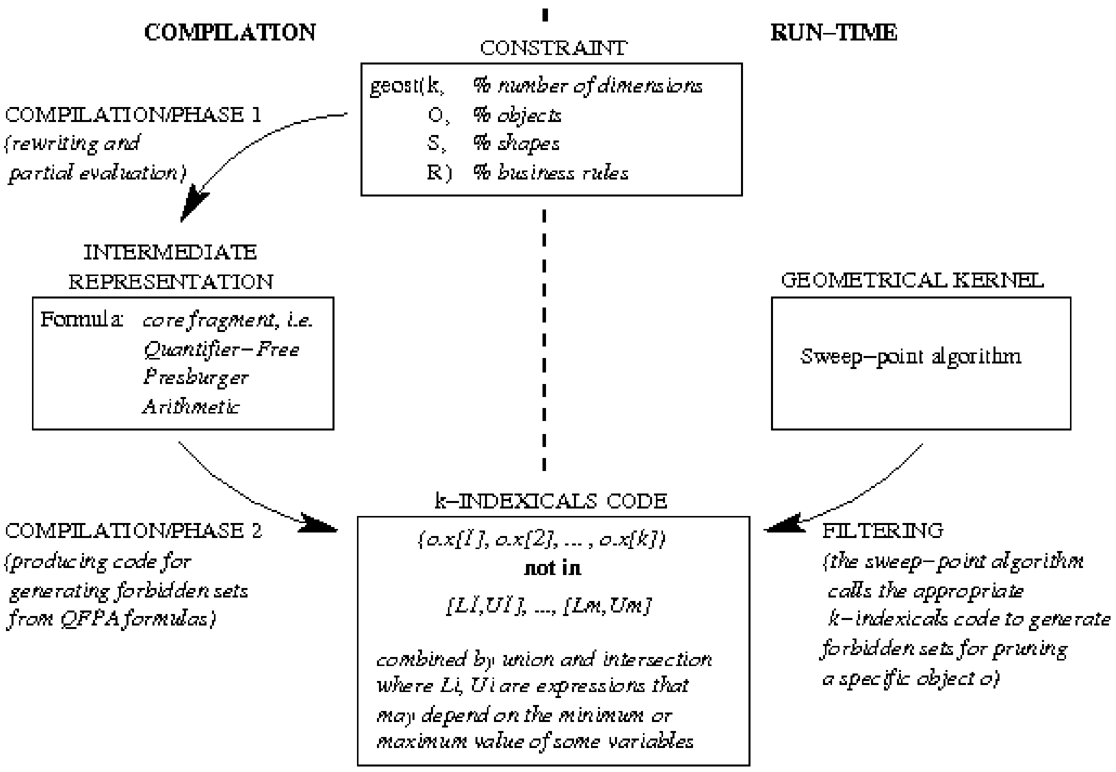

COMPILATION
CONSTRAINT
RUN–TIME
geost(k, % number of dimensions
O, % objects
S, % shapes
R) % business rules
COMPILATION/PHASE 1
(rewriting and partial evaluation)
INTERMEDIATE REPRESENTATION
Formula: core fragment, i.e. Quantifier–Free Presburger Arithmetic
GEOMETRICAL KERNEL
Sweep–point algorithm
k–INDEXICALS CODE
COMPILATION/PHASE 2
(producing code for generating forbidden sets from QFPA formulas)
(o.x[1], o.x[2], ... , o.x[k])
not in
[L1,U1], ..., [Lm,Um]
combined by union and intersection where Li, Ui are expressions that may depend on the minimum or maximum value of some variables
FILTERING
(the sweep–point algorithm calls the appropriate k–indexicals code to generate forbidden sets for pruning a specific object o)

Figure 6.2: Overall architecture of the implementation.

formula that must hold for the constraint to be true. A *term* is a *variable*, an *integer*, an *identifier*, or a *compound term*. A *compound term* consists of a *functor* (an identifier) and one or more arguments (terms). A term is *ground* if it is free of variables. An *entity* denotes an object resp. shifted box, the exact structure of which is left unspecified, but a possible Prolog representation is shown in Appendix 6.8. An *attref* is a reference to an attribute of an entity.

Bounded existential resp. *universal quantifiers* are provided. They are meaningful if the quantified variable occurs in the quantified *fol*. They are treated by expansion to a disjunction resp. conjunction of instances of that *fol* with an element of the *collection* substituted for the quantified variable. For example, formulas (6.1) and (6.2) below are equivalent:

$$\forall(x, [0, 1, 2], p(x)) \tag{6.1}$$

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

$$p(0) \wedge p(1) \wedge p(2) \tag{6.2}$$

A *cardinality formula* specifies a variable quantified over a list of terms, a lower and an upper bound, and a *fol* template mentioning the quantified variable. The formula is true if and only if the number of true instances of the *fol* template is within the given bounds. Cardinality formulas [20] are treated by expansion to $\neg$, $\wedge$ and $\vee$ connectives [13]. For example, formulas (6.3) and (6.4) are equivalent:

$$\#(y, [o_1, o_2, o_3], 2, 3, y.\text{type} > 5) \tag{6.3}$$

$$\bigvee \begin{pmatrix} o_1.\text{type} > 5 \land o_2.\text{type} > 5 \\ o_1.\text{type} > 5 \land o_3.\text{type} > 5 \\ o_2.\text{type} > 5 \land o_3.\text{type} > 5 \end{pmatrix} \tag{6.4}$$

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

A *folding operator* allows to express e.g. the sum of some attribute over a set of objects. The operator specifies a variable quantified over a list of terms, a binary operator, an identity element, and a template mentioning the quantified variable. The identity element is needed for the empty list case. For example, formulas (6.5) and (6.6) are equivalent:

$$@(y, [o_1, o_2, o_3], +, 0, y.\text{weight}) \tag{6.5}$$

$$o_1.\text{weight} + o_2.\text{weight} + o_3.\text{weight} \tag{6.6}$$

6.3 QFPA Core Fragment

In this section, we show how a formula p in the rule language is rewritten by a series of equivalence-preserving transformations into a *qfpa*, i.e. a formula of the core fragment of the language shown in Fig. 6.4. In fact, the fragment coincides with Quantifier-Free Presburger Arithmetic (QFPA), although QFPA is usually described with a less restrictive syntax. The declarative semantics of a *qfpa* is the natural one.

QFPA is widely used in symbolic verification, and there has been much work on deciding whether a given QFPA formula is satisfiable [16]. Many methods based on integer programming techniques [26] rely on having the formula on disjunctive normal form. However, for constraint programming purposes, we are interested in necessary conditions that can be used for filtering domain variables, and we are not aware on any such work on QFPA.

6.3.1 Rewriting into QFPA

We now show the details of rewriting the formula given as the *geost* parameter $\mathcal{R}$ in the following eight steps into a *qfpa* $\hat{\mathcal{R}}$. Fig. 6.5 shows the details of some of these steps as tables. The cell in the column entitled $\mathbf{condition}$, if nonempty, mentions the condition under which the rewrite is done. Later, we will show how $\hat{\mathcal{R}}$ is translated to generators of forbidden sets.

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

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

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

Figure 6.3: The rule language

$$\boxed{\begin{array}{lll} \mathit{qfpa} ::= & \mathit{qfpa} \wedge \mathit{qfpa} & \{\text{ conjunction }\} \\ \quad\mid & \mathit{qfpa} \vee \mathit{qfpa} & \{\text{ disjunction }\} \\ \quad\mid & \sum_i \mathit{integer}_i \cdot \mathit{attref}_i \geq \mathit{integer} & \{\text{ base case }\} \end{array}}$$

Figure 6.4: Core fragment of the language. An *attref* corresponds to a nonground attribute of an object or an attribute of a shifted box of a polymorphic object.

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

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

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

Moving $+$ inside $\min$ and $\max$. Any expression with $\min$ or $\max$ occurring inside $+$ are rewritten by using the commutative and distributive laws (6.7) so that the $+$ is moved inside the other operator.

$$\begin{aligned} a + b &= b + a \\ a + \min(b, c) &= \min(a + b, a + c) \\ a + \max(b, c) &= \max(a + b, a + c) \end{aligned} \tag{6.7}$$

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

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

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

line	$\mathbf{p}$	$\mathbf{R_1(p)}$	condition
1	p	$R_1(q)$	$q = \text{macro}(p)$
2	$\neg p$	$\neg R_1(p)$	
3	$p \Rightarrow q$	$R_1(q \vee \neg p)$	
4	$p \Leftrightarrow q$	$R_1((p \Rightarrow q) \wedge (q \Rightarrow p))$	
5	$\exists(x, [y_1, \ldots, y_n], p)$	$R_1(p_{x/y_1} \vee \cdots \vee p_{x/y_n})$	
6	$\forall(x, [y_1, \ldots, y_n], p)$	$R_1(p_{x/y_1} \wedge \cdots \wedge p_{x/y_n})$	
7	$@(x, [y_1, \ldots, y_n], \circ, z, p)$	$R_1(p_{x/y_1} \circ \cdots \circ p_{x/y_n} \circ z)$	
8	$\#(x, [], l, u, p)$	`true`	$l \leq 0 \leq u$
9	$\#(x, [], l, u, p)$	`false`	$l > 0 \vee 0 > u$
10	$\#(x, [y_1, \ldots, y_n], l, u, p)$	$R_1 \left(\begin{array}{c} (p_{x/y_1} \wedge \#(x, [y_2, \ldots, y_n], l-1, u-1, p) \vee \\ (\neg p_{x/y_1} \wedge \#(x, [y_2, \ldots, y_n], l, u, p) \end{array} \right)$	$n > 0$
11	*expr*	i	$i = \text{ieval}(p)$
12	$\texttt{objects}([\texttt{o}_1, \ldots, \texttt{o}_n])$	objects with the given oids	
13	$\texttt{sboxes}([\texttt{s}_1, \ldots, \texttt{s}_n])$	sboxes with the given sids	

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

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

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

Figure 6.5: **Top.** Rewrite phase 1, of a formula p into a formula $R_1(p)$, eliminates macros (line 1), implication (line 3), equivalence (line 4), bounded quantifiers (line 5-6), folding operators (line 7), cardinality operators (line 8-10), ground attribute references (line 11), and entity collections (line 12-13). If a compound term does not match any line 1-13, its arguments are rewritten recursively. $p_{x/y}$ denotes the term p with y substituted for x. $\text{macro}(p)$ denotes the macro expansion of the formula p. $\text{ieval}(p)$ denotes the integer value of the ground expression p. **Bottom left.** Rewrite phase 3, of a formula p into a formula $R_3(p)$, normalizes comparison operators into either $\geq$ or $>$. **Bottom center.** Rewrite phase 4, of a formula p into a formula $R_4(p, 1)$, eliminates the $-$, $\times$ and $/$ operators, and assigns a coefficient c to each operand of the rewritten formula. $\text{reval}(y)$ denotes the rational value of the ground expression y. **Bottom right.** Rewrite phase 6, of a formula p into a formula $R_6(p)$, eliminates $\min$ and $\max$.

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

Example 7 *Returning to our running example, we show in Figs. 6.6-6.7 how the initial business rules are successively rewritten into a* qfpa*. The example shows that the rewrite process essentially amounts to partial evaluation. The resulting* qfpa*, $\hat{\mathcal{R}}$, is a conjunction of six subformulas corresponding respectively to:*

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

$\square$

6.4 Compiling to an Efficient Run-Time Representation

It is straightforward to obtain necessary conditions for *qfpa*s as well as pruning rules operating on one variable at a time. Based on such conditions and pruning rules, we will show how to construct generators of k-dimensional forbidden sets. We call such generators *k-indexicals*, for they are generalization of the indexicals of cc(FD) [32]. Finally, we show how the forbidden sets generated by such indexicals are aggregated by the sweep-based algorithm [4] and used for filtering.

Indexicals were first introduced for the language cc(FD) [32] and later used in the context of CLP(FD) [11, 10], AKL [8] and finite set constraints [30]. They have proven a powerful and efficient way of implementing constraint propagation. A key feature of an indexical is that it is a function of the current domains of the variables on which it depends. Thus, indexicals also capture the propagation from variables to variables that occurs as variables are pruned. In the cited implementations, an indexical is a procedure that computes the feasible set of values for a variable. We generalize this notion to generating a forbidden set of k-dimensional points for an object, and so k-indexicals captures the propagation from objects to objects that occurs as object attributes are pruned.

6.4.1 Necessary Conditions

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

We will ignore the degenerate cases where $\hat{\mathcal{R}}$ is `true` resp. `false`, in which case *geost* merely succeeds resp. fails. For the normal *qfpa* cases, we obtain the necessary conditions shown in Table 6.1.

6.4.2 Pruning Rules

For the base case $\sum_i c_i \cdot x_i \geq r$, we have the well-known pruning rules (6.8), which provide sharp bounds; see e.g. [19] for details.

```
          all_not_overlap_sboxes([1,2],[1,2,3,4,5]),
          all_type1_not_meet_sboxes([1,2],[1,2,3,4,5])]).
```

$$\bigwedge\left(\begin{array}{c}
\neg\left(\bigwedge\left(\begin{array}{c} 4 > x_{51} \\ x_{51}+2 > 1 \\ 3 > x_{52} \\ x_{52}+2 > 2 \end{array}\right)\right) \\
\neg\left(\bigwedge\left(\begin{array}{c} 4 > x_{51} \\ x_{51}+2 > 3 \\ 4 > x_{52} \\ x_{52}+2 > 3 \end{array}\right)\right) \\
\neg\left(\bigwedge\left(\begin{array}{c} 2+\ell_{31} > 3 \\ 5+\ell_{32} > 7 \end{array}\right)\right) \\
\neg\left(\bigwedge\left(\begin{array}{c} 2+\ell_{31} > x_{51} \\ x_{51}+2 > 2 \\ 5+\ell_{32} > x_{52} \\ x_{52}+2 > 5 \end{array}\right)\right) \\
\neg\left(\bigwedge\left(\begin{array}{c} 6 > x_{51} \\ x_{51}+2 > 3 \\ 8 > x_{52} \\ x_{52}+2 > 7 \end{array}\right)\right) \\
\neg\left(\bigwedge\left(\begin{array}{c} 4 \geq x_{51} \\ x_{51}+2 \geq 3 \\ 4 \geq x_{52} \\ x_{52}+2 \geq 3 \\ \bigvee\left(\begin{array}{c} 4 = x_{51} \\ x_{51}+2 = 3 \\ 4 = x_{52} \\ x_{52}+2 = 3 \end{array}\right) \end{array}\right)\right) \\
\neg\left(\bigwedge\left(\begin{array}{c} 6 \geq x_{51} \\ x_{51}+2 \geq 3 \\ 8 \geq x_{52} \\ x_{52}+2 \geq 7 \\ \bigvee\left(\begin{array}{c} 6 = x_{51} \\ x_{51}+2 = 3 \\ 8 = x_{52} \\ x_{52}+2 = 7 \end{array}\right) \end{array}\right)\right)
\end{array}\right)
\qquad
\bigwedge\left(\begin{array}{c}
\bigvee\left(\begin{array}{c} 4 \leq x_{51} \\ x_{51}+2 \leq 1 \\ 3 \leq x_{52} \\ x_{52}+2 \leq 2 \end{array}\right) \\
\bigvee\left(\begin{array}{c} 4 \leq x_{51} \\ x_{51}+2 \leq 3 \\ 4 \leq x_{52} \\ x_{52}+2 \leq 3 \end{array}\right) \\
\bigvee\left(\begin{array}{c} 2+\ell_{31} \leq 3 \\ 5+\ell_{32} \leq 7 \end{array}\right) \\
\bigvee\left(\begin{array}{c} 2+\ell_{31} \leq x_{51} \\ x_{51}+2 \leq 2 \\ 5+\ell_{32} \leq x_{52} \\ x_{52}+2 \leq 5 \end{array}\right) \\
\bigvee\left(\begin{array}{c} 6 \leq x_{51} \\ x_{51}+2 \leq 3 \\ 8 \leq x_{52} \\ x_{52}+2 \leq 7 \end{array}\right) \\
\bigvee\left(\begin{array}{c} 4 < x_{51} \\ x_{51}+2 < 3 \\ 4 < x_{52} \\ x_{52}+2 < 3 \\ \bigwedge\left(\begin{array}{c} 4 \neq x_{51} \\ x_{51}+2 \neq 3 \\ 4 \neq x_{52} \\ x_{52}+2 \neq 3 \end{array}\right) \end{array}\right) \\
\bigvee\left(\begin{array}{c} 6 < x_{51} \\ x_{51}+2 < 3 \\ 8 < x_{52} \\ x_{52}+2 < 7 \\ \bigwedge\left(\begin{array}{c} 6 \neq x_{51} \\ x_{51}+2 \neq 3 \\ 8 \neq x_{52} \\ x_{52}+2 \neq 7 \end{array}\right) \end{array}\right)
\end{array}\right)$$

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

Figure 6.6: Running example business rules (top), formula after macro expansion and constant folding (middle left), elimination of negation (middle right), normalization of arithmetic (bottom left), and elimination of operators (bottom right).

$$
\bigwedge\begin{pmatrix}
\bigvee\begin{pmatrix} x_{51} \geq 4 \\ -1 \cdot x_{51} \geq 1 \\ x_{52} \geq 3 \\ -1 \cdot x_{52} \geq 0 \end{pmatrix} \\
\bigvee\begin{pmatrix} x_{51} \geq 4 \\ -1 \cdot x_{51} \geq -1 \\ x_{52} \geq 4 \\ -1 \cdot x_{52} \geq -1 \end{pmatrix} \\
\bigvee\begin{pmatrix} -1 \cdot \ell_{31} \geq -1 \\ -1 \cdot \ell_{32} \geq -2 \end{pmatrix} \\
\bigvee\begin{pmatrix} -1 \cdot \ell_{31} + x_{51} \geq 2 \\ -1 \cdot x_{51} \geq 0 \\ -1 \cdot \ell_{32} + x_{52} \geq 5 \\ -1 \cdot x_{52} \geq -3 \end{pmatrix} \\
\bigvee\begin{pmatrix} x_{51} \geq 6 \\ -1 \cdot x_{51} \geq -1 \\ x_{52} \geq 8 \\ -1 \cdot x_{52} \geq -5 \end{pmatrix} \\
\bigvee\begin{pmatrix} x_{51} \geq 5 \\ -1 \cdot x_{51} \geq 0 \\ x_{52} \geq 5 \\ -1 \cdot x_{52} \geq 0 \\
\bigwedge\begin{pmatrix}
\bigvee\begin{pmatrix} -1 \cdot x_{51} \geq -3 \\ x_{51} \geq 5 \end{pmatrix} \\
\bigvee\begin{pmatrix} x_{51} \geq 2 \\ -1 \cdot x_{51} \geq 0 \end{pmatrix} \\
\bigvee\begin{pmatrix} -1 \cdot x_{52} \geq -3 \\ x_{52} \geq 5 \end{pmatrix} \\
\bigvee\begin{pmatrix} x_{52} \geq 2 \\ -1 \cdot x_{52} \geq 0 \end{pmatrix}
\end{pmatrix}
\end{pmatrix} \\
\bigvee\begin{pmatrix} x_{51} \geq 7 \\ -1 \cdot x_{51} \geq 0 \\ x_{52} \geq 9 \\ -1 \cdot x_{52} \geq -4 \\
\bigwedge\begin{pmatrix}
\bigvee\begin{pmatrix} -1 \cdot x_{51} \geq -5 \\ x_{51} \geq 7 \end{pmatrix} \\
\bigvee\begin{pmatrix} x_{51} \geq 2 \\ -1 \cdot x_{51} \geq 0 \end{pmatrix} \\
\bigvee\begin{pmatrix} -1 \cdot x_{52} \geq -7 \\ x_{52} \geq 9 \end{pmatrix} \\
\bigvee\begin{pmatrix} x_{52} \geq 6 \\ -1 \cdot x_{52} \geq -4 \end{pmatrix}
\end{pmatrix}
\end{pmatrix}
\end{pmatrix}
\qquad
\bigwedge\begin{pmatrix}
\bigvee\begin{pmatrix} x_{51} \geq 4 \\ x_{52} \geq 3 \end{pmatrix} \\
\bigvee\begin{pmatrix} x_{51} \geq 4 \\ -1 \cdot x_{51} \geq -1 \\ x_{52} \geq 4 \\ -1 \cdot x_{52} \geq -1 \end{pmatrix} \\
\bigvee\begin{pmatrix} -1 \cdot \ell_{31} + x_{51} \geq 2 \\ -1 \cdot \ell_{32} + x_{52} \geq 5 \\ -1 \cdot x_{52} \geq -3 \end{pmatrix} \\
\bigvee\begin{pmatrix} x_{51} \geq 6 \\ -1 \cdot x_{51} \geq -1 \\ -1 \cdot x_{52} \geq -5 \end{pmatrix} \\
\bigvee\begin{pmatrix} x_{51} \geq 5 \\ x_{52} \geq 5 \\
\bigwedge\begin{pmatrix}
\bigvee\begin{pmatrix} -1 \cdot x_{51} \geq -3 \\ x_{51} \geq 5 \end{pmatrix} \\
x_{51} \geq 2 \\
\bigvee\begin{pmatrix} -1 \cdot x_{52} \geq -3 \\ x_{52} \geq 5 \end{pmatrix} \\
x_{52} \geq 2
\end{pmatrix}
\end{pmatrix} \\
\bigvee\begin{pmatrix} x_{51} \geq 7 \\ -1 \cdot x_{52} \geq -4 \\
\bigwedge\begin{pmatrix}
\bigvee\begin{pmatrix} -1 \cdot x_{51} \geq -5 \\ x_{51} \geq 7 \end{pmatrix} \\
x_{51} \geq 2 \\
\bigvee\begin{pmatrix} x_{52} \geq 6 \\ -1 \cdot x_{52} \geq -4 \end{pmatrix}
\end{pmatrix}
\end{pmatrix}
\end{pmatrix}
$$

Figure 6.7: Running example formula after elimination of rational numbers (left) and simplification (right), resulting in a QFPA formula $\hat{\mathcal{R}}$.

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

Table 6.1: Necessary condition $N(t)$ for *qfpa* t

$$\forall j \begin{cases} x_j \geq \lceil \frac{r - MAX(\sum_{i \neq j} c_i \cdot x_i)}{c_j} \rceil, & \text{if } c_j > 0 \\ x_j \leq \lfloor \frac{-r + MAX(\sum_{i \neq j} c_i \cdot x_i)}{-c_j} \rfloor, & \text{otherwise} \end{cases} \tag{6.8}$$

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

- If x_j occurs in p but not in q, rule (6.8) is only valid for p if the necessary condition for q does not hold.
- Similarly if x_j occurs in q but not in p.
- If x_j occurs in both p and q, we can use rule (6.8) for both p and q and conclude that x_j must be in the union of the two feasible intervals.

Finally, consider a conjunction $p \wedge q$, i.e. both p and q must hold. If x_j occurs in both p and q, we can use rule (6.8) for both p and q and conclude that x_j must be in the intersection of the two feasible intervals.

Example 8 *Returning to our running example, consider the fragment $x_{51} \geq 4 \vee x_{52} \geq 3$ of the* qfpa*, which comes from a rule preventing o_5 from overlapping o_1. Suppose that we want to prune x_{52}. Then we can combine the necessary condition for $x_{51} \geq 4$ with rule (6.8) for $x_{52} \geq 3$ into the conditional pruning rule:*

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

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

□

6.4.3 k-Indexicals

Recall that the set of rules given in $\mathcal{R}$ has been rewritten into a *qfpa* $\hat{\mathcal{R}}$. Consider this formula, or some subformula $\hat{\mathcal{R}}_i$ of it if $\hat{\mathcal{R}}$ is a conjunction (see Section 6.4.4). The idea is to compile this subformula, for each object o mentioned by it, into a k-indexical for $\hat{\mathcal{R}}_i$ and o. The forbidden sets that it generates can then be aggregated and used by the sweep-point kernel [4] to prune the nonground attributes of o. Let us introduce some notation to make this idea clear.

Definition 3 *A* forbidden set for *qfpa* r and object o *is a set*[5] *of k-dimensional points such that, if o is placed at any of these points, r is disentailed.*

Definition 4 *A* k-indexical for *qfpa* r and object o *is a procedure that functions as a generator of forbidden sets for r and o. It has the form $object.x \notin ibody$ where $ibody$ is defined in Fig. 6.8. The k-indexical* depends on object o' *if $ibody$ mentions o'.*

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

k-indexicals are described by the inductive definition shown in Fig. 6.8. They are built up from generators of k-dimensional half-planes, combined by union and intersection operations.

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

Figure 6.8: k-indexicals

6.4.4 Compilation

The *qfpa* $\hat{\mathcal{R}}$, normally[6] a conjunction $\hat{r}_1 \wedge \cdots \wedge \hat{r}_n$, is compiled to k-indexicals by the following steps:

1. Partition the conjuncts of $\hat{\mathcal{R}}$ into equivalence classes $\hat{\mathcal{R}}_1, \ldots, \hat{\mathcal{R}}_m$ such that for all $i < j$, $\hat{r}_i$ and $\hat{r}_j$ are in the same equivalence class if and only if they mention[7] the same set of objects of $\mathcal{O}$.

2. For each equivalence class $\hat{\mathcal{R}}_i$ and object $o \in \mathcal{O}$ mentioned by $\hat{\mathcal{R}}_i$, map $\hat{\mathcal{R}}_i$ (as a conjunction) into a k-indexical for o, having the form $o.x \notin F_o(\hat{\mathcal{R}}_i)$, according to Table 6.2.

The mapping closely follows the pruning rules (6.8), except now we want to obtain a forbidden set instead of a feasible interval. Row 5 of Table 6.2 corresponds to the case where r does not mention o, in which case all points are forbidden for o if r is disentailed, and no points are forbidden for o otherwise.

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

[6]Since it comes from the conjunction of business rules stated in the last argument of *geost*.

[7]A formula *mentions* an object o if it refers to a nonground attribute of o.

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

Table 6.2: Mapping a *qfpa* r to a generator of forbidden sets, $F_o(r)$, for the object o. We assume here that o is not polymorphic.

It is well known that indexicals can be efficiently compiled and executed by a virtual machine [11, 10]. In our context, we predict that there will be a large amount of common subterms in the k-indexicals, and so common subexpression elimination will be quite important. Therefore, a register-based virtual machine would seem an appropriate choice.

It is worth noting that the forbidden sets generated by our compiler do not necessarily include all infeasible points. Consider e.g. the *qfpa* $o.x[1] + o.x[2] = 3$ with $o.x[1] \in [0,3], o.x[2] \in [0,3]$, which we compile to:

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

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

Example 9 *Returning to our running example, we obtained a* qfpa *which was a conjunction of six subformulas (see Fig. 6.7). They are partitioned into two equivalence classes:*

1. *One for the single conjunct that mentions both o_3 and o_5, reflecting the business rule that o_3 and o_5 must not overlap. It is mapped to k-indexicals (6.9) and (6.10).*
2. *One for the five conjuncts that only mention o_5 (because o_1, o_2 and o_4 are ground). It is mapped to k-indexical (6.11) and reflects several business rules:*
    - *o_5 must not overlap o_1, o_2 nor o_4,*
    - *o_5 must not meet o_2 nor o_4, for the $type$ attribute of these three objects takes the value 1.*

The three k-indexicals reflect the following business rules:

1. *o_3 must not take a shape that will cause it to overlap o_5. Note that this k-indexical propagates from o_5 to the shape id of o_3. Pruning of shape ids of polymorphic objects is discussed in Section 6.5. Initially, no forbidden boxes are generated.*

$$\boxed{s_3 \notin \bigcap \left(\begin{array}{c} \{i \in \text{dom}(s_3) \mid s_3 = i \Rightarrow \ell_{31} > \overline{x_{51}} - 2\} \\ \{i \in \text{dom}(s_3) \mid s_3 = i \Rightarrow \ell_{32} > \overline{x_{52}} - 5\} \\ \textbf{if } \underline{x_{52}} > 3 \textbf{ then } \mathbb{Z} \textbf{ else } \emptyset \end{array} \right)} \tag{6.9}$$

2. *o_5 must not overlap o_3. Note that this k-indexical propagates from o_3 to o_5. Initially, it will generate the forbidden box shown in Fig. 6.9 (top left).*

$$\boxed{o_5.x \notin ([1, (\underline{\ell_{31}} + 1)], [4, (\underline{\ell_{32}} + 4)])} \tag{6.10}$$

3. *o_5 must not overlap o_1, o_2 nor o_4, nor meet o_2 nor o_4. This k-indexical will generate the forbidden boxes shown in Fig. 6.9 (top right).*

$$\boxed{o_5.x \notin \bigcup \left(\begin{array}{c} ([1,3],[1,2]) \\ ([2,3],[2,3]) \\ ([2,5],[6,6]) \\ \bigcap \left(\begin{array}{c} ([1,4],[1,4]) \\ \bigcup \left(\begin{array}{c} ([4,4],[1,6]) \\ ([1,1],[1,6]) \\ ([1,9],[4,4]) \\ ([1,9],[1,1]) \end{array} \right) \end{array} \right) \\ \bigcap \left(\begin{array}{c} ([1,6],[5,6]) \\ \bigcup \left(\begin{array}{c} ([1,9],[5,5]) \\ ([6,6],[1,6]) \\ ([1,1],[1,6]) \end{array} \right) \end{array} \right) \end{array} \right)} \tag{6.11}$$

□

6.4.5 Filtering Algorithm

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

Example 10 *Returning to our running example, suppose now that the sweep-point kernel wants to adjust the lower bound of x_{51}. Fig. 6.9 (bottom) traces the steps performed by the algorithm when it walks from a lexicographically smallest position to the first feasible position of o_5. The result is that the lower bound of x_{51} is adjusted to 5.*

□

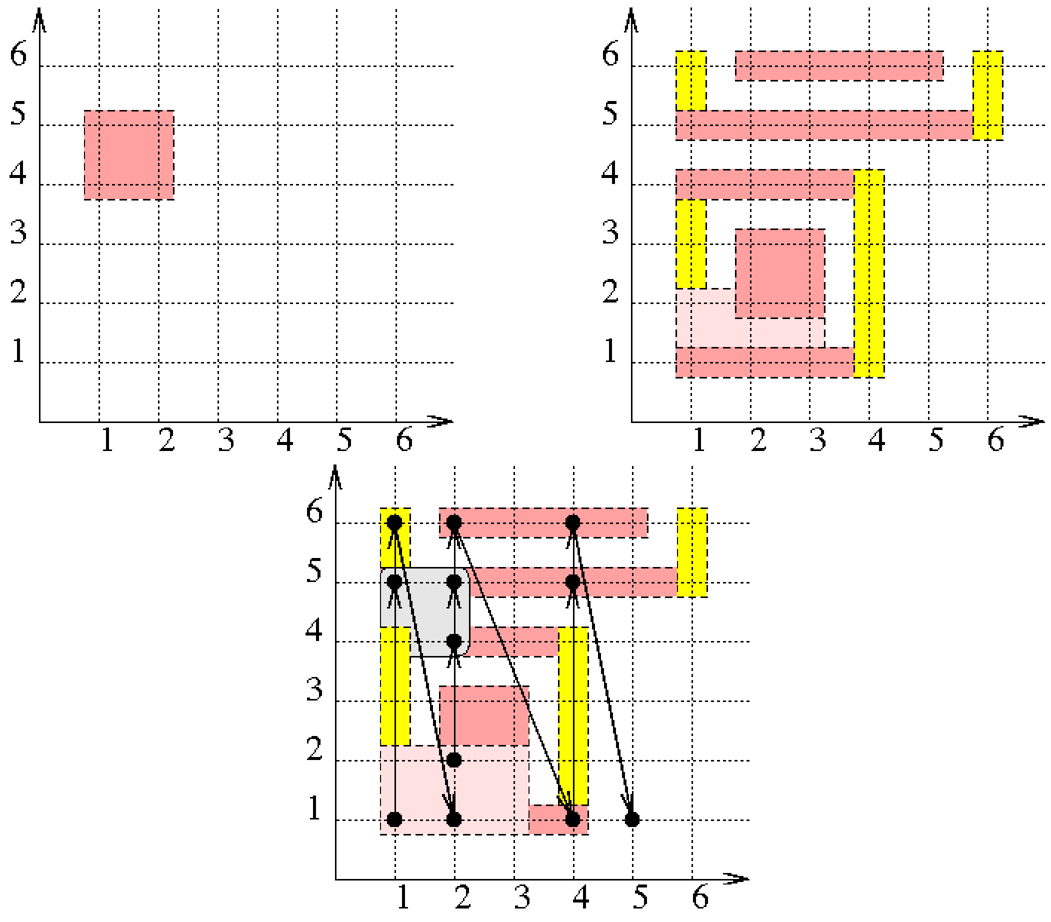


Figure 6.9: Running example: forbidden boxes generated by (6.10) (top left) and by (6.11) (top right). Sequence of candidate positions explored by the sweep-based algorithm in order to reach the feasible position $(5, 1)$ (bottom). The only purpose of using different colors and shadows of grey is to show the borders of the forbidden boxes.

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

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

6.5 Polymorphism

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

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

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

Polymorphic shifted boxes. With polymorphic objects, the expanded sentences of the rule language will mention attributes of shifted boxes, where the values of those attributes depend on the shape id. To deal with this complication, we introduce for polymorphic objects o a virtual $pbox[j]$ attribute, which stands for the j^{th} shifted box that has the same shape id as o. Thus a *pbox* attribute behaves like a shifted box but with nonground attributes that have evaluable lower and upper bounds, which is precisely what is needed in order to use the necessary conditions (Table 6.1) and pruning rules (6.8). Phase 1 of the rewrite process introduces *pbox*es when it encounters an expression `sboxes`($[\texttt{o}.sid]$) and o is polymorphic. Assuming that each possible shape of o consists of the same number, n, of shifted boxes, the expression is rewritten to $[o.pbox[1], \ldots, o.pbox[n]]$. Thus the requirement that n be fixed is a restriction of the approach.

Propagating to $o.sid$. We take the approach of treating variable $o.sid$ as the $(k+1)^{th}$ dimension, where the sweep-based algorithm treats the $(k+1)^{th}$ dimension as an *assignment dimension* — it seeks a witness for each value in the domain. For the compilation, all we have to change is to make the indexicals generate forbidden sets in $\mathbb{Z}^{k+1}$ instead of $\mathbb{Z}^k$, and to add two more types of generators of forbidden sets. Table 6.3 shows the updated table of generators of forbidden sets. Its rows 5 and 6 generate forbidden sets for the assignment dimension $k+1$, i.e. for $o.sid$.

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

Table 6.3: Mapping a *qfpa* r to a generator of forbidden sets, $F_o(r)$, for the object o, which may be polymorphic.

6.6 Experimental Results

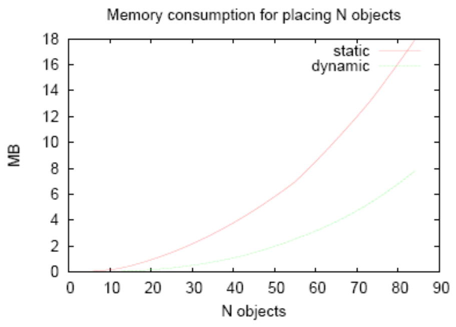
The *geost* constraint, including the rewriting, compilation, and sweep-based algorithms, have been implemented in Prolog using the global constraint programming API of SICStus Prolog 4 [9], compiled with `gcc -O2` version 4.0.2 on a 3GHz Pentium IV with 1MB of cache.

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

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

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

Fig. 6.10 summarizes the result. We find that the time and space complexity, static as well as dynamic, is $O(n^2)$. The coefficient of the n^2 term is rather high, we implement the sweep-based algorithm and all management of forbidden boxes in C, like in the previous version of *geost*, we expect this coefficient to decrease sharply.



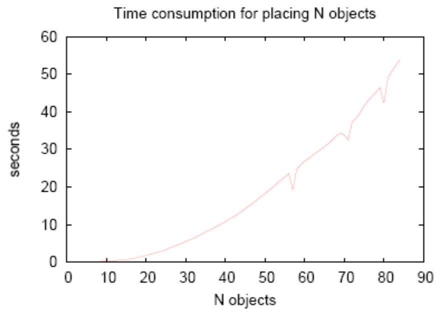


Figure 6.10: Memory and time consumption for placing n containers to be placed in seven bins without overlap. The **static** curve is the memory in use just after posting the constraint. The **dynamic** curve is the extra memory in use just after finding the first solution.

6.7 Conclusion and Open Issues

We have presented a global constraint that enforces rules written in a language based on arithmetic and first-order logic to hold among a set of objects. By rewriting the rules to QFPA formulas, we have shown how to compile them to k-indexicals. Finally, we have shown the forbidden sets generated by such indexicals can be aggregated by a sweep-based algorithm and used for filtering. Initial experiments support the feasibility of the approach. We would like to point out the following issues, which are areas of further work.

Generality. Our restriction that object attributes (except shape id and origin) must be ground is somewhat artificial, and we plan to lift it. The rewritten QFPA formulas would simply have more variables per object, and the sweep-based algorithm would deal not with a k- or $k+1$-dimensional *placement* space, but with an m-dimensional *solution* space, where m is the number of possibly nonground attributes per object. In particular, in order to deal with objects whose length in some dimension is a domain variable that occurs in some other constraint, the length and possibly the end-point would have to be expressed as nonground object attributes. Similarly, to treat the time dimension, we would add three nonground object attributes $start$, $duration$, and $completion$, as in [4], to be included in the solution space.

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

Theoretical properties. It has been shown in Prop. 1 and 2 that the `PKML/Rules2CP` rewriting system is confluent and Noetherian. Since our rule language is essentially a subset of

`Rules2CP`, the results apply to *geost* rules as well. A size bound on programs generated from `Rules2CP` is also known (see Prop. 3) and applies to *geost* provided that $\min$, $\max$ and cardinality is not used in the rules, since these operators can cause an exponential (for $\min$ and $\max$) resp. quadratic (for cardinality) [13] blow-up. Consequently, one can certainly construct pathological cases where the rewrite phases and/or runtime representation require huge amounts of memory. Even if, at this time, this was not really a problem for the instances and rules we experimented with [8], one way to manage the complexity of the rewrite phases is to apply simplifying rewrites, e.g. Phase 8, as eagerly as possibly. Another way could be to memoize patterns that have already been rewritten. Finally, common subexpression elimination will mitigate this problem.

6.8 Appendix A: Prolog Syntax

Table 6.4 shows the Prolog syntax of the various operators, objects, shifted boxes and attributes.

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

Table 6.4: Abstract syntax vs. Prolog syntax. `Atts` stands for possible additional attributes of the form `Name-Value`.

[8]They involved at most 100 objects.

6.9 Appendix B: Region Connection Calculus Rules

Region Connection Calculus (RCC-8, [28]) provides eight topological relations (i.e., *disjoint*, *meet*, *overlap*, *equal*, *covers*, *coveredby*, *contains*, *inside*) between two ground objects such that any two ground objects are in one and exactly one of these topological relations. Fig. 6.11 illustrates the meaning of each topological relation. In this section, we provide the corresponding rules in our language for these binary relations.

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

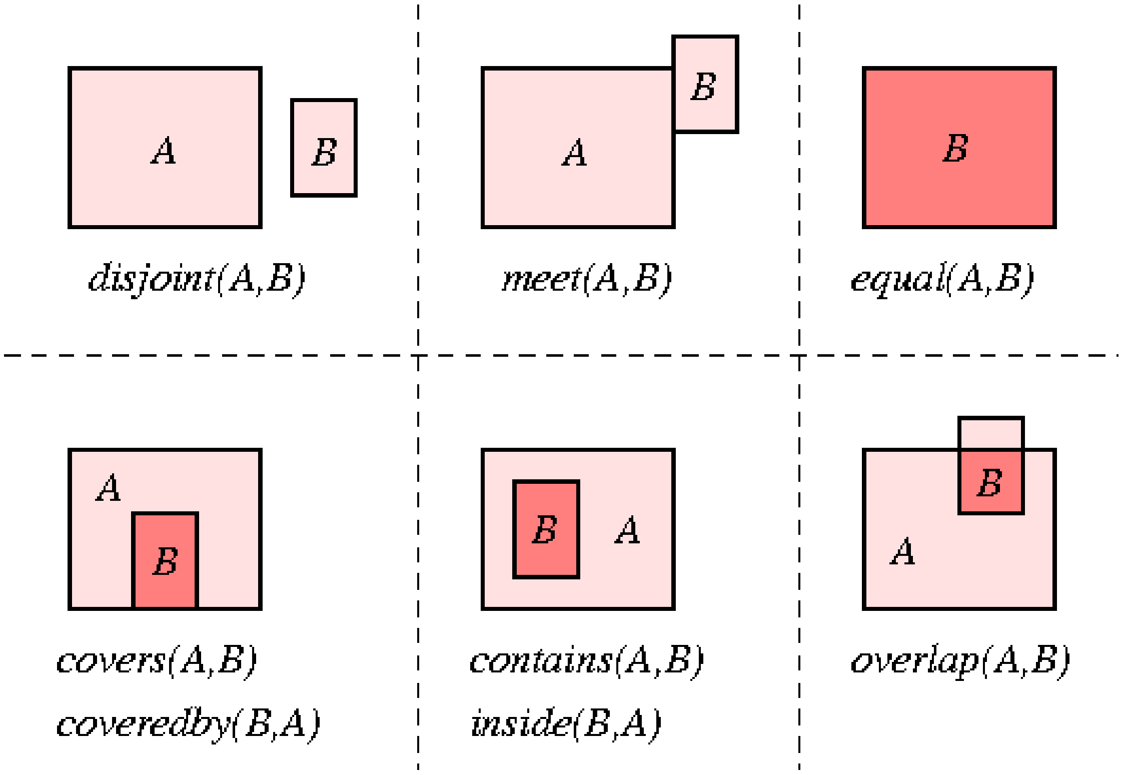


Figure 6.11: The eight topological relations of RCC-8

6.9.1 Rules for RCC-8 Relations between Two Shifted Boxes

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

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

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

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

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

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

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

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

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

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

6.9.2 Rules for RCC-8 Relations between Two Objects

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

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

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

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

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

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

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

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

6.10 Appendix C: A Real-Life Problem Instance

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

inside Each object is placed inside the container.

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

non_overlap The objects do not pairwise overlap.

stack_weight A heavier object cannot be piled on top of a lighter one.

stack_oversize For any two objects in a pile, the overhang can be at most 10 units.

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

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

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

General macros.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Business rules: putting all the rules together.

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

6.11 Appendix D: A Packing-Unpacking Problem

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

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

Definition 5 *Given a list* OIDs *of identifiers of objects of the* geost *constraint and a observation place, specified by a* dimension Dim *(an integer between* 1 *and* k*) and a* direction Dir *(*0 *or* 1*), the* $\mathit{visible}(\mathrm{OIDs}, \mathrm{Dim}, \mathrm{Dir})$ *constraint holds if, for all objects o mentioned in* OIDs, at least one surface of each shifted box associated with o is entirely visible from the specified observation place $\langle \mathrm{Dim}, \mathrm{Dir} \rangle$ at time $o.\mathit{start}$[9] as well as at time $o.\mathit{completion} - 1$.[10]

Definition 6 *Consider two distinct objects* o *and* o' *of the* $\mathit{visible}(\mathrm{OIDs}, \mathrm{Dim}, \mathrm{Dir})$ *constraint (i.e.,* $\mathrm{o}, \mathrm{o}' \in \mathrm{OIDs}$*) as well as an observation place defined by the pair* $\langle \mathrm{Dim}, \mathrm{Dir} \rangle$*. The object* o is masked by *the object* o' *according to the observation place* $\langle \mathrm{Dim}, \mathrm{Dir} \rangle$ *if there exist two shifted boxes* s *and* s' *respectively associated with* o *and* o' *such that conditions* **A**, **B**, **C** *and* **D** *all hold:*

A $\mathrm{o}.\mathit{duration} > 0 \wedge \mathrm{o}'.\mathit{duration} > 0 \wedge \mathrm{o}.\mathit{completion} > \mathrm{o}'.\mathit{start} \wedge \mathrm{o}'.\mathit{completion} > \mathrm{o}.\mathit{start}$ *(i.e., the time intervals associated with* O *and* o' *intersect).*

B *Discarding dimension* Dim*,* s *and* s' *intersect in the other dimensions (i.e., objects o and* o' *are in vis-à-vis).*

C *In dimension* Dim*,* o *and* o' *are ordered in the wrong way according to direction* Dir.

D *At least one of the two instants respectively corresponding to the start time of* o *and to the completion time of* o *is located within interval* $[\mathrm{o}'.\mathit{start}, \mathrm{o}'.\mathit{completion}]$.

Our Prolog implementation solved this problem instance in 7.5 CPU second and about 2.2 megabytes of memory. The rules generated 1744 k-indexicals with a total of 65456 virtual instructions. During the search the sweep-point kernel was applied 4502 times. The result is shown in Fig. 6.12. The four parts of the figure respectively correspond to the successive states of the bin (i.e., we have four time intervals):

top Initially, rectangles 1 to 16 enter the bin.

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

[10] Again, we assume that all objects for which the completion time equals $o.\mathit{completion}$ are transparent.

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

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

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

We now give the encoding of the problem.

Shorthands and invariants for space and time.

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

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

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

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

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

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

Non-overlapping constraints considering both space and time.

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

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

Visibility rules.

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

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

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

Business rules: putting all the rules together.

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

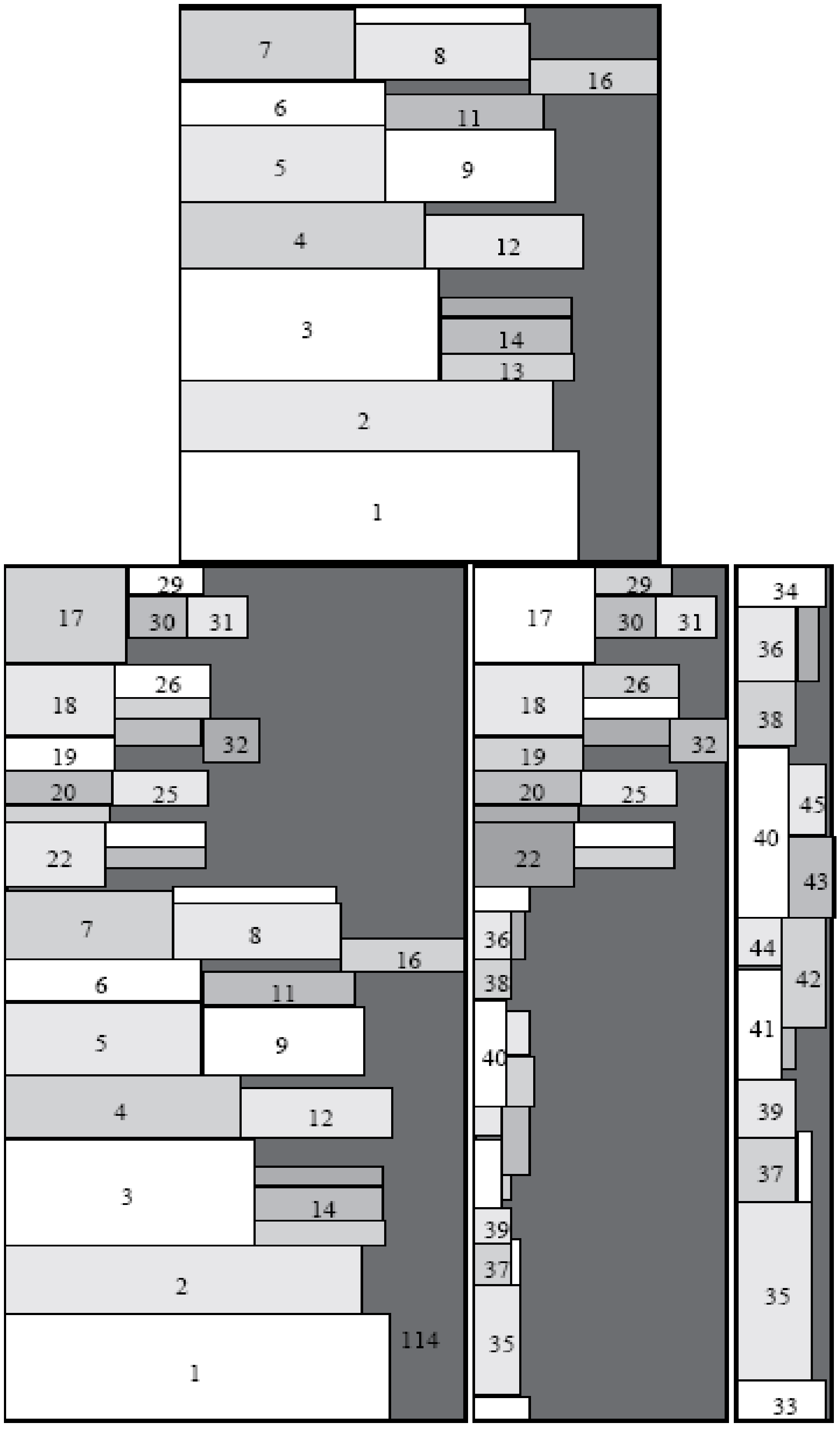


Figure 6.12: Solution to the packing-unpacking problem

Bibliography

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

[2] Krzysztof Apt and Mark Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, 2006.

[3] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. 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.

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

[5] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.

[6] N. Beldiceanu, P. Flener, and X. Lorca. Combining tree partitioning, precedence, and incomparability constraints. *Constraints Journal*, 2008. To appear, available at `http://www.springerlink.com/content/08p0h427w7n22681/`.

[7] M. Benedetti, A. Lallouet, and J. Vautard. QCSP made practical by virtue of restricted quantification. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 38–43, 2007.

[8] B. Carlson. *Compiling and Executing Finite Domain Constraints*. PhD thesis, Uppsala University, 1995.

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

[10] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Programming Languages: Implementations, Logics, and Programming*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997.

[11] P. Codognet and D. Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.

[12] Maria Garcia de la Banda, Kim Marriott, Reza Rafeh, and Mark Wallace. The modelling language Zinc. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming* CP*'06)*, pages 700–705. Springer-Verlag, 2006.

[13] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–25, 2006.

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

[15] François Fages, Sylvain Soliman, and Rémi Coolen. CLPGUI: a generic graphical user interface for constraint logic programming. *Journal of Constraints, Special Issue on User-Interaction in Constraint Satisfaction*, 9(4):241–262, October 2004.

[16] V. Ganesh, S. Berezin, and D.L. Hill. Deciding presburger arithmetic by model checking and comparisons with other methods. In *Proc. FMCAD'02*, volume 2517 of *LNCS*, pages 171–186. Springer, 2002.

[17] Business Rules Group. The business rules manifesto, 2003. Business Rules Group `http://www.businessrulesgroup.org/brmanifesto.htm`.

[18] Rémy Haemmerlé and François Fages. Modules for Prolog revisited. In *Proceedings of International Conference on Logic Programming ICLP 2006*, number 4079 in Lecture Notes in Computer Science, pages 41–55. Springer-Verlag, 2006.

[19] W. Harvey and P. J. Stuckey. Constraint representation for propagation. In M. Maher and J.-F. Puget, editors, *Proc. CP'98*, volume 1520 of *LNCS*, pages 235–249. Springer, 1998.

[20] P. Van Hentenryck and Y. Deville. The *cardinality* operator: a new logical connective in constraint logic programming. In *Int. Conf. on Logic Programming (ICLP'91)*. MIT Press, 1991.

[21] Dieter Hofbauer. Termination proofs by multiset path orderings imply primitive recursive derivation lengths. *Theoretical Computer Science*, 105(1):129–140, 1992.

[22] Jinbo Huang and Adnan Darwiche. The language of search. *Journal of Artificial Intelligence Research*, 29:191–219, 2007.

[23] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.

[24] Zohar Manna. *Lectures on the Logic of Computer Programming*. Number 0031 in CBMS-NSF regional conference series in applied mathematics. SIAM, 1980.

[25] Jakob Puchinger, Peter J. Stuckey, Mark Wallace, and Sebastian Brand. From high-level model to branch-and-price solution in g12. In *Proceedings of CPAIOR'08*, Lecture Notes in Computer Science, Paris, France, 2008. Springer-Verlag.

[26] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[27] Reza Rafeh, Maria Garcia de la Banda, Kim Marriott, and Mark Wallace. From Zinc to design model. In *Proceedings of PADL'07*, pages 215–229. Springer-Verlag, 2007.

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

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

[30] G. Tack, C. Schulte, and G. Smolka. Generating propagators for finite set constraints. In F. Benhamou, editor, *Proc. CP'2006*, volume 4204 of *LNCS*, pages 575–589. Springer, 2006.

[31] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[32] P. Van Hentenryck, V. Saraswat, and Y. Deville. Constraint processing in cc(FD). unpublished manuscript, Computer Science Department, Brown University, 1991.

[33] Pascal Van Hentenryck. *The OPL Optimization programming Language*. MIT Press, 1999.

[34] Pascal Van Hentenryck and Laurent Michel. *Constraint-based Local Search*. MIT Press, 2005.

[35] Victor Vianu. Rule-based languages. *Annals of Mathematics and Artificial Intelligence*, 19(1-2):215 – 259, March 1997.