

A short description of the main differences between JSBML and LibSBML

Andreas Dräger

November 21, 2010

Although the libraries JSBML and LibSBML for working with files and data structures defined in the standard SBML (Systems Biology Markup Language) are very similar and share a common scope, users should be informed about their major differences to switch from one library from the other one more easily. To this end, the document at hand gives a brief overview of the main differences between the JavaTM application programming interfaces of both libraries.

1 An extended type hierarchy

Whenever multiple elements defined in the SBML specification share some attributes, JSBML provides a common super class or at least a common interface that gathers methods for manipulation of the shared properties. In this way, the type hierarchy of JSBML has become more complex (see Fig. 1 on the following page). Just like in LibSBML, all elements extend the abstract type `SBase`, but in JSBML, `SBase` has become an interface. This allows more complex relations between derived data types. In contrast to LibSBML, `SBase` in JSBML extends three other interfaces: `Cloneable`, `Serializable`, and `TreeNode`. As all elements defined in JSBML override the `clone()` method from the class `java.lang.Object`, all JSBML elements can be deeply copied and are therefore “cloneable”. By extending the interface `Serializable`, it is possible to store JSBML elements in binary form without explicitly writing it to an SBML file. In this way, programs can easily load and save their in-memory objects or send complex data structures through a network connection without the need of additional file encoding and subsequent parsing. The third interface, `TreeNode` is actually defined in Java’s `swing` package, but defines a data type independent of any graphical information. It basically defines recursive methods on hierarchically structured data types, such as iteration over all of its successors. In this way, all instances

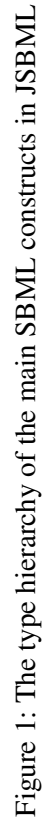


Figure 1: The type hierarchy of the main SBML constructs in JSBML

```
1 package org.sbml.simulator.gui;
2
3 import javax.swing.*;
4
5 import org.sbml.jsbml.SBMLDocument;
6 import org.sbml.jsbml.xml.stax.SBMLReader;
7
8 public class JSBMLvisualizer extends JFrame {
9
10     public JSBMLvisualizer(SBMLDocument document) {
11         super(document.isSetModel() ? document.getModel().getId() : "SBML_
12             Visualizer");
13         getContentPane().add(
14             new JScrollPane(new JTree(document),
15                 JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
16                 JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED));
17         setDefaultCloseOperation(EXIT_ON_CLOSE);
18         pack();
19         setLocationRelativeTo(null);
20         setVisible(true);
21     }
22
23     public static void main(String[] args) throws Exception {
24         UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
25         new JSBMLvisualizer(SBMLReader.readSBML(args[0]));
26     }
27 }
```

Listing 1: Parsing and visualizing the content of an SBML file

of JSBML's *SBase* interface can be directly passed to the swing class *JTree* and hence be easily visualized. Listing 1 demonstrates in a simple code example how to parse an SBML file and to immediately display its content on a *JFrame*. Fig. 2 on the following page shows an example output when applying the program from Listing 1 to SBML test model case00026. The *ASTNode* class in JSBML also implements all these three interfaces and can hence be cloned, serialized, and visualized in the same way.

2 Abstract syntax trees

Both libraries define a class *ASTNode* for in-memory manipulation and evaluation of abstract syntax trees that represent mathematical formulae and equations. These can either be parsed from a representation in C language-like *Strings*, or from a *MathML* representation. The JSBML *ASTNode*

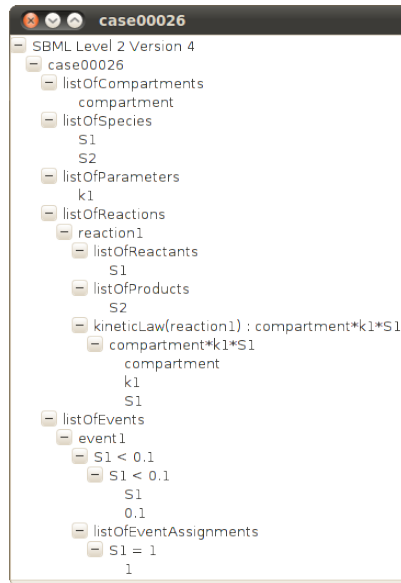


Figure 2: A tree representation of the content of SBML test model case00026

provides various methods to transform these trees to other formats, for instance, \LaTeX Strings. In JSBML, several static methods allow easy creation of new syntax trees, for instance, the following code

```
ASTNode myNode = ASTNode.plus(myLeftAstNode, myRightASTNode);
```

creates a new instance of `ASTNode` which represents the sum of the two other `ASTNodes`. In this way, even complex trees can be easily manipulated.

3 The ASTNodeCompiler class

This interface allows users to create customized interpreters for the content of mathematical equations encoded in abstract syntax trees. It is directly and recursively called from the `ASTNode` class and returns an `ASTNodeValue` object, which wraps the possible evaluation results of the interpretation. JSBML already provides several implementations of this interface, for instance, `ASTNode` objects can be directly translated to LaTeX or MathML for further processing.

4 Change events and listeners

JSBML introduces the possibility to listen to change events in the life of an SBML document. To benefit from this advantage, simply let your class implement the interface `SBaseChangeListener`

and add it to the list of listeners in your instance of `SBMLDocument`. You only have to implement three methods

`sbaseAdded` This method notifies the listener that the given `SBase` has just been added to the `SBMLDocument`

`sbaseRemoved` The `SBase` instance passed to this method is no longer part of the `SBMLDocument` as it has just been removed.

`stateChanged` This method provides detailed information about some value change within the `SBMLDocument`. The object passed to this method is an `SBaseChangeEvent`, which provides information about the `SBase` that has been changed, its property whose value has been changed (this is a `String` representation of the name of the property), along with the previous value and the new value.

With the help of these methods, you can keep track of what your `SBMLDocument` does at any time. Furthermore, one could consider to make use of this functionality in a graphical user interface, where the user should be asked if he or she really wants to delete some element or to approve changes before making these persistent. Another idea of using this, would be to write log files of the model building process automatically.

5 Deprecation

The intension of JSBML is to provide a Java library for the latest specification of SBML. Hence, JSBML provides methods and classes to cover earlier releases of SBML as well, but these are often marked as being deprecated to avoid creating models that refer to these elements.

6 Exceptions

Generally, JSBML throws more exceptions than LibSBML. This behavior helps programmers and users to avoid creating invalid SBML data structures already when dealing with these in memory. Examples are the `ParseException` that may be thrown if a given formula cannot be parsed properly into an `ASTNode` data structure, or `InvalidArgumentExceptions` if inappropriate values are passed to methods. For instance, an object representing a constant such as a `Parameter` whose constant attribute has been set to `true` cannot be used as the `Variable` element in an `Assignment`. Another example is the `InvalidArgumentException` that is thrown when trying to set an invalid identifier `String` for an instance of `AbstractNamedSBase`. Hence, you have to be aware of potential exceptions and errors when using JSBML, on the other hand will this behavior prevent you from doing obvious mistakes.

7 Initial assignments

JSBML unifies all those elements that assign values to some other `SBase` in SBML under the interface `Assignment`. This interface uses the term `Variable` for the element whose value is to be changed depending on some mathematical expression that is also present in the `Assignment` (because `Assignment` extends the interface `MathContainer`). Therefore, an `Assignment` contains methods such as `set-/getVariable(Variable v)` and also `isSetVariable()` and `unsetVariable()`. In addition to that JSBML also provides the method `set-/getSymbol(String symbol)` in the `InitialAssignment` class to make sure that switching from `LibSBML` to `JSBML` is quite smoothly. However, the preferred way in `JSBML` is to apply the methods `setVariable` either with `String` or `Variable` instances as arguments.

8 Model history

In earlier versions of SBML only the model itself could be associated with a history, i.e., a description about the person(s) who build this model, including names, e-mail addresses, modification and creation dates. Nowadays, it has become possible to annotate each individual construct of an SBML model with such a history. This is reflected by naming the corresponding object `History` in `JSBML`, whereas it is still called `ModelHistory` in `LibSBML`. Hence, all instances of `SBase` in `JSBML` contain methods to access and manipulate its `History`. Furthermore, you will not find the classes `ModelCreator` and `ModelCreatorList` because `JSBML` gathers its `Creator` objects in a generic `List<Creator>` in the `History`.

9 The MathContainer interface

This interface gathers all those elements that may contain mathematical expressions encoded in abstract syntax trees (instances of `ASTNode`). The abstract class `AbstractMathContainer` serves as actual super class for most of the derived types.

10 The classes `libSBML` and `JSBML`

There is no class `LibSBML` because this library is called `JSBML`. You can therefore only find a class `JSBML`. This class provides some similar methods as the `LibSBML` class in `LibSBML`, such as `getJSBMLDottedVersion()` to obtain the current version of the `JSBML` library. However, many other methods that you might expect to find there, if you are used to `LibSBML`, are located in the actual classes that are related with the function. For instance, the method to convert between a `String` and a corresponding `Unit.Kind` can be done by using the method

```
Unit.Kind.valueOf(myString);
```

In a similar way, the `ASTNode` class provides a method to parse C-like formula Strings according to the specification of SBML Level 1 into an abstract syntax tree. Therefore, in contrast to the `LibSBML` class, the class `JSBML` contains only a few methods.

11 Replacement of the interface `libSBMConstants` by Java enums

You won't find a corresponding implementation of this interface in `JSBML`. The reason is that the `JSBML` team decided to encode constants using the Java construct `enum`. For instance, all the fields starting with the prefix `AST_TYPE_*` have a corresponding field in the `ASTNode` class itself. There you can find the `Type` enum. Instead of typing `AST_TYPE_PLUS`, you would therefore type `ASTNode.Type.PLUS`.

The same holds true for `Unit.Kind.*` corresponding to the `LibSBMLConstants.UNIT_KIND_*` fields.

12 Various types of `ListOf*` classes

There is no method `get(String id)` because the generic implementation of the `ListOf<? extends SBase>` class in `JSBML` accepts also elements that do not necessarily have an identifier. Only instances of `NamedSBase` may have the fields `identifier` and `name` set. Hence, generally, the `ListOf` class cannot assume these fields to be present. To query an instance of `ListOf` in `JSBML` for names or identifiers or both, you can apply the following filter:

```
NamedSBase nsb = myList.firstHit(new NameFilter(identifier));
```

This will give you the first element in the list with the given identifier. Various filters are already implemented, but you can easily add your customized filter. To this end, you only have to implement the `Filter` interface in `org.sbml.jsbml.util.filters`. There you can also find an `OrFilter` and an `AndFilter`, which take as arguments multiple other filters. With the `SBOFilter` you can query for certain SBO annotations in your list, whereas the `CVTermFilter` helps you to identify `SBase` instances with a desired MIRIAM annotation. For instances of `ListOf<Species>` you can apply the `BoundaryConditionFilter` to look for those species that operate on the boundary of the reaction system.

13 Units

Since SBML Level 3 the data type of the exponent attribute in the `Unit` class has been changed from `int` to double values. `JSBML` reflects this in the method `getExponent()` by returning

`double` values only. For a better compatibility with LibSBML, whose corresponding method still returns `int` values, JSBML also provides the method `getExponentAsDouble()`. This method returns the value from the `getExponent()` method and is therefore absolutely redundant.

14 Unit Definitions

A model in JSBML always also contains all predefined units in the model if there are any, i.e., for models encoded of SBML versions before level 3. These can be accessed from an instance of model by calling the method `getPredefinedUnit(String unit)`.