

# Description and Representation of the Problems selected for the First International Constraint Satisfaction Solver Competition

Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre

CRIL-CNRS FRE 2499,

Université d'Artois

Lens, France

{*boussemart, hemery, lecoutre*}@cril.univ-artois.fr

**Abstract.** In this paper, we present the problems that have been selected for the first international competition of CSP solvers. First, we introduce a succinct description of each problem and then, we present the two formats that have been used to represent the CSP instances.

## 1 Introduction

For the first international competition of CSP solvers which was held as part of the CPAI'2005 workshop (<http://cpai.ucc.ie/05/CPAI.html>), it has been decided to only deal with constraint networks involving finite domains and constraints defined in extension. It means that each domain corresponds to a finite set of values, and that each constraint is explicitly defined using a list of allowed or unallowed tuples.

In order to avoid any ambiguity, we briefly introduce constraint networks. A constraint network consists of a finite set of variables such that each variable  $X$  has an associated domain  $\text{dom}(X)$  denoting the set of values allowed for  $X$ , and a finite set of constraints such that each constraint  $C$  has an associated relation  $\text{rel}(C)$  denoting the set of tuples allowed for the variables  $\text{vars}(C)$  involved in  $C$ . A solution to a constraint network is an assignment of values to all the variables such that all the constraints are satisfied. A constraint network is said to be satisfiable if it admits at least a solution. The Constraint Satisfaction Problem (CSP), whose task is to determine whether or not a given constraint network is satisfiable, is NP-complete. A constraint network is also called CSP instance.

In this paper, we first describe the pool of problems adopted for the competition and then we introduce the formats that have been proposed to represent them. We have to mention the participation of Radoslaw Szymanek, Marc van Dongen and Rick Wallace who kindly provide us with the description of some problems. Also, note that the two formats described in this paper are the results of fruitful discussions by all members of the organizing committee.

## 2 Pool of problems

The initial problem pool created by organizers contained some amount of diversified problems. This pool has been made available about 6 weeks before the deadline fixed for entering the competition. However, one important task was to introduce instances of which contestants will not be aware, so they can not tune their solvers to the problem pool. This is the reason why it was decided that two independent (not competing) members of the organizing committee, Radoslaw Szymanek and Mark Hennessy, will decide about the shape of the final problem pool. They had to secretly decide what instances will be used in the competition. Therefore, they decided to replace part of the initial pool with some new instances and, in addition, they added instances within new problem classes. Finally, contestants were also allowed to contribute up to ten instances.

Below, we describe the different problems selected for the competition. An *initial* series of instances will refer to a selected series that was down-loadable before the beginning of the competition whereas an *original* series will refer to a secret series that has been generated by Radoslaw Szymanek and Mark Hennessy. The problems that were contributed by the contestants will be referred to as *contestant's* problems.

### 2.1 Binary problems

First, we introduce (in lexicographic order) the binary problems. Any instance generated from a binary problem only involves binary constraints.

**3-SAT** A 3-SAT instance is a SAT instance such that each clause contains exactly 3 literals. Two initial series, denoted `ehi-85` and `ehi-90`, have been selected. These series correspond to 2 sets of 100 3-SAT instances (all unsat). These 3-SAT instances, originated in [3], have been converted into CSP instances using the dual method as described in [2].

**CRIL** F. Boussemart, F. Hemery and C. Lecoutre have submitted 8 contestant instances (5 binary ones, 3 non binary ones). The contestant series, denoted `cril`, is composed of 5 binary instances (all unsat). Two of them, `cril_b_unsat_2` and `cril_b_unsat_3`, are so-called composed random instances (see later in this subsection). Two other ones, `cril_b_unsat_4` and `cril_b_unsat_5`, are instances (called `qk_40_40_5_mul` and `qk_9_9_33_mul`), of the Queens-Knights problem [6]. The last one, `cril_b_unsat_1` has been obtained from an RLFAP instance by removing one minimal unsatisfiable core.

**Domino** The Domino problem corresponds to an undirected constraint graph with a cycle and a trigger constraint. It has been introduced in [22]. One initial series, denoted `domino1`, of 10 binary instances (all sat) has been selected.

**FAPP** The Frequency Assignment Problem with Polarization constraints (denoted FAPP) is the problem retained for the ROADEF'2001 challenge (<http://uma.ensta.fr/conf/roadef-2001-challenge/>). It is one extended subject of the CALMA European project (Combinatorial ALgorithms for Military Applications). Two initial series, denoted **fapp01** and **fapp02**, of binary instances have been selected. **fapp01** is composed of 11 instances (7 sat, 4 unsat) and **fapp02** is composed of 11 instances (9 sat, 2 unsat).

**Geom** The Geometric problem has been proposed by Rick Wallace. The geometric instances are a kind of random instances generated as follows. Instead of a density parameter, a "distance" parameter,  $dst$ , is used such that  $dst \leq \sqrt{2}$ . For each variable, two coordinates are chosen at random so the associated point lies in the unit square. Then for each variable pair,  $(x,y)$ , if the distance between their associated points is less than or equal to  $dst$ , the arc  $(x,y)$  is added to the constraint graph. Constraint relations are created in the same way as they are for homogeneous random CSP instances. In the geometric problem generator developed by Rick Wallace, after adding all arcs consistent with the distance criterion, connected components are checked. If there is more than 1 component, a connected graph is iteratively created starting with the component whose variables are labeled "1", finding the 'nearest' variable not in that component, connecting it to the variable that is 'closest' to it, and giving all variables in the new component a label "1". The idea is to create a connected graph that is as close as possible to the original distance criterion. One initial series, denoted **geom**, of 100 binary instances (92 sat, 8 unsat) has been selected.

**Job-Shop** The job-shop scheduling problem is the task of finding a schedule that minimizes the overall completion time of  $n$  jobs that require some shared resources. Four initial series, denoted **js-e0ddr1**, **js-e0ddr2**, **js-enddr1** and **js-ewddr2** of 10 binary instances (all sat) have been selected as well as one initial series, denoted **js-enddr2**, of 6 binary instances (all sat). These instances have been proposed by N. Sadeh and experimented in [18]. Note that we were unluckily unable to collect 14 instances of the original pool (as corresponding files are corrupted).

**Latin Square** The Latin square problem is the task of putting different values (taken from 0 to  $n-1$ ) in the cells of a square matrix of size  $n$  such that all rows and columns contain different values. One initial series, denoted **lsq-dg1**, of 5 binary instances (2 sat, 3 unsat) has been selected. Note that, for these instances, all diagonals (including broken ones) must also contain different values.

**Marc** One contestant series, denoted **marc**, of 10 binary instances (5 sat, 5 unsat) has been submitted by Marc van Dongen. These instances were contributed to point out that constraint representation does not come for free. All proposed

instances, which are large but trivially solvable, fall into two categories: satisfiable **sat** and unsatisfiable **unsat**. Within each category, the generated random instances have  $n$  variables,  $d = n$  values per domain,  $m = n(n - 1)/2$  binary constraints (corresponding to a complete constraint graph) and a tightness of 0.5 for the satisfiable instances and  $0.5 - 1/n^2$  for the unsatisfiable ones. Five classes corresponding to  $n \in \{80, 84, 88, 92, 96\}$  have been considered. The largest size had to be 96 because it wasn't possible to convert any larger instances to XML due to limitations of the conversion tool. Had it been possible to convert larger instances then larger classes would have been proposed. The satisfiable instances have been generated by picking a random number  $r \in \{1, \dots, n\}$  and by building each binary constraint as follows: the tuple  $\langle r, r \rangle$  was first added to the list of supports and then random tuples were added until the tightness was 0.5. This makes  $\langle r, \dots, r \rangle$  a trivial solution. The unsatisfiable instances are also constructed at random and made unsatisfiable by embedding an "unsatisfiable chain."

**Pigeons** The Pigeons problem is the task of putting  $n$  pigeons into  $n-1$  boxes, one pigeon per box. One initial series, denoted **pigeons**, of 20 binary instances (all **unsat**) has been selected.

**QCP / QWH** The Quasi-group Completion problem (QCP) is the task of determining whether the remaining entries of the partial Latin square can be filled in such a way that we obtain a complete Latin square, ie. a full multiplication table of a quasi-group [10]. The Quasi-group With Holes problem (QWH) is a variant of the QCP as instances are generated in such a way that they are guaranteed to be satisfiable [10]. Nineteen binary instances (all **sat**) of two initial series [6], denoted **bqwh15-106** and **bqwh18-141**, of balanced QWH instances have been selected.

To generate original series, a QCP/QWH generator was used. Orders 10, 15, 20, and 25 were chosen in order to vary the difficulty of the instances. Radoslaw Szymanek has implemented in JaCoP (for a related reference about this solver, see [13]) its own model, consisting of permutation constraints with GAC scheme and smallest domain first search to determine the difficulty of the problem instances. Each class originally contained one thousand of randomly generated instances from which the easiest ones and the hardest ones were hand picked for the competition. The initial model consisting of permutation constraints was translated into binary inequality constraints in order to obtain concise extensional representation. This translation makes it harder to recover the initial coarse grain structure of the instance and any solver which can do it efficiently will gain advantage over other solvers.

In addition to QWH/QCP problems, the *perturbated* versions of those problem classes were added. The instances within perturbated classes are easily generated using the generator mentioned above. The perturbation changes the domains of variables. In the original problem, all variables are assumed to have a

domain from 0 to  $n-1$ , where  $n$  specifies the order of the problem. In the perturbed problem, all variables of unassigned elements are assumed to have a domain from 1 to  $n$ . However, if the generator preassigned a square to value zero then such assignment was allowed. This setting transforms some of the permutation constraints into an all-different constraint since the number of values was larger than the number of variables. This small perturbation increases the difficulty noticeably for the model originally used to assess the difficulty of the instances. This increased difficulty can be partially explained by the fact that all-different constraint is less tight than permutation constraint which significantly decreases the amount of propagation done at each search stage. Radoslaw Szymanek and Mark Hennessy decided to use those instances to see if the instances remain hard for different solvers. Unfortunately, some of the orders of perturbed instances could not be included due to time shortage to generate large initial pool of random instances to be used during human selection.

**Queen Attacking** The Queen Attacking problem is the task of putting a queen and the  $n^2$  numbers  $1, \dots, n^2$ , on a  $n \times n$  chessboard so that no two numbers are on the same cell, any number  $i + 1$  is reachable by a knight move from the cell containing  $i$  and the number of cells containing a prime number that are not attacked by the queen is 0 (for satisfaction). See prob029 at <http://www.csplib.org>. One initial series, denoted **qa1**, of 8 binary instances (6 sat, 2 unsat) has been selected.

**Queens Knights** The Queens-Knights problem is the task of putting on a chessboard of size  $n \times n$ ,  $q$  queens and  $k$  knights such that no two queens can attack each other and all knights form a cycle (when considering knight moves) [6]. In one version of this problem (identified by “add”), a square of the chessboard can be shared by both a queen and a knight and in another one (identified by “mul”), it is not allowed. One initial series, denoted **qk1**, of 18 binary instances (all unsat) has been selected.

**Random instances (Model RB)** Four “standard” models, denoted A, B, C and D, have been introduced to generate random binary CSP instances. However, [1] have identified a shortcoming of all these models. Indeed, they prove that random instances generated using these models suffer from (trivial) unsatisfiability as the number of variables increases. To overcome the deficiency of these standard models, [20,21] have proposed two revised models, called RB and RD. Eight initial series, denoted **frbN-D** where **N** and **D** respectively correspond to the number of variables and the uniform size of domains, of 5 binary instances generated following model RB and forced to be satisfiable have been selected. These instances are particularly interesting as they are hard to solve. For more information about model RB and forced satisfiable instances, see [19], and about the selected binary instances, see <http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/benchmarks.htm>.

**Random instances (Model B)** There were also five original series, denoted `random-N-D-M` where  $N$ ,  $D$  and  $M$  respectively correspond to the number of variables, the uniform size of domains and the number of binary constraints, of 10 random instances generated according to Model B. The instances were generated using Marc van Dongen’s tool for generating problems in Table format. The tool is based on Frost *et al.*’s random problem generator which can be downloaded from Christian Bessière’s website. Given  $M$ , the critical tightness  $T$  was computed using the formula  $T = D^2 \times \left(1 - D^{-1/(M'(N-1))}\right)$ , where  $M'$  is the density on a scale from 0 to 1, i.e.  $M' = 2M/(N(N-1))$ . For the competition  $N = D \in \{23, \dots, 27\}$  and  $M = N(N-1)/2$  (which corresponds to complete constraint graphs). Within the class `random-N-D-M`, an instance called `random-N-D-M-T-R` was generated using seed  $R$ . The seeds were selected by Radoslaw Szymanek.

**Random instances (composed)** Nine initial series of 10 random binary CSP instances have been selected. The series `composed-25-10-20` contains 10 satisfiable instances whereas all other series contains 10 unsatisfiable instances. Each such generated instance is composed of a main (under-constrained, here) fragment and some auxiliary fragments, each of which being grafted to the main one by introducing some binary constraints. Such classes have been introduced in [14] and related instances have been experimented in [12].

**RLFAP** The Radio Link Frequency Assignment Problem (RLFAP) is the task of assigning frequencies to a number of radio links in such a manner as to simultaneously satisfy a large number of constraints and use as few distinct frequencies as possible. In 1993, the CELAR (the French “Centre d’Electronique de l’Armement”) built a suite of simplified versions of Radio Link Frequency Assignment Problems starting from data on a real network. These benchmarks have been made available to the public in the framework of the European EU-CLID project CALMA (Combinatorial Algorithms for Military Applications). For more information, see [7]. Five initial series of binary instances have been selected. The two series, denoted `rlfapScens` and `rlfapGraphs`, respectively contain 11 instances (6 sat, 5 unsat) and 14 instances (8 sat, 6 unsat). The three other ones, denoted `rlfapModScens`, `rlfapModGraphs` and `rlfapScens11`, respectively contain 14 instances (5 sat, 9 unsat), 12 instances (6 sat, 6 unsat) and 9 instances (all unsat), and correspond to instances for which some constraints have been deleted or/and some frequencies removed as described in [4].

**Towers of Hanoi** The problem of the towers of Hanoi [11, pp. 1–4] is the task of transferring an entire tower represented by  $n$  disks initially stacked in increasing size on one of three pegs to one of the two other pegs, moving only one disk at a time and never a larger one onto a smaller. One initial series, denoted `hanoi1`, of 5 binary instances (all sat) has been selected.

## 2.2 Non binary problems

**All-interval Series** The all-interval series problem is the task of finding a vector  $s = (s_1, \dots, s_n)$ , such that  $s$  is a permutation of  $\{0, 1, \dots, n-1\}$  and the interval vector  $v = (|s_2 - s_1|, |s_3 - s_2|, \dots, |s_n - s_{n-1}|)$  is a permutation of  $\{1, 2, \dots, n-1\}$ . See prob007 at <http://www.csplib.org>. One initial series, denoted `nb-allSeries1`, of 20 instances (all sat) involving both binary and ternary constraints has been selected.

**Black Hole** The Black Hole problem is the task of moving all cards of 17 fans of 3 cards each to the center pile, the Black hole, which initially only contains the Ace of Spades. One original series, denoted `BH-4-4`, is composed of 10 binary instances (all sat). To generate these instances (see [9]), as the instances for 52 card deck or 28 card deck using simple extensional oriented constraint model seemed to be out of reach for any reasonable quality solver, we decided to use card decks consisting of only 16 cards, which have much smaller search spaces. We would like to thank Barbara Smith for supplying us with these small instances.

**Chessboard Coloration** The chessboard coloration problem is the task of coloring all squares of a chessboard composed of  $r$  rows and  $c$  columns. There are exactly  $n$  available colors and the four corners of any rectangle extracted from the chessboard must not be assigned the same color. One initial series, denoted `nb-cc1`, of 11 non binary instances (5 sat, 6 unsat) involving constraints of arity 4 has been selected.

**CRIL** The contestant series, denoted `nb-cril`, is composed of 3 non binary instances. Two of them, `cril_unsat_nb.6` and `cril_unsat_nb.7`, are instances of the allocation problem defined in [15] and the third one, `cril_sat_nb.0` is an instance of the Radar Surveillance problem as defined in [6].

**Golomb Ruler** The Golomb Ruler problem is the task of putting  $n$  marks on a ruler of length  $l$  such that the distance between any two pairs of marks is distinct. See prob006 at <http://www.csplib.org>. One initial series, denoted `nb-gr1`, of 10 instances (5 sat, 5 unsat) involving both binary and ternary constraints has been selected. One original series, denoted `nb-Golomb`, of 20 non binary instances (10 sat, 10 unsat) has also been generated. The optimal Golomb rulers of size 3 to 12 were used to create the new instances. The sat instances were fixed to the optimal value for a given Golomb Ruler while the unsat instances were created in two different ways: either by constraining the length of the ruler to one below the optimal value, or by removing all possible values taken by one of the variables in all optimal rulers. The symmetry breaking constraints were included for the latter method to reduce the amount of values removed from the variable domains.

**Ramsey** The Ramsey problem is the task of coloring, using  $k$  colors, the edges of a complete graph involving  $n$  nodes in such a way that there is no monochromatic triangle in the graph (i.e., in any triangle at most two edges have the same color). See prob017 at <http://www.csplib.org>. Two initial series, denoted `nb-ramsey3` and `nb-ramsey4`, of respectively 8 instances (4 sat, 4 unsat) and 5 instances (5 sat) involving ternary constraints have been selected.

**Random instances (Model RB)** Three initial series of model RB, denoted `nb-random-3-N-D-M-T-forced` where N, D, M and T respectively correspond to the number of variables, the uniform size of domains, the number of ternary constraints and the tightness of each constraint, of 10 ternary instances of model RB forced to be satisfiable have been selected. These instances are particularly interesting as they are hard to solve. For more information about model RB and forced satisfiable instances, see [19].

**Renault** The initial series, denoted `renault`, which is simply composed of 1 non binary satisfiable instance has been selected (after merging constraints with similar scopes). This is a CSP instance from a Renault Megane configuration problem that was converted from symbolic to numeric domains.

**Schurr's Lemma** The Schurr's lemma problem is the task of putting  $n$  balls labeled from 1 to  $n$  into 3 boxes so that for any triple of balls  $(x,y,z)$  with  $x+y=z$ , not all are in the same box. See prob015 at <http://www.csplib.org>. One initial series, denoted `nb-schurr`, of 10 instances (2 sat, 8 unsat) involving ternary constraints has been selected. The modified versions of the series (instances with suffix *mod*) have been introduced in [5].

**Travelling Salesperson** The Travelling Salesperson problem is the task, given a set of cities, of finding a tour of minimal length that traverses each city (only once). Two original series, denoted `nb-TSP-20-easy` and `nb-TSP-25-easy`, of 5 non binary instances (all sat) have been selected. Also, two original series, denoted `nb-TSP-20-hard` and `nb-TSP-25-hard`, of 10 non binary instances (all sat) have been selected. The number of cities has been set to 20 and 25 and instances have been generated by using a perturbed version of this problem. However, this perturbation made the original problem easier. Due to restrictions of constraint arity and the goal to minimize the size of the file used to describe the problem, it was decided to allow any number of travelling persons to travel between cities. This significantly simplifies the extensional description of the problem. The cost of the whole trip was fixed to the cost of the optimal trip using only one salesperson. A pool of one thousand randomly generated instances was used in selecting instances to the final pool.

### 3 Format Specification

We have proposed two formats to represent CSP instances:

- a XML format
- a table format

Clearly, these formats are low-level ones. It means that no freedom remains for modeling. More precisely, for each instance, domains, variables, relations and constraints are exhaustively defined. The objective of introducing these formats is to allow comparing different algorithms/solvers without any interaction with modeling and any possible interpretation of the instance (e.g., the order of the variables). It should not be confused with the objective of proposing a powerful modeling language as the high-level proposals dedicated to mathematical programming such as e.g., AMPL (<http://www.ampl.com>) and GAMS (<http://www.gams.com>) or dedicated to constraint programming such as e.g., NCL [23], OPL [17] and ESRA [8]. Remark that the specification language Z (<http://v1.users.org>) has also been used to build nice (high-level) problem models [16].

Current table and XML formats are strongly connected as it is possible to convert any instance from one format to the other one. The advantages of the table format are its simplicity (as it is quite easy to parse) and its conciseness<sup>1</sup>. The advantages of the XML format are its readability (as it allows generating structured documents) and the possibility to benefit from many tools related to XML.

#### 3.1 XML Representation

For the first competition of CSP solvers, we have focused on XML representation of constraint networks involving variables with finite domains and constraints given in extension. In order to manage without any ambiguity different possible extensions of this representation, we identify the current format proposed below with 1.1 (i.e., XML representation of CSP instances, version 1.1). Such extensions will include constraints given in intension, continuous domains, weighted CSP instances, etc. Finally, note that our approach involves using empty elements with several attributes in order to represent a domain, a variable, a relation and a constraint (as described below). Although this approach can be discussed, it has the advantage of being simple and of saving space.

Each CSP instance will be represented following the format given in Figure 1 where  $q$ ,  $n$ ,  $r$  and  $m$  respectively denote the number of distinct domains, the number of variables, the number of distinct relations and the number of constraints. Thus, each instance is defined by an XML element which is called *instance* and which contains five elements described in the following subsections.

---

<sup>1</sup> On average, the size of representing one instance using this format is about 25% smaller than the size of representing the same instance using the XML format.

```

<instance>
  <presentation
    name = 'put here the instance name'
    description = 'put here the instance description'
    nbSolutions = 'put here the number of solutions'
    solution = 'put here a solution'
    format = 'put here the XML format of the instance'
  />
  <domains nbDomains='q'>
    <domain
      name = 'put here the domain name'
      nbValues = 'put here the number of values'
      values = 'domainDescription'
    />
    ...
  </domains>
  <variables nbVariable='n'>
    <variable
      name = 'put here the variable name'
      domain = 'put here the name of a domain'
    />
    ...
  </variables>
  <relations nbRelations='r'>
    <relation
      name = 'put here the name of the relation'
      domain = 'put here the domain of the relation'
      nbSupports | nbConflicts = 'put here either the number
        of supports or the number of conflicts'
      supports | conflicts = 'put here either the set
        of supports or the set of conflicts'
    />
    ...
  </relations>
  <constraints nbConstraints='m'>
    <constraint
      name = 'put here the name of the constraint'
      scope = 'put here the names of the variables
        involved in the constraint'
      relation = 'put here the name of the relation'
    />
    ...
  </constraints>
</instance>

```

Fig. 1. XML representation of a CSP instance

**Presentation** The XML element called *presentation* is empty and includes a set of attributes as follows:

```
<presentation
  name = 'put here the instance name'
  description = 'put here the instance description'
  nbSolutions = 'put here the number of solutions'
  solution = 'put here a solution'
  format = 'put here the XML format of the instance'
/>
```

Each attribute of *presentation* is optional and of type string as it only provides human-readable information. The attribute *nbSolutions* can be given an integer value denoting the total number of solutions of the instance, an expression of the form 'at least n' or '?. For example,

- *nbSolutions* = '0' indicates that the instance is unsatisfiable,
- *nbSolutions* = '3' indicates that the instance has exactly 3 solutions,
- *nbSolutions* = 'at least 1' indicates that the instance has at least 1 solution (and, hence, is satisfiable),
- *nbSolutions* = '?' indicates that it is unknown whether or not the instance is satisfiable,

The attribute *solution* indicates a solution if one exists and has been found.

The attribute *format* indicates which version of XML representation of CSP instances is used. The current version is denoted 1.1.

Remark that, in the context of the competition, the value of the attributes *name*, *description*, *nbSolutions* and *solution* will be set to '?'.

**Domains** The XML element called *domains* admits an attribute which is called *nbDomains* and contains some occurrences of an element called *domain*, one for each domain associated with at least one variable of the instance. The attribute *nbDomains* is of type integer and its value is equal to the number of occurrences of the element *domain*. Each element *domain* is empty and includes three attributes, called *name*, *nbValues* and *values*, as follows:

```
<domain
  name = 'put here the domain name'
  nbValues = 'put here the number of values'
  values = 'domainDescription'
/>
```

The value of the attribute *name* corresponds to the name of the domain, must not contain space characters and must be unique. The attribute *nbValues* is of type integer and its value is equal to the number of values of the domain. The value of the attribute *values* gives a description of the domain, i.e., the set of integer values that can be assigned to this variable. The description of a domain takes the form:

```

domainDescription ::= domainPiece ( ' ' domainPiece)*
<domainPiece> ::= integer |
                integer'..'integer

```

To summarize, a domain is defined from some pieces that correspond to single values and ranges of integer values. For example,

- values = '1 5 10' corresponds to the set {1, 5, 10} of integer values,
- values = '1..3 10..14' corresponds to the set {1, 2, 3, 10, 11, 12, 13, 14} of integer values.

**Variables** The XML element called *variables* admits an attribute which is called *nbVariables* and contains some occurrences of an element called *variable*, one for each variable of the instance. The attribute *nbVariables* is of type integer and its value is equal to the number of occurrences of the element *variable*. Each element *variable* is empty and includes two attributes, called *name* and *domain*, as follows:

```

<variable
  name = 'put here the variable name'
  domain = 'put here the name of a domain'
/>

```

The value of the attribute *name* corresponds to the name of the variable, must not contain space characters and must be unique. The value of the attribute *domain* gives the name of the associated domain. It must correspond to the value of the *name* attribute of a *domain* element.

**Relations** The XML element called *relations* admits an attribute which is called *nbRelationss* and contains some occurrences of an element called *relation*, one for each relation associated with at least a constraint of the instance. The attribute *nbRelations* is of type integer and its value is equal to the number of occurrences of the element *relation*.

Each element *relation* is empty and includes four attributes, called *name*, *domain*, and either *nbSupports* and *supports* or *nbConflicts* and *conflicts*. The value of the attribute *name* corresponds to the name of the relation, must not contain space characters and must be unique. The value of the attribute *domain* corresponds to a list of domains whose Cartesian product denotes the domain of the relation. It must correspond to a list of domain names where each name corresponds to the value of the *name* attribute of a *domain* element. A space character is used as a separator. The attribute *nbSupports* or *nbConflicts* is of type integer and its value is equal to the number of supports (if supports are given) or the number of conflicts (if conflicts are given) of the relation.

Finally, either we have an attribute called *supports* or an attribute called *conflicts*. The value of such attributes denotes a set of tuples which must be a subset of the Cartesian product denoting the domain of the relation. Note that

no separator is inserted between tuples and that each tuple corresponds to a list of numbers separated by commas and enclosed in parentheses.

A relation defined by its set of supports looks like:

```
<relation
  name = 'put here the name of the relation'
  domain = 'put here the domain of the relation'
  nbSupports = 'put here the number of supports'
  supports = 'put here the set of supports'
/>
```

A relation defined by its set of conflicts looks like:

```
<relation
  name = 'put here the name of the relation'
  domain = 'put here the domain of the relation'
  nbConflicts = 'put here the number of conflicts'
  conflicts = 'put here the set of conflicts'
/>
```

**Constraints** The XML element called *constraints* admits an attribute which is called *nbConstraints* and contains some occurrences of an element called *constraint*, one for each constraint of the instance. The attribute *nbConstraints* is of type integer and its value is equal to the number of occurrences of the element *constraint*.

Each element *constraint* is empty and includes three attributes, called *name*, *scope* and *relation* as follows:

```
<constraint
  name = 'put here the name of the constraint'
  scope = 'put here the names of the variables
          involved in the constraint'
  relation = 'put here the name of the relation'
/>
```

The value of the attribute *name* corresponds to the name of the constraint, must not contain space characters and must be unique. The value of the attribute *scope* denotes the set of variables involved in the constraint. It must correspond to a list of variable names where each name corresponds to the value of the *name* attribute of a *variable* element. A space character is used as a separator. For example, *scope* = 'X0 X4' corresponds to the variables involved in a binary constraint.

The value of the attribute *relation* corresponds to the name of the relation associated with the constraint. It must correspond to the value of the *name* attribute of a *relation* element.

```

<instance>
  <presentation
    name="4queens"
    description="This problem involves placing 4 queens
                on a chessboard"
    nbSolutions="at least 1"
    format="1.1"/>

  <domains nbDomains="1">
    <domain name="dom0" nbValues="4" values="1..4"/>
  </domains>

  <variables nbVariables="4">
    <variable name="X0" domain="dom0"/>
    <variable name="X1" domain="dom0"/>
    <variable name="X2" domain="dom0"/>
    <variable name="X3" domain="dom0"/>
  </variables>

  <relations nbRelations="3">
    <relation name="rel0" domain="dom0 dom0" nbConflicts="10"
      conflicts="(1,1)(1,2)(2,1)(2,2)(2,3)(3,2)(3,3)
                (3,4)(4,3)(4,4)"/>
    <relation name="rel1" domain="dom0 dom0" nbConflicts="8"
      conflicts="(1,1)(1,3)(2,2)(2,4)(3,1)(3,3)(4,2)
                (4,4)"/>
    <relation name="rel2" domain="dom0 dom0" nbConflicts="6"
      conflicts="(1,1)(1,4)(2,2)(3,3)(4,1)(4,4)"/>
  </relations>

  <constraints nbConstraints="6">
    <constraint name="C0" scope="X0 X1" relation="rel0"/>
    <constraint name="C1" scope="X0 X2" relation="rel1"/>
    <constraint name="C2" scope="X0 X3" relation="rel2"/>
    <constraint name="C3" scope="X1 X2" relation="rel0"/>
    <constraint name="C4" scope="X1 X3" relation="rel1"/>
    <constraint name="C5" scope="X2 X3" relation="rel0"/>
  </constraints>
</instance>

```

**Fig. 2.** 4 queens (XML format)

```

<instance>
  <presentation
    name="nonBinaryInstance"
    description="Illustrative instance"
    nbSolutions="at least 1"
    format="1.1"/>

  <domains nbDomains="3">
    <domain name="dom0" nbValues="7" values="0..6"/>
    <domain name="dom1" nbValues="3" values="1 5 10"/>
    <domain name="dom2" nbValues="10" values="1..5 11..15"/>
  </domains>

  <variables nbVariables="5">
    <variable name="X0" domain="dom0"/>
    <variable name="X1" domain="dom0"/>
    <variable name="X2" domain="dom1"/>
    <variable name="X3" domain="dom2"/>
    <variable name="X4" domain="dom0"/>
  </variables>

  <relations nbRelations="4">
    <relation name="rel0" domain="dom0 dom0" nbConflicts="7"
      conflicts="(0,0)(1,1)(2,2)(3,3)(4,4)(5,5)(6,6)"/>
    <relation name="rel1" domain="dom2 dom0" nbConflicts="25"
      conflicts="(1,0)(1,1)(1,2)(1,3)(1,4)(1,5)(1,6)(2,1)(2,2)
      (2,3)(2,4)(2,5)(2,6)(3,2)(3,3)(3,4)(3,5)(3,6)(4,3)(4,4)
      (4,5)(4,6)(5,4)(5,5)(5,6)"/>
    <relation name="rel2" domain="dom1 dom0" nbSupports="1"
      supports="(5,3)"/>
    <relation name="rel3" domain="dom0 dom1 dom2" nbSupports="17"
      supports="(0,1,3)(0,5,3)(0,10,12)(1,1,4)(1,5,2)(1,10,13)
      (2,1,5)(2,5,1)(2,10,14)(3,10,5)(3,10,15)(4,5,11)(4,10,4)
      (5,5,12)(5,10,3)(6,5,13)(6,10,2)"/>
  </relations>

  <constraints nbConstraints="5">
    <constraint name="C0" scope="X0 X1" relation="rel0"/>
    <constraint name="C1" scope="X3 X0" relation="rel1"/>
    <constraint name="C2" scope="X2 X0" relation="rel2"/>
    <constraint name="C3" scope="X1 X2 X3" relation="rel3"/>
    <constraint name="C4" scope="X1 X4" relation="rel0"/>
  </constraints>
</instance>

```

**Fig. 3.** Non binary instance (XML format)

**Some examples** In Figure 2, one can see the XML representation of the 4 queens instance where constraints are all represented by a set of conflicts. In Figure 3, one can see the XML representation of a CSP instance involving 5 variables and 5 constraints. It illustrates non binary constraints and constraints given by supports or conflicts. These constraints, given in extension, can be respectively more formally expressed by:

$C0: X0 \neq X1$   
 $C1: X3 - X0 \geq 2$   
 $C2: X2 - X0 = 2$   
 $C3: X1 + 2 = |X2 - X3|$   
 $C4: X1 \neq X4$

### 3.2 Table format

Now, we introduce the syntax of the **table** format proposed for the first competition of CSP solvers. Each CSP instance has been represented following the format given in Figure 4. We have just to mention that if the value of `<type>` is 0 then it means the tuples in the relation correspond to conflicts. If it is 1 then it means all the tuples in the corresponding relation are supports. Also, note that Space (' ': ASCII CODE 32) and Line Feed ('\n': ASCII CODE 10) characters act as separators.

```

<problem name>
<number of domain definitions>
<domain definition number>      <domain size>      <domain values>
.
.
<number of variables>
<variable number>      <domain definition number>
.
.
<number of relation definitions>
<relation definition number> <type> <arity> <domain definition numbers>
      <number of tuples> <tuples>
.
.
<number of constraints>
<arity>      <variables involved>      <relation definition number>
.
.

```

**Fig. 4.** Table representation of a CSP instance

```

4queens
1                                     /* Number of domain definitions */
0 4 1 2 3 4
4                                     /* Number of variables */
0 0
1 0
2 0
3 0
3                                     /* Number of relation definitions */
0 0 2 0 0 10 1 1 1 2 2 1 2 2 2 3 3 2 3 3 3 4 4 3 4 4
1 0 2 0 0 8 1 1 1 3 2 2 2 4 3 1 3 3 4 2 4 4
2 0 2 0 0 6 1 1 1 4 2 2 3 3 4 1 4 4
6                                     /* Number of constraints */
2 0 1 0
2 0 2 1
2 0 3 2
2 1 2 0
2 1 3 1
2 2 3 0

```

Fig. 5. 4 queens (Table format)

```

nonBinaryInstance
3                                     /* Number of domain lists */
0 7 0 1 2 3 4 5 6
1 3 1 5 10
2 10 1 2 3 4 5 11 12 13 14 15
4                                     /* Number of variables */
0 0
1 0
2 1
3 2
4                                     /* Number of relation definitions */
0 0 2 0 0 7 0 0 1 1 2 2 2 2 3 3 4 4 5 5 6 6
1 0 2 0 2 25 1 0 1 1 1 2 1 3 1 4 1 5 1 6 2 1 2 2 2 3 2 4
    2 5 2 6 3 2 3 3 3 4 3 5 3 6 4 3 4 4 4 5 4 6
    5 4 5 5 5 6
2 1 2 1 0 1 5 3
3 1 3 0 1 2 17 0 1 3 0 5 3 0 10 12 1 1 4 1 5 2 1 10 13
    2 1 5 2 5 1 2 10 14 3 10 5 3 10 15 4 5 11
    4 10 4 5 5 12 5 10 3 6 5 13 6 10 2
4                                     /* Number of constraints */
2 0 1 0
2 0 3 1
2 2 0 2
3 1 2 3 3

```

Fig. 6. Non binary instance (Table format)

**Some examples** In Figures 5 and 6, we present the table version of the two small examples introduced in Section 3.1. Note that comments are inserted to make the example more understandable. However, in the proposed table format comments are not allowed.

### 3.3 Format rules for the competition

Finally, one can notice that it is possible to convert any instance from one format to the other one. We have proposed a program called `checker` that allows validating and converting XML and table representations of CSP instances. In fact, a file representing a CSP instance is valid iff:

1. the number of (types of) domains corresponds to the specified number of domains
2. the number of values in each domain corresponds to the specified number of values (size)
3. each domain is given in ascending order and does not contain several occurrences of the same value
4. the number of variables corresponds to the specified number of variables
5. the domain associated with each variable is defined
6. the number of (types of) relations corresponds to the specified number of relations
7. the domain of each relation is well defined (that is to say, it corresponds to a list that denotes a Cartesian product from (types of) domains that are defined)
8. the number of tuples (supports or conflicts) in each relation corresponds to the specified number of tuples
9. each tuple belongs to the domain of the relation
10. each relation is given in ascending (lexicographic) order and does not contain several occurrences of the same tuple
11. the number of constraints corresponds to the specified number of constraints
12. the scope of each constraint is well defined (that is to say, it corresponds to a list of variables that are defined)
13. the relation associated with each constraint is defined
14. the domain that can be built from the scope of each constraint corresponds to the domain of the associated relation

Furthermore, in the context of the first competition, the following rules have also been adopted:

1. the name of the instance is "?" or only contains letters, digits or characters in  $\{'\_','\_'\}$
2. values in each domain are integers which belong to  $-2^{14}..2^{14}$
3. variables are numbered from 0 to  $n - 1$ , where  $n$  is the number of variables (In XML format, variables are denoted  $X_0..X_{n-1}$ ).
4. variables are given in ascending order of their numbers
5. the arity of each constraint is in the range 2..20
6. the scopes of any two different constraints must differ.

## 4 Conclusion

For the first international constraint satisfaction solver competition, the organizing committee has collected a large set of random, academic and real-world instances. We hope that this set will be useful to the CSP community as benchmarks for testing new methods/algorithms. On the other hand, we project, in the near future, to extend the current table and XML formats proposed here in order to deal with constraints defined in intention, weighted CSPs, global constraints, optimization etc.

## Acknowledgements

We would like to thank Radoslaw Szymanek for providing us with the description of the original series (QCP/QWH, Black Hole, Golomb Ruler, Travelling Salesperson), Marc van Dongen for the description of the Model B random series and the Marc contestant series, and Rick Wallace for the description of the geom series. Also, we would like to thank all members of the organizing committee for fruitful discussions about the formats to represent CSP instances.

This paper has been supported by the CNRS, the “programme COCOA de la Région Nord/Pas-de-Calais” and by the “IUT de Lens”.

## References

1. D. Achlioptas, L.M. Kirousis, E. Kranakis, D. Krizanc, M.S.O. Molloy, and Y.C. Stamatiou. Random constraint satisfaction: a more accurate picture. In *Proceedings of CP'97*, pages 107–120, 1997.
2. F. Bacchus. Extending Forward Checking. In *Proceedings of CP'00*, pages 35–51, 2000.
3. R.J. Bayardo and R.C. Shrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of AAAI'97*, pages 203–208, 1997.
4. C. Bessière, A. Chmeiss, and L. Sais. Neighborhood-based variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of CP'01*, pages 565–569, 2001.
5. C. Bessière, P. Meseguer, E.C. Freuder, and J. Larrosa. On forward checking for non binary constraint satisfaction. *Artificial Intelligence*, 141:205–224, 2002.
6. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
7. B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio Link Frequency Assignment. *Constraints*, 4(1):79–89, 1999.
8. P. Flener and B. Hnich. The syntax and semantics of ESRA. Technical report, Computing Science Department, Uppsala University, 2001.
9. I. Gent, C. Jefferson, I. Lynce, I. Miguel, P. Nightingale, B. Smith, and A. Tarim. Search in the patience game 'black hole'. Technical report, Cork Constraint Computation Centre, 2005.
10. C.P. Gomez and D. Shmoys. Completing quasigroups or latin squares: a structured graph coloring problem. In *Proceedings of Computational Symposium on Graph Coloring and Generalization*, 2002.

11. R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 1989.
12. N. Jussien, R. Debruyne, and P. Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Proceedings of CP'00*, pages 249–261, 2000.
13. K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(3):355–383, 2003.
14. C. Lecoutre, F. Boussemart, and F. Hemery. Backjump-based techniques versus conflict-directed heuristics. In *Proceedings of ICTAI'04*, pages 549–557, 2004.
15. S. Merchez, C. Lecoutre, and F. Boussemart. Abstraction de réseaux de contraintes. *Revue d'Intelligence Artificielle*, To appear 2006.
16. G. Renker and H. Ahriz. Building models through formal specification. In *Proceedings of CPAIOR'04*, pages 395–401, 2004.
17. ILOG SA, editor. *ILOG OPL Studio 3.7, Language Manual*. 2003.
18. N. Sadeh and M.S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86:1–41, 1996.
19. K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. A simple model to generate hard satisfiable instances. In *Proceedings of IJCAI'05*, pages 337–342, 2005.
20. K. Xu and W. Li. Exact phase transitions in random constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 12:93–103, 2000.
21. K. Xu and W. Li. Many hard examples in exact phase transitions with application to generating hard satisfiable instances. Technical report, CoRR Report cs.CC/0302001, 2003.
22. Y. Zhang and R.H.C. Yap. Making AC3 an optimal algorithm. In *Proceedings of IJCAI'01*, pages 316–321, Seattle WA, 2001.
23. J. Zhou. Introduction to the constraint language NCL. *Journal of Logic Programming*, 45(1-3):71–103, 2000.