

Représentation des problèmes combinatoires sous contraintes

Christophe Lecoutre

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

Perspectives et Défis de l'IA

Paris – 6 Octobre 2017

Représentation des problèmes combinatoires sous contraintes

Christophe Lecoutre

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

Perspectives et Défis de l'IA

Paris – 6 Octobre 2017

Notre contexte pour cet exposé !

Deux phases en programmation par contraintes (CP) :

- modélisation
- résolution

Nous nous focaliserons sur la modélisation !

Notre contexte pour cet exposé !

Deux phases en programmation par contraintes (CP) :

- modélisation
- résolution

Nous nous focaliserons sur la modélisation !

Notre contexte pour cet exposé !

Deux phases en programmation par contraintes (CP) :

- modélisation
- résolution

Nous nous focaliserons sur la modélisation !

Notre contexte pour cet exposé !

Deux phases en programmation par contraintes (CP) :

- modélisation
- résolution

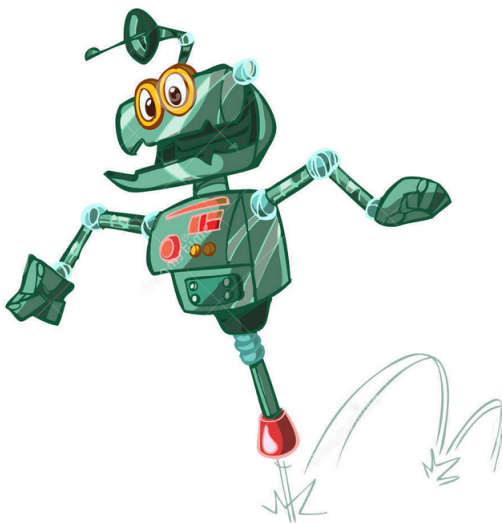
Nous nous focaliserons sur la modélisation !

Point de départ

Quelle image peut-on donner de la CP aujourd'hui ?

Point de départ

Quelle image peut-on donner de la CP aujourd'hui ?



Notre slogan: CP pour tous !



CP pour tous !

Idéalement, la programmation par contraintes pour :

- les experts
- et les nuls

La communauté CP doit faire un effort pour développer :

- des livres grand public (notamment sur la modélisation)
- des langages et formats simples
- des outils facilitant le développement et l'utilisation de solveurs de contraintes

Notre contribution :

- MCSP3, une API de modélisation basée sur Java
- XCSP3, un format de représentation "universel" basé sur XML
- des outils disponibles (parseurs, vérificateurs, site web)
- un livre de vulgarisation ?

CP pour tous !

Idéalement, la programmation par contraintes pour :

- les experts
- et les nuls

La communauté CP doit faire un effort pour développer :

- des livres grand public (notamment sur la modélisation)
- des langages et formats simples
- des outils facilitant le développement et l'utilisation de solveurs de contraintes

Notre contribution :

- MCSP3, une API de modélisation basée sur Java
- XCSP3, un format de représentation "universel" basé sur XML
- des outils disponibles (parseurs, vérificateurs, site web)
- un livre de vulgarisation ?

CP pour tous !

Idéalement, la programmation par contraintes pour :

- les experts
- et les nuls

La communauté CP doit faire un effort pour développer :

- des livres grand public (notamment sur la modélisation)
- des langages et formats simples
- des outils facilitant le développement et l'utilisation de solveurs de contraintes

Notre contribution :

- MCSP3, une API de modélisation basée sur Java
- XCSP3, un format de représentation "universel" basé sur XML
- des outils disponibles (parseurs, vérificateurs, site web)
- un livre de vulgarisation ?

CP pour tous !

Idéalement, la programmation par contraintes pour :

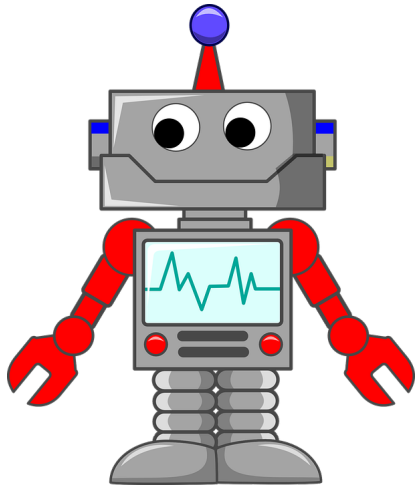
- les experts
- et les nuls

La communauté CP doit faire un effort pour développer :

- des livres grand public (notamment sur la modélisation)
- des langages et formats simples
- des outils facilitant le développement et l'utilisation de solveurs de contraintes

Notre contribution :

- MCSP3, une API de modélisation basée sur Java
- XCSP3, un format de représentation "universel" basé sur XML
- des outils disponibles (parseurs, vérificateurs, site web)
- un livre de vulgarisation ?



Des débuts extrêmement prometteurs :

- Prolog
- OPL

Une suite un peu plus confuse :

- ESRA
- Essence
- MiniZinc

Et aussi des formats de bas niveau :

- XCSP 2.1
- FlatZinc

Des débuts extrêmement prometteurs :

- Prolog
- OPL

Une suite un peu plus confuse :

- ESRA
- Essence
- MiniZinc

Et aussi des formats de bas niveau :

- XCSP 2.1
- FlatZinc

Des débuts extrêmement prometteurs :

- Prolog
- OPL

Une suite un peu plus confuse :

- ESRA
- Essence
- MiniZinc

Et aussi des formats de bas niveau :

- XCSP 2.1
- FlatZinc



XCSP3, nouveau venu en 2015, est :

- un format basé sur XML pour représenter les instances de problèmes combinatoires sous contraintes
- un format intermédiaire intégré préservant la structure des modèles

XCSP3 is a major extension of XCSP 2.1 since it allows us to deal with:

- mono/multi optimization
- many types of variables
- cost functions
- reification and views
- annotations
- variable quantification
- distributed, probabilistic and qualitative reasoning



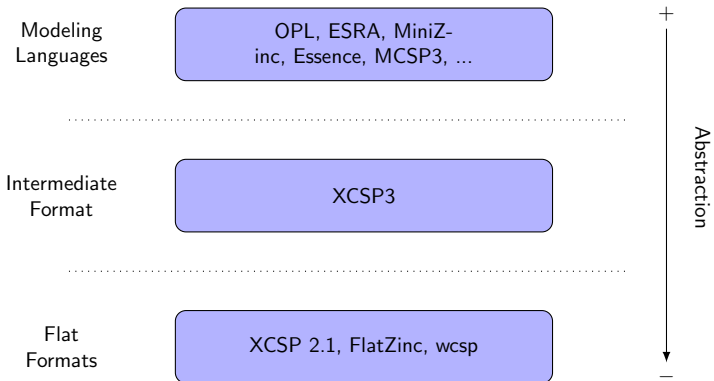
XCSP3, nouveau venu en 2015, est :

- un format basé sur XML pour représenter les instances de problèmes combinatoires sous contraintes
- un format intermédiaire intégré préservant la structure des modèles

XCSP3 is a major extension of XCSP 2.1 since it allows us to deal with:

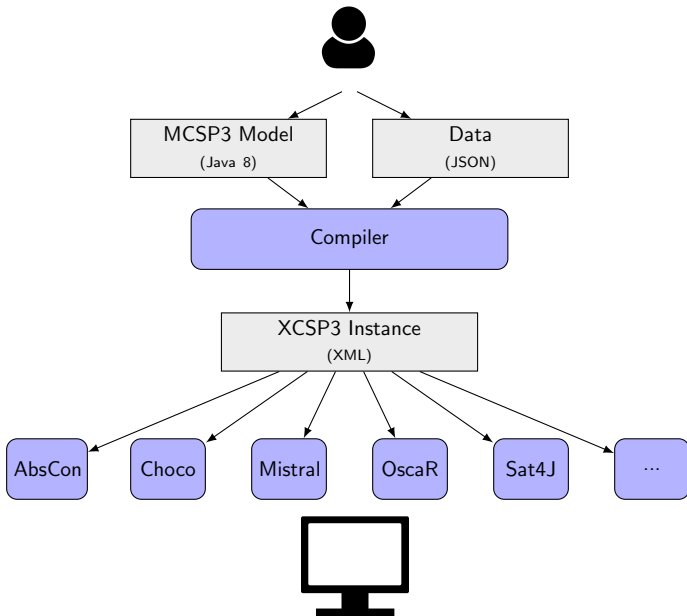
- mono/multi optimization
- many types of variables
- cost functions
- reification and views
- annotations
- variable quantification
- distributed, probabilistic and qualitative reasoning

XCSP3: an Intermediate Format



www.xcsp.org

XCSP3: the central piece of a Modeling/Solving process



- 1 Modéliser avec MCSP3
- 2 Représenter les instances avec XCSP3
 - Représenter les variables
 - Représenter les contraintes
 - Représenter les objectifs
- 3 Préservation de la structure en XCSP3
- 4 CP pour tous !

Modeling Languages

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

Typically, a model represents a family of problem instances, by referring to some parameters denoting the data. Building a model for a problem involves:

- ① the identification of the parameters, i.e., the structure of the data
- ② the description of the **model**, taking the parameters into account, and using an appropriate language
- ③ the generation of the effective data, each time a new instance has to be handled

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

Modeling Languages

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

Typically, a model represents a family of problem instances, by referring to some parameters denoting the data. Building a model for a problem involves:

- ① the identification of the parameters, i.e., the structure of the data
- ② the description of the **model**, taking the parameters into account, and using an appropriate language
- ③ the generation of the effective **data**, each time a new instance has to be handled

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

Modeling Languages

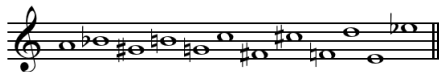
Modeling languages can be used to represent problems, using some form of control and abstraction.

Typically, a model represents a family of problem instances, by referring to some parameters denoting the data. Building a model for a problem involves:

- ① the identification of the **parameters**, i.e., the structure of the data
- ② the description of the **model**, taking the parameters into account, and using an appropriate language
- ③ the generation of the effective **data**, each time a new instance has to be handled

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

All-Interval Series (CSPLib 007)



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

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

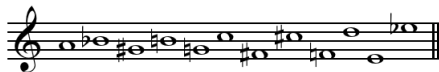
Example of solution for $n = 5$:

$$x = \langle 4, 0, 3, 1, 2 \rangle$$

So, now, we have to:

- ① identify the parameters (structure of the data) for this problem
- ② describe a **model**, using the parameters
- ③ generate effective data for different problem instances

All-Interval Series (CSPLib 007)



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

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

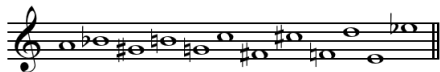
Example of solution for $n = 5$:

$$x = \langle 4, 0, 3, 1, 2 \rangle$$

So, now, we have to:

- 1 identify the parameters (structure of the data) for this problem
- 2 describe a **model**, using the parameters
- 3 generate effective data for different problem instances

All-Interval Series (CSPLib 007)



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

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

Example of solution for $n = 5$:

$$x = \langle 4, 0, 3, 1, 2 \rangle$$

So, now, we have to:

- 1 identify the parameters (structure of the data) for this problem
- 2 describe a model, using the parameters
- 3 generate effective data for different problem instances

Parameters/Data for All-Interval Series

Concerning parameters: we just need an integer for representing the order (n) of the problem instance.

Concerning data: we use JSON, a lightweight data-interchange format, for representing the effective data. For example, we can generate:

- a file *order-5.json* containing `{ "n": 5 }`
- a file *order-6.json* containing `{ "n": 6 }`
- a file *order-7.json* containing `{ "n": 7 }`
- ...

Remark.

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

Parameters/Data for All-Interval Series

Concerning parameters: we just need an integer for representing the order (n) of the problem instance.

Concerning **data**: we use JSON, a lightweight data-interchange format, for representing the effective data. For example, we can generate:

- a file *order-5.json* containing `{ "n": 5 }`
- a file *order-6.json* containing `{ "n": 6 }`
- a file *order-7.json* containing `{ "n": 7 }`
- ...

Remark.

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

Parameters/Data for All-Interval Series

Concerning **parameters**: we just need an integer for representing the order (n) of the problem instance.

Concerning **data**: we use JSON, a lightweight data-interchange format, for representing the effective data. For example, we can generate:

- a file *order-5.json* containing `{ "n": 5 }`
- a file *order-6.json* containing `{ "n": 6 }`
- a file *order-7.json* containing `{ "n": 7 }`
- ...

Remark.

Technically, when the 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, 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

For building models, we use:

MCSP3, a Java-based API

Model for All-Interval Series

With n being the unique parameter for this problem, 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

For building models, we use:

MCSP3, a Java-based API

MCSP3 Model for All-Interval Series

A file *AllInterval.java* containing:

```
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**, i.e., a file such that e.g., *AllInterval.java*
- some effective **data**, i.e., a file such that e.g., *order-5.json*

we have to generate a specific representation in XCSP3 format.

We use a compiler, provided in the API. We just type:

```
java org.xcsp.modeler.Compiler AllInterval -data=order-5.json
```

As a result, we obtain an XCSP3 file *allInterval-5.xml*

Modelling Languages and Solvers

Unfortunately, most of the solvers cannot directly read/understand modeling languages. For each problem instance, identified by:

- a **model**, i.e., a file such that e.g., *AllInterval.java*
- some effective **data**, i.e., a file such that e.g., *order-5.json*

we have to generate a specific representation in XCSP3 format.

We use a compiler, provided in the API. We just type:

```
java org.xcsp.modeler.Compiler AllInterval -data=order-5.json
```

As a result, we obtain an XCSP3 file *allInterval-5.xml*

Modelling Languages and Solvers

Unfortunately, most of the solvers cannot directly read/understand modeling languages. For each problem instance, identified by:

- a **model**, i.e., a file such that e.g., *AllInterval.java*
- some effective **data**, i.e., a file such that e.g., *order-5.json*

we have to generate a specific representation in XCSP3 format.

We use a compiler, provided in the API. We just type:

```
java org.xcsp.modeler.Compiler AllInterval -data=order-5.json
```

As a result, we obtain an XCSP3 file *allInterval-5.xml*

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>
```

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 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 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.

- 1 Modéliser avec MCSP3
- 2 Représenter les instances avec XCSP3
 - Représenter les variables
 - Représenter les contraintes
 - Représenter les objectifs
- 3 Préservation de la structure en XCSP3
- 4 CP pour tous !

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

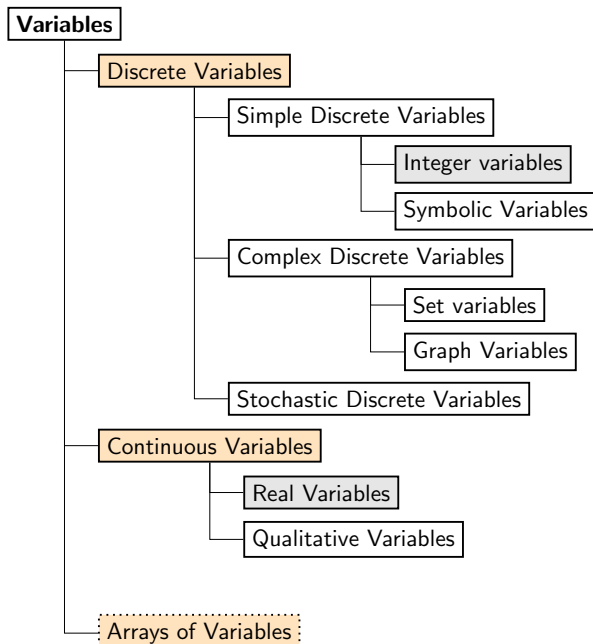


Syntax

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

- 1 Modéliser avec MCSP3
- 2 Représenter les instances avec XCSP3
 - Représenter les variables
 - Représenter les contraintes
 - Représenter les objectifs
- 3 Préservation de la structure en XCSP3
- 4 CP pour tous !

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>
```



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>
```

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>
```


- 1 Modéliser avec MCSP3
- 2 Représenter les instances avec XCSP3
 - Représenter les variables
 - Représenter les contraintes
 - Représenter les objectifs
- 3 Préservation de la structure en XCSP3
- 4 CP pour tous !

Constraints

Constraints over Simple Discrete Variables

Constraints over Integer Variables

Constraints over Symbolic Variables

Constraints over Complex Discrete Variables

Constraints over Set Variables

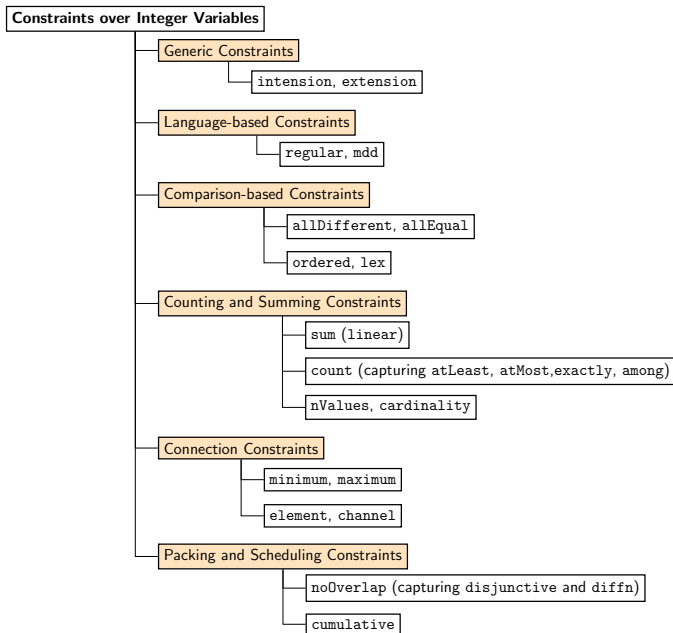
Constraints over Graph Variables

Constraints over Continuous Variables

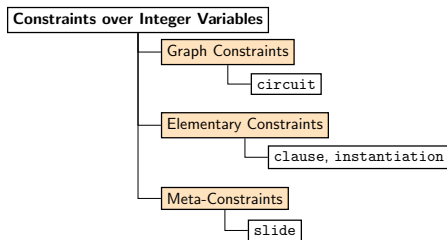
Constraints over Real Variables

Constraints over Qualitative Variables

Popular constraints: XCSP3-core



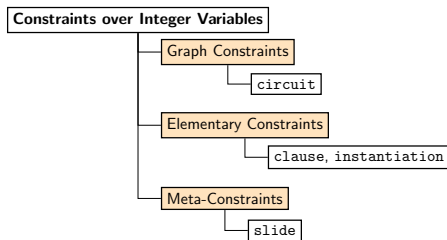
Popular constraints: XCSP3-core



Note that XCSP3-core is;

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

Popular constraints: XCSP3-core



Note that XCSP3-core is;

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

Constraint intension



Syntax

```
<intension> booleanExpression </intension>
```

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

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

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

$\text{sum}(X, C, (\odot, k))$, with $X = \langle x_1, x_2, \dots \rangle$, and $C = \langle c_1, c_2, \dots \rangle$, iff

$$\left(\sum_{i=1}^{|X|} c_i \times x_i \right) \odot k$$

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>
```

- 1 Modéliser avec MCSP3
- 2 Représenter les instances avec XCSP3
 - Représenter les variables
 - Représenter les contraintes
 - Représenter les objectifs
- 3 Préservation de la structure en XCSP3
- 4 CP pour tous !

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>  
  functionalExpression  
</minimize>
```

```
<maximize>  
  functionalExpression  
</maximize>
```

Example

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

- 1 Modéliser avec MCSP3
- 2 Représenter les instances avec XCSP3
 - Représenter les variables
 - Représenter les contraintes
 - Représenter les objectifs
- 3 Préservation de la structure en XCSP3
- 4 CP pour tous !

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 mentioned)
- groups of constraints (already mentioned)
- blocks of constraints
- meta-constraints
- classes (tags)

Blocks of Constraints

Useful for linking constraints semantically.



Syntax

```
<block [class="(identifier whitespace)+"]>
  (<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 do 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(eriodes), 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>
    ...
  </constraints>
</instance>
```

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>
```

- 1 Modéliser avec MCSP3
- 2 Représenter les instances avec XCSP3
 - Représenter les variables
 - Représenter les contraintes
 - Représenter les objectifs
- 3 Préservation de la structure en XCSP3
- 4 CP pour tous !

Rendre accessible CP au plus grand nombre !

$10^3..10^4$ utilisateurs



$10^5..10^6$ utilisateurs



Notre effort en ce sens :

- MCSP3, API de modélisation écrite en Java (8).
Official Release: December 2017
- XCSP3, format de représentation 'universel' basé sur XML,
préservant la structure.
Specifications 3.0.3, Août 2016.

Rendre accessible CP au plus grand nombre !

$10^3..10^4$ utilisateurs



$10^5..10^6$ utilisateurs



Notre effort en ce sens :

- MCSP3, API de modélisation écrite en Java (8).
Official Release: December 2017
- XCSP3, format de représentation 'universel' basé sur XML,
préservant la structure.
Specifications 3.0.3, Août 2016.



De nombreux outils sur github:

<https://github.com/xcsp3team/>.

Parseurs :

- Java 8 Parser
- C++ 11 Parser

Vérificateurs :

- solutions et bornes (org.xcsp.checker.SolutionChecker)
- validité des instances XCSP3 (release en Novembre 2017)



Mise à disposition des **instances** :

- plus de 23000 instances téléchargeables sur www.xcsp.org,
- possibilité d'utiliser un moteur de sélection.

Mise à disposition des solveurs :

- exécutables, et parfois sources sur <https://www.cril.univ-artois.fr/CompetitionXCSP17/>,
- trace des solveurs exécutés pendant la compétition.



Mise à disposition des **instances** :

- plus de 23000 instances téléchargeables sur www.xcsp.org,
- possibilité d'utiliser un moteur de sélection.

Mise à disposition des **solveurs** :

- exécutables, et parfois sources sur <https://www.cril.univ-artois.fr/CompetitionXCSP17/>,
- trace des solveurs exécutés pendant la compétition.