A short description of the main differences between JSBML and LibSBML

Andreas Dräger* Nicolas Rodriguez† Alexander Dörr*

Marine Dumousseau† Clemens Wrzodek* Michael Hucka‡

February 3, 2011

*Center for Bioinformatics Tuebingen, University of Tuebingen, Tübingen, Germany
†European Bioinformatics Institute, Wellcome Trust Genome Campus, Hinxton, Cambridge, UK
‡Beckman Institute BNMC, California Institute of Technology, Pasadena, CA, USA

Contents

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 Java™ application programming interfaces of both libraries.

In addition, JSBML can be used as a communication layer between the widely spread application CellDesigner and an application that works with JSBML as its internal data structure. This document gives an example that demonstrates how to convert between CellDesigner's plug-in data structures and JSBML objects.

In the same way, it is possible to inter-convert between data structures obtained from LibSBML and JSBML data structures. This document also provides an example of how to read SBML files with LibSBML, to turn them into JSBML data structures, manipulate them and to turn it back for writing into the LibSBML format.

Furthermore, JSBML provides a compatibility module, whose member classes show an identical type declaration as defined in LibSBML. In this way, the compatibility module facilitates switching from LibSBML to JSBML and vice versa by simply exchanging the included JAR file in the project. As for the other two modules, this document also gives an example for the usage of the compatibility module.

1 An extended type hierarchy

Whenever multiple elements defined in at least one of the SBML specifications (Hucka *et al.*, 2003a, 2008, 2010) 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 Figs. 1 to 5 on pages 6–11).
Just like in LibSBML (Bornstein *et al.*, 2008), 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 of JSBML's `SBase` interface can be directly passed to the `swing` class `JTree` and hence be easily visualized. Listing 1 on page 7 demonstrates in a simple code example how to parse an SBML file and to immediately display its content on a `JFrame`. Fig. 6 on page 12 shows an example output when applying the program from Listing 1 on page 7 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.

1.1 The `Assignment` class

JSBML unifies all those elements that assign values to some other `SBase` in SBML (Hucka *et al.*, 2003b) 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()` as well as `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.

1.2 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

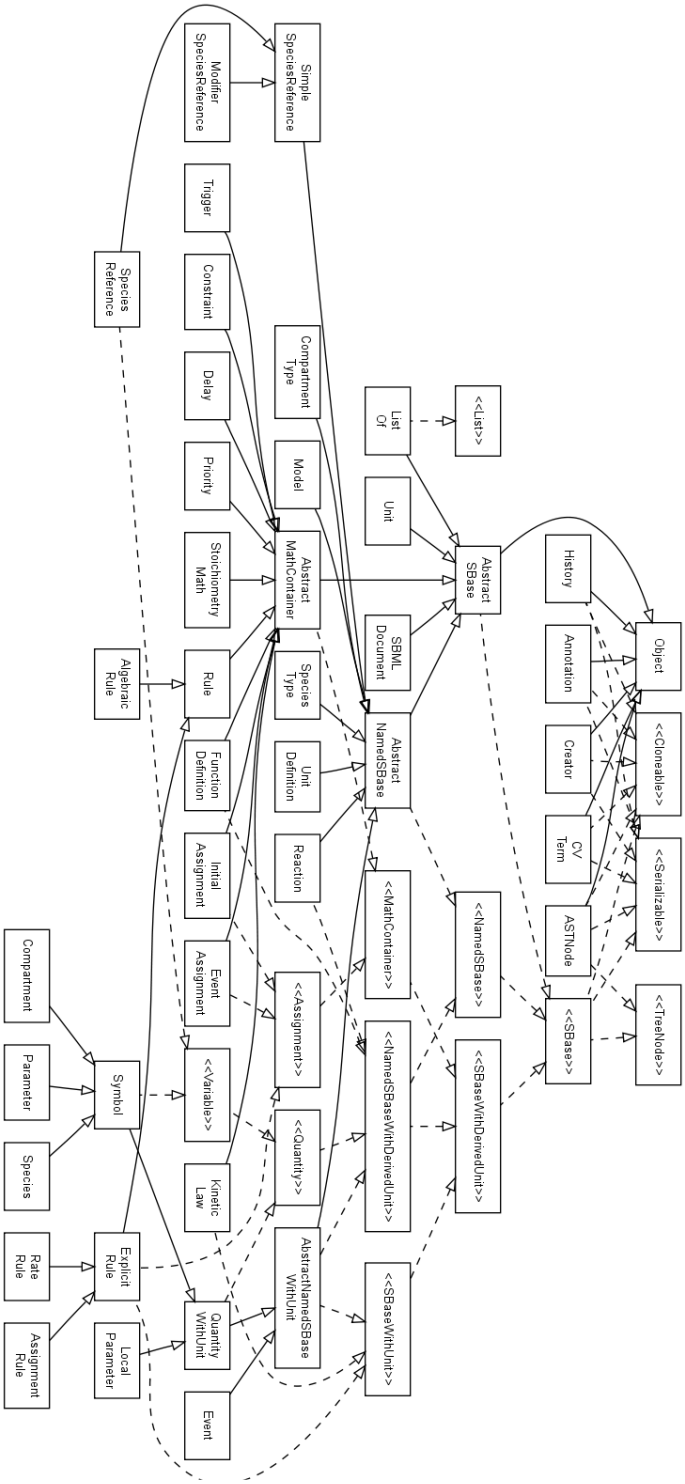

Figure 1: The type hierarchy of the main SBML constructs in JSBML

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

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

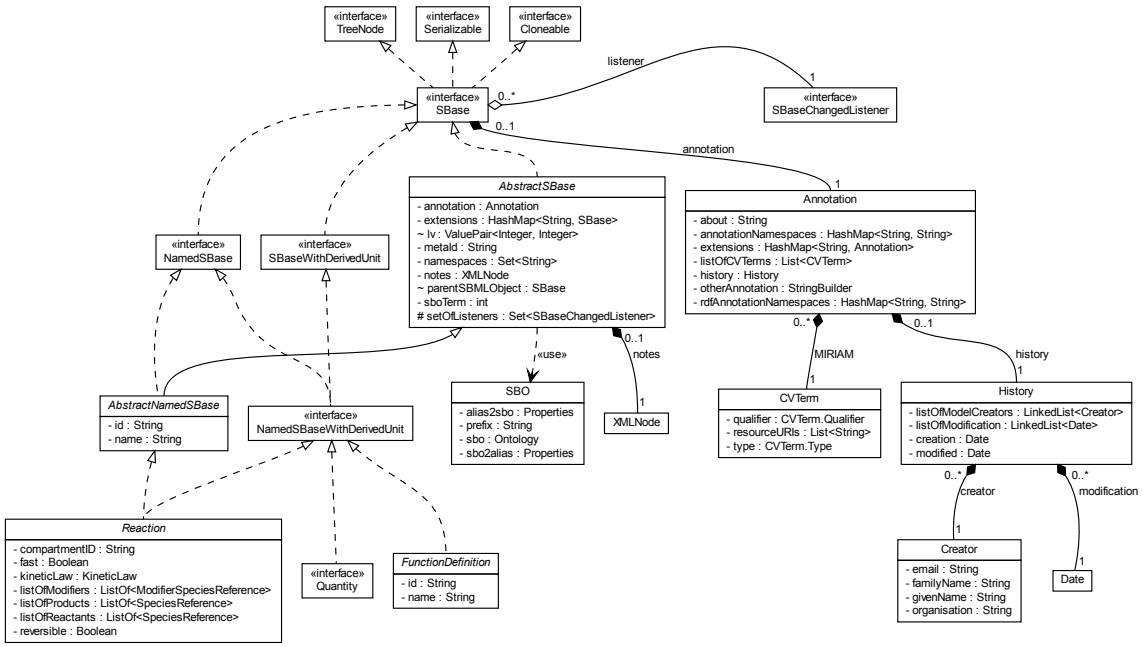

Figure 2: The interface `SBase`, adapted from (Dräger, 2011). This figure displays the most important top-level data structures of JSBML with main focus on the differences to LibSBML. All other data types that represent SBML constructs in JSBML extend either one of the two abstract classes `AbstractSBase` or `AbstractNamedSBase`. The class `SBO` parses the ontology file provided on the SBO web site (`http://www.ebi.ac.uk/sbo/main/`) in OBO format (Open Biomedical Ontologies) using a parser provided by the BioJava project (Holland *et al.*, 2008). For the sake of a clear arrangement, this figure omits methods, fields and other properties.

as actual super class for most of the derived types.

2 Differences in the abstract programming interface

JSBML strives to attain an almost complete compatibility to LibSBML. However, the differences in the programming languages `C` and Java™ lead to the necessity of introducing some differences. In some cases, a direct "translation" from `C` code to Java would not be very elegant. JSBML wants to provide a Java API, whose classes and methods are structured and named and behave like classes and methods in other Java libraries. In this section, we will discuss the most important differences in the APIs of JSBML and LibSBML.

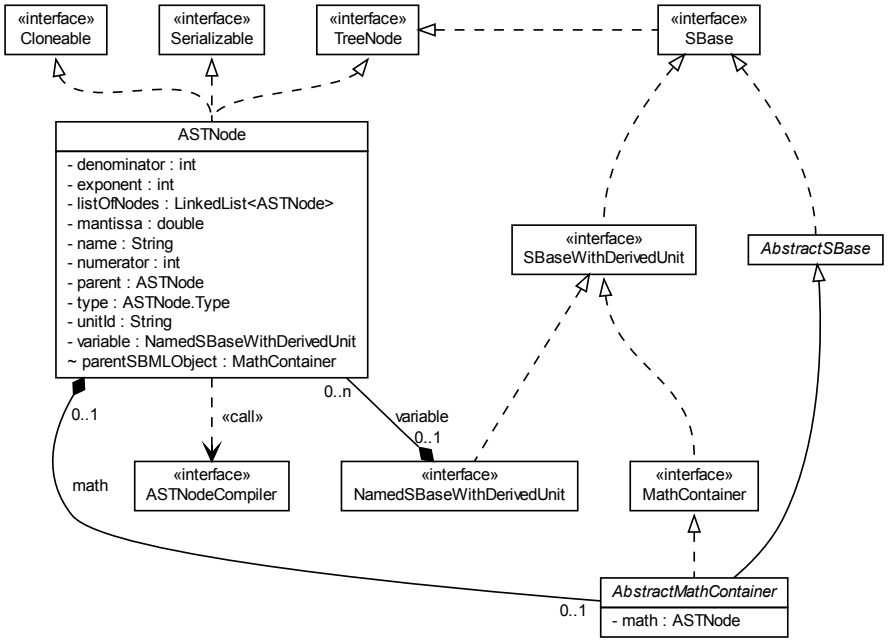

Figure 3: Abstract syntax trees, adapted from (Dräger, 2011). The class `AbstractMathContainer` severs as the super class for several model components in JSBML. It provides methods to manipulate and access an instance of `ASTNode`, which can be converted to or read from `C`-like formula `Strings`. Internally, `AbstractMathContainers` only deal with instances of `ASTNode`. It should be noted that these abstract syntax trees do not implement the `SBase` interface, but also implement the Java interfaces `Cloneable`, `Serializable`, and `TreeNode`. In this figure, the inheritance relationship between `SBase` and `Cloneable` as well as between `SBase` and `Serializable` has been omitted for the sake of simplicity.

2.1 Abstract syntax trees

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

```
1 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.

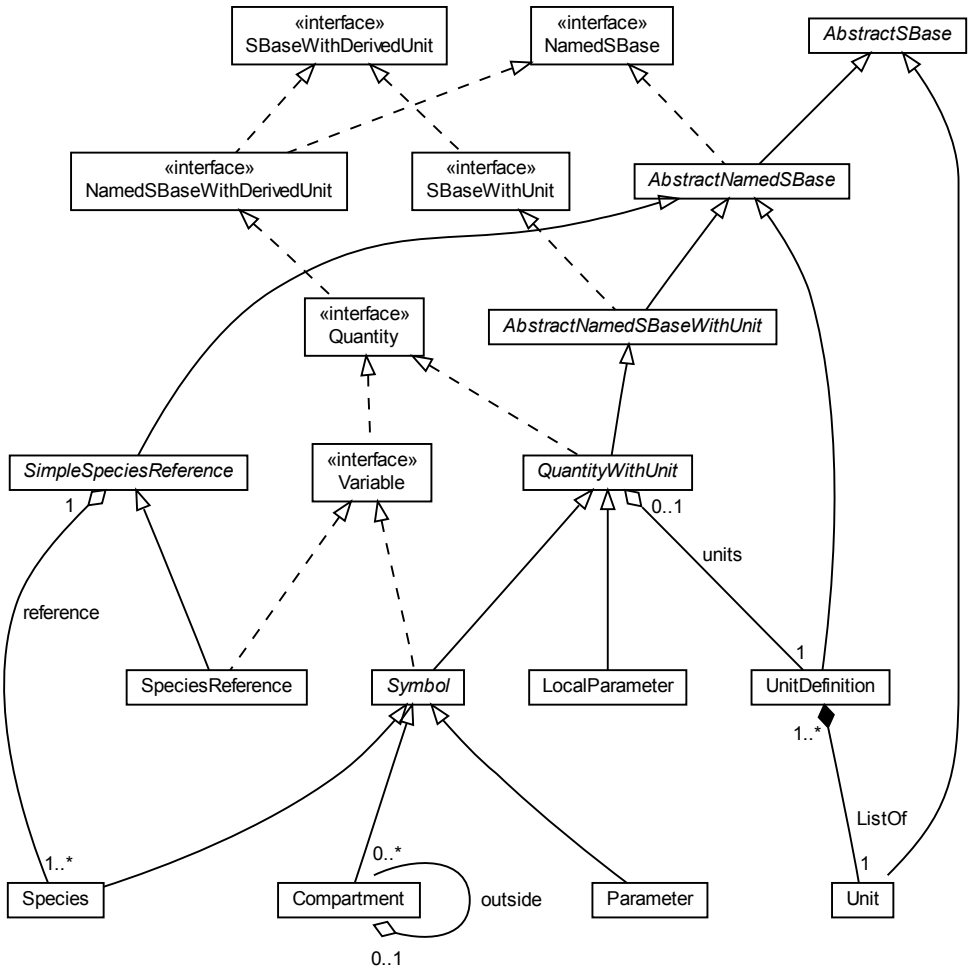

Figure 4: The interface `Variable`, adapted from (Dräger, 2011). JSBML refers to those components of a model that may change their value during a simulation as `Variables`. The class `Symbol` serves as the abstract super class for variables that can also be equipped with a unit. Instances of `Parameter` do not contain any additional field. In `Species` a Boolean switch decides whether its value is to be interpreted as an initial amount or as an initial concentration. In contrast to `Variables`, `LocalParameters` represent constant unit-value pairs that can only be accessed within their declaring `KineticLaw`.

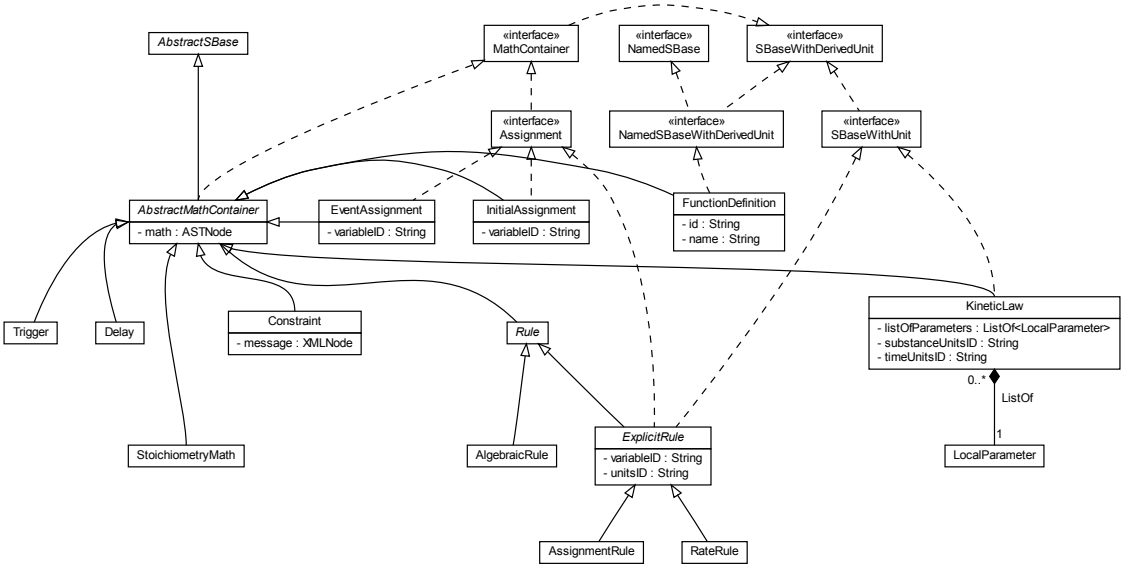

Figure 5: `MathContainer`, adapted from (Dräger, 2011). Instances of the interface `MathContainer`, particularly its directly derived class `AbstractMathContainer` constitute the super class for all elements that store and manipulate mathematical formulas in JSBML, which is done in form of `ASTNode` objects. These can be evaluated using an implementation of `ASTNodeCompiler`. Note that some classes that extend `AbstractMathContainer` do not contain any own fields or methods: `Delay`, `Trigger`, `StoichiometryMath`, and `AlgebraicRule`.

2.2 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.

2.3 Cloning when adding child nodes

When adding elements such as a `Species` to a `Model`, LibSBML will clone the object and add the clone to the `Model`. In contrast, JSBML does not automatically perform cloning. The advantage is that modifications on the object belonging to the original pointer will also propagate to the element added to the `Model`. Furthermore, this is more efficient with respect to the run time and also more intuitive. If cloning is necessary, users should call the `clone()` method manually. Since all instances of `SBase` and also `Annotation`, `ASTNode`, `CVTerm`, and `History` implement the interface `Cloneable` (see Fig. 1 on page 6), all these elements can be naturally cloned. However,

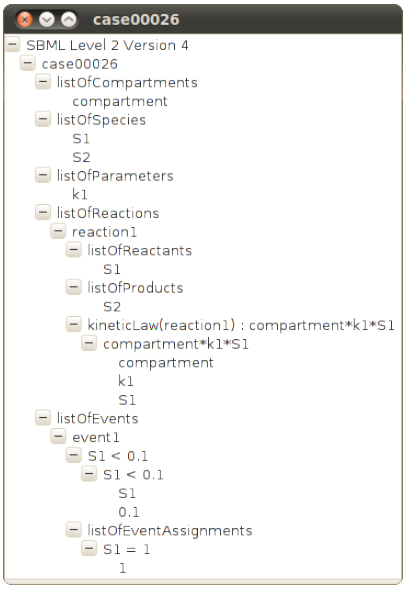

Figure 6: A tree representation of the content of SBML test model `case00026`

when cloning an object in JSBML, such as an `AbstractNamedSBase`, all children of this element will recursively be cloned before adding them to the new element. This is necessary, because the data structures specified in SBML define a tree, in which each element has exactly one parental node.

2.4 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.

2.5 Exceptions

Generally, JSBML throws more exceptions than LibSBML, whose methods often return error codes instead. 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`.

- An instance of `Priority` can only be assigned to an `Events` if its `level` attribute has at least been set to three.

Hence, you have to be aware of potential exceptions and errors when using JSBML, on the other hand this will prevent you from doing obvious mistakes.

2.6 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 do 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`.

2.7 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

```
1 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 (Hucka *et al.*, 2003a) into an abstract syntax tree. Therefore, in contrast to the `libSBML` class, the class `JSBML` contains only a few methods.

2.8 Replacement of the interface `libSBMConstants` by Java `enum`s

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 `libSBMLConstants.AST_TYPE_PLUS`, you would therefore type `ASTNode.Type.PLUS`.

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

2.9 Various types of `ListOf*` classes

In JSBML the `ListOf*` objects do not offer a method `get(String id)` because their generic implementation `ListOf<? extends SBase>` excepts 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:

```
1 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 (Le Novère, 2006; Le Novère *et al.*, 2006) in your list, whereas the `CVTermFilter` helps you to identify `SBase` instances with a desired MIRIAM (Minimal Information Required In the Annotation of Models) annotation (Le Novère *et al.*, 2005). For instances of `ListOf<Species>` you can apply the `BoundaryConditionFilter` to look for those species that operate on the boundary of the reaction system.

2.10 Units

Since SBML Level 3 (Hucka *et al.*, 2010) 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.

2.11 Unit Definitions

2.11.1 Predefined 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)`.

MIRIAM annotations (Le Novère *et al.*, 2005) have become an integral part of SBML models since Level 2 Version 2. Recently, the Unit Ontology[1] (UO) has been included in the set of supported ontology and online resources of MIRIAM. Since all the predefined units in SBML have corresponding entries in the UO, JSBML automatically equips those predefined units with the correct MIRIAM URI in form of a controlled vocabulary term (`CVTerm`) if the Level/Version combination of the model supports MIRIAM annotations.

[1] `http://www.obofoundry.org/cgi-bin/detail.cgi?id=unit`

Note that the enum Unit.Kind also provides methods to directly obtain the entry from the UO that corresponds to a certain unit kind and also to generate MIRIAM URIs accordingly. In this way, JSBML facilitates the annotation of user-defined units and unit definitions with MIRIAM-compliant information.

2.11.2 Access to the units of an element

In JSBML, all SBML elements that can be associated with some unit implement the interface SBaseWithUnit. This interface provides methods for direct access to an object representing their unit. Currently, the following elements implement this interface:

- AbstractNamedSBaseWithUnit
- ExplicitRule
- KineticLaw

Fig. 1 on page 6 provides a better overview about the relationships between all the classes explained here. Note that AbstractNamedSBaseWithUnit serves as the abstract super class for Event and QuantityWithUnit. In Event, all methods to deal with units are already deprecated because only in SBML Level 1 Versions 1 and 2 (Hucka *et al.*, 2003a) Events could be explicitly equipped with units. The same holds true for instances of ExplicitRule and KineticLaw, which both can only explicitly be populated with units for SBML in Level 1, Version 1 and 2. In contrast, QuantityWithUnit serves as the abstract super class for LocalParameter and Symbol, which is then again the super type of Compartment, Species, and (global) Parameter.

With SBaseWithUnit being a subtype of SBaseWithDerivedUnit users can access the units of such an element in two different ways:

getUnit() This method returns the String of the unit kind or the unit definition in the model that has been directly set by the user during the life time of the element. If nothing has been declared, an empty String will be delivered.

getDerivedUnit() This method gives either the same result as getUnit() if some unit has been declared explicitly, or it returns the predefined unit of the element for the given SBML Level/Version combination. Only if neither a user-defined nor a predefined unit is available, this method returns an empty String.

Both methods have corresponding methods to directly obtain an instance of UnitDefinition for convenience.

However, care must be taken when obtaining an instance of UnitDefinition from one of the classes implementing SBaseWithUnit because it might happen that the model containing this SBaseWithUnit does actually not contain the required instance of UnitDefinition and the method returns a UnitDefinition that has just been created for convenience from the information provided by the class. It might therefore be useful to either check if the Model contains this UnitDefinition or to add it to the Model.

In case of KineticLaw it is even more difficult, because SBML Level 1 allows to separately set the substance unit and the time unit of the element. To unify the API, we decided to also provide methods that allow the user to simply pass one UnitDefinition or its identifier to KineticLaw. These methods then try to guess if a substance unit or time unit is given. Furthermore, it is possible to pass a UnitDefinition representing a variant of substance per time directly. In this case, the KineticLaw will memorize a direct link to this UnitDefinition in the model and also try to save separate links to the time unit and the substance unit. However, this may cause a problem if the containing Model does not contain separate UnitDefinitions for both entries.

Generally, this approach provides a more general way to access and to manipulate units of SBML elements.

3 Additional features of JSBML

The JSBML library also provides some features that cannot be found in LibSBML. This section briefly introduces its most important additional capabilities.

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

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

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

`stateChanged(SBaseChangedEvent event)` This method provides detailed information about some value change within the SBMLDocument. The object passed to this method is an SBaseChangedEvent, 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.

Note that the class SBaseChangedEvent implements the class java.util.EventObject and that the interface SBaseChangedListener extends the interface java.util.EventListener. In

this way, the event and listener data structures fit into the common Java API (Application Programming Interface) and allow users also to make use of, e.g., `EventHandlers` to deal with changes in a model. It should also be noted that `SBaseChangedListeners` only keep track of changes in instances of `SBase` directly. This means that changes inside of, e.g., `CVTerm` or `History` may not be traced.

3.2 Logging functionality

JSBML provides logging.

A Frequently Asked Questions (FAQ)

Why does the class `LocalParameter` not inherit from `Parameter`? The reason is the Boolean attribute `constant`, which is present in `Parameter` and can be set to `false`. A parameter in the meaning of SBML is not a constant, it might be some system variable and can therefore be the subject of `Rules`, `Events`, `InitialAssignments` and so on, i.e., all instances of `Assignment`, whereas a `LocalParameter` is defined as a constant quantity that never changes its value during the evaluation of a model. It would therefore only be possible to let `Parameter` inherit from `LocalParameter` but this could lead to a semantic misinterpretation.

B An example of how to turn a JSBML-based application into a CellDesigner plug-in

Once an application has been implemented based on JSBML, it can easily be accessed from CellDesigner's plug-in menu (Funahashi *et al.*, 2003). To this end, it is necessary to extend two classes that are defined in CellDesigner's plug-in API (Application Programming Interface). The Listings 2 to 3 on pages 18–19 show a very simple example of how to pass CellDesigner plug-in model data structures to the translator in JSBML, which creates then a JSBML `Model` data structure. The examples described by Listings 2 to 3 on pages 18–19 create a plug-in for CellDesigner, which displays the SBML data structure in a tree, like the example in Fig. 6 on page 12. This example only shows how to translate a plug-in data structure from CellDesigner into a corresponding JSBML data structure. With the help of the class `PluginSBMLWriter` it is possible to notify CellDesigner about changes in the model data structure. Note that Listing 3 on page 19 is only completed by implementing the methods from the super class. In this example it is sufficient to leave the implementation open.

```
1  package org.sbml.jsbml.cdplugin;
2
3  import java.awt.event.ActionEvent;
4  import javax.swing.JMenuItem;
5  import jp.sbi.celldesigner.plugin.PluginAction;
6
7  /** A simple implementation of an action for a CellDesigner plug-in */
8  public class SimpleCellDesignerPluginAction extends PluginAction {
9
10   private SimpleCellDesignerPlugin plugin;
11
12   /** Constructor memorizes the plug-in data structure. */
13   public SimpleCellDesignerPluginAction(SimpleCellDesignerPlugin plugin) {
14     this.plugin = plugin;
15   }
16
17   /** Executes an action if the given commant occurs. */
18   public void myActionPerformed(ActionEvent ae) {
19     if (ae.getSource() instanceof JMenuItem) {
20       String itemText = ((JMenuItem) ae.getSource()).getText();
21       if (itemText.equals(SimpleCellDesignerPlugin.ACTION)) {
22         plugin.startPlugin();
23       }
24     } else {
25       System.err.printf("Unsupported␣source␣of␣action␣%s\n", ae
26           .getSource().getClass().getName());
27     }
28   }
29
30 }
```

Listing 2: A simple implementation of CellDesigner's abstract class `PluginAction`

```
1  package org.sbml.jsbml.cdplugin;
2
3  import javax.swing.*;
4  import jp.sbi.celldesigner.plugin.*;
5  import org.sbml.jsbml.*;
6  import org.sbml.jsbml.gui.*;
7
8  /** A very simple implementation of a plug-in for CellDesigner. */
9  public class SimpleCellDesignerPlugin extends CellDesignerPlugin {
10
11   public static final String ACTION = "Display␣full␣model␣tree";
12   public static final String APPLICATION_NAME = "Simple␣Plugin";
13
14   /** Creates a new CellDesigner plug-in with an entry in the menu bar. */
15   public SimpleCellDesignerPlugin() {
16     super();
17     try {
18       System.out.printf("\n\nLoading␣%s\n\n", APPLICATION_NAME);
19       SimpleCellDesignerPluginAction action = new
             SimpleCellDesignerPluginAction(this);
20       PluginMenu menu = new PluginMenu(APPLICATION_NAME);
21       PluginMenuItem menuItem = new PluginMenuItem(ACTION, action);
22       menu.add(menuItem);
23       addCellDesignerPluginMenu(menu);
24     } catch (Exception exc) {
25       exc.printStackTrace();
26     }
27   }
28
29   /** This method is to be called by our CellDesignerPluginAction. */
30   public void startPlugin() {
31     PluginSBMLReader reader = new PluginSBMLReader(getSelectedModel(), SBO
32         .getDefaultPossibleEnzymes());
33     Model model = reader.getModel();
34     SBMLDocument doc = new SBMLDocument(model.getLevel(), model
35         .getVersion());
36     doc.setModel(model);
37     new JSBMLvisualizer(doc);
38   }
39
40   // Include also methods from super class, not needed in this example.
41   public void addPluginMenu() { }
42   public void modelClosed(PluginSBase psb) { }
43   public void modelOpened(PluginSBase psb) { }
44   public void modelSelectChanged(PluginSBase psb) { }
45   public void SBaseAdded(PluginSBase psb) { }
46   public void SBaseChanged(PluginSBase psb) { }
47   public void SBaseDeleted(PluginSBase psb) { }
48 }
```

Listing 3: A simple example for a CellDesigner plug-in using JSBML as a communication layer

References

Bornstein, B. J., Keating, S. M., Jouraku, A., and Hucka, M. (2008). LibSBML: an API Library for SBML. *Bioinformatics*, **24**(6), 880–881.

Dräger, A. (2011). *Computational Modeling of Biochemical Networks*. Ph.D. thesis, University of Tübingen, Sand 1, 720726 Tübingen.

Funahashi, A., Tanimura, N., Morohashi, M., and Kitano, H. (2003). CellDesigner: a process diagram editor for gene-regulatory and biochemical networks. *BioSilico*, **1**(5), 159–162.

Holland, R. C. G., Down, T., Pocock, M., Prlić, A., Huen, D., James, K., Foisy, S., Dräger, A., Yates, A., Heuer, M., and Schreiber, M. J. (2008). BioJava: an Open-Source Framework for Bioinformatics. *Bioinformatics*, **24**(18), 2096–2097.

Hucka, M., Finney, A., Sauro, H., and Bolouri, H. (2003a). Systems Biology Markup Language (SBML) Level 1: Structures and Facilities for Basic Model Definitions. Technical Report 2, Systems Biology Workbench Development Group JST ERATO Kitano Symbiotic Systems Project Control and Dynamical Systems, MC 107-81, California Institute of Technology, Pasadena, CA, USA.

Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., Kitano, H., Arkin, A. P., Bornstein, B. J., Bray, D., Cornish-Bowden, A., Cuellar, A. A., Dronov, S., Gilles, E. D., Ginkel, M., Gor, V., Goryanin, I. I., Hedley, W. J., Hodgman, T. C., Hofmeyr, J.-H. S., Hunter, P. J., Juty, N. S., Kasberger, J. L., Kremling, A., Kummer, U., Le Novère, N., Loew, L. M., Lucio, D., Mendes, P., Minch, E., Mjolsness, E. D., Nakayama, Y., Nelson, M. R., Nielsen, P. F., Sakurada, T., Schaff, J. C., Shapiro, B. E., Shimizu, T. S., Spence, H. D., Stelling, J., Takahashi, K., Tomita, M., Wagner, J. M., Wang, J., and the rest of the SBML Forum (2003b). The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, **19**(4), 524–531.

Hucka, M., Finney, A., Hoops, S., Keating, S. M., and Le Novère, N. (2008). Systems biology markup language (SBML) Level 2: structures and facilities for model definitions. Technical report, Nature Precedings.

Hucka, M., Bergmann, F. T., Hoops, S., Keating, S. M., Sahle, S., Schaff, J. C., Smith, L. P., and Wilkinson, D. J. (2010). The Systems Biology Markup Language (SBML): Language Specification for Level 3 Version 1 Core. Technical report, Nature Precedings.

Le Novère, N. (2006). Model storage, exchange and integration. *BMC Neuroscience*, **7 Suppl 1**, S11.

Le Novère, N., Finney, A., Hucka, M., Bhalla, U. S., Campagne, F., Collado-Vides, J., Crampin, E. J., Halstead, M., Klipp, E., Mendes, P., Nielsen, P., Sauro, H., Shapiro, B. E., Snoep, J. L.,

Spence, H. D., and Wanner, B. L. (2005). Minimum information requested in the annotation of biochemical models (MIRIAM). *Nature Biotechnology*, **23**(12), 1509–1515.

Le Novère, N., Courtot, M., and Laibe, C. (2006). Adding semantics in kinetics models of biochemical pathways. In C. Kettner and M. G. Hicks, editors, *2nd International ESCEC Workshop on Experimental Standard Conditions on Enzyme Characterizations. Beilstein Institut, Rüdesheim, Germany*, pages 137–153, Rüdessheim/Rhein, Germany. ESEC.

Index