XML Representation of Constraint Networks Format XCSP 2.1

Organising Committee of the
Third International Competition of CSP Solvers

Abstract

We propose a new extended format to represent constraint networks using XML. This format allows us to represent constraints defined either in extension or in intension. It also allows us to reference global constraints. Any instance of the problems CSP (Constraint Satisfaction Problem), QCSP (Quantified CSP) and WCSP (Weighted CSP) can be represented using this format.

A subset of this format will be used for the third international competition of CSP solvers which will be held during summer 2008 (deadline: May 10, 2008).

The release of this document is January 15, 2008.

Contents

1 Introduction

The Constraint Programming (CP) community suffers from the lack of a standardized representation of problem instances. This is the reason why we propose an XML representation of constraint networks. The Extensible Markup Language (XML) [18] is a simple and flexible text format playing an increasingly important role in the exchange of a wide variety of data on the Web. The objective of the XML representation is to ease the effort required to test and compare different algorithms by providing a common test-bed of constraint satisfaction instances.

One should notice that the proposed representation is low-level. More precisely, for each instance, domains, variables, relations (if any), predicates (if any) and constraints are exhaustively defined. The current format should not be confused with powerful modelling language such as the high-level proposals dedicated to mathematical programming - e.g. AMPL (`http://www.ampl.com`) and GAMS (`http://www.gams.com`) - or dedicated to constraint programming[1] - e.g. OPL [10], EaCL [16], NCL [19], ESRA [8], Zinc [6] and ESSENCE [9]. Nevertheless, we also project to extend this format in a near future to take into account higher level constructs.

In this document, we present an extension, denoted XCSP 2.1, of the format XCSP 2.0 which was used for the 2006 CSP solver competition. It aims at being a (hopefully good) compromise between readability, verbosity and structuration. More precisely, our objective is that the representation be:

- readable: thanks to XML, we think that it is the case. If you want to modify an instance by hand, you can do it without too many difficulties whereas it would be almost impossible with a tabular format. Only a few constructions require an a-priori knowledge of the format.
- concise: with the abridged version which doesn't use systematically XML tags and attributes, the proposed representation can be comparable in length to one that would be given in tabular format. This is important, for example, to represent instances involving constraints in extension.
- structured: because the format is based on XML, it remains rather easy to parse instances.

Roughly speaking, we propose two variants of this format:

- a fully-tagged representation,
- an abridged representation.

The first representation (tagged notation) is a full XML, completely structured representation which is suitable for using generic XML tools but is more verbose and more tedious to write for a human being. The second representation (abridged notation) is just a shorthand notation of the first representation

[1] Remark that the specification language Z (`http://vl.users.org`) has also been used to build nice (high-level) problem models [15]

which is easier to read and write for a human being, but less suitable for generic XML tools.

These two representations are equivalent and this document details the translation of one representation to another. Automatic tools to convert from one representation to another will be made available, and parsers will accept the two representations. This will allow human beings to use the shorthand notation to encode an instance while still being able to use generic XML tools.

2 Basic Components

This section describes how the most common kinds of information are represented in this XML format. This section only details the general data structures that are used in the description of instances. The way these structures are used to represent an instance is presented in the other sections of this document. As mentioned in the introduction, we present two representations for each structure. The first one is fully-tagged and the second one is abridged.

2.1 Identifiers and Integers

First, let us introduce the syntax of identifiers and integers. An identifier has to be associated with some XML elements, usually under the form of an attribute called *name*. An identifier must be a valid identifier according to the most common rules (start with a letter or underscore and further contain letters, digits or underscores). More precisely, identifiers and integers are defined (in BNF notation) as follows:

```
<identifier> ::= <letter> | "_" { <letter> | <digit> | "_" }
<integer>  ::= [ "+" | "-"]  <digit> {<digit>}
<letter> ::= "a".."z" | "A".."Z"
<digit> ::= "0".."9"
```

Of course, identifiers are case-sensitive.

2.2 Separators

2.2.1 Tagged notation

For the tagged version, we do not need to use any specific separator since the document is fully structured.

2.2.2 Abridged notation

For the abridged version, we sometimes need to employ separators. They are defined as follows:

```
<whitespace> ::= " " | "\t" | "\n" | "\r"
<separator> ::= <whitespace> | { <whitespace> }
```

2.3 Constants

Different kinds of constant can be used in the encoding of a CSP instance.

2.3.1 Tagged notation

Boolean constants are written using two special elements: `<true/>` and `<false/>`. Integer constants are written inside a `<i>` element (e.g. `<i>19</i>`).

Real constants are written inside a `<r>` element (e.g. `<r>19.5</r>`).

2.3.2 Abridged notation

Wherever it is legal to have a numerical constant, its value can be written directly without the enclosing tag (e.g. 19, 19.5). 19 is considered as an integer constant.

To avoid introducing reserved keywords, Boolean constants (`<true/>` and `<false/>`) cannot be abridged.

2.4 Intervals

2.4.1 Tagged notation

Intervals are represented by a `<interval>` element with two attributes: **min** and **max**. The **min** represents the minimal value of the interval (the lower bound) and the **max** attribute the maximal value (the upper bound). For example, `<interval min="10" max="13"/>` corresponds to the set $\{10, 11, 12, 13\}$.

2.4.2 Abridged notation

To represent an interval, on has just to write two constants (of the same type) separated by the sequence "..". For example, 10..13 corresponds to the set of integers $\{10, 11, 12, 13\}$.

2.5 Variables

Several constructs in the format have to reference variables. For simplicity, we consider here that the term variable refers to both effective and formal parameters of functions and predicates.

2.5.1 Tagged notation

The reference to a variable is represented by a `<var>` element with an empty body and a single attribute **name** which provides the identifier of the variable. For example, a reference to the variable $X1$ is represented by `<var name="X1"/>`.

2.5.2 Abridged notation

Wherever it is legal to have a `<var name="`*identifier*`"/>` element, this element can be replaced equivalently by *identifier*. For example, `X1` and `<var name="X1"/>` are two legal and equivalent ways to refer to the variable `X1`.

2.6 Formal parameters

A predicate or function in this XML format must first define the list of its formal parameters (with their type). Then, these formal parameters can be referenced with the notations defined in section 2.5.

In both tagged and abridged representations, formal parameters are defined in the body of a `<parameters>` element.

2.6.1 Tagged notation

In the tagged notation, each formal parameter is defined by a `<parameter>` element with two attributes. The attribute **name** defines the formal name of the parameter and the attribute **type** defines its type. For example, `<parameter name="X0" type="int">` defines a parameter named `X0` of integer type.

2.6.2 Abridged notation

Wherever it is legal to have a `<parameter>` element, a formal parameter can be written in abridged notation by its type followed by whitespace followed by the parameter name (as in C and Java programming language). Consecutive parameters must be separated by whitespace. For example, `int X0` is the abridged representation of `<parameter name="X0" type="int">`.

The syntax of the formal parameters list is described by the following grammar (in BNF notation):

```
<formalParameters> ::= [<formalParametersList>]
<formalParametersList> ::= <formalParameter>
                         | <formalParameter> <separator> <formalParametersList>
<formalParameter> ::= <type> <separator> <identifier>
<type>  ::= "int"
```

2.7 Lists

A list is an array of (possibly heterogeneous) objects. The order of objects in the list is significant (but this order may be deliberately ignored when needed).

2.7.1 Tagged notation

A list is represented by a `<list>` element with all members of the list given in the body of the element. For example, a list containing the integers 1, 2 and the Boolean value *true* is represented by:

```
<list> <i>1</i> <i>2</i> <true/> </list>
```

2.7.2 Abridged notation

Wherever it is legal to have a list element, the opening square brace is defined as a synonym of `<list>` and the closing square brace is defined as a synonym of `</list>`. A separator is used between two elements of the list. Whitespace can be found before and after a square brace.

```
[1 2 <true/>]
```

2.8 Matrices

Vectors are represented as lists, 2-dimensions matrices are represented as lists of vectors, 3-dimensions matrices are represented as lists of 2-dimensions matrices and so on. To improve readability, 2-dimensions matrices are represented as lists of lines of coefficients.

```
[1 2 3]

[
 [1 2 3]
 [0 1 2]
 [0 0 1]
]
```

2.9 Dictionaries

A dictionary is an associative array that maps a key to a value. In other words, it is an array of <key,value> pairs. A key is a name which references a value in the data structure. A notation common to a number of languages to access the value corresponding to a key k in a dictionary d is $d[k]$. In a sense, a dictionary is a generalization of an array: indices in (classical) arrays must be contiguous integers while keys in a dictionary can be arbitrary names. A dictionary can also be seen as a generalization of the notion of structures (struct in C/C++) and records (Pascal). A record with n fields $f_1, \ldots, f_n$ can be seen as a dictionary containing the n keys $f_1, \ldots, f_n$ and the corresponding values. A dictionary is an extension of a record since new keys can be added to a dictionary while a record usually has a fixed list of fields. Each pair <key,value> in a dictionary is called an entry in that dictionary.

A function which accepts a dictionary as parameter can decide that some keys must be present in the dictionary and that some others keys are optional. This provides a simple way to support optional parameters. When a key is missing in a dictionary, it is considered that there is no corresponding value. The special tag `<nil/>` is another way to specify explicitly that a key has no corresponding value. This special value corresponds to the `null` value in SQL. Omitting a key k from a dictionary or defining that key k corresponds to value `<nil/>` are equivalent ways of associating no value to key k.

The order of keys in a dictionary is not significant. A dictionary may contain no key at all. A dictionary can be associated to a key in a given dictionary (in other words, dictionaries may be contained in a dictionary).

2.9.1 Tagged notation

An entry of a dictionary that associates a value v with key k is encoded by an element `<entry>` with a single attribute **key** with value k and a body containing the value v: `<entry key="k">v</entry>`

A dictionary is defined by a `<dict>` element with all entries of the dictionary given inside the body: `<dict><entry key="name1">value1</entry> <entry key="name2">value2</entry></dict>`. The body of this element cannot contain other elements than dictionary entries.

2.9.2 Abridged notation

Several notations are already used in different languages to associate a key with a value in an associative array (`key => value` in PHP and Perl, `/key value` in PostScript and PDF,...). Since the character > is a reserved character in XML, we use the PostScript notation.

A dictionary in abridged notation starts by a opening curly brace followed by a list of entries and is ended by a closing curly brace. Each entry is written as a key immediately preceded by a slash (no space between the slash and the key), whitespace and the value corresponding to this key. Whitespace can be found before and after a curly brace. For example:

{`/name1 value2 /name2 value2`}

2.9.3 Conventional order

In some cases (e.g. when a function expects a dictionary with a given set of keys), a conventional order can be associated with a dictionary. This conventional order specifies a default order of keys which can be used to further shorten the notations. When the conventional order of keys can be known from the context, a dictionary can be written in abridged notation by opening a curly brace, listing the values of each key which is expected in the dictionary and closing the curly brace. The absence of a slash following the opening curly brace identifies a dictionary represented in conventional order (note however that white space is allowed between the opening curly brace and the first slash).

In this context, there must be as many values inside the curly braces as the number of keys in the conventional order. To assign no value to a given key, the special value `<nil/>` must be used.

For example, the coordinates of a point of a plane may be represented by a dictionary containing two keys x and y. The point at coordinates $(2, 5)$ can be represented by several notations. We can have:

```
<dict>
  <entry key="x"><i>2</i></entry>
```

```
  <entry key="y"><i>5</i></entry>
</dict>

or:

<dict>
  <entry key="y"><i>5</i></entry>
  <entry key="x"><i>2</i></entry>
</dict>

or:

{/x 2 /y 5}

or:

{/y 5 /x 2}

or:

{
  /x 2
  /y 5
}
```

When a conventional order is fixed which indicates that key x is given before key y, the same dictionary can be written by:

```
{2  5}
```

2.10 Tuples

Here, we consider a tuple as being a sequence of objects of the same type. For example, $(2, 5, 8)$ is a tuple containing three integers.

2.10.1 Tagged notation

A tuple is represented by a `<tuple>` element with all members of the tuple given in the body of the element. For example, the tuple $(2, 5, 8)$ is represented by:

```
<tuple> <i>2</i> <i>5</i> <i>8</i> </tuple>
```

2.10.2 Abridged notation

For the abridged variant, the members of any tuple are written directly within the enclosing tag. However, if we have two successive tuples (i.e. `...  </tuple> <tuple> ...`), we use the character '|' as a separator between them. For example, the representation of a sequence of binary tuples takes the form (in BNF notation):

```
<binaryTupleSequence> ::= <binaryTuple> | <binaryTuple> "|" <binaryTupleSequence>
<binaryTuple> ::= <integer> <separator> <integer>
```

For ternary relations, one has just to consider tuples formed from 3 values, etc. For example, a list of binary tuples is:

```
0 1|0 3|1 2|1 3|2 0|2 1|3 1
```

while a list of ternary tuples is:

```
0 0 1|0 2 1|1 0 1|1 2 0|2 1 1|2 2 2
```

2.11 Weighted Tuples

It may be interesting (e.g. see the WCSP framework [11]) to associate a weight (or cost) with a tuple or a sequence of tuples.

2.11.1 Tagged notation

We then just have to enclose this tuple (these tuples) within a `<weight>` element which admits one attribute **value**. This attribute must be an integer or the special value "infinity". For example, if a weight equal to 10 must be associated with the tuple $(2, 5, 8)$, we obtain:

```
<weight value="10"> <tuple> <i>2</i> <i>5</i> <i>8</i> </tuple>
</weight>
```

Notice that it is possible to directly associate the same weight with several tuples.

2.11.2 Abridged notation

In abridged notation, each tuple can be given an explicit cost by prefixing it with its cost followed by a colon character ':'. When a cost is not specified for a tuple, it simply means that the cost of the current tuple is equal to the cost of the previous one. At the extreme, only the first tuple is given an explicit cost, all other tuples implicitly referring to this cost. In any case, the first tuple of a relation must be given an explicit cost. Remark that with the abridged variant, it is not possible to put in the same context unweighted and weighted tuples (but, we believe that it is not a real problem). Finally, to associate the special value "infinity" with a tuple, the special element `<infinity/>` must be used.

For example, let us consider the following "classical" list of binary tuples:

```
0 1|0 3|1 2|1 3|2 0|2 1|3 1
```

If 1 is the cost of tuples $(0, 1), (0, 3), (3, 1)$ whereas 10 is the cost of all other tuples, then we can write:

```
1:0 1|1:0 3|10:1 2|10:1 3|10:2 0|10:2 1|1:3 1
```

but also, using implicit costs:

```
1:0 1|0 3|10:1 2|1 3|2 0|2 1|1:3 1
```

This example may also be written equivalently on several lines:

```
1:  0 1|0 3|
10: 1 2|1 3|2 0|2 1|
1:  3 1
```

Note that using the abridged representation to associate costs with tuples allows us to save a large amount of space.

3 Representing CSP instances

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 $dom(X)$ denoting the set of values allowed for X, and a finite set of constraints such that each constraint C has an associated relation $rel(C)$ denoting the set of tuples allowed for the variables $scp(C)$ involved in C. A solution to a constraint network is the assignment of a value to each variable 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-hard. A constraint network is also called a CSP instance. For an introduction to constraint programming, see for example [7, 1].

Each CSP instance is represented following the format given in Figure 1 where q, n, r, p and e respectively denote the number of distinct domains, the number of variables, the number of distinct relations, the number of distinct predicates and the number of constraints. Note that $q \leq n$ as the same domain definition can be used for different variables, $r \leq e$ and $p \leq e$ as the same relation or predicate definition can be used for different constraints. Thus, each instance is defined by an XML element which is called `<instance>` and which contains four, five or six elements. Indeed, it is possible to have one instance defined without any reference to a relation or/and to a predicate. Then, the elements `<relations>` and `<predicates>` may be missing (if both are missing, it means that only global constraints are referenced).

Each basic element (`<presentation>`, `<domain>`, `<variable>`, `<relation>`, `<predicate>` and `<constraint>`) of the representation admits an attribute called **name**. The value of the attribute **name** must be a valid identifier (as introduced in the previous section). In the representation of any instance, it is not possible to find several attributes "name" using the same identifier.

Remark 1 *In the body of any element of the document, one can insert an* `<extension>` *element in order to put any information specific to a solver.*

3.1 Presentation

The XML element called `<presentation>` admits a set of attributes and may contain a description (a string) of the instance:

```
<instance>
   <presentation
       name = 'put here the instance name'
       ...
       format = 'XCSP 2.1' >
     Put here the description of the instance
   </presentation>

   <domains nbDomains='q'>
      <domain
         name = 'put here the domain name'
         nbValues = 'put here the number of values' >
        Put here the list of values
      </domain>
      ...
   </domains>

   <variables nbVariables='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'
           arity = 'put here the arity of the relation'
           nbTuples = 'put here the number of tuples'
           semantics = 'put here either supports or conflicts' >
         Put here the list of tuples
       </relation>
        ...
   </relations>

   <predicates nbPredicates='p'>
       <predicate
           name = 'put here the name of the predicate' >
         <parameters>
           put here a list of formal parameters
         </parameters>
         <expression>
            Put here one (or more) representation of the predicate expression
         </expression>
       </predicate>
       ...
   </predicates>

   <constraints nbConstraints='e'>
      <constraint
         name = 'put here the name of the constraint'
         arity = 'put here the arity of the constraint'
         scope = 'put here the scope of the constraint'
         reference = 'put here the name of a relation, a
                      predicate or a global constraint'>
         ...
      </constraint>
      ...
   </constraints>
</instance>
```

Figure 1: XML representation of a CSP instance

```
<presentation
    name = 'put here the instance name'
    maxConstraintArity = 'put here the greatest constraint arity'
    minViolatedConstraints = 'the minimum number of violated constraints'
    nbSolutions = 'put here the number of solutions'
    solution = 'put here a solution'
    type = 'CSP'>
    format = 'XCSP 2.1'
  Put here the description of the instance
</presentation>
```

Only the attribute **format** is mandatory (all other attributes are optional as they mainly provide human-readable information). It must be given the value 'XCSP 2.1' for the current format. The attribute **name** must be a valid identifier (or the special value '?'). The attribute **maxConstraintArity** is of type integer and denotes the greatest arity of all constraints involved in the instance. The attribute **minViolatedConstraints** can be given an integer value denoting the minimum number of constraints that are violated by any full instantiation of the variables, an expression of the form 'at most k' with k being a positive integer or '?'. 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 k' with k being a positive integer 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 **type** attribute indicates the kind of problem described by this instance. It should be set to 'CSP' for a constraint satisfaction problem, 'QCSP' for a quantified CSP and 'WCSP' for a weighted CSP. For compatibility with the XCSP 2.0 format, this attribute is optional for CSP instances (but it is strongly advised to use it). It is mandatory for other types of instances (QCSP, WCSP,...).

Remark that the optional attribute **maxSatisfiableConstraints**, although still authorized, is deprecated. We encourage to use **minViolatedConstraints** instead.

3.2 Domains

The XML element called `<domains>` admits an attribute which is called **nbDomains** and contains some occurrences (at least, one) of an element called `<domain>`, each one being 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>` admits two attributes, called **name** and **nbValues** and contains a list of values, as follows:

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

The attribute **name** corresponds to the name of the domain and its value must be a valid identifier.

The attribute **nbValues** is of type integer and its value is equal to the number of values of the domain. The content of the element `<domain>` gives the set of integer values included in the domain. More precisely, it contains a sequence of integers and integer intervals.

For the abridged variant, we have for example:

- 1 5 10 corresponds to the set $\{1, 5, 10\}$.
- 1..3 7 10..14 corresponds to the set $\{1, 2, 3, 7, 10, 11, 12, 13, 14\}$.

Note that **nbValues** gives the number of values of the domain (i.e. the domain size), and not, the number of domain pieces (integers and integer intervals).

3.3 Variables

The XML element called `<variables>` admits an attribute which is called `<nbVariables>` and contains some occurrences (at least, one) 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 but admits two attributes, called **name** and **domain**, as follows:

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

The attribute **name** corresponds to the name of the variable and its value must be a valid identifier.

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.

3.4 Relations

If present, the XML element called `<relations>` admits an attribute which is called **nbRelations** and contains some occurrences (at least, one) of an element called `<relation>`, each one being associated with at least one 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>` admits four attributes, called **name**, **arity**, **nbTuples** and **semantics**, and contains a list of distinct tuples that represents either allowed tuples (supports) or disallowed tuples (conflicts). It is defined as follows:

```
<relation
    name = 'put here the name of the relation'
    arity = 'put here the arity of the relation'
    nbTuples = 'put here the number of tuples'
    semantics = 'put here either supports or conflicts' >
  Put here the list of tuples
</relation>
```

The attribute **name** corresponds to the name of the relation and its value must be a valid identifier.

The attribute **arity** is of type integer and its value is equal to the arity of the relation. The attribute **nbTuples** is of type integer and its value is equal to the number of tuples of the relation. The attribute **semantics** can only be given two values: 'supports' and 'conflicts'. Of course, if the value of **semantics** is 'supports' (resp. 'conflicts'), then it means that the list of tuples correspond to allowed (resp. disallowed) tuples. The content of the element `<relation>` gives the set of distinct tuples of the relation.

The representation of lists of tuples is given in Section 2.10.

Note that an empty list of tuples is authorized by the syntax: a relation with an empty list of tuples is trivially unsatisfiable if its semantics is 'supports', and trivially satisfied if its semantics is 'conflicts'.

3.5 Predicates

If present, the XML element called `<predicates>` admits an attribute which is called **nbPredicates** and contains some occurrences (at least, one) of an element called `<predicate>`, one for each predicate associated with at least a constraint of the instance. The attribute **nbPredicates** is of type integer and its value is equal to the number of occurrences of the element `<predicate>`.

Each element `<predicate>` admits one attribute, called **name**, and contains two elements, called `<parameters>` and `<expression>`. It is defined as follows:

```
<predicate
    name = 'put here the name of the predicate' >
  <parameters>
     put here a list of formal parameters
  </parameters>
  <expression>
     Put here one (or several) representation(s) of the predicate expression
  </expression>
</predicate>
```

The attribute **name** corresponds to the name of the predicate and its value must be a valid identifier.

The <parameters> element defines the list of formal parameters of the predicate. The syntax of this element is detailed is section 2.6.

The only authorized type is for the moment 'int' (denoting integer values).

However, in the future, other types will be taken into account: "bool", "string", etc.

The element <expression> may contain several representations of the predicate expression.

3.5.1 Functional Representation

It is possible to insert a functional representation of the predicate expression by inserting in <expression> an element <functional> which contains any Boolean expression defined as follows:

```
<integerExpression> ::=
      <integer> | <identifier>
    | "neg(" <integerExpression> ")"
    | "abs(" <integerExpression> ")"
    | "add(" <integerExpression> "," <integerExpression> ")"
    | "sub(" <integerExpression> "," <integerExpression> ")"
    | "mul(" <integerExpression> "," <integerExpression> ")"
    | "div(" <integerExpression> "," <integerExpression> ")"
    | "mod(" <integerExpression> "," <integerExpression> ")"
    | "pow(" <integerExpression> "," <integerExpression> ")"
    | "min(" <integerExpression> "," <integerExpression> ")"
    | "max(" <integerExpression> "," <integerExpression> ")"
    | "if(" <booleanExpression> "," <integerExpression>  "," <integerExpression> ")"

<booleanExpression> ::=
      "false" | "true"
    | "not(" <booleanExpression> ")"
    | "and(" <booleanExpression> "," <booleanExpression> ")"
    | "or(" <booleanExpression> "," <booleanExpression> ")"
    | "xor(" <booleanExpression> "," <booleanExpression> ")"
    | "iff(" <booleanExpression> "," <booleanExpression> ")"
    | "eq(" <integerExpression> "," <integerExpression> ")"
    | "ne(" <integerExpression> "," <integerExpression> ")"
    | "ge(" <integerExpression> "," <integerExpression> ")"
    | "gt(" <integerExpression> "," <integerExpression> ")"
    | "le(" <integerExpression> "," <integerExpression> ")"
    | "lt(" <integerExpression> "," <integerExpression> ")"
```

Hence, any constraint in intension can be defined by a predicate which corresponds to an expression built from (Boolean and integer) constants and the introduced set of functions (operators). The semantics of operators is given by Table 1.

An expression usually contains identifiers which correspond to the formal parameters of a predicate. To illustrate this, let us consider the predicate that allows defining constraints involved in any instance of the *queens* problem. It corresponds to: $X \neq Y \wedge |X - Y| \neq Z$. We obtain using the functional representation:

Operation	Arity	Syntax	Semantics	MathML
Arithmetic (operands are integers)				
Opposite	1	neg(x)	-x	<minus>
Absolute Value	1	abs(x)	| x |	<abs>
Addition	2	add(x,y)	x + y	<plus>
Substraction	2	sub(x,y)	x - y	<minus>
multiplication	2	mul(x,y)	x * y	<times>
Integer Division	2	div(x,y)	x div y	<quotient>
Remainder	2	mod(x,y)	x mod y	<rem>
Power	2	pow(x,y)	x^y	<power>
Minimum	2	min(x,y)	min(x,y)	<min>
Maximum	2	max(x,y)	max(x,y)	<max>
Relational (operands are integers)				
Equal to	2	eq(x,y)	$x = y$	<eq>
Different from	2	ne(x,y)	$x \neq y$	<neq>
Greater than or equal	2	ge(x,y)	$x \geq y$	<geq>
Greater than	2	gt(x,y)	$x > y$	<gt>
Less than or equal	2	le(x,y)	$x \leq y$	<leq>
Less than	2	lt(x,y)	$x < y$	<lt>
Logic (operands are Booleans)				
Logical not	1	not(x)	$\neg$ x	<not>
Logical and	2	and(x,y)	$x \wedge y$	<and>
Logical or	2	or(x,y)	$x \vee y$	<or>
Logical xor	2	xor(x,y)	$x \oplus y$	<xor>
Logical equivalence (iff)	2	iff(x,y)	$x \Leftrightarrow y$	
Control				
Alternative	3	if(x,y,z)	value of y if x is true, otherwise value of z	

Table 1: Operators used to build predicate expressions

```
<predicate name="P0">
  <parameters>
    int X int Y int Z
  </parameters>
  <expression>
    <functional>
      and(ne(X,Y),ne(abs(sub(X,Y)),Z))
    </functional>
  </expression>
</predicate>
```

3.5.2 MathML Representation

MathML is a language dedicated to represent mathematical expressions. We can represent predicate expressions using a subset of this language. It is possible to insert an XML representation of the predicate expression by inserting in `<expression>` an element called `<math>` which contains any Boolean expression defined (in BNF notation) as follows:

```
<integerExpression> ::=
     "<cn>" <integer> "</cn>"
   | "<ci>" <identifier> "</ci>"
   | "<apply> <minus/>"  <integerExpression> "</apply>"
   | "<apply> <abs/>"  <integerExpression> "</apply>"
   | "<apply> <plus/>" <integerExpression> <integerExpression> "</apply>"
   | "<apply> <minus/>" <integerExpression> <integerExpression> "</apply>"
   | "<apply> <times/>" <integerExpression> <integerExpression> "</apply>"
   | "<apply> <quotient/>" <integerExpression> <integerExpression> "</apply>"
   | "<apply> <rem/>" <integerExpression> <integerExpression> "</apply>"
   | "<apply> <power/>" <integerExpression> <integerExpression> "</apply>"
   | "<apply> <min/>" <integerExpression> <integerExpression> "</apply>"
   | "<apply> <max/>" <integerExpression> <integerExpression> "</apply>"

<booleanExpression> ::=
     "<false/>"
   | "<true/>"
   | "<apply> <not/>" <booleanExpression> "</apply>"
   | "<apply> <and/>" <booleanExpression> <booleanExpression> "</apply>"
   | "<apply> <or/> <booleanExpression> <booleanExpression> "</apply>"
   | "<apply> <xor/> <booleanExpression> <booleanExpression> "</apply>"
   | "<apply> <eq/>" <integerExpression> <integerExpression> "</apply>"
   | "<apply> <neq/>" <integerExpression> <integerExpression> "</apply>"
   | "<apply> <gt/>" <integerExpression> <integerExpression> "</apply>"
   | "<apply> <geq/>" <integerExpression> <integerExpression> "</apply>"
   | "<apply> <lt/>" <integerExpression> <integerExpression> "</apply>"
   | "<apply> <leq/>" <integerExpression> <integerExpression> "</apply>"
```

For more information, see `http://www.w3.org/Math` or `http://www.dessci.com/en/support/tutorials/mathml/content.htm`. For example, to represent $X \neq Y \wedge |X - Y| \neq Z$, we can write:

```
<predicate name="P0">
  <parameters> int X int Y int Z </parameters>
  <expression>
    <math>
      <apply>
         <neq/> <ci> X </ci> <ci> Y </ci>
      </apply>
      <apply>
         <neq/>
         <apply>
           <abs/>
           <apply> <minus/ > <ci> X </ci> <ci> Y </ci> </apply>
         </apply>
         <ci> Z </ci>
      </apply>
    </math>
  </expression>
</predicate>
```

3.5.3 Postfix Representation

It is possible to insert a postfix representation (not fully described in this document) of the predicate expression by inserting in `<expression>` an element `<postfix>`. For example, to represent $X \neq Y \wedge |X - Y| \neq Z$, we can write:

```
<predicate name="P0">
  <parameters>
    int X int Y int Z
  </parameters>
  <expression>
    <postfix>
      X Y ne X Y sub abs Z ne and
    </postfix>
  </expression>
</predicate>
```

3.5.4 Infix Representation

It is possible to insert an infix representation (not fully described in this document) of the predicate expression by inserting in `<expression>` an element `<infix>`. For example, to represent $X \neq Y \wedge |X - Y| \neq Z$, we can write (using a C syntax for the Boolean expression):

```
<predicate name="P0">
  <parameters>
    int X int Y int Z
  </parameters>
  <expression>
    <infix syntax="C">
      X != Y && abs(X-Y) != Z
    </infix>
  </expression>
</predicate>
```

3.6 Constraints

The XML element called `<constraints>` admits an attribute which is called **nbConstraints** and contains some occurrences (at least, one) 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>` admits four attributes, called **name**, **arity**, **scope** and **reference**, and potentially contains some elements:

```
<constraint
   name = 'put here the name of the constraint'
   arity = 'put here the arity of the constraint'
   scope = 'put here the scope of the constraint'
   reference = 'put here the name of a relation, of
               predicate or a global constraint'>
   ...
</constraint>
```

The attribute **name** corresponds to the name of the constraint and its value must be a valid identifier.

The attribute **arity** is of type integer and its value is equal to the arity of the constraint (that is to say, the number of variables in its scope). It must be

greater than or equal to 1. The value of the attribute **scope** denotes the set of variables involved in the constraint. It must correspond to a list of distinct variable names where each name corresponds to the value of the **name** attribute of a `<variable>` element. Variables are separated by whitespace.

There are three alternatives to represent constraints. Indeed, it is possible to introduce:

- constraints in extension
- constraints in intension
- global constraints

3.6.1 Constraints in extension

The value of the attribute **reference** must be the name of a relation. It means that it must correspond to the value of the **name** attribute of a `<relation>` element. The element `<constraint>` is empty when it represents a constraint defined in extension. For example:

```
<constraint name="C0" scope="V0 V1" reference="R0" />
```

3.6.2 Constraints in intension

The value of the attribute **reference** must be the name of a predicate. It means that it must correspond to the value of the **name** attribute of a `<predicate>` element.

The element `<constraint>` contains an element `<parameters>` when it represents a constraint defined in intension. The element `<parameters>` contains a sequence of effective parameters, each one being either an integer or a variable reference (which must occur in the scope of the constraint). For the abridged variant, a separator is inserted between two elements. Of course, the arity of a predicate referenced by a constraint must correspond to the number of effective parameters of this constraint. Also, all variables occurring in the scope of a constraint referencing a predicate must occur as effective parameters of this constraint. For example:

```
<constraint name="C0" scope="V0 V1" reference="P0">
   <parameters>
      V0 V1 1
   </parameters>
</constraint>
```

The semantics is the following. Given a tuple built by assigning a value to each variable belonging to the scope of the constraint, the predicate expression is evaluated after replacing each occurrence of a formal parameter corresponding to an effective parameter denoting a variable with the assigned value. The tuple is allowed iff the expression evaluates to *true*.

In the current version of the format, effective parameters are restricted to be either variables in the scope of the constraint, or integer constants. Future

version of the format will remove this limitation and allow any expression as an effective parameter.

3.6.3 Global constraints

The value of the attribute **reference** must be the name of a global constraint, prefixed by "global:". As the character ':' cannot occur in any valid identifier, it avoids some potential collision with other identifiers. The name of global constraints is case-insensitive. Therefore, `global:allDifferent` and `global:alldifferent` represent the same constraint.

The element `<constraint>` may contain an element `<parameters>` when it represents a global constraint. If present, the element `<parameters>` contains a sequence of parameters specific to the global constraint. As a consequence, the description of such parameters must be given for each global constraint. It is then clear that, for each global constraint, we have to indicate its name (the one to be referenced), its parameters (and the way they are structured in XML) and its semantics. Below, we provide such information for four global constraints.

Constraint weightedSum (not defined in the global constraint catalog)

Semantics $\sum_{i=1}^{r} k_i * X_i$ op b where r denotes the arity of the constraint, k_i denotes an integer, X_i the i^{th} variable occurring in the scope of the constraint, *op* a relational operator in $\{=, \neq, >, \geq, <, \leq\}$, and b an integer.

Parameters There is a first parameter that represents a list of k dictionaries representing each product in the sum. Each dictionary contains an integer coefficient (associated with the `coef` key) and one variable identifier (associated with the `var` key). The conventional order of keys in these dictionaries is `coef,var`. Therefore, {`/coef 2 /var X1`} can be represented as {`2 X1`}.

There is a second parameter which is a tag denoting the relational operator. It corresponds to an atom that must necessarily belong to {`<eq/>,<ne/>,<ge/>`, `<gt/>,<le/>,<lt/>`} (see Table 1). There is a third parameter which is an integer.

Example $V0 + 2V1 - 3V2 > 12$

```
<constraint name="C2" arity="3" scope="V0 V1 V2" reference="global:weightedSum">
  <parameters>
     [ { 1 V0 } { 2 V1 } { -3 V2 } ]
     <gt/>
     12
  </parameters>
</constraint>
```

The syntax of the weightedSum global constraint slightly changed from the XCSP 2.0 format to the current format. The first parameter of this constraint is now a list of dictionaries (previously, it was a list of lists). The old syntax is deprecated.

Alternatives This arithmetic constraint can be represented in intension (using the grammar described earlier in the paper). It is interesting to note that:

- $\sum_{i=1}^{r} k_i * X_i = b \Leftrightarrow \sum_{i=1}^{r} k_i * X_i \geq b \wedge \sum_{i=1}^{r} k_i * X_i \leq b$
- $\sum_{i=1}^{r} k_i * X_i \neq b \Leftrightarrow \sum_{i=1}^{r} k_i * X_i > b \vee \sum_{i=1}^{r} k_i * X_i < b$
- $\sum_{i=1}^{r} k_i * X_i > b \Leftrightarrow \sum_{i=1}^{r} k_i * X_i \geq b - 1$
- $\sum_{i=1}^{r} k_i * X_i < b \Leftrightarrow \sum_{i=1}^{r} -k_i * X_i > -b$

References This arithmetic constraint is related to the constraint called sum_ctr in [2]. Some information can also be found in [14].

3.6.4 Global constraints from the Catalog

The catalog of global constraints (see `http://www.emn.fr/x-info/sdemasse/gccat`) describes a huge number of global constraints. This section describes how these constraints can be translated in the XML representation. This description is based on the 2006-09-30 version of the catalog.

This version of the XML format supports global constraints from the catalog with parameters of type `int`, `dvar`, `list` and `collection` (in the catalog terminology).

Unless stated otherwise for a particular constraint, the name of the global constraint in the XML representation is directly obtained from the name of the constraint in the catalog by prefixing it with 'global:'. For example, the catalog defines a constraint named `cumulative`. In the XML representation, it is named `global:cumulative`. The semantics of the global constraint is the one defined in the catalog.

Except for some particular cases, parameters of global constraints are represented according to the following rules.

atom In the catalog, a parameter of type `atom` is represented in the XML format by a tag. Atoms representing relational operators will be denoted by elements of {`<eq/>`,`<ne/>`,`<ge/>`, `<gt/>`,`<le/>`,`<lt/>`} (see Table 1).

int In the catalog, a parameter of type `int` is an integer constant. It is represented in the XML format as an integer constant (`<i>value</i>` in tagged notation or `value` in abridged notation).

dvar In the catalog, a parameter of type `dvar` corresponds to a CSP variable. It is represented in the XML format as a variable reference (`<var name="X"/>` in tagged notation or `X` in abridged notation). A parameter of type `dvar` can also correspond to an integer constant.

list A list of elements in the catalog of global constraints is represented as a list in the XML format (cf. 2.7).

collection The catalog defines a collection as a collection of ordered items, each item being a set of `<attribute, value>` pairs. In our XML representation, this is directly translated as a list of dictionaries. The keys in each dictionary correspond to the attributes in the collection.

As an example, the `cumulative` constraint is defined in the catalog as `cumulative(TASKS; LIMIT)` where TASKS is a collection(origin–dvar; duration–dvar; end–dvar; height–dvar) and LIMIT is an int. For each task, height must be defined but only two attributes among origin, duration and end can be defined (since by definition origin-end=duration). A constraint that enforces a maximal height of 4 for 3 tasks starting at origins represented as a CSP variable and with given duration and height, can be represented by:

```
<constraint name="C1" arity="3" scope="X2 X5 X9" reference="global:cumulative">
  <parameters>
     [
       {/origin X2 /duration 10 /height 1}
       {/origin X5 /duration 5 /height 2}
       {/origin X9 /duration 8 /height 3}
     ]
     4
  </parameters>
</contraint>
```

Note that attributes that are not required are represented as missing keys in the dictionaries.

Assuming the conventional order origin, duration, end, height is defined for `cumulative`, this constraint can also be written as

```
<constraint name="C1" arity="3" scope="X2 X5 X9" reference="global:cumulative">
  <parameters>
     [
       {X2 10 <nil/> 1}
       {X5 5 <nil/> 2}
       {X9 8 <nil/> 3}
     ]
     4
  </parameters>
</contraint>
```

Here, missing attributes are represented by the `<nil/>` element.

For each global constraint, the conventional order of dictionaries is the order of attributes defined in the global catalog.

When a global constraint has a collection parameter which contains only one attribute, it is represented as a list of values. For example, the `global_cardinality` constraint has a first parameter of type collection of dvar. It is represented directly a list of variables (`[X1 X2 X3]`) instead of a list of dictionaries with one single key (`[{/var X1} {/var X2} {/var X3}]` or, using conventional order, `[{X1} {X2} {X3}]`).

Arguments of a global constraint in the catalog must be translated in XML as a sequence of elements inside the tag `<parameters>` in the order defined in the catalog.

Below, we present the description of some translations of global constraints from the catalog. Note that most of the global constraints from the catalog can be automatically translated in format XCSP 2.1.

Constraint allDifferent It is defined in the catalog as follows:

```
alldifferent(VARIABLES)

VARIABLES collection(var:dvar)
```

Following rules given above, we may have in XML:

```
<constraint name="C1" arity="4" scope="V1 V2 V3 V4" reference="global:allDifferent">
   <parameters>
       [ V1 V2 V3 V4 ]
   </parameters>
</constraint>
```

Note that it is also possible to include integer constants. For example:

```
<constraint name="C1" arity="4" scope="V1 V2 V3 V4" reference="global:allDifferent">
   <parameters>
       [ V1 V2 100 V3 V4 ]
   </parameters>
</constraint>
```

Note that the old syntax, with implicit parameters, is deprecated. On the other hand, this constraint can be represented in intension by introducing a predicate that represents a conjunction of inequalities. It can also be converted into a clique of binary notEqual constraints. For more information about this constraint, see e.g. [13, 17, 2].

Constraint among It is defined in the catalog as follows:

```
among(NVAR,VARIABLES,VALUES)

NVAR        dvar
VARIABLES   collection(var:dvar)
VALUES      collection(val:int)
```

Following rules given above, as an illustration, we can have in XML:

```
<constraint name="C1" arity="5" scope="V0 V1 V2 V3 V4" reference="global:among">
   <parameters>
       V0
       [ V1 V2 V3 V4 ]
       [ 1 5 8 10 ]
   </parameters>
</constraint>
```

Constraint atleast It is defined in the catalog as follows:

```
atleast(N,VARIABLES,VALUE)

N         int
VARIABLES collection(var:dvar)
VALUE     int
```

Following rules given above, we can have in XML:

```
<constraint name="C1" arity="4" scope="V1 V2 V3 V4" reference="global:atleast">
   <parameters>
      1
      [ V1 V2 V3 V4 ]
      2
   </parameters>
</constraint>
```

Constraint atmost It is defined in the catalog as follows:

```
atmost(N,VARIABLES,VALUE)

N         int
VARIABLES collection(var:dvar)
VALUE     int
```

Following rules given above, we can have in XML:

```
<constraint name="C1" arity="4" scope="V1 V2 V3 V4" reference="global:atmost">
   <parameters>
      1
      [ V1 V2 V3 V4 ]
      2
   </parameters>
</constraint>
```

Constraint cumulative It is defined in the catalog as follows:

```
cumulative(TASKS,LIMIT)

TASKS      collection(origin:dvar, duration:dvar, end:dvar, height:dvar)
LIMIT      int
```

Here, we have a collection of ordered items where each item corresponds to a task with 4 attributes (origin, duration, end and height), and a limit value. As an illustration, we can have in XML:

```
<constraint name="C1" arity="8" scope="O1 D1 E1 H1 O2 D2 E2 H2"
            reference="global:cumulative">
   <parameters>
      [ { O1 D1 E1 H1 } { O2 D2 E2 H2 } ]
      8
   </parameters>
</constraint>
```

As indicated in the constraint restrictions, one attribute among $\{origin, duration, end\}$ may be missing. Assume that it is the case for *end*, we could have:

```
<constraint name="C1" arity="6" scope="O1 D1 H1 O2 D2 H2"
            reference="global:cumulative">
   <parameters>
       [ { O1 D1 <nil/> H1 } { O2 D2 <nil/> H2 } ]
       8
   </parameters>
</constraint>
```

Constraint cycle It is defined in the catalog as follows:

```
   cycle(NCYCLE,NODES)

   NCYCLE       dvar
   NODES        collection(index:int, succ:dvar)
```

Here, we have a value that denotes a number of cycles and a collection of ordered items where each item consists of 2 attributes (index, succ). As an illustration, we can have in XML:

```
<constraint name="C1" arity="5" scope="V0 V1 V2 V3 V4" reference="global:cycle">
   <parameters>
       V0
       [ {1 V1} {2 V2} {3 V3} {4 V4} ]
   </parameters>
</constraint>
```

Constraint diffn It is defined in the catalog as follows:

```
   diffn(ORTHOTOPES)

   ORTHOTOPES collection(orth:ORTHOTOPE)
   ORTHOTOPE collection(origin:dvar, size:dvar, end:dvar)
```

As an illustration, we can have in XML:

```
<constraint name="C1"  arity="18" scope="V0 V1 V2 V3 V4 V5 V6 V7 V8 V9 V10
            V11 V12 V13 V14 V15 V16 V17"  reference="global:diffn">
   <parameters>
      [
         [ {V0 V1 V2} {V3 V4 V5} ]
         [ {V6 V7 V8} {V9 V10 V11} ]
         [ {V12 V13 V14} {V15 V16 V17} ]
      ]
   </parameters>
</constraint>
```

Constraint disjunctive It is defined in the catalog as follows:

```
disjunctive(TASKS)

TASKS        collection(origin:dvar, duration:dvar)
```

Here, we have a collection of ordered items where each item corresponds to a task with 2 attributes (origin, duration). As an illustration, we can have in XML:

```
<constraint name="C1" arity="6" scope="O1 D1 O2 D2 O3 D3" reference="global:disjunctive">
   <parameters>
        [ { O1 D1} {O2 D2} {O3 D3} ]
   </parameters>
</constraint>
```

Constraint element It is defined in the catalog as follows:

```
element(INDEX,TABLE,VALUE)

INDEX        dvar
TABLE        collection(value:dvar)
VALUE        dvar
```

As an illustration, we can have in XML:

```
<constraint name="C1" arity="5" scope="I X1 X2 X3 V"  reference="global:element">
   <parameters>
      I
      [ X1 X2 X3 ]
      V
   </parameters>
</constraint>
```

Constraint global_cardinality It is defined in the catalog as follows:

```
global_cardinality(VARIABLES,VALUES)

VARIABLES collection(var:dvar)
VALUES collection(val:int, noccurrence:dvar)
```

As an illustration, we can have in XML:

```
<constraint name="C1" arity="5" scope="V0 V1 V2 V3 V4"
           reference="global:global_cardinality">
   <parameters>
       [ V0 V1 V2 ]
       [ { 1 V3 } { 2 V4 } ]
   </parameters>
</constraint>
```

Constraint global_cardinality_with_costs It is defined as follows:

```
global_cardinality_with_costs(VARIABLES,VALUES,MATRIX,COST)

VARIABLES collection(var:dvar)
VALUES collection(val:int, noccurrence:dvar)
MATRIX collection(i:int, j:int, c:int)
COST dvar
```

As an illustration, we can have in XML:

```
<constraint name="C1" arity="5" scope="V0 V1 V2 V3 V4"
            reference="global:global_cardinality_with_costs">
   <parameters>
       [ V0 V1 V2 ]
       [ { 1 V3 } { 2 V4 } ]
       [ {1 1 1} {1 1 0} {1 1 3} {1 1 2} {1 1 4} {1 1 2} ]
       V5
   </parameters>
</constraint>
```

Constraint minimum_weight_all_different It is defined as follows:

```
minimum_weight_alldifferent(VARIABLES,MATRIX,COST)

VARIABLES    collection(var:dvar)
MATRIX       collection(i:int, j:int, c:int)
COST         dvar
```

As an illustration, we can have in XML:

```
<constraint name="C1" scope="V0 V1 V2 V3"  reference="global:minimum_weight_all_different">
   <parameters>
       [ V0 V1 V2 ]
       [ {1 1 1} {1 1 0} {1 1 3} {1 1 2} {1 1 4} {1 1 2} ]
       V3
 </parameters>
</constraint>
```

Constraint not_all_equal It is defined in the catalog as follows:

```
not_all_equal(VARIABLES)

VARIABLES collection(var:dvar)
```

As an illustration, we can have in XML:

```
<constraint name="C1" arity="5" scope="V0 V1 V2 V3 V4"  reference="global:not_all_equal">
   <parameters>
       [ V0 V1 V2 V3 V4 ]
   </parameters>
</constraint>
```

Constraint nvalue It is defined in the catalog as follows:

```
nvalue(NVAL,VARIABLES)

NVAL        dvar
VARIABLES   collection(var:var)
```

As an illustration, we can have in XML:

```
<constraint name="C1" arity="5" scope="V0 V1 V2 V3 V4"  reference="global:nvalue">
   <parameters>
       V0
       [ V1 V2 V3 V4 ]
   </parameters>
</constraint>
```

Constraint nvalues It is defined in the catalog as follows:

```
nvalues(VARIABLES,RELOP,LIMIT)

VARIABLES collection(var:dvar)
RELOP atom
LIMIT dvar

RELOP in {eq,ne,ge,gt,le,lt}
```

As an illustration, we can have in XML:

```
<constraint name="C1" arity="5" scope="V0 V1 V2 V3 V4"  reference="global:nvalues">
   <parameters>
       [ V1 V2 V3 V4 ]
       <gt/>
       V0
   </parameters>
</constraint>
```

3.7 Illustrations

In Figures 2 and 3, one can see the XML representation of the 4-*queens* instance. In Figures 4, 5 and 6, one can see the XML representation of a CSP instance involving 5 variables and the 5 following constraints:

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

Finally, in Figures 7 and 8, one can see the XML representation of the 3-*magic square* instance. The global constraints $weightedSum$ and $allDifferent$ are used.

4 Representing QCSP instances

This is a proposal for QCSP and QCSP$^+$ **by M. Benedetti, A. Lallouet and J. Vautard**.

The Quantified Constraint Satisfaction Problem (QCSP) is an extension of CSP in which variables may be quantified universally or existentially. A QCSP instance corresponds to a sequence of quantified variables, called prefix, followed by a conjunction of constraints. QCSP and its semantics were introduced in [5]. QCSP$^+$ is an extension of QCSP, introduced in [3] to overcome some difficulties that may occur when modelling real problems with classical QCSP. From a logic viewpoint, an instance of QCSP+ is a formula in which (i) quantification scopes of alternate type are nested one inside the other, (ii) the quantification in each scope is *restricted* by a CSP called *restriction* or *precondition*, and (iii) a CSP to be satisfied, called goal, is attached to the innermost scope. An example with 4 scopes is:

$$
\begin{array}{l}
\forall X_1 \;\; (L_1^{\forall}(X_1) \rightarrow \\
\qquad \exists Y_1 \;\; (L_1^{\exists}(X_1, Y_1) \wedge \\
\qquad\qquad \forall X_2 \;\; (L_2^{\forall}(X_1, Y_1, X_2) \rightarrow \\
\qquad\qquad\qquad \exists Y_2 \;\; (L_2^{\exists}(X_1, Y_1, X_2, Y_2) \wedge \;\; G(X_1, X_2, Y_1, Y_2)) \\
\qquad\qquad) \\
\qquad) \\
) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (1)
\end{array}
$$

where X_1, X_2, Y_1, and Y_1 are in general sets of variables, and each L_i^Q is a conjunction of constraints. A more compact and readable syntax for QCSP$^+$ employs square braces to enclose restrictions. An example with 3 scopes is as follows

$$\forall X_1[L_1^{\forall}(X_1)] \; \exists Y_1[L_1^{\exists}(X_1, Y_1)] \; \forall X_2[L_2^{\forall}(X_1, Y_1, X_2)] \;\; G(X_1, Y_1, X_2)$$

which reads "for all values of X_1 which satisfy the constraints $L_1^{\forall}(X_1)$, there exists a value for Y_1 that satisfies $L_1^{\exists}(X_1, Y_1)$ and is such that for all values for X_2 which satisfy $L_2^{\forall}(X_1, Y_1, X_2)$, the goal $G(X_1, X_2, Y_1)$ is satisfied".

A standard QCSP can be viewed as a particular case of QCSP$^+$ in which all quantifications are unrestricted, i.e. all the CPSs L_i^Q are empty.

4.1 Presentation

With respect to format XCSP 2.0, here is the extension to the XML element called <presentation> in order to deal with a QCSP instance:

- the attribute **format** must be given the value `"XCSP 2.1"`.
- the attribute **type** is required and its value must be `"QCSP"` or `"QCSP+"`.

All the relevant information on quantification in included in a new section called "quantification" (see below).

4.2 Quantification

This section gives the quantification structure associated with a QCSP/QCSP$^+$ instance. It essentially provides an ordered list of quantification blocks, called blocks. The size of this list is mandatorily declared in the attribute **nbBlocks**:

```
<quantification nbBlocks="b">
  put here a sequence of b blocks
</quantification>
```

Notice that the order in which quantification blocks are listed inside this XML element provides key information, as it specifies the left-to-right order of (restricted) quantifications associated with the QCSP/QCSP$^+$ instance. Each block of a QCSP/QCSP$^+$ instance is represented by the following XML element:

```
<block  quantifier = 'put here the quantifier type'
        scope = 'put here a list of variable names'>

   put here an optional list of constraints (QCSP+ only)
</block>
```

where

- the value of the attribute **quantifier** must be either "exists" or "forall". It gives the kind of quantification of all the variables in this block;
- the value of the attribute *scope* specifies the variables which are quantified in this block. At least one variable must be present. The order in which variables are listed is not relevant;
- for QCSP instances, the body of a block element must be empty
- for QCSP+ instances, the body of the block element may contain one or more `<constraint>` elements that restrict the quantification of the block to the sole values of the quantified variables which satisfy all constraints defined in the body of the `<block>` element.

  Such constraints may only refer to variables defined in the same block and to variables defined in previous blocks (w.r.t. to the order in the enclosing `<quantification>` element), but not to variables mentioned in later blocks;

Notice the following features of well-formed QCSP/QCSP$^+$ instances:

1. each variable can be mentioned in *at most one* block;
2. each variable must be mentioned in *at least* one block; this means the problem is *closed*, i.e. no free variables are allowed[2];

[2]This restriction may be relaxed by future formalizations of open QCSPs.

4.3 Illustrations

Let the domain of all the variables we introduce be $\{1, 2, 3, 4\}$.

- An XML encoding of the QCSP:

$$\exists W, X \;\; \forall Y \;\; \exists Z \quad W + X = Y + Z, \;\; Y \neq Z$$

  is given in Figure 9.

- An XML encoding of the QCSP$^+$:

  $\exists W, X[W{+}X < 8, W{-}X > 2]\, \forall Y[W \neq Y, X \neq Y]\, \exists Z[Z < W{-}Y]\, W{+}X = Y{+}Z$

  is given in Figure 10.

5 Representing WCSP instances

The classical CSP framework can be extended by associating weights (or costs) with tuples [4]. The WCSP (Weighted CSP) is a specific extension that rely on a specific valuation structure S(k) defined as follows.

Definition 1 *S(k) is a triple* $([0, \ldots, k], \oplus, \geq)$ *where:*
$k \in [1, \ldots, \infty]$ *is either a strictly positive natural or infinity,*
$[0, 1, \ldots, k]$ *is the set of naturals less than or equal to* k*,*
$\oplus$ *is the sum over the valuation structure defined as:* $a \oplus b = min\{k, a + b\}$*,*
$\geq$ *is the standard order among naturals.*

A WCSP instance is defined by a valuation structure $S(k)$, a set of variables (as for classical CSP instances) and a set of constraints. A domain is associated with each variable and a cost function with each constraint. More precisely, for each constraint C and each tuple t that can be built from the domains associated with the variables involved in C, a value in $[0, 1, \ldots, k]$ is assigned to t. When a constraint C assigns the cost k to a tuple t, it means that C forbids t. Otherwise, t is permitted by C with the corresponding cost. The cost of an instantiation of variables is the sum (using operator $\oplus$) over all constraints involving variables instantiated. An instantiation is consistent if its cost is strictly less than k. The goal of the WCSP problem is to find a full consistent assignment of variables with minimum cost.

It is rather easy to represent WCSP in XML in format XCSP 2.1. This is described below.

5.1 Presentation

Here are the extensions to the XML element called `<presentation>` in order to deal with a WCSP instance:

- the attribute **format** must be given the value `"XCSP 2.1"`.
- the attribute **type** is required and its value must be `"WCSP"`.

5.2 Relations

For WCSP represented in extension, it is necessary to introduce "soft" relations. These relations are defined as follows:

- the value of the attribute **semantics** is set to `"soft"`.
- weighted tuples are given as described in Section 2.11
- an new attribute **defaultCost** is mandatory and represents the cost of any tuple which is not explicitly listed in the relation. Its value belongs to $[0, \ldots, k]$ where k is the maximal cost of the valuation structure, defined by the attribute **maximalCost** of the element `<constraints>` (see below). Note that it may be the special value 'infinity'.

5.3 Functions

Instead of representing constraints of a WCSP in extension, it is possible to represent them in intension by introducing cost functions. For any tuple passed to such a function, its cost is computed and returned. In other words, we employ here exactly the same mechanism as the one employed for hard constraints represented in intension. The only difference is that a predicate returns a Boolean value whereas a cost function must return an integer value.

If present, the XML element called `<functions>` admits an attribute which is called **nbFunctions** and contains some occurrences (at least, one) of an element called `<function>`, one for each function associated with at least a constraint of the instance. The attribute **nbFunctions** is of type integer and its value is equal to the number of occurrences of the element `<function>`. The `<functions>` element must be a direct child of the `<instance>` element.

Each element `<function>` admits two attributes, called **name** and **return**, and contains two elements, called `<parameters>` and `<expression>`. It is defined as follows:

```
<function name = 'put here the name of the function'   return='return type'>
  <parameters>
     put here a list of formal parameters
  </parameters>
  <expression>
     Put here one (or several) representation(s) of the function expression
  </expression>
</function>
```

The attribute **name** corresponds to the name of the function and its value must be a valid identifier.

The attribute **return** indicates the return type of the function. In this version of the format, the only return type used is 'int'. Then elements `<parameters>` and `<expression>` are defined exactly as those defined for `<predicate>` elements. The only difference is that the expression must be of type integer instead of being of type Boolean.

5.4 Constraints

For any WCSP instance, it is required to introduce an attribute **maximalCost** to the element `<constraints>`. The value of this attribute is of type integer (and must be strictly positive) and represents the maximum cost of the WCSP framework (the k value). Remember that it corresponds to a total violation and may be equal to 'infinity' (whereas 0 corresponds to a total satisfaction). Also, an optional attribute **initialCost** to the element <constraints> is introduced. If present, the value of this attribute is of type integer and represents a constant cost that must be added to the sum of constraints cost. This is the cost of the 0-ary constraint of the WCSP framework which is sometimes assumed (e.g. see [12]). When not present, it is assumed to be equal to 0.

Remark 2 *When representing a WCSP instance, it is possible to refer to hard constraints (defined in extension or intension). For such constraints, an allowed tuple has a cost of* 0 *while a disallowed tuple has a cost equal to the value of the attribute* **maximalCost**.

5.5 Illustration

The representation in XCSP 2.1 of an illustrative WCSP instance is given by Figure 12.

6 Restrictions for the competition

6.1 Concerning CSP and MaxCSP

The restrictions considered for the 2008 CSP (and Max-CSP) solver competition are:

- only the abridged representation is considered.
- name of domains, variables, relations, predicates, constraints and so on are normalized.
- all domain values and all intermediate computations (when evaluating expressions according to their functional representation) must fit into 32-bits signed integers.
- constraints are sorted by lexicographic order of their normalized scope.
- only the four global constraints ($allDifferent$, $weightedSum$, $cumulative$ and $element$) illustrated in this document will be considered for the CSP category "global constraints", Besides, the abridged representation of global constraints will only be given using the conventional order.

Here are the differences between the format XCSP 2.0* (the restriction of the format XCSP 2.0 considered for the 2006 CSP solver competition) and the

format XCSP 2.1* (the restriction of the format 2.1 considered for the 2008 CSP solver competition).

1. the attribute *name* of the element *presentation* has become optional,
2. two new operators (iff and if) have been introduced to be used when building predicate expressions,
3. constraints of same scope are gathered,
4. three new global constraints are introduced.

In other words, if you have a CSP (or Max-CSP) solver that already recognizes the format XCSP 2.0, the only thing you have to do is to extend it to take into account the two new operators (iff and if). Note that parsers in C(++) and Java for XCSP 2.1 are available from `http://www.cril.univ-artois.fr/CPAI08/`. On the other hand, if you want to submit a solver for the category of global constraints, you also should recognize the three new global constraints. A tool that allows validating XML representations of CSP instances in format 2.1 for the 2008 competition is also available from the URL given above.

On the other hand, it is important to provide some information about computations.

Roundings As all domain values and all (intermediate) computations (when evaluating expressions) are (signed) integers, no rounding problem may occur.

Overflows It is well-known that, when evaluating arithmetic expressions, overflow may occur. It is a real problem in the context of a competition as a solver that products an overflow can be considered as bugged. Any instance selected for the 2008 competition will be guaranteed to be overflow-free. However, be careful, it only means that no overflow will occur if values and all intermediate computations are represented using 32-bits signed integers and predicate expressions are evaluated in the order fixed by the functional representation.

Divisions There are several possible definitions of the quotient and remainder of a division when dividend and divisor are not necessarily positive. We adopt the convention which is used virtually by all modern processors (including the ones used in the competition), the C99 standard and the Java specification. The quotient produced for $div(n, d)$ is an integer value q whose magnitude is as large as possible while satisfying $|dq| \leq |n|$. Moreover, q is positive when $|n| \geq |d|$ and n and d have the same sign, but q is negative when $|n| \geq |d|$ and n and d have opposite signs. The remainder produced for $rem(n, d)$ produces a result value r such that $q * d + r = n$ where $q = div(n, d)$. It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative, and can be positive only if the dividend is positive; moreover, the magnitude of the result is always less than the magnitude of the divisor.

Divisions by Zero Any instance selected for the 2008 competition will be guaranteed to be free of any division by zero. However, be careful, it only means that it will not occur if values and all intermediate computations are represented using 32-bits signed integers and predicate expressions are evaluated in the order fixed by the functional representation.

Finally, for the competition, some traps must be avoided:

- unary constraints may occur (e.g. `zebra.xml`).
- several constraints may share the same scope (e.g. `fapp01-0200-0.xml`).
- several constraints may share the same scope under permutations (e.g. see constraints C0 and C13 in `langford-2-4-ext.xml`).

6.2 Concerning WCSP

The restrictions considered for the 2008 WCSP solver competition are:

- only the abridged representation is considered,
- name of domains, variables, relations, constraints and so on are normalized.
- all domain values and all intermediate computations (i.e. the cost any full instantiation) must fit into 32-bits signed integers.
- constraints are sorted by lexicographic order of their normalized scope.
- no function and no global constraint are considered.

References

[1] K.R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[2] N. Beldiceanu, M. Carlsson, and J. Rampon. Global constraint catalog. Technical report, Swedish Institute of Computer Science, 2005.

[3] M. Benedetti, A. Lallouet, and J. Vautard. QCSP made practical by virtue of restricted quantification. In *Proceedings of IJCAI'07*, pages 6–12, 2007.

[4] S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier. Semiring-based csps and valued csps: Frameworks, properties, and comparison. *Constraints Journal*, 4(3):199–240, 1999.

[5] L. Bordeaux and E. Monfroy. Beyond NP: Arc-Consistency for quantified constraints. In *Proceedings of CP'02*, pages 371–386, 2002.

[6] M. Garcia de la Banda, K. Marriott, R. Rafeh, and M. Wallace. The modelling language Zinc. In *Proceedings of CP'06*, pages 700–705, 2006.

[7] R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[8] P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In *LOPSTR'03: Revised Selected Papers*, pages 214–232, 2004.

[9] A. Frisch, M. Grum, C. Jefferson, B. Martinez Hernandez, and I. Miguel. The design of ESSENCE: A constraint language for specifying combinatorial problems. In *Proceedings of IJCAI'07*, pages 80–87, 2007.

[10] P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.

[11] J. Larrosa. Node and arc consistency in weighted CSP. In *Proceedings of AAAI'02*, pages 48–53, 2002.

[12] J. Larrosa and T. Schiex. In the quest of the best form of local consistency for Weighted CSP. In *Proceedings of IJCAI'03*, pages 363–376, 2003.

[13] J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI'94*, pages 362–367, 1994.

[14] J.C. Régin and M. Rueher. A global constraint combining a sum constraint and difference constraints. In *Proceedings of CP'00*, pages 384–395, 2000.

[15] G. Renker and H. Ahriz. Building models through formal specification. In *Proceedings of CPAIOR'04*, pages 395–401, 2004.

[16] E. Tsang, P. Mills, R. Williams, J. Ford, and J. Borrett. A computer-aided constraint programming system. In *Proceedings of PACLP'99*, pages 81–93, 1999.

[17] W.J. van Hoeve. The alldifferent constraint: a survey. In *Proceedings of the Sixth Annual Workshop of the ERCIM Working Group on Constraints*, 2001.

[18] World Wide Web Consortium (W3C). *Extensible Markup Language (XML)*. http://www.w3.org/XML/, 1997.

[19] J. Zhou. Introduction to the constraint language NCL. *Journal of Logic Programming*, 45(1-3):71–103, 2000.

```
<instance>
  <presentation  name="Queens"  nbSolutions="at least 1"  format="XCSP 2.1">
     This is the 4-queens instance represnted in extension.
  </presentation>

  <domains nbDomains="1">
    <domain name="D0" nbValues="4">
      1..4
    </domain>
  </domains>

  <variables nbVariables="4">
    <variable name="V0" domain="D0"/>
    <variable name="V1" domain="D0"/>
    <variable name="V2" domain="D0"/>
    <variable name="V3" domain="D0"/>
  </variables>

  <relations nbRelations="3">
    <relation name="R0" arity="2" nbTuples="10" semantics="conflicts">
       1 1|1 2|2 1|2 2|2 3|3 2|3 3|3 4|4 3|4 4
    </relation>
    <relation name="R1" arity="2" nbTuples="8" semantics="conflicts">
       1 1|1 3|2 2|2 4|3 1|3 3|4 2|4 4
    </relation>
    <relation name="R2" arity="2" nbTuples="6" semantics="conflicts">
       1 1|1 4|2 2|3 3|4 1|4 4
    </relation>
  </relations>

  <constraints nbConstraints="6">
    <constraint name="C0" arity="2" scope="V0 V1" reference="R0"/>
    <constraint name="C1" arity="2" scope="V0 V2" reference="R1"/>
    <constraint name="C2" arity="2" scope="V0 V3" reference="R2"/>
    <constraint name="C3" arity="2" scope="V1 V2" reference="R0"/>
    <constraint name="C4" arity="2" scope="V1 V3" reference="R1"/>
    <constraint name="C5" arity="2" scope="V2 V3" reference="R0"/>
  </constraints>
</instance>
```

Figure 2: The 4-*queens* instance in extension

```
<instance>
   <presentation  name="Queens"  nbSolutions="at least 1"  format="XCSP 2.1">
     This is the 4-queens instance represented in intention.
  </presentation>

  <domains nbDomains="1">
    <domain name="D0" nbValues="4">
       1..4
    </domain>
  </domains>

  <variables nbVariables="4">
    <variable name="V0" domain="D0"/>
    <variable name="V1" domain="D0"/>
    <variable name="V2" domain="D0"/>
    <variable name="V3" domain="D0"/>
  </variables>

  <predicates nbPredicates="1">
    <predicate name="P0">
      <parameters> int X0 int X1 int X2  </parameters>
      <expression>
        <functional> and(ne(X0,X1),ne(abs(sub(X0,X1)),X2)) </functional>
      </expression>
    </predicate>
  </predicates>

  <constraints nbConstraints="6">
    <constraint name="C0" arity="2" scope="V0 V1" reference="P0">
      <parameters> V0 V1 1 </parameters>
    </constraint>
    <constraint name="C1" arity="2" scope="V0 V2" reference="P0">
      <parameters> V0 V2 2 </parameters>
    </constraint>
    <constraint name="C2" arity="2" scope="V0 V3" reference="P0">
      <parameters> V0 V3 2 </parameters>
    </constraint>
    <constraint name="C3" arity="2" scope="V1 V2" reference="P0">
      <parameters> V1 V2 1 </parameters>
    </constraint>
    <constraint name="C4" arity="2" scope="V1 V3" reference="P0">
      <parameters> V1 V3 2 </parameters>
    </constraint>
    <constraint name="C5" arity="2" scope="V2 V3" reference="P0">
      <parameters> V2 V3 1 </parameters>
    </constraint>
  </constraints>
</instance>
```

Figure 3: The *4-queens* instance in intention

```
<instance>
  <presentation  name="Test"  format="XCSP 2.1">
     This is another instance represented in extension.
  </presentation>

  <domains nbDomains="3">
    <domain name="D0" nbValues="7">
       0..6
    </domain>
    <domain name="D1" nbValues="3">
      1 5 10
    </domain>
    <domain name="D2" nbValues="10">
      1..5 11..15
    </domain>
  </domains>

  <variables nbVariables="5">
    <variable name="V0" domain="D0"/>
    <variable name="V1" domain="D0"/>
    <variable name="V2" domain="D1"/>
    <variable name="V3" domain="D2"/>
    <variable name="V4" domain="D0"/>
  </variables>

  <relations nbRelations="4">
    <relation name="R0" arity="2" nbTuples="7" semantics="conflicts">
       0 0|1 1|2 2|3 3|4 4|5 5|6 6
    </relation>
    <relation name="R1" arity="2" nbTuples="25" semantics="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>
    <relation name="R2" arity="2" nbTuples="1" semantics="supports">
       5 3
    </relation>
    <relation name="R3" arity="3" nbTuples="17" semantics="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
    </relation>
  </relations>

  <constraints nbConstraints="5">
    <constraint name="C0" arity="2" scope="V0 V1" reference="R0"/>
    <constraint name="C1" arity="2" scope="V3 V0" reference="R1"/>
    <constraint name="C2" arity="2" scope="V2 V0" reference="R2"/>
    <constraint name="C3" arity="3" scope="V1 V2 V3" reference="R3"/>
    <constraint name="C4" arity="2" scope="V1 V4" reference="R0"/>
  </constraints>
</instance>
```

Figure 4: Test Instance in extension

```
<instance>
   <presentation  name="Test"  format="XCSP 2.1">
     This is another instance represented in intention.
  </presentation>

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

  <variables nbVariables="5">
    <variable name="V0" domain="dom0"/>
    <variable name="V1" domain="dom0"/>
    <variable name="V2" domain="dom1"/>
    <variable name="V3" domain="dom2"/>
    <variable name="V4" domain="dom0"/>
  </variables>

  <predicates nbPredicates="4">
    <predicate name="P0">
      <parameters> int X0 int X1 </parameters>
      <expression>
        <functional> ne(X0,X1) </functional>
      </expression>
    </predicate>
    <predicate name="P1">
      <parameters> int X0 int X1 int X2 </parameters>
      <expression>
        <functional> ge(sub(X0,X1),X2) </functional>
      </expression>
    </predicate>
    <predicate name="P2">
      <parameters> int X0 int X1 int X2 </parameters>
      <expression>
        <functional> eq(sub(X0,X1),X2) </functional>
      </expression>
    </predicate>
    <predicate name="P3">
      <parameters> int X0 int X1 int X2 int X3 </parameters>
      <expression>
        <functional> eq(add(X0,X1),abs(sub(X3,X4))) </functional>
      </expression>
    </predicate>
  </predicates>
  ...
```

Figure 5: Test Instance in intention (to be continued)

```
  ...
 <constraints nbConstraints="5">
   <constraint name="C0" arity="2" scope="V0 V1" reference="P0">
     <parameters> V0 V1 </parameters>
   </constraint>
   <constraint name="C1" arity="2" scope="V0 V3" reference="P1">
     <parameters> V3 V0 2 </parameters>
   </constraint>
   <constraint name="C2" arity="2" scope="V0 V2" reference="P2">
     <parameters> V2 V0 2 </parameters>
   </constraint>
   <constraint name="C3" arity="3" scope="V1 V2 V3" reference="P3">
     <parameters> V1 2 V2 V3 </parameters>
   </constraint>
   <constraint name="C4" arity="2" scope="V1 V4" reference="P0">
     <parameters> V1 V4 </parameters>
   </constraint>
 </constraints>
</instance>
```

Figure 6: Test Instance in intention (continued)

```
<instance>
  <presentation  name="Magic Square"  format="XCSP 2.1">
     This is the magic square of order 3.
  </presentation>

  <domains nbDomains="1">
    <domain name="dom0" nbValues="9">
      1..9
    </domain>
  </domains>

  <variables nbVariables="9">
    <variable name="X0" domain="dom0"/>
    <variable name="X1" domain="dom0"/>
    <variable name="X2" domain="dom0"/>
    <variable name="X3" domain="dom0"/>
    <variable name="X4" domain="dom0"/>
    <variable name="X5" domain="dom0"/>
    <variable name="X6" domain="dom0"/>
    <variable name="X7" domain="dom0"/>
    <variable name="X8" domain="dom0"/>
  </variables>

  <constraints nbConstraints="8">
    <constraint name="C0" arity="3" scope="X0 X1 X2" reference="global:weightedSum">
     <parameters>
        [ { 1 X0 } { 1 X1 } { 1 X2 } ]
        <eq/>
        15
     </parameters>
    </constraint>
    <constraint name="C1" arity="3" scope="X3 X4 X5" reference="global:weightedSum">
     <parameters>
        [ { 1 X3 } { 1 X4 } { 1 X5 } ]
        <eq/>
        15
     </parameters>
    </constraint>
    <constraint name="C2" arity="3" scope="X6 X7 X8" reference="global:weightedSum">
     <parameters>
        [ { 1 X6 } { 1 X7 } { 1 X8 } ]
       <eq/>
        15
     </parameters>
    </constraint>
    <constraint name="C3" arity="3" scope="X0 X3 X6" reference="global:weightedSum">
     <parameters>
        [ { 1 X0 } { 1 X3 } { 1 X6 } ]
        <eq/>
        15
     </parameters>
    </constraint>
    ...
```

Figure 7: The 3-*magic square* instance (to be continued)

```
    ...
   <constraint name="C4" arity="3" scope="X1 X4 X7" reference="global:weightedSum">
    <parameters>
       [ { 1 X1 } { 1 X4 } { 1 X7 } ]
       <eq/>
       15
    </parameters>
   </constraint>
   <constraint name="C5" arity="3" scope="X2 X5 X8" reference="global:weightedSum">
    <parameters>
       [ { 1 X2 } { 1 X5 } { 1 X8 } ]
       <eq/>
       15
    </parameters>
   </constraint>
   <constraint name="C6" arity="3" scope="X0 X4 X8" reference="global:weightedSum">
    <parameters>
       [ { 1 X0 } { 1 X4 } { 1 X8 } ]
       <eq/>
       15
    </parameters>
   </constraint>
   <constraint name="C7" arity="3" scope="X2 X4 X6" reference="global:weightedSum">
    <parameters>
       [ { 1 X2 } { 1 X4 } { 1 X6 } ]
       <eq/>
       15
    </parameters>
   </constraint>
   <constraint name="C8" arity="9" scope="X0 X1 X2 X3 X4 X5 X6 X7 X8"
               reference="global:allDifferent" />
  </constraints>
</instance>
```

Figure 8: The 3-*magic square* instance (continued)

```
<instance>
  <presentation name="ExampleQCSP" format="XCSP 2.1" type="QCSP">
    This is a QCSP instance.
  </presentation>

  <domains nbDomains="1">
    <domain name="D0" nbValues="4">
      1..4
    </domain>
  </domains>

  <variables nbVariables="4" >
    <variable name="W" domain="D0"/>
    <variable name="X" domain="D0"/>
    <variable name="Y" domain="D0"/>
    <variable name="Z" domain="D0"/>
  </variables>

  <predicates nbPredicates="2">
    <predicate name="P0">
      <parameters> int A int B  int C int D </parameters>
      <expression>
        <functional> eq(add(A,B),add(C,D)) </functional>
      </expression>
    </predicate>
    <predicate name="P1">
      <parameters> int A int B </parameters>
      <expression>
        <functional> ne(A,B) </functional>
      </expression>
    </predicate>
    </predicates>

  <constraints nbConstraints="2">
    <constraint name="C0" arity="4" scope="W X Y Z" reference="P0">
      <parameters> W X Y Z </parameters>
    </constraint>
    <constraint name="C1" arity="2" scope="Y Z" reference="P1">
      <parameters> Y Z </parameters>
    </constraint>
  </constraints>

  <quantification nbBlocks="3">
    <block quantifier="exists" scope="W X" \>
    <block quantifier="forall" scope="Y" \>
    <block quantifier="exists" scope="Z" \>
  </quantification >
</instance>
```

Figure 9: A QCSP instance

```
<instance>
  <presentation name="ExampleQCSP+" format="XCSP 2.1" type="QCSP+">
    This is a QCSP+ instance.
  </presentation>

  <domains nbDomains="1">
    <domain name="D0" nbValues="4">
      1..4
    </domain>
  </domains>

  <variables nbVariables="4" >
    <variable name="W" domain="D0"/>
    <variable name="X" domain="D0"/>
    <variable name="Y" domain="D0"/>
    <variable name="Z" domain="D0"/>
  </variables>

  <predicates nbPredicates="4">
     <predicate name="myP">
     <parameters> int A int B int C int D </parameters>
     <expression>
       <functional> eq(add(A,B),add(C,D)) </functional>
     </expression>
    </predicate>
    <predicate name="sum_lt">
      <parameters> int A int B int C</parameters>
      <expression>
        <functional> lt(add(A,B),C) </functional>
      </expression>
    </predicate>
    <predicate name="sub_gt">
      <parameters> int A int B int C</parameters>
      <expression>
        <functional> gt(sub(A,B),C)</functional>
      </expression>
    </predicate>
    <predicate name="neq">
      <parameters> int A int B</parameters>
      <expression>
        <functional> ne(A,B)</functional>
      </expression>
    </predicate>
  </predicates>
  ...
```

Figure 10: A QCSP$^+$ instance (to be continued)

```
  ...
  <constraints nbConstraints="1">
    <constraint name="goal" arity="4" scope="W X Y Z" reference="myP">
      <parameters> W X Y Z </parameters>
    </constraint>
  </constraints>

  <quantification nbBlocks="3">
    <block quantifier="exists" scope="W X">
      <constraint name="restr1_c1" arity="2" scope="W X" reference="sum_lt">
        <parameters> W X 8 </parameters>
      </constraint>
      <constraint name="restr1_c2" arity="2" scope="W X" reference="sub_gt">
        <parameters> W X 2 </parameters>
      </constraint>
    </block>

    <block quantifier="universal" scope="Y">
      <constraint name="restr2_c1" arity="2" scope="W Y" reference="neq">
        <parameters> W Y </parameters>
      </constraint>
      <constraint name="restr2_c2" arity="2" scope="X Y" reference="neq">
        <parameters> X Y </parameters>
      </constraint>
    </block>

    <block quantifier="existential" scope="Z">
      <constraint name="restr3_c1" arity="3" scope="W Y Z" reference="sub_gt">
        <parameters> W Y Z </parameters>
      </constraint>
    </block>
  </quantification >
</instance>
```

Figure 11: A QCSP$^+$ instance (continued)

```
<instance>
  <presentation  name="ExampleWCSP"  format="XCSP 2.1"  type="WCSP">
     This is a WCSP instance.
  </presentation>

  <domains nbDomains="1">
    <domain name="D0" nbValues="3">0..2</domain>
  </domains>

  <variables nbVariables="4">
    <variable name="V0" domain="D0"/>
    <variable name="V1" domain="D0"/>
    <variable name="V2" domain="D0"/>
    <variable name="V3" domain="D0"/>
  </variables>

  <relations nbRelations="6">
    <relation name="R0" arity="2" nbTuples="10" semantics="soft" defaultCost="0">
      5:0 0|0 1|1 0|1 1|1 2|2 1|2 2|2 3|3 2|3 3
    </relation>
    <relation name="R1" arity="1" nbTuples="2" semantics="soft" defaultCost="0">
      1:1|3
    </relation>
    <relation name="R2" arity="1" nbTuples="2" semantics="soft" defaultCost="0">
      1:1|2
    </relation>
    <relation name="R3" arity="1" nbTuples="2" semantics="soft" defaultCost="0">
      1:0|2
    </relation>
  </relations>

   <functions nbFunctions="2">
     <function name="F0" return="int">
       <parameters> int X int Y </parameters>
       <expression>
         <functional> if(eq(X,Y),0,5) </functional>
       </expression>
     </function>
     <function name="F1" return="int">
       <parameters> int X int Y int Z  </parameters>
       <expression>
         <functional> if(gt(mul(add(X,Y),Z),5),0,2) </functional>
       </expression>
     </function>
   </functions>

  <constraints nbConstraints="7" initialCost="0" maximalCost="5">
    <constraint name="C0" arity="2" scope="V0 V1" reference="R0" />
    <constraint name="C1" arity="2" scope="V0 V2" reference="F0" />
    <constraint name="C2" arity="3" scope="V1 V2 V3" reference="F1" />
    <constraint name="C3" arity="1" scope="V0" reference="R1" />
    <constraint name="C4" arity="1" scope="V1" reference="R2" />
    <constraint name="C5" arity="1" scope="V2" reference="R3" />
  </constraints>
</instance>
```

Figure 12: A WCSP instance