

SBML Level 2 Parsing Library for Java: Tutorial

Note: The software packages provided are betas, and intended to serve to further the community's efforts, find bugs in the software, and to clarify SBML itself through discussion. My email is mvass@vt.edu.

Version 1 follows the latest specification of Level 2 given. It incorporates a minimal level of error checking above that of type checking on fields where integers are called for instead of strings. It uses SAX2 to parse the XML file itself. RDF's are supported, but are treated as strings and are not separately parsed. Facilities for the separate parsing of MathML 2.0 are provided.

Vectors are used to represent the list structure for elements in the SBML document. So, when accessing an object from the functions vector in the model it will be of type `FunctionDefinition`.

SBML Reading and Writing

Reading: Package - `jigcell.modelbuilder.sbml`

To read in an SBML 2.0 file, use the `SBMLLevel2Reader` class. Simply provide the constructor with the name of the file and a **new** instance of `Model`. Your `Model` instance will now contain a representation of the SBML 2.0 file that corresponds (as close as possible) to the data types in the Draft specification.

Now, you would likely need to convert the MathML strings found in many of the objects. (All other conversions are up to you and the software you are importing or exporting to). This is done by the following:

Package: `jigcell.modelbuilder.sbml.math`

Simply make an instance of `MathMLConvertor` (yes, this is misspelled throughout the software, wait until the next version ☺):

```
MathMLConvertor m = new MathMLConvertor();
```

Then add all `FunctionDefinitions` you have to the `MathMLConvertor`, as follows:
(Assume we have a `Model` named `model`)

```
Vector functions = model.getFunctionDefinitions();
for(int i=0; i < functions.size(); i++){
    FunctionDefinition f = (FunctionDefinition)functions.get(i);
    String nameOrId = f.getId();
    m.insertFunction(nameOrId);
}
```

This allows the convertor to know what expressions are functions in the MathML string.

There are two different options on conversion, one is the conversion of lambda MathML functions and the second is that of regular MathML strings. This is done in the following manner, for functions we write:

```
String expression = convertor.toNormalMath(f.getMath(),true);
```

And for regular MathML strings we write:

```
String expression = convertor.toNormalMath(k.getMath(),false);
```

Before performing conversions on another model, you must call the reset method on convertor to remove the previously defined functions.

Writing: Package - jigcell.modelbuilder.sbml

To write an SBML 2.0 file, use the SBMLLevel2Document class. You first will need to fill an instance of the Model class with all the values you desire to have. Conversion from a normal mathematical expression to MathML 2.0 must be done by the user when setting the math strings. This is done as follows:

Packages: jigcell.modelbuilder.sbml.jep, jigcell.modelbuilder.sbml.function

As before all functions must be specified up front:

```
JEP jep = new JEP();
jep.addFunction(functionName,new PostfixMathCommand(numParameters));
```

As is likely evident, numParameters is the number of parameters in your function. Negative 1 means that an arbitrary number of parameters is allowed, but is incompatible with the SBML 2.0 limitations on MathML 2.0.

Now, convert your expressions as follow:

```
String mathML = jep.getSBMLString(yourString);
```

Once you are done filling in all the values you want to in Model, do the following:

```
SBMLLevel2Document.setModel(model);
SBMLLevel2Document.write(filename);
```

Issues

Currently only the lower 8 bits of characters are written out due to a file writing operation function used. This will cause problems for those relying on the full 16 bits of Unicode.

This will be fixed in the next release of the library.

The MathML parsing routines are inadequate for simple conversion from MathML 2.0 to a variety of languages. While the output closely matches C, C++, XPP, or Java at times, the best solution is to also provide access to the original parse tree, so that the library user can code the translation to their own desired presentation or coding language. We have experienced this need with FORTRAN.

What level of helper functions is needed for access to the underlying model? Is the vector/array based implementation suitable, or is more needed? In my use of the library I have written functions to extract the differential equations for any species and all of the conservation relations. What others might be useful?

Other suggestions/requests?