Structure Preserved by XCSP3
An Interesting Feature for Competitions

Christophe Lecoutre

CRIL-CNRS UMR 8188
Universite d'Artois
Lens, France

Pragmatics of Constraint Reasoning
August 28, 2017

Outline

1. Modeling with MCSP3
2. Representing Instances with XCSP3
   - Representing Variables
   - Representing Constraints
   - Representing Objectives
3. Structure Preserved by XCSP3
4. What about Competitions?

Modeling Languages

Modeling languages are languages that can be used to model problems, using some form of control and abstraction.

Typically, a model represents a family of problem instances, by referring to some data parameters. Modeling a (family of) problem involves:

1. the description of the structure of the data, seen as parameters, for the problem
2. the description of the model, taking data parameters into account, using an appropriate language
3. the generation of the effective data (files) corresponding to the different instances to be solved

Let us illustrate this with the academic problem "All-Interval Series".

Modeling Languages

Modeling languages are languages that can be used to model problems, using some form of control and abstraction.

Typically, a model represents a family of problem instances, by referring to some data parameters. Modeling a (family of) problem involves:

1. the description of the structure of the data, seen as parameters, for the problem
2. the description of the model, taking data parameters into account, using an appropriate language
3. the generation of the effective data (files) corresponding to the different instances to be solved

Let us illustrate this with the academic problem "All-Interval Series".

Modeling Languages

Modeling languages are languages that can be used to model problems, using some form of control and abstraction.

Typically, a model represents a family of problem instances, by referring to some data parameters. Modeling a (family of) problem involves:

1. the description of the structure of the data, seen as parameters, for the problem
2. the description of the model, taking data parameters into account, using an appropriate language
3. the generation of the effective data (files) corresponding to the different instances to be solved

Let us illustrate this with the academic problem "All-Interval Series".

All-Interval Series (CSPLib 007)

Given $n \in \mathbb{N}$, find a vector $x = \langle x_1, x_2, \ldots, x_n \rangle$, such that

- x is a permutation of $\{0, 1, \ldots, n-1\}$
- $y = \langle y_1, y_2, \ldots, y_{n-1} \rangle = \langle |x_2 - x_1|, |x_3 - x_2|, \ldots, |x_n - x_{n-1}| \rangle$ is a vector that is a permutation of $\{1, 2, \ldots, n-1\}$.

So, now, we have to:

1. define the structure (type) of the data for this problem
2. define the model, parameterized with data structure
3. propose effective data corresponding to different problem instances

All-Interval Series (CSPLib 007)

Given $n \in \mathbb{N}$, find a vector $x = \langle x_1, x_2, \ldots, x_n \rangle$, such that

- x is a permutation of $\{0, 1, \ldots, n-1\}$
- $y = \langle y_1, y_2, \ldots, y_{n-1} \rangle = \langle |x_2 - x_1|, |x_3 - x_2|, \ldots, |x_n - x_{n-1}| \rangle$ is a vector that is a permutation of $\{1, 2, \ldots, n-1\}$.

So, now, we have to:

1. define the structure (type) of the data for this problem
2. define the model, parameterized with data structure
3. propose effective data corresponding to different problem instances

Data for All-Interval Series

We just need an integer for representing the order (n) of the problem instance.

Which format to choose for representing data?

- Tabular (Text)
- XML
- JSON

Hence, JSON is a good choice for representing effective data. For example, for order 5, we can generate a file containing:

```
{
  "n": 5
}
```

Remark.

Technically, when the data parameters are very basic, there is no real need to generate data files.

Data for All-Interval Series

We just need an integer for representing the order (n) of the problem instance.

Which format to choose for representing data?

- Tabular (Text)
- XML
- JSON

Hence, JSON is a good choice for representing effective data. For example, for order 5, we can generate a file containing:

```
{
  "n": 5
}
```

Remark.

Technically, when the data parameters are very basic, there is no real need to generate data files.

Data for All-Interval Series

We just need an integer for representing the order (n) of the problem instance.

Which format to choose for representing data?

- Tabular (Text)
- XML
- JSON $\Leftarrow$ a lightweight data-interchange format (our choice)

Hence, JSON is a good choice for representing effective data. For example, for order 5, we can generate a file containing:

```
{
  "n": 5
}
```

Remark.
Technically, when the data parameters are very basic, there is no real need to generate data files.

Data for All-Interval Series

We just need an integer for representing the order (n) of the problem instance.

Which format to choose for representing data?

- Tabular (Text)
- XML
- JSON ⇐ a lightweight data-interchange format (our choice)

Hence, JSON is a good choice for representing effective data. For example, for order 5, we can generate a file containing:

```
{
  "n": 5
}
```

Remark.
Technically, when the data parameters are very basic, there is no real need to generate data files.

Data for All-Interval Series

We just need an integer for representing the order (n) of the problem instance.

Which format to choose for representing data?

- Tabular (Text)
- XML
- JSON $\Leftarrow$ a lightweight data-interchange format (our choice)

Hence, JSON is a good choice for representing effective data. For example, for order 5, we can generate a file containing:

```
{
  "n": 5
}
```

Remark.

Technically, when the data parameters are very basic, there is no real need to generate data files.

Model for All-Interval Series

With n being the unique parameter for this problem, the structure of a natural model is:

- Variables
  - x, one-dimensional array of n integer variables
  - y, one-dimensional array of $n-1$ integer variables
- Constraints
  - two constraints `allDifferent`
  - a group of constraints linking x and y

Which language to choose for building models?

- AMPL
- OPL
- MiniZinc
- Essence
- MCSP3

Model for All-Interval Series

With n being the unique parameter for this problem, the structure of a natural model is:

- Variables
  - x, one-dimensional array of n integer variables
  - y, one-dimensional array of $n-1$ integer variables
- Constraints
  - two constraints `allDifferent`
  - a group of constraints linking x and y

Which language to choose for building models?

- AMPL
- OPL
- MiniZinc
- Essence
- MCSP3

Model for All-Interval Series

With n being the unique parameter for this problem, the structure of a natural model is:

- Variables
  - x, one-dimensional array of n integer variables
  - y, one-dimensional array of $n-1$ integer variables
- Constraints
  - two constraints `allDifferent`
  - a group of constraints linking x and y

Which language to choose for building models?

- AMPL
- OPL
- MiniZinc
- Essence
- MCSP3 $\Leftarrow$ a Java-based API (our choice)

MCSP3 Model for All-Interval Series

```
class AllInterval implements ProblemAPI {
  // Data
  int n;

  public void model() {
    // Variables
    Var[] x = array("x", size(n), dom(range(n)),
      "x[i] is the ith value of the series");
    Var[] y = array("y", size(n-1), dom(range(1, n-1)),
      "y[i] is the distance from x[i] to x[i+1]");

    // Constraints
    allDifferent(x);
    allDifferent(y);
    forall(range(n - 1),
      i -> equal(y[i], dist(x[i], x[i+1])));
  }
}
```

Modelling Languages and Solvers

Unfortunately, most of the solvers cannot directly read/understand modeling languages. For each problem instance, identified by a model and effective data, we have to generate a specific representation (new file).

Which format to choose for representing instances?

- XCSP 2.1
- FlatZinc
- XCSP3

Important:

- XCSP 2.1 and FlatZinc are flat formats
- XCSP3 is an intermediate format that preserves the structure of the problems/models

Modelling Languages and Solvers

Unfortunately, most of the solvers cannot directly read/understand modeling languages. For each problem instance, identified by a model and effective data, we have to generate a specific representation (new file).

Which format to choose for representing instances?

- XCSP 2.1
- FlatZinc
- XCSP3

Important:

- XCSP 2.1 and FlatZinc are flat formats
- XCSP3 is an intermediate format that preserves the structure of the problems/models

Modelling Languages and Solvers

Unfortunately, most of the solvers cannot directly read/understand modeling languages. For each problem instance, identified by a model and effective data, we have to generate a specific representation (new file).

Which format to choose for representing instances?

- XCSP 2.1
- FlatZinc
- XCSP3 ⇐ an XML-based representation (our choice)

Important:

- XCSP 2.1 and FlatZinc are flat formats
- XCSP3 is an intermediate format that preserves the structure of the problems/models

Modelling Languages and Solvers

Unfortunately, most of the solvers cannot directly read/understand modeling languages. For each problem instance, identified by a model and effective data, we have to generate a specific representation (new file).

Which format to choose for representing instances?

- XCSP 2.1
- FlatZinc
- XCSP3 ⇐ an XML-based representation (our choice)

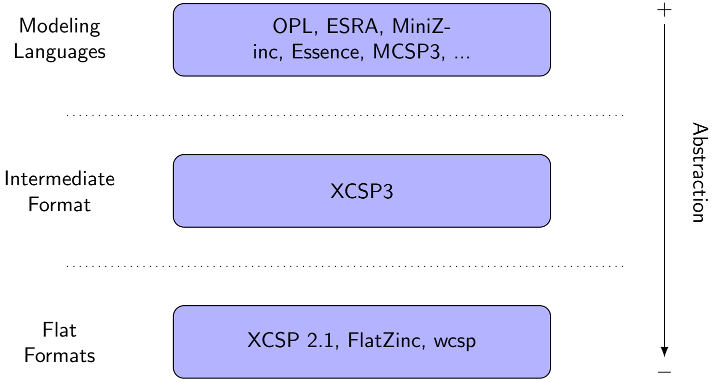
Important:

- XCSP 2.1 and FlatZinc are flat formats
- XCSP3 is an intermediate format that preserves the structure of the problems/models

XCSP3 Instance: AllInterval-05

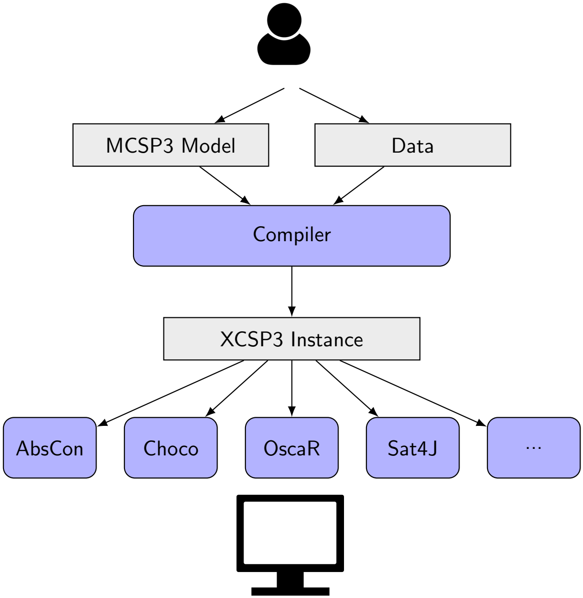
```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" note="x[i]: the ith value of the series"
      size="[5]"> 0..4 </array>
    <array id="y" note="y[i]: distance from x[i] to x[i+1]"
      size="[4]"> 1..4 </array>
  </variables>
  <constraints>
    <allDifferent> x[] </allDifferent>
    <allDifferent> y[] </allDifferent>
    <group>
      <intension> eq(%0,dist(%1,%2)) </intension>
      <args> y[0] x[0] x[1] </args>
      <args> y[1] x[1] x[2] </args>
      <args> y[2] x[2] x[3] </args>
      <args> y[3] x[3] x[4] </args>
    </group>
  </constraints>
</instance>
```

Modeling Languages and Formats



www.xcsp.org

A Complete Modeling/Solving Toolchain



Mainstream Technologies

The complete Toolchain MCSP3 + XCSP3 has many advantages:

- JSON, Java and XML are robust mainstream technologies
- Using JSON permits to have a unified notation, easy to read for both humans and machines
- Using Java permits the user to avoid learning again a new programming language
- Using a coarse-grained XML structure permits to have quite readable problem descriptions, easy to read for both humans and machines

Remark.

At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Mainstream Technologies

The complete Toolchain $\mathrm{MCSP3} + \mathrm{XCSP3}$ has many advantages:

- JSON, Java and XML are robust mainstream technologies
- Using JSON permits to have a unified notation, easy to read for both humans and machines
- Using Java permits the user to avoid learning again a new programming language
- Using a coarse-grained XML structure permits to have quite readable problem descriptions, easy to read for both humans and machines

Remark.
At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Mainstream Technologies

The complete Toolchain $\mathrm{MCSP3} + \mathrm{XCSP3}$ has many advantages:

- JSON, Java and XML are robust mainstream technologies
- Using JSON permits to have a unified notation, easy to read for both humans and machines
- Using Java permits the user to avoid learning again a new programming language
- Using a coarse-grained XML structure permits to have quite readable problem descriptions, easy to read for both humans and machines

Remark.
At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Mainstream Technologies

The complete Toolchain MCSP3 + XCSP3 has many advantages:

- JSON, Java and XML are robust mainstream technologies
- Using JSON permits to have a unified notation, easy to read for both humans and machines
- Using Java permits the user to avoid learning again a new programming language
- Using a coarse-grained XML structure permits to have quite readable problem descriptions, easy to read for both humans and machines

Remark.

At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Mainstream Technologies

The complete Toolchain MCSP3 + XCSP3 has many advantages:

- JSON, Java and XML are robust mainstream technologies
- Using JSON permits to have a unified notation, easy to read for both humans and machines
- Using Java permits the user to avoid learning again a new programming language
- Using a coarse-grained XML structure permits to have quite readable problem descriptions, easy to read for both humans and machines

Remark.

At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Mainstream Technologies

The complete Toolchain $\mathrm{MCSP3} + \mathrm{XCSP3}$ has many advantages:

- JSON, Java and XML are robust mainstream technologies
- Using JSON permits to have a unified notation, easy to read for both humans and machines
- Using Java permits the user to avoid learning again a new programming language
- Using a coarse-grained XML structure permits to have quite readable problem descriptions, easy to read for both humans and machines

Remark.
At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Outline

Skeleton of XCSP3 Instances

Syntax

```
<instance format="XCSP3" type="frameworkType">










</instance>
```

Skeleton of XCSP3 Instances

Syntax

```
<instance format="XCSP3" type="frameworkType">
  <variables>
    (   <var.../>
      | <array.../>
    )+
  </variables>









</instance>
```

Skeleton of XCSP3 Instances

Syntax

```
<instance format="XCSP3" type="frameworkType">





  <constraints>
    (   <constraint.../>
      | <metaConstraint.../>
      | <group.../>
      | <block.../>
    )*
  </constraints>






</instance>
```

Skeleton of XCSP3 Instances

Syntax

```
<instance format="XCSP3" type="frameworkType">













  [<objectives [combination="combinationType"]>
    (   <minimize.../>
      | <maximize.../>
    )+
   </objectives>]

</instance>
```

Skeleton of XCSP3 Instances

Syntax

```
<instance format="XCSP3" type="frameworkType">













  [<annotations.../>]
</instance>
```

Skeleton of XCSP3 Instances

Syntax

```
<instance format="XCSP3" type="frameworkType">
  <variables>
    (   <var.../>
      | <array.../>
    )+
  </variables>
  <constraints>
    (   <constraint.../>
      | <metaConstraint.../>
      | <group.../>
      | <block.../>
    )*
  </constraints>
  [<objectives [combination="combinationType"]>
    (   <minimize.../>
      | <maximize.../>
    )+
   </objectives>]
  [<annotations.../>]
</instance>
```

Outline

Variables

- **Variables**
  - Discrete Variables
    - Simple Discrete Variables
      - Integer variables
      - Symbolic Variables
    - Complex Discrete Variables
      - Set variables
      - Graph Variables
    - Stochastic Discrete Variables
  - Continuous Variables
    - Real Variables
    - Qualitative Variables
  - Arrays of Variables

Integer Variables

Syntax

```
<var id="identifier" [type="integer"]>
  ((intVal | intIntvl) wspace)*
</var>
```

Example

```
<var id="foo"> 0 1 2 3 4 5 6 </var>
<var id="bar"> 0..6 </var>
<var id="qux"> -6..-2 0 1..3 4 7 8..11 </var>

<var id="b1"> 0 1 </var>
<var id="b2"> 0 1 </var>

<var id="x"> 0..+infinity </var>
<var id="y"> -infinity..+infinity </var>
```

Symbolic Variables

Syntax

```
<var id="identifier" type="symbolic">
  (symbol wspace)*
</var>
```

Example

```
<var id="trafficLight" type="symbolic">
  green orange red
</var>

<var id="person" type="symbolic">
  tom oliver paul john
</var>
```

Real Variables

Syntax

```
<var id="identifier" type="real">
  (realIntvl wspace)*
</var>
```

Example

```
<var id="w" type="real"> [0,+infinity[ </var>
<var id="x" type="real"> [-4,4] </var>
<var id="y" type="real"> [2/3,8.355] [10,12.8] </var>
```

Set Variables

A set domain is approximated by a set interval specified by its upper and lower bounds (subset-bound representation).

Syntax

```
<var id="identifier" type="set">
  [<required> ((intVal | intIntvl) wspace)* </required>
   <possible> ((intVal | intIntvl) wspace)* </possible>]
</var>
```

For a set variable s of domain $[\{1,5\},\{1,3,5,6\}]$, we have:

Example

```
<var id="s" type="set">
  <required> 1 5 </required>
  <possible> 3 6 </possible>
</var>
```

Remark

It is possible to define symbolic set variables too.

Arrays of Variables

Interestingly, XCSP3 allows us to declare k-dimensional arrays of variables, with $k \geq 1$.

Syntax

```
<array id="identifier" [type="varType"] size="dimensions">
  ...
</array>
```

Example

```
<array id="x" size="[10]"> 1..100 </array>
<array id="y" size="[5][8]"> 2 4 6 8 10 </array>
<array id="z" size="[4][4][2]"> 0 1 </array>

<array id="t" size="[12]" type="symbolic set">
  <required> a b </required>
  <possible> c d </possible>
</array>
```

Outline

Constraints

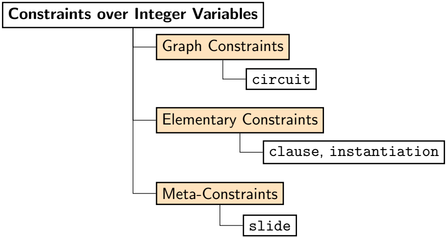
- Constraints over Simple Discrete Variables
  - Constraints over Integer Variables
  - Constraints over Symbolic Variables
- Constraints over Complex Discrete Variables
  - Constraints over Set Variables
  - Constraints over Graph Variables
- Constraints over Continuous Variables
  - Constraints over Real Variables
  - Constraints over Qualitative Variables

Popular constraints: XCSP3-core

Constraints over Integer Variables

- Generic Constraints
  - `intension`, `extension`
- Language-based Constraints
  - `regular`, `mdd`
- Comparison-based Constraints
  - `allDifferent`, `allEqual`
  - `ordered`, `lex`
- Counting and Summing Constraints
  - `sum` (`linear`)
  - `count` (capturing `atLeast`, `atMost`,`exactly`, `among`)
  - `nValues`, `cardinality`
- Connection Constraints
  - `minimum`, `maximum`
  - `element`, `channel`
- Packing and Scheduling Constraints
  - `noOverlap` (capturing `disjunctive` and `diffn`)
  - `cumulative`

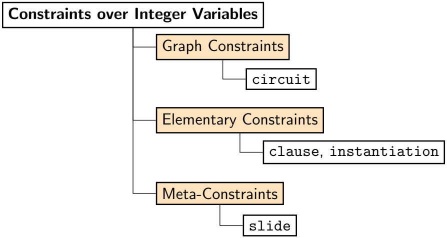
Popular constraints: XCSP3-core



Note that XCSP3-core is;

- sufficient for modeling many problems
- used in the 2017 XCSP3 Solver Competititon

Popular constraints: XCSP3-core



Note that XCSP3-core is;

- sufficient for modeling many problems
- used in the 2017 XCSP3 Solver Competititon

Constraint intension

Syntax

```
<intension> booleanExpression </intension>                    // Simplified Form
```

The constraints

$$c_1 : x + y = z$$
$$c_2 : w \geq z$$

are represented by:

Example

```
<intension id="c1"> eq(add(x,y),z) </intension>
<intension id="c2"> ge(w,z) </intension>
```

Constraint extension

For *positive* table constraints, we have:

Syntax

```
<extension>
  <list> ... </list>
  <supports> ... </supports>
</extension>
```

For *negative* table constraints, we have:

Syntax

```
<extension>
  <list> ... </list>
  <conflicts> ... </conflicts>
</extension>
```

Remark

The syntax is precisely given in the document introducing XCSP3 *specifications.*

Constraint extension

The constraints

$$(x_1, x_2, x_3) \in \{(0,1,0),(1,0,0),(1,1,0),(1,1,1)\}$$
$$(y_1, y_2, y_3, y_4) \notin \{(1,2,3,4),(3,1,3,4)\}$$

are respectively represented by:

Example

```
<extension>
  <list> x1 x2 x3 </list>
  <supports> (0,1,0)(1,0,0)(1,1,0)(1,1,1) </supports>
</extension>

<extension>
  <list> y1 y2 y3 y4 </list>
  <conflicts> (1,2,3,4)(3,1,3,4) </conflicts>
</extension>
```

Constraint allDifferent

Syntax

```
<allDifferent>
  <list> ... </list>
  [<except> ... </except>]
</allDifferent>
```

Tags of <list> are optional if <list> is the unique parameter of the constraint.

Example

```
<allDifferent>
  x1 x2 x3 x4 x5
</allDifferent>
<allDifferent>
  <list> y[] </list>
  <except> 0 </except>
</allDifferent>
```

Constraint sum (linear)

Syntax

```
<sum>
   <list> ... </list>
   [ <coeffs> ... </coeffs> ]
   <condition> ... </condition>
</sum>
```

Semantics

sum$(X, C, (\odot, k))$, with $X = \langle x_1, x_2, \ldots \rangle$, and $C = \langle c_1, c_2, \ldots \rangle$, iff
$(\sum_{i=1}^{|X|} c_i \times \boldsymbol{x}_i) \odot \boldsymbol{k}$

The linear function $x_1 \times 1 + x_2 \times 2 + x_3 \times 3 > y$ is expressed as:

Example

```
<sum>
  <list> x1 x2 x3 </list>
  <coeffs> 1 2 3 </coeffs>
  <condition> (gt,y) </condition>
</sum>
```

Outline

Dealing with Optimization

The syntax for dealing with optimization is:

Syntax

```
<objectives [combination="combinationType"]>
  (<minimize.../> | <maximize.../>)+
</objectives>
```

When there are several objectives, the element `<objectives>` has an attribute `combination`, whose role is illustrated in the two next slides.

Objectives in Functional Form

Syntax

```
<minimize>
  funtionalExpression
</minimize>

<maximize>
  funtionalExpression
</maximize>
```

Example

```
<objectives combination="lexico">
  <minimize> z </minimize>
  <maximize> add(x,mul(y,2)) </maximize>
</objectives>
```

Outline

Advanced Forms in XCSP3

Many forms of constraints:

- Constraints lifted to lists, sets, and multisets
- Restricted constraints
- Soft constraints
- Weighted constraints (cost functions)
- Sliding constraints (`slide` `seqbin`)
- meta-constraints (`and`, `or` and `not`)

This allows us to represent problem instances in a rather high-level representation. This participates to keeping structure.

Structure of Problems/Models

Some XCSP3 constructions allow us to preserve the structure of problems/models:

- arrays of variables (already introduced)
- groups of constraints
- blocks of constraints
- meta-constraints
- classes (tags)

Groups of Constraints

Useful for posting together constraints of similar syntax.

Syntax

```
<group [id="identifier"]>
   <constraint.../>                                    // constraint template
   (<args> ... </args>)2+
</group>
```

Example

```
<group>
  <extension>
    <list> %0 %1 </list>
    <supports> (1,2)(2,1)(2,3)(3,1)(3,2) </supports>
  </extension>
  <args> w x </args>
  <args> x y </args>
  <args> y z </args>
</group>
```

Blocks of Constraints

Useful for linking constraints semantically.

Syntax

```
<block [class="(identifier wspace)+"]>
   (<constraint.../> | <metaConstraint.../> | <group.../>)+
</block>
```

Example

```
<constraints>
  <block class="clues">
    <intension> ... </intension>
    <intension> ... </intension>
    ...
  </block>
  <block class="symmetryBreaking">
    <lex> ... </lex>
    <lex> ... </lex>
    ...
  </block>
  <block note="Management of first week"> ... </block>
  <block note="Management of second week"> ... </block>
</constraints>
```

Back to Modeling

Basically, modeling a problem consists in:

- identifying arrays of variables
- identifying groups of constraints

With XCSP3, we can keep such structure.

This was illustrated before with one example. Let us dot it now with sports scheduling:

- first, a model MCSP3
- second, an XCSP3 instance for nTeams=4.

Sports Tournament Scheduling (CSPLib 026)

The problem is to schedule a tournament of n teams over $n-1$ weeks, with each week divided into $n/2$ periods, and each period divided into two slots. The first team in each slot plays at home, whilst the second plays the first team away. A tournament must satisfy the following three constraints:

- every team plays once a week;
- every team plays at most twice in the same period over the tournament;
- every team plays every other team.

Sports Tournament Scheduling (CSPLib 026)

An example schedule for 8 teams is:

Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7
1 - 2	2 - 4	1 - 4	2 - 8	1 - 6	2 - 5	1 - 8
4 - 7	3 - 1	3 - 5	4 - 6	3 - 2	4 - 3	3 - 6
6 - 5	5 - 8	6 - 2	5 - 1	5 - 7	6 - 8	5 - 4
8 - 3	7 - 6	8 - 7	7 - 3	8 - 4	7 - 1	7 - 2

MCSP3 Model (SportsScheduling)

```
class SportsScheduling implements ProblemAPI {
  int nTeams;

  public void model() {
    // Here, some statements for defining nP(eriods), nW(eeks)...
    Var[][] h = array("h", size(nP, nW), dom(range(nTeams)),
      "h[p][w] is the number of the home opponent");
    Var[][] a = array("a", size(nP, nW), dom(range(nTeams)),
      "a[p][w] is the number of the away opponent");
    Var[][] m = array("m", size(nP, nW), dom(range(nPM)),
      "m[p][w] is the number of the match");

    forall(range(nP).range(nW), (p, w) ->
      extension(vars(h[p][w], a[p][w], m[p][w]), numbers))
    .note("Linking variables through ternary table constraints");

    allDifferent(m).note("All matches are different");

    forall(range(nW), w ->
      allDifferent(vars(columnOf(h, w), columnOf(a, w))))
    .note("Each week, all teams are different");

    forall(range(nP), p ->
      cardinality(vars(h[p], a[p]), vals(range(nTeams)),
          occursEachBetween(1, 2)))
    .note("Each team plays at most two times in each period");

    block(() -> {  ... }).tag(SYMMETRY_BREAKING);
}
```

XCSP3 Instance (SportsScheduling-4)

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="h" note="h[p][w] the number of the home opponent"
        size="[2][3]"> 0..3 </array>
    <array id="a" note="a[p][w] the number of the away opponent"
        size="[2][3]"> 0..3 </array>
    <array id="m" note="m[p][w] is the number of the match" size="
        [2][3]"> 0..5 </array>
  </variables>
  <constraints>
    <group note="Linking variables through table constraints">
      <extension>
        <list> %0 %1 %2 </list>
        <supports> (0,1,0)(0,2,1)...(1,3,4)(2,3,5) </supports>
      </extension>
      <args> h[0][0] a[0][0] m[0][0] </args>
      <args> h[0][1] a[0][1] m[0][1] </args>
      <args> h[0][2] a[0][2] m[0][2] </args>
      <args> h[1][0] a[1][0] m[1][0] </args>
      <args> h[1][1] a[1][1] m[1][1] </args>
      <args> h[1][2] a[1][2] m[1][2] </args>
    </group>
    <allDifferent note="All matches are different">
      m[][]
    </allDifferent>
    ...
```

XCSP3 Instance (SportsScheduling-4)

```
...
<group note="Each week, all teams are different">
  <allDifferent> %... </allDifferent>
  <args> h[][0] a[][0] </args>
  <args> h[][1] a[][1] </args>
  <args> h[][2] a[][2] </args>
</group>
<group note="Each team plays at most two times in each per.">
  <cardinality>
    <list> %... </list>
    <values> 0 1 2 3 </values>
    <occurs> 1..2 1..2 1..2 1..2 </occurs>
  </cardinality>
  <args> h[0][] a[0][] </args>
  <args> h[1][] a[1][] </args>
</group>
<block class="symmetryBreaking">
  ...
</block>
</constraints>
</instance>
```

Outline

Two Important Things

For competitions, I think that it is very important to

1. help users with useful tools (parsers, checkers, ...),
2. do not sacrifice structure when "flattening" models into instances.

XCSP3: Available Tools and Benchmarks

Many tools are available on github:

https://github.com/xcsp3team/.

Parsers available on github:

- Java 8 Parser
- C++ 11 Parser

Various tools for:

- checking solutions and bounds: org.xcsp.checker.SolutionChecker
- checking the validity of an instance for a competition track: org.xcsp.checker.CompetitionChecker
- checking the validity of an XCSP3 instance (made available soon)

Many series of CSP/COP instances that can be downloaded from `www.xcsp.org` by means of our selection engine!

XCSP3 Competition 2018

Which directions for new features?

- Constraints
  - short tables, with *
  - constraint `circuit`
  - constraint `allDifferent-List`
  - ...
- Annotations
  - decision variables
  - search heuristics
- Use of classes
  - new XParser(fileName,"symmetryBreaking"); // elements with this tag are discarded
  - new XParser(fileName)
- ...