Java Parser for XCSP3
Version 1.2

Christophe Lecoutre
CRIL CNRS, UMR 8188
University of Artois, France
lecoutre@cril.fr

May 19, 2016

The version 1.2 of the XCSP3 Java Parser recognizes almost all concepts of XCSP3. Its current limitations are the following:

- the attribute `as` can only be used for variables and arrays of variables (but for the moment, no instance among the $15,000$ that have been generated, exploits this attribute differently).
- compressed tuples are not handled (but shortened tuples are).

Note that this is a Java parser that uses DOM (Document Object Model) to parse XCSP3 instances. However, feel free to adapt it.

As mentioned above, the Java parser (class `XParser`) is rather complete. It allows scanning an XCSP3 file and to build a pre-digested representation of the various objects of the instance. However to simplify things, we propose an interface that is composed of callback functions. This interface corresponds to classes `XCallbacks` and `XCallbacks2`. Currently, this interface has been developed for XCSP3-core, i.e., for integer variables, the 20 most popular constraints over integer variables, and mono-optimization. This interface will be progressively extended.

To start, once you have imported the classes required to parse (with DOM) XCSP3 instances, you should start writing (in Java 8) something like:

```
class XCSP3 implements XCallbacks2 {

  XCSP3(String fileName) throws Exception {
    Document document = XUtility.loadDocument(fileName);
  }
}
```

If when running this code on a given filename, an exception is not raised, that's fine (maybe, as a first attempt, use an uncompressed file as for example, `queens8.xml` instead of `queens8.xml.lzma`). Otherwise, you may need to redefine the way you can convert an XCSP3 file (whose name is given) into an object Document, as follows:

```
class XCSP3 implements XCallbacks2 {
  public Document loadDocument(String fileName) throws Exception {
    // put your code here to convert the file, whose name is given, into a Document
    // See the code in XUtility.loadDocument(fileName) for a hint
  }

  XCSP3(String fileName) throws Exception {
    Document doc = XUtility.loadDocument(fileName);
  }
}
```

Now, assume that everything is fine with the loading of the document. To parse the document, we build an object `XParser`, and then we can interact with all entries that were initially in `<variables>`, `<constraints>` and `<objectives>`. These entries are respectively stored in `parser.vEntries`, `parser.cEntries` and `parser.oEntries`. Here, we just display the description of all these entries.

```
class XCSP3 implements XCallbacks2 {

  XCSP3(String fileName) throws Exception {
    Document document = XUtility.loadDocument(fileName);
    XParser parser = new XParser(document);
    parser.vEntries.stream().forEach(e -> System.out.println(e.toString()));
    parser.cEntries.stream().forEach(e -> System.out.println(e.toString()));
    parser.oEntries.stream().forEach(e -> System.out.println(e.toString()));
  }
}
```

Now, consider the following instance `testExtension1.xml`:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <var id="x0"> 0 1 </var>
    <var id="x1"> 0 1 </var>
    <var id="x2"> 0 1 </var>
    <var id="x3"> 0 1 </var>
    <var id="x4"> 0 1 </var>
    <var id="x5"> 0 1 </var>
    <var id="x6"> 0 1 </var>
  </variables>
  <constraints>
    <extension>
      <list> x4 x5 x0 </list>
      <supports> (0,0,1)(0,1,0)(1,0,0)(1,1,1) </supports>
    </extension>
    <extension>
      <list> x0 x6 x1 </list>
      <supports> (0,0,1)(0,1,0)(1,0,0)(1,1,1) </supports>
    </extension>
    <extension>
      <list> x3 x6 x2 </list>
      <supports> (0,0,1)(0,1,0)(1,0,0)(1,1,1) </supports>
    </extension>
    <extension>
      <list> x4 x5 x3 </list>
      <supports> (0,0,0)(0,1,1)(1,0,1)(1,1,0) </supports>
    </extension>
  </constraints>
</instance>
```

and simplify your code so as to just have:

```
class XCSP3 implements XCallbacks2 {
  XCSP3(String fileName) throws Exception {
    loadInstance(fileName);
  }
}
```

Remark 1 *You can provide the name of the classes that you want to discard when calling the XParser constructor. For example,* `parser = new XParser(fileName,"symmetryBreaking");` *But this is an advanced use.*

When you execute the code for the instance `testExtension1.xml` above, you must obtain something like:

```
Missing Implementation
  Method buildVarInteger
  Class org.xcsp.parser.XCallbacks2
  Line 105
```

Here, is is indicated that a method whose name is `buildVarInteger` needs to be implemented. Indeed, to manage integer variables, we have to implement the two following methods:

```
 public void buildVarInteger(XVarInteger x, int minValue, int maxValue);
 public void buildVarInteger(XVarInteger x, int[] values);
```

For managing variables, it seems relevant to introduce a map for making the correspondence between the variables in the parser (`XVarInteger` objects) and the variables in your solver (assumed to correspond to `VarInteger` objects). Note that:

- minValue may be equal to `XConstants.VAL_MINUS_INFINITY_INT`, for denoting $-\infty$, and
- maxValue may be equal to `XConstants.VAL_PLUS_INFINITY_INT`, for denoting $+\infty$.

but we have no such a case for the first 60 problems (and $15,000$ generated instances) we have modeled.

```
class XCSP3 implements XCallbacks2 {
  private Map<XVarInteger, VarInteger> mapVar = new LinkedHashMap<>();

  public void buildVarInteger(XVarInteger xx, int minValue, int maxValue) {
    VarInteger x = ...  // build your solver variable x here using xx.id, minValue and maxValue
    mapVar.put(xx,x);
  }

  public void buildVarInteger(XVarInteger xx, int[] values) {
    VarInteger x = ... // build your solver variable x here using xx.id and values
    mapVar.put(xx,x);
  }

  XCSP3(String fileName) throws Exception {
    loadInstance(fileName);
  }
}
```

Remark 2 *The parser automatically flattens all variables from arrays. Also, all variables with degree 0 (i.e., all useless variables that occur in no constraint and no objective) are automatically discarded. If you want to execute specific code, you can override the following methods from* `XCallbacks`*:*

```
public void loadVariables(XParser parser)
public void loadVar(XVar v, Map<XDom, Object> cache4DomObject)
public void loadArray(XArray va, Map<XDom, Object> cache4DomObject)
```

Now, when you execute the code for the instance `testExtension1.xml` above, you must obtain something like:

```
Missing Implementation
  Method buildCtrExtension
  Class org.xcsp.parser.XCallbacks2
  Line 129
```

The implementation of a required method is missing here. Because, most of the time, we shall have to translate parser variables into solver variables, we propose first to add the following auxiliary functions:

```
  private VarInteger trVar(Object var) {
    return mapVar.get((XVarInteger) var);
  }

  private VarInteger[] trVars(Object vars) {
    return Arrays.stream((XVarInteger[]) vars).map(v -> mapVar.get(v)).toArray(
         VarInteger[]::new);
  }

  private VarInteger[][] trVars2D(Object vars) {
    return Arrays.stream((XVarInteger[][]) vars).map(t -> trVars(t)).toArray(
         VarInteger[][]::new);
  }
```

And now, assuming that you can call a method `extension` to build an extensional constraint in your solver, you can add:

```
  public void buildCtrExtension(String id, XVarInteger[] list, int[][] tuples, boolean
       positive, Set<TypeFlag> flags) {
    if (flags.contains(TypeFlag.STARRED_TUPLES)) {
      ... // can you manage tables with symbol * ?
    }
    if (flags.contains(TypeFlag.UNCLEAN_TUPLES)) {
      ... // do you have to clean the tuples, so as to remove those that cannot be built from
          variable domains ?
    }
    extension(trVars(list), tuples, positive);
  }
```

The specified Boolean argument indicates is the table is positive (set of tuples seen as supports) or negative (set of tuples seen as conflicts). Note that we have a set as last argument for the method. It allows us to know if * is present in any tuple, and if all tuples are present in the Cartesian product of the variable domains. If everything is ok, you should get 8 solutions for the instance `testExtension1.xml`. Check that you obtain the same result with the equivalent instance `testExtension2.xml` that uses an array instead of stand-alone variables.

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" size="[7]"> 0 1 </array>
  </variables>
  <constraints>
    <extension>
      <list> x[4..5] x[0] </list>
      <supports> (0,0,1)(0,1,0)(1,0,0)(1,1,1) </supports>
    </extension>
    <extension>
      <list> x[0] x[6] x[1] </list>
      <supports> (0,0,1)(0,1,0)(1,0,0)(1,1,1) </supports>
    </extension>
    <extension>
      <list> x[3] x[6] x[2] </list>
      <supports> (0,0,1)(0,1,0)(1,0,0)(1,1,1) </supports>
    </extension>
    <extension>
      <list> x[4] x[5] x[3] </list>
      <supports> (0,0,0)(0,1,1)(1,0,1)(1,1,0) </supports>
    </extension>
  </constraints>
</instance>
```

For dealing with unary extensional constraints, you have to implement:

```
public void buildCtrExtension(String id, XVarInteger x, int[] values, boolean positive
    , Set<TypeFlag> flags);
```

Remark 3 *When a constraint is recognized as totally entailed (no conflicts) or totally disentailed (no supports), as it is the case for extensional constraints when the number of tuples is 0, the parser respectively calls the methods*

```
public void buildCtrTrue(String id, XVar[] list)
public void buildCtrFalse(String id, XVar[] list)
```

You may wish to override the default behavior of these methods.

Let us consider now the following instance, called `testPrimitive.xml`, involving three variables x, y and z of domain $\{1,2,3,4,5\}$, and five constraints (two of them defined from a group) $x \leq 4$, $x > y$, $y > z$, $x \neq z$ and $x = y + z$.

```
<instance format="XCSP3" type="CSP">
  <variables>
    <var id="x"> 0..5 </var>
    <var id="y"> 0..5 </var>
    <var id="z"> 0..5 </var>
  </variables>
  <constraints>
    <intension> le(x,4) </intension>
    <group>
      <intension> gt(%0,%1) </intension>
      <args> x y </args>
      <args> y z </args>
    </group>
    <intension> ne(x,z) </intension>
    <intension> eq(x,add(y,z)) </intension>
  </constraints>
</instance>
```

When you execute the code for this instance, you must obtain something like:

```
Missing Implementation
  Method buildCtrPrimitive
  Class org.xcsp.parser.XCallbacks2
  Line 113
```

The code for the following method is missing:

```
void buildCtrPrimitive(String id, XVarInteger x, TypeConditionOperatorRel op, int k)
```

Primitive unary intensional constraints are special cases of intensional constraints. The parser intercepts them (we shall see later how to deactivate this, if needed). Primitive unary constraints that can be intercepted have the form:

$$x \odot k$$

with x being a variable, $\odot \in \{<, \leq, \geq, >, \neq, =\}$ and $k \in \mathbb{Z}$. For example, we can have $x > 10$ or $y \neq 5$. If you implement the code for dealing with unary primitive constraints in this method, and run again the code for this instance, you must obtain something like:

```
Missing Implementation
  Method buildCtrPrimitive
  Class org.xcsp.parser.XCallbacks2
  Line 117
```

This time, the code for the following method is missing:

```
void buildCtrPrimitive(String id, XVarInteger x, TypeArithmeticOperator opa,
    XVarInteger y, TypeConditionOperatorRel op, int k)
```

Primitive binary intensional constraints are special cases of intensional constraints. The parser intercepts them (we shall discuss to which extent, later). Primitive binary constraints that can be intercepted have the form:

$$(x \oplus y) \odot k$$

with x and y being two variables, $\oplus \in \{+, -, *, /, \%, ||\}$, $\odot \in \{<, \leq, \geq, >, \neq, =\}$ and $k \in \mathbb{Z}$. For example, we can have $x - y > 0$ or $|x - y| = 5$ (note that $||$ stands here for the distance between two objects). If you implement the code for dealing with binary primitive constraints (or a relevant subset of them) in this method, and run again the code for this instance, you must obtain something like:

```
Missing Implementation
  Method buildCtrPrimitive
  Class org.xcsp.parser.XCallbacks2
  Line 121
```

This time, the code for the following method is missing:

```
void buildCtrPrimitive(String id, XVarInteger x, TypeArithmeticOperator opa,
    XVarInteger y, TypeConditionOperatorRel op, XVarInteger z)
```

Primitive ternary intensional constraints are special cases of intensional constraints. The parser intercepts them. Primitive ternary constraints that can be intercepted have the form:

$$(x \oplus y) \odot z$$

with x, y and z being three variables, $\oplus \in \{+, -, *, /, \%, ||\}$, and $\odot \in \{<, \leq, \geq, >, \neq, =\}$. For example, we can have $x + y = z$ or $x * y > z$. If you implement the code for dealing with ternary primitive constraints (or a relevant subset of them) in this method, and run again the code for this instance, you should be fine. Do you get two solutions for this instance?

Maybe, you don't get all these primitive propagators in your solver. Or you just want to manage yourself the Boolean expressions (predicates) of intension constraints. Here is how you can proceed. First, have a look at the following enumeration, put in class `XCallbacks`, together with a predefined map called `callbacksParameters`.

```
enum XCallbacksParameters {
  RECOGNIZE_SPECIAL_UNARY_INTENSION_CASES,
  RECOGNIZE_SPECIAL_BINARY_INTENSION_CASES,
  RECOGNIZE_SPECIAL_TERNARY_INTENSION_CASES,
  RECOGNIZE_SPECIAL_COUNT_CASES,
  RECOGNIZE_SPECIAL_NVALUES_CASES,
  INTENSION_TO_EXTENSION_ARITY_LIMIT, // set it to 0 for deactivating "intension to extension"
      conversion
  INTENSION_TO_EXTENSION_SPACE_LIMIT,
  INTENSION_TO_EXTENSION_PRIORITY;
}

Map<XCallbacksParameters, Object> callbacksParameters = defaultParameters();
```

If you want to deactivate the recognition of unary and binary primitives constraints, you just have to modify the code of your constructor as follows:

```
XCSP3(String fileName) throws Exception {
  callbacksParameters.remove(XCallbacksParameters.
      RECOGNIZE_SPECIAL_UNARY_INTENSION_CASES);
  callbacksParameters.remove(XCallbacksParameters.
      RECOGNIZE_SPECIAL_BINARY_INTENSION_CASES);
  loadInstance(fileName);
}
```

But of course, you have to implement the following method:

```
void buildCtrIntension(String id, XVarInteger[] scope, XNodeParent syntaxTreeRoot)
```

For `intension`, we know that `(XNodeExpr) childs[0].value` is an object XNodeExpr that represents the root node of a syntactic tree denoting the Boolean expression. Let us go further. Suppose that you don't have any propagators for primitive constraints, and you don't have a generic filtering algorithm for dealing with arbitrary form of intensional constraints (after possible decomposition), then your last hope is to convert them into extensional form. The parser can do it for you as follows:

```
XCSP3(String fileName) throws Exception {
  callbacksParameters.remove(XCallbacksParameters.
      RECOGNIZE_SPECIAL_UNARY_INTENSION_CASES);
  callbacksParameters.remove(XCallbacksParameters.
      RECOGNIZE_SPECIAL_BINARY_INTENSION_CASES);
  callbacksParameters.remove(XCallbacksParameters.
      RECOGNIZE_SPECIAL_TERNARY_INTENSION_CASES);
  map.put(XCallbacksParameters.INTENSION_TO_EXTENSION_ARITY_LIMIT, 3); // included
  map.put(XCallbacksParameters.INTENSION_TO_EXTENSION_SPACE_LIMIT, 1000000);
  map.put(XCallbacksParameters.INTENSION_TO_EXTENSION_PRIORITY, Boolean.TRUE);
  loadInstance(fileName);
}
```

However, note that we can control the translation, by fixing the maximal arity (here, 3) of intensional constraints to be converted, and the maximal size (here, $1,000,000$) of the Cartesian product of their variable domains.

Remark 4 *The way primitive constraints are recognized is not complete. There are some cases where the recognition process may fail. In the future, we might propose a canonical representation of the syntactic tree to improve things.*

Now, consider the following instance:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" note="x[i] is the ith value of the series" size="[5]"> 0..4 </array>
    <array id="y" note="y[i] is the distance between x[i] and x[i+1]" size="[4]"> 1..4
         </array>
  </variables>
  <constraints>
    <allDifferent> x[] </allDifferent>
    <allDifferent> y[] </allDifferent>
    <group class="channeling">
      <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>
```

If you run your code for this instance, you must obtain something like:

```
Missing Implementation
  Method buildCtrAllDifferent
  Class org.xcsp.parser.XCallbacks2
  Line 141
```

After implementing this method, you should get 8 solutions. You can keep proceeding that way for all constraints.

If we consider now an optimization problem, with simply a variable to minimize.

```
<instance format="XCSP3" type="COP">
  <variables>
    <array id="x" size="[5]"> 0..500 </array>
    <array id="y" size="[5][5]"> 1..500 </array>
  </variables>
  <constraints>
    <group class="channeling">
      <intension> eq(%0,add(%1,%2)) </intension>
      <args> x[1] x[0] y[0][1] </args>
      <args> x[2] x[0] y[0][2] </args>
      <args> x[3] x[0] y[0][3] </args>
      <args> x[4] x[0] y[0][4] </args>
      <args> x[2] x[1] y[1][2] </args>
      <args> x[3] x[1] y[1][3] </args>
      <args> x[4] x[1] y[1][4] </args>
      <args> x[3] x[2] y[2][3] </args>
      <args> x[4] x[2] y[2][4] </args>
      <args> x[4] x[3] y[3][4] </args>
    </group>
    <allDifferent> y[0][1..4] y[1][2..4] y[2][3..4] y[3][4] </allDifferent>
  </constraints>
  <objectives>
    <minimize> x[4] </minimize>
  </objectives>
</instance>
```

You have to implement the following method:

```
void buildObjToMinimize(String id, XVarInteger x)
```

Once it is done, do you get 11 as optimal value?
As a last example, let us consider the following quadratic assignment instance.

```
<instance format="XCSP3" type="COP">
  <variables>
    <array id="x" size="[12]"> 0..11 </array>
    <array id="d" size="[12][12]"> 0 5 9..12 15..18 21 26 29 30 33..40 44 46 48 54..56
         58 59 61 63 64 66 68..70 72 73 76 78..80 83 85 86 90 93 95..97 </array>
  </variables>
  <constraints>
    <allDifferent> x[] </allDifferent>
    <group>
      <extension>
        <list> %0 %1 %2 </list>
        <supports> (0,1,36)(0,2,54) ... (ellipsis here) ... (11,9,86)(11,10,18) </
             supports>
      </extension>
      <args> x[0..1] d[0][1] </args>
      <args> x[0] x[2] d[0][2] </args>
      <args> x[0] x[3] d[0][3] </args>
      <args> x[0] x[4] d[0][4] </args>
      <args> x[1] x[5] d[1][5] </args>
      <args> x[2] x[6] d[2][6] </args>
      <args> x[2] x[7] d[2][7] </args>
      <args> x[5] x[8] d[5][8] </args>
      <args> x[7] x[9] d[7][9] </args>
      <args> x[8] x[10] d[8][10] </args>
      <args> x[9] x[11] d[9][11] </args>
    </group>
  </constraints>
  <objectives>
    <minimize type="sum">
      <list> d[0][1..4] d[1][5] d[2][6..7] d[5][8] d[7][9] d[8][10] d[9][11] </list>
      <coeffs> 90 10 23 43 88 26 16 1 96 29 37 </coeffs>
    </minimize>
  </objectives>
</instance>
```

You have to implement the following method:

```
void buildObjToMinimize(String id, TypeObjective type, XVarInteger[] list, int[]
    coeffs)
```

Once it is done, do you get 4, 776 as optimal value?