User guide for JSBML

Andreas Dräger[*] Nicolas Rodriguez[†] Marine Dumousseau[†]
Alexander Dörr[*] Clemens Wrzodek[*] Finja Büchel[*]
Florian Mittag[*]

Principal Investigators:
Nicolas Le Novère[†], Andreas Zell[*], and Michael Hucka[‡]

March 15, 2012

[*]Center for Bioinformatics Tuebingen, University of Tuebingen, Tübingen, Germany
[†]European Bioinformatics Institute, Wellcome Trust Genome Campus, Hinxton, Cambridge, UK
[‡]Computing and Mathematical Sciences, California Institute of Technology, Pasadena, California, USA

The specifications of the Systems Biology Markup Language (SBML) define a standard for storing and exchanging biochemical models in XML-formatted text files. To perform higher-level operations on these models, e.g., numerical simulation or visual representation, an appropriate mapping to in-memory objects is required. To this end, the JSBML library has been developed. JSBML supports all SBML levels and versions that are available today. In addition, JSBML provides modules that facilitate the development of CellDesigner plugins or ease the migration from a libSBML backend.

This document should help you getting started with JSBML. It is not only intended for users, developing their applications from scratch, but also for users, switching from libSBML to JSBML.

Contents

1 Getting started with JSBML

The following are quick-start instructions for getting started with JSBML. This document is based on JSBML version 1.0. Before doing any of the steps below, you will need to obtain a copy of JSBML either via the SourceForge download page[1] or using Subversion (SVN) as described below.

1.1 Introduction

JSBML is a library that will help you to manipulate SBML files (Dräger, 2011; Dräger *et al.*, 2011). If you are not familiar with SBML, a good starting point would be to read the latest SBML specification[2] (Hucka *et al.*, 2010). If you have some other questions about SBML, you may find the answer in the SBML FAQ[3]. JSBML is written in Java™. To use it, you will need a Java Runtime Environment (JRE) 1.5 or higher. See, for example, the Java SE download page[4]. JSBML also provides several modules. Two of them should ease developers to interact with CellDesigner or libSBML and one module eases switching from libSBML to JSBML or the other way around.

1.2 Obtaining and setting up JSBML

1.2.1 Using the JSBML JAR file distribution

Before starting to use JSBML, you will need to configure your class path. JSBML provides two versions of the JAR file:

1. including all dependencies - it is sufficient to include just this file in your class path.
2. without any dependencies - you need to take care of all the dependencies of JSBML by yourself.

The JSBML JAR file with dependencies is a merged JAR file that includes all of its required third-party libraries. In this case, it is sufficient to include it into your build or class path in order to use JSBML.

[1] `https://sourceforge.net/projects/jsbml/files/jsbml/`
[2] `http://sbml.org/Documents/Specifications/`
[3] `http://sbml.org/Documents/FAQ`
[4] `http://www.oracle.com/technetwork/java/javase/downloads/index.html`

Dependencies

When using the JSBML JAR file without dependencies, you need the JSBML dependencies in addition to the JSBML library. The following list gives you an overview of all these libraries:

biojava-1.7-ontology.jar This is a stripped down version of the biojava-1.7[5] containing mostly ontology-related classes (Holland *et al.*, 2008).

junit-4.8.jar This library is only needed, if you want to run the JUnit[6] tests of JSBML (located in the `test` folder).

stax2-api-3.0.3.jar Used to read and write the XML files[7].

stax-api-1.0.1.jar Used to read and write the XML files[8].

woodstox-core-lgpl-4.0.9.jar Used to read and write the XML files. This is the actual stax parser implementation JSBML uses[9].

staxmate-2.0.0.jar Used to read and write the XML files. This library allows us to use stax in a more user-friendly manner[10].

xstream-1.3.1.jar Used to read and write the XML files. This parser is used to parse the result from the SBML validator, JSBML might use it more intensively in the future or drop it to use only stax/woodstox[11].

jigsaw-dateParser.jar This is a stripped down version of the jigsaw-library, containing one class to manipulate dates. It has been created with the version from 2010-12-16[12].

log4j-1.2.8.jar JSBML uses the Apache log4j logger[13]. If you want to use logging, you should include this logger.

JSBML was developed and tested with these versions of the libraries described above. Some more recent versions might work, too. When you have all of these dependencies in your build or class path alongside the JSBML JAR file, you are ready to work with JSBML.

[5] `http://biojava.org`
[6] `http://www.junit.org`
[7] `http://docs.codehaus.org/display/WSTX/StAX2`
[8] `http://stax.codehaus.org`
[9] `http://woodstox.codehaus.org`
[10] `http://staxmate.codehaus.org`
[11] `http://xstream.codehaus.org`
[12] `http://jigsaw.w3.org`
[13] `http://logging.apache.org/log4j/`

1.2.2 Download and usage of the source distribution

As an alternative to using the JAR files, you can check out the source tree from SVN and compile JSBML yourself. To do that, you will need to have a Java JDK[4] installed, the Apache Ant[14] build system, and Subversion[15], a version control system.

Use the following command to download the latest JSBML classes (requires Subversion[15]):

```
svn co "https://jsbml.svn.sourceforge.net/svnroot/jsbml/trunk" jsbml
cd jsbml
```

To compile the JSBML library to a single JAR file, type the following command (requires Apache Ant[14]):

```
ant jar
```

If you want to run the JUnit[6] tests on your compiled JAR file, please use the following command:

```
ant test
```

If you performed all the steps above, you should have a JSBML library, based on the latest version of all classes. You can now include the created JAR file into your build or class path and start using JSBML.

1.2.3 Download and usage of the JSBML modules

JSBML provides today, two additional modules. Binary versions of the modules can be found at the download site of JSBML. In order to obtain the most recent version of the modules, please type the following Subversion[15] commands on your command line.

The CellDesigner bridge module should help CellDesigner plugin developers to use JSBML as internal data structure.

```
svn co "https://jsbml.svn.sourceforge.net/svnroot/jsbml/modules/cellDesigner"
    cellDesigner
```

Developers, who still want to make use of libSBML, might want to have a look at the libSBML communication layer.

```
svn co "https://jsbml.svn.sourceforge.net/svnroot/jsbml/modules/libSBMLio/"
    libSBMLio
```

[14] http://ant.apache.org/
[15] http://subversion.apache.org/

Listing 1.1: Parsing and visualizing the content of an SBML file

```
1  import javax.swing.*;
2  import org.sbml.jsbml.*;
3
4  /** Displays the content of an SBML file in a {@link JTree} */
5  public class JSBMLvisualizer extends JFrame {
6    /** @param document The sbml root node of an SBML file */
7    public JSBMLvisualizer(SBMLDocument document) {
8      super(document.getModel().getId());
9      getContentPane().add(new JScrollPane(new JTree(document)));
10     pack();
11     setVisible(true);
12   }
13   /** @param args Expects a valid path to an SBML file. */
14   public static void main(String[] args) throws Exception {
15     new JSBMLvisualizer((new SBMLReader()).readSBML(args[0]));
16   }
17 }
```

1.3 *Hello World*: writing your first JSBML applications

This section presents two examples for using JSBML. One example reads an existing `SBMLDocument` from a file an visualizes it on a `JFrame`. The second example creates a new `SBMLDocument` from scratch and writes its content into a file. This should help you getting started and writing your own JSBML applications.

1.3.1 Reading and visualizing an `SBMLDocument`

Listing 1.1 demonstrates in a simple code example how to parse an SBML file (submitted as first argument) and to immediately display its content on a `JFrame`. Fig. 1.1 on page 11 shows an example output when applying the program to an SBML test model. Line 16 in Listing 1.1 shows how to read an `SBMLDocument` from a file, using the `SBMLReader`. Afterwards, the `JSBMLvisualizer` constructor is called, which first creates a new `JFrame` with the model's id as title (line 9). Since JSBML's `SBase` object, and all derived elements, implement the `TreeNode` interface, it is possible to create a `JTree` from the information in the `SBMLDocument` only. This is done in line 10.

1.3.2 Creating and writing an `SBMLDocument`

Listing 1.2 on the next page shows a more complex example. A new `SBMLDocument` is created from scratch. It mainly consists of one `Compartment`, one `Model`, two `Species`, and a `Reaction` in which both `Species` are involved. This `SBMLDocument` is written into a file, using `SBMLWriter`.

Listing 1.2: Creating a new `SBMLDocument` and writing its content into a file

```
1  import java.beans.PropertyChangeEvent;
2  import javax.swing.tree.TreeNode;
3  import org.sbml.jsbml.*;
4  import org.sbml.jsbml.util.TreeNodeChangeListener;
5
6  /** Creates an {@link SBMLDocument} and writes it's content to disk. **/
7  public class JSBMLexample implements TreeNodeChangeListener {
8    public JSBMLexample() throws Exception  {
9      // Create a new SBMLDocument, using SBML level 2 version 4.
10     SBMLDocument doc = new SBMLDocument(2, 4);
11     doc.addTreeNodeChangeListener(this);
12
13     // Create a new SBML-Model and compartment in the document
14     Model model = doc.createModel("test_model");
15     Compartment compartment = model.createCompartment("default");
16     compartment.setSize(1d);
17
18     // Create model history
19     History hist = new History();
20     Creator creator = new Creator("Given␣Name", "Family␣Name",
21       "My␣Organisation", "My@EMail.com");
22     hist.addCreator(creator);
23     model.setHistory(hist);
24
25     // Create some example content in the document
26     Species specOne = model.createSpecies("test_spec1", compartment);
27     Species specTwo = model.createSpecies("test_spec2", compartment);
28     Reaction sbReaction = model.createReaction("reaction_id");
29
30     // Add a substrate (SBO: 15) and product (SBO: 11).
31     SpeciesReference subs = sbReaction.createReactant(specOne);
32     subs.setSBOTerm(15);
33     SpeciesReference prod = sbReaction.createProduct (specTwo);
34     prod.setSBOTerm(11);
35
36     // Write the SBML document to disk
37     SBMLWriter.write(doc, "test.sbml.xml", "ProgName", "Version");
38   }
39
40   /** Just an example main **/
41   public static void main(String[] args) throws Exception {
42     new JSBMLexample();
43   }
44
45   /* Those three methods respond to events from SBaseChangedListener */
46   public void nodeAdded(TreeNode sb) {System.out.println("[ADD]␣" + sb);}
47   public void nodeRemoved(TreeNode sb) {System.out.println("[RMV]␣" + sb);}
48   public void propertyChange(PropertyChangeEvent ev)
       {System.out.println("[CHG]␣" + ev);}
49 }
```

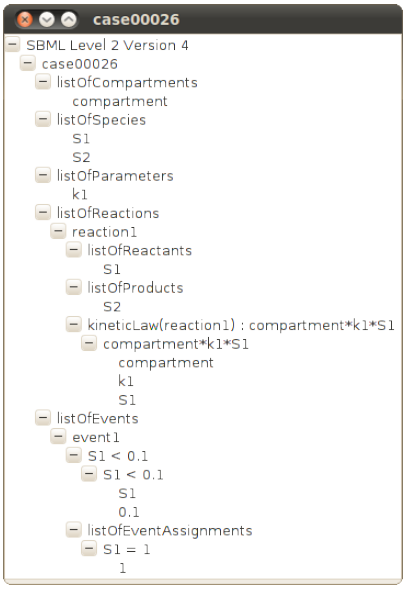

Figure 1.1: A tree representation of the content of SBML test model `case00026`. In JSBML, the hierarchically structured `SBMLDocument` can be traversed recursively because all instances of `SBase` implement the interface `TreeNode`.

1.3.3 Further examples

Listing 2.4 on page 33 shows how to convert libSBML data structures into JSBML data objects. Listing 2.5 on page 34 demonstrates the implementation of CellDesigner's abstract class `PluginAction` and Listing 2.6 on page 35 gives a complete example for writing CellDesigner plugins with JSBML.

2 Main differences between JSBML and libSBML

Until today, the Java binding of libSBML has been the main library for developing Java applications that use SBML. Thus, many Java developers are used to the methods and commands, libSBML provides. The JSBML team made some effort to allow those developers a fast and easy switch to this new library. For example, a libSBML compatibility module has been developed, that implements existing libSBML methods and simply redirects the parameters to the corresponding JSBML methods. But it is important that developers, coming from libSBML, know the main differences between the two libraries. The following sections give a brief description of those main differences.

2.1 Introduction

The intention of implementing a pure Java™ Application Programming Interface (API) for working with SBML files was not to re-implement the existing Java API of libSBML (Bornstein *et al.*, 2008). From the very beginning, JSBML has been designed based on the SBML specifications (Hucka *et al.*, 2003, 2008, 2010) but with respect to naming conventions of methods and variables from libSBML. Similarly to the SBML specifications, the libSBML library has grown historically. The implementation of JSBML permitted to entirely re-design the type hierarchy of the SBML elements and the way to implement what is specified in the SBML documents. However, it is important to keep in mind that SBML is a language that defines how to store of biological processes and how to exchange these models between different software tools. It does not specify how to represent its elements in memory. Furthermore, during the evolution of SBML some elements or properties of elements have become obsolete. It is therefore up to an implementing library to decide how to deal with those constructs. To facilitate switching from libSBML to JSBML and the other way around, JSBML has been designed to behave similarly to libSBML but, due to the different background of both libraries and the fact that libSBML is based on `C` and `C++` code, some differences are unavoidable. In cases of doubt JSBML tries to mirror the SBML specifications rather than libSBML. Finally, JSBML has also been developed as a library that does not "only" provide reading, manipulating, and writing abilities for SBML files. It is intended to be directly used as a flexible internal data structure for numerical computation, visualization and much more. With the help of its modules JSBML can also be used as a communication layer between applications. For instance, JSBML facilitates the implementation of plugins for the program know as CellDesigner (Funahashi *et al.*, 2003). The following sections will not only give a detailed overview about the

most important differences between JSBML and libSBML, but also provide some programming examples and hints about how to use and work with JSBML.

2.2 An extended type hierarchy

Whenever multiple elements defined in at least one of the SBML specifications share some attributes, JSBML provides a common superclass or at least a common interface that gathers methods for the manipulation of the shared properties. In this way, the type hierarchy of JSBML has become quite complex (see Figs. 2.1 to 2.5 on pages 14–18). Just as 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 the interface `TreeNodeWithChangeSupport` that in turn extends three other interfaces: `Cloneable`, `Serializable`, override the `clone()` method from the class `java.lang.Object`, all JSBML elements can be deeply copied and are therefore *clone-able*. By extending the interface `Serializable`, it is possible to store JSBML elements in binary form without explicitly writing them 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. `TreeNode` is a type that is 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 can hence be easily visualized. Listing 1.1 on page 9 demonstrates in a simple code example how to parse an SBML file and to immediately display its content on a `JFrame`. The `ASTNode` class in JSBML is also derived from all these three interfaces and can hence be cloned, serialized, and visualized in the same way.

However, it is important to note that JSBML does not depend on any particular graphical user interface because no other classes from `swing` are initialized when loading the interface `TreeNode`.

2.2.1 `AbstractTreeNode`

When looking at the SBML specification, one may notice that SBML defines a data structure in an an entirely tree-based manner. Besides `SBase`, SBML contains also other kinds of tree nodes that are hierarchically linked within the `SBMLDocument`. In order to unify the programming interface, JSBML defines abstract data types as top-level ancestors for its `SBase` implementation as well as all other hierarchical elements, such as `Annotation`, `ASTNode`, `Creator`, `CVTerm`, `History`, and `XMLNode` (for notes in XHTML format).

First, the interface `TreeNodeWithChangeSupport` defines a *cloneable* and *serializable* version of `TreeNode`. In addition, it also provides methods to notify dedicated `TreeNodeChangeListeners` about any changes within the data structure.

Figure 2.1: The type hierarchy of the main SBML constructs in JSBML. With letting `SBase` extend the interface `TreeNodeWithChangeSupport` that in turn extends the Java interfaces `Cloneable`, `Serializable`, and `TreeNode`, all derived elements of `SBase` also implement these types. In this way, derivatives of `SBase` can be used wherever an instance of `TreeNode` is requested. Furthermore, SBML elements that do not extend `SBase` are also derived from the identical base type `TreeNodeWithChangeSupport`, hence sharing several common methods and attributes. Elements colored in blue have been introduced as additional, in most cases abstract, data types in JSBML but do not have a corresponding element in libSBML. The yellow types `Creator` and `ModelHistory` correspond to `ModelCreator` and `ModelHistory` in libSBML.

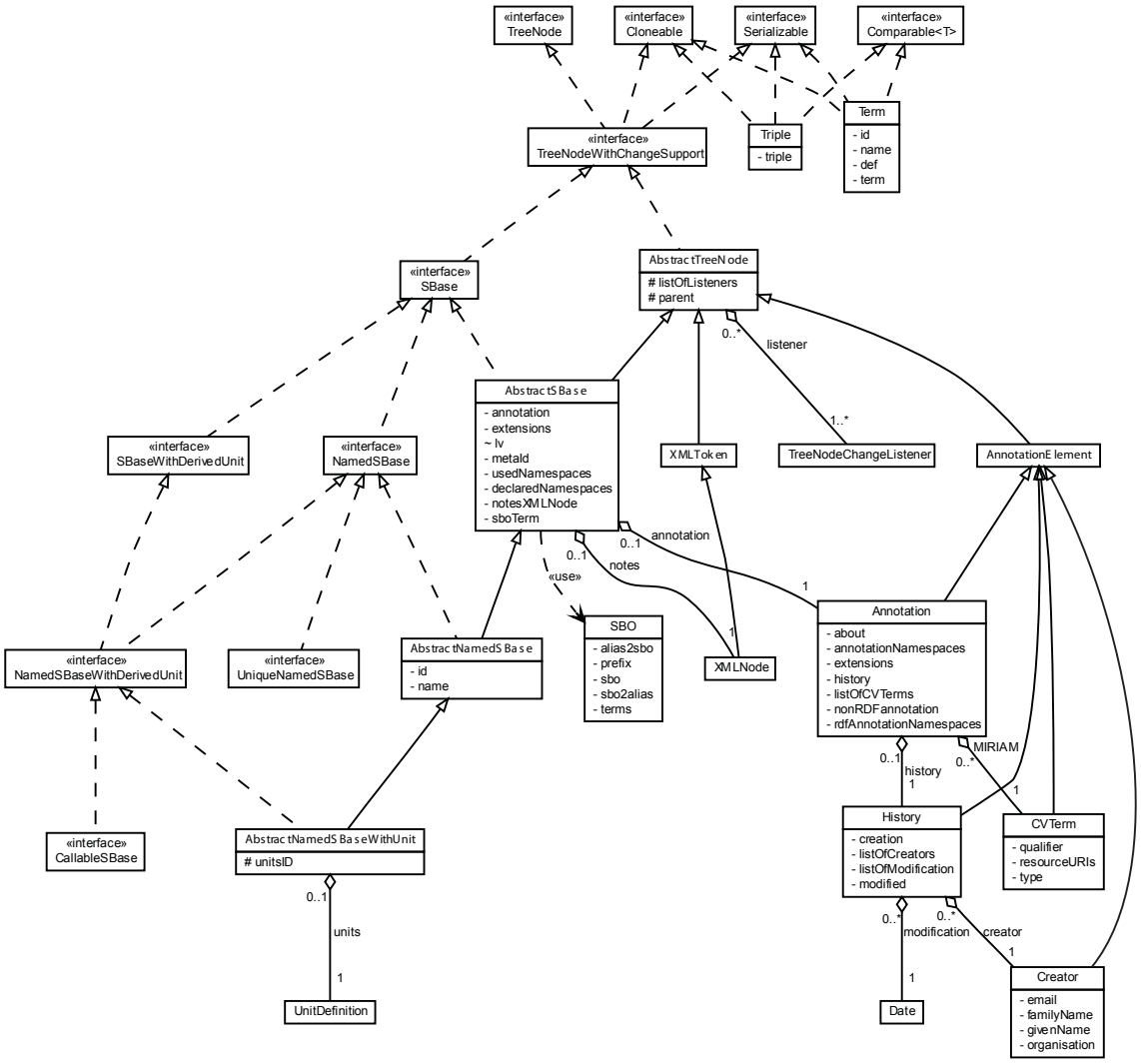

Figure 2.2: The interface `SBase`. This figure displays the most important top-level data structures of JSBML with main focus on the differences to libSBML. All data types that represent SBML constructs in JSBML extend `AbstractTreeNode`. Derivatives of `SBase` extend either one of the two abstract classes `AbstractSBase` or `AbstractNamedSBase`, which in turn also extend `AbstractTreeNode`. 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 all methods in the UML diagram. `SBO` stores its ontology in the classes `Term` that are interrelated in `Triples` consisting of subject, predicate, and object (each being an instance of `Term`).

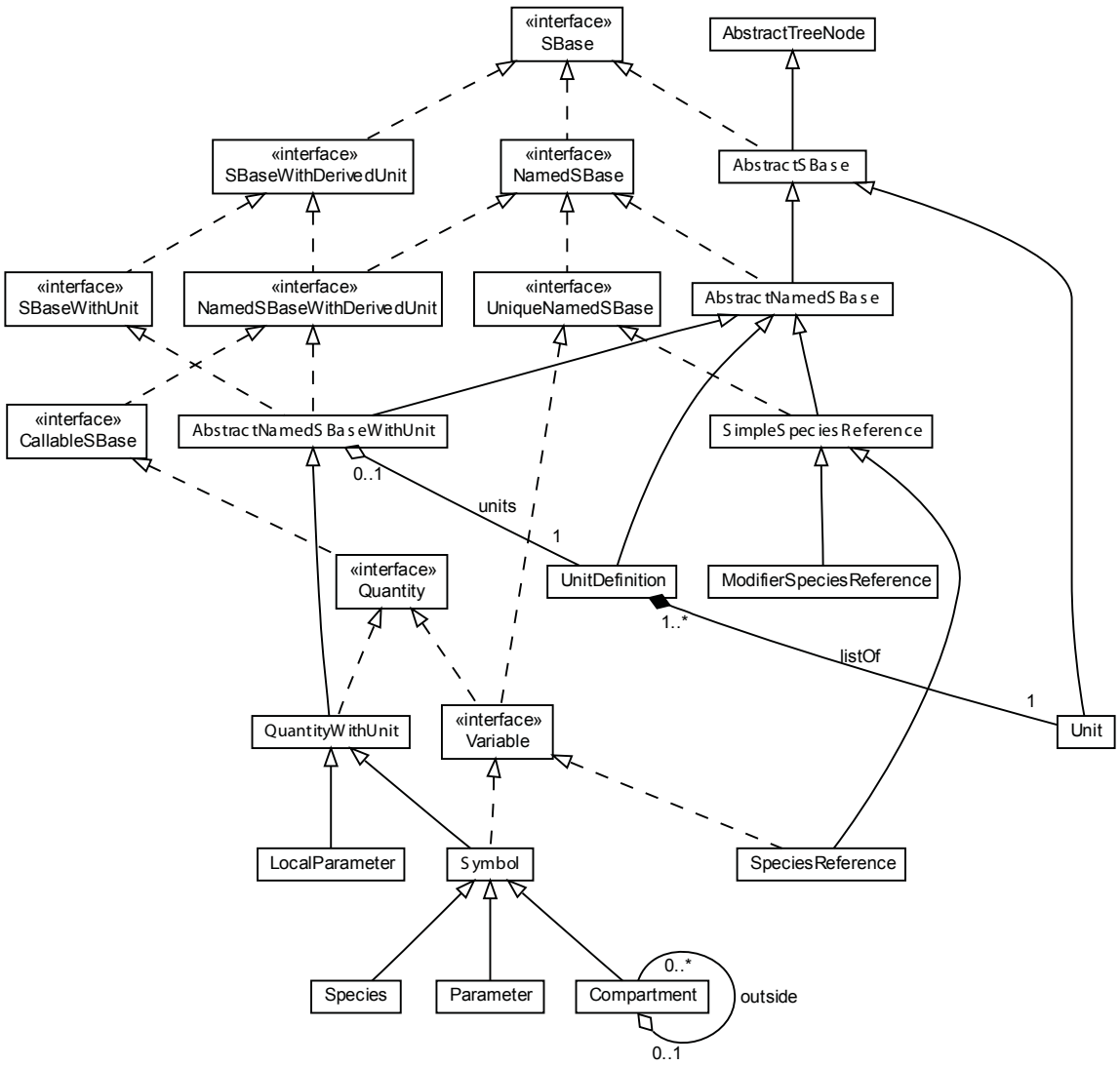

Figure 2.3: The interface `Variable`. JSBML refers to those components of a model that may change their value during a simulation as `Variables`. The class `Symbol` serves as the abstract superclass for variables that can also be equipped with a unit. Instances of `Parameter` do not contain any additional 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`.

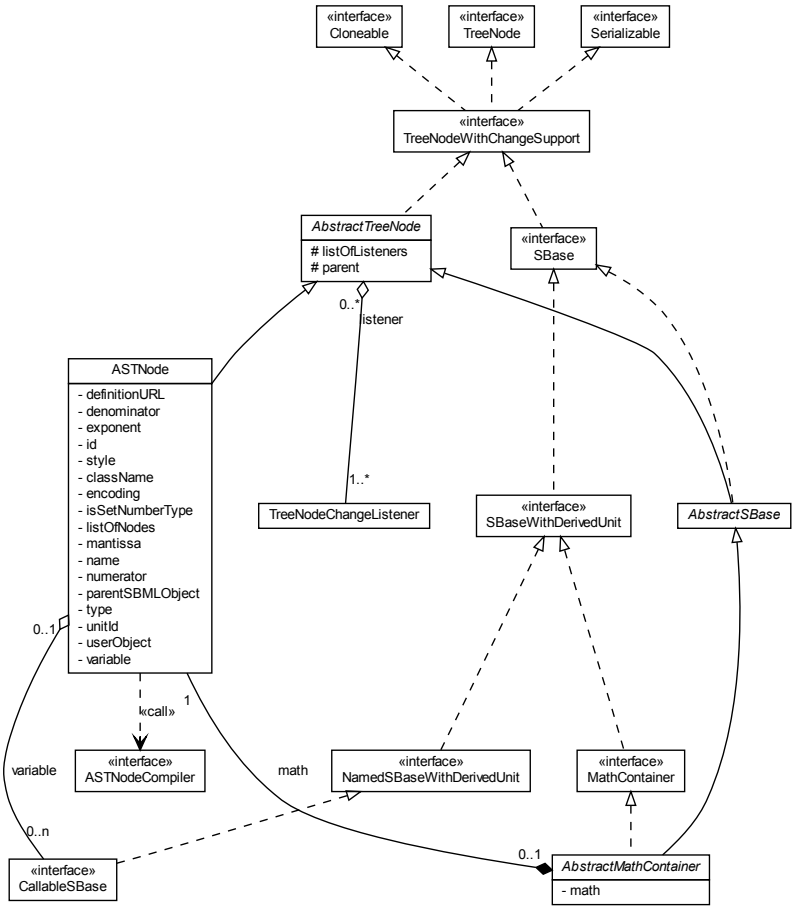

Figure 2.4: Abstract syntax trees. The class `AbstractMathContainer` serves as the superclass for several model components in JSBML. It provides methods to manipulate and access an instance of `ASTNode`, which can be converted to or read from `C`-like formula `String`s. Internally, `AbstractMathContainer`s only deal with instances of `ASTNode`. It should be noted that these abstract syntax trees do not implement the `SBase` interface, but extend `AbstractTreeNode`.

Figure 2.5: Containers for mathematical expressions. The interface `MathContainer`, particularly its directly derived class `AbstractMathContainer`, constitutes the superclass 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`, `Priority`, `StoichiometryMath`, or `AlgebraicRule`.

Its abstract implementation, AbstractTreeNode, does already implement many of the methods inherited from TreeNodeWithChangeSupport and also maintains a list of TreeNodeChangeListeners. Furthermore, this class contains a basic implementation of the methods equals and hashCode, which both already make use of a recursive call over all descendants within the hierarchical SBML data structure. Based on this class, the implementation of all derived data types has become much simpler. The abstract implementation of SBase is also an instance of AbstractTreeNode.

2.2.2 Characteristic features of SBases

The SBML specifications define the data type SBase as the supertype for all other SBML elements. In JSBML, SBase has become an interface and most elements therefore extend its abstract implementation AbstractSBase.

In contrast to libSBML, the Level and Version of such an AbstractSBase is stored in a special generic object, a ValuePair. The class ValuePair takes two values of any type that both implement the interface Comparable. Storing the Level/Version combination in such a ValuePair, which itself implements the Comparable interface, allows users to perform checks for an expected Level/Version combination of an element more easily, as the example in Listing 2.1 demonstrates. The method getLevelAndVersion() in AbstractSBase delivers

Listing 2.1: Check for a minimal expected Level/Version combination

```
1 if (mySBase.getLevelAndVersion().compareTo(Integer.valueOf(2),
2         Integer.valueOf(2)) < 0) {
3   throw new IllegalArgumentException(String.format(
4           "Cannot␣create␣a␣%s␣with␣Level␣=␣%s␣and␣Version␣=␣&s.",
5           mySBase.getElementName(), getLevel(), getVersion()));
6 }
```

an instance of ValuePair with the Level and Version combination for the respective element.

Some types derived from SBase contain an identifier, a so-called id. JSBML gathers all these elements under the common interface NamedSBase. The class AbstractNamedSBase, which extends AbstractSBase, implements this interface. The interface UniqueNamedSBase indicates all those elements whose identifier must be unique within the model, i.e., no other element within the model may have the same identifier. The identifiers of all instances of NamedSBase must be unique if these are defined. The Boolean method isIdMandatory() in NamedSBase indicates if an identifier must be defined for an element in order to create a valid SBML data structure. The only two elements with not-unique identifiers are UnitDefinitions, whose identifiers exist in a separate namespace, and LocalParameters, whose identifiers may shadow the identifiers of global elements.

Many SBML elements represent some quantitative value, which is associated with a unit. However, the value does not necessarily have to be defined explicitly. In many cases, it needs to be com-

puted from a formula contained in the instance of SBase in form of an abstract syntax tree, i.e., ASTNode. Therefore, also the associated unit may not be set explicitly but can be derived when evaluating the formula. In JSBML, the interface SBaseWithDerivedUnit unifies all those elements that either explicitly or implicitly contain some unit. If these elements can also be addressed using an identifier, they also implement the interface NamedSBaseWithDerivedUnit. Within formulas, i.e., ASTNodes, references can only be made to instances of CallableSBase, which is a special case of NamedSBaseWithDerivedUnit. Fig. 2.3 on page 16 shows this part of JSBML's type hierarchy in more detail.

As a special case, these elements may explicitly declare a unit. The interface SBaseWithUnit serves as the supertype for all those elements that may be explicitly equipped with a unit. The convenient class AbstractNamedSBaseWithUnit extends AbstractNamedSBase and implements both interfaces SBaseWithUnit and NamedSBaseWithDerivedUnit. All elements derived from this abstract class may therefore declare a unit and can be addressed using an unambiguous identifier.

Furthermore, the interface Quantity describes an element that is associated with a value and at least a derived unit. In addition, a Quantity can be addressed using its unambiguous identifier. JSBML uses the term QuantityWithUnit for a Quantity that explicitly declares its unit. In contrast to Quantity that explicitly declares its unit. In contrast to Quantity, the data type QuantityWithUnit is not an interface, but an abstract class.

If a Quantity provides a Boolean switch to decide whether it describes a constant, JSBML represents such a type in the interface Variable. Finally, JSBML refers to Variables with a defined unit as a Symbol and provides a corresponding abstract class. In this way, the SBML elements Compartment, Parameter, and Species are special cases of Symbol in JSBML. The specification of SBML Level 3 introduces another type of Variable, which does not explicitly declare its unit: SpeciesReference. On the other hand, a LocalParameter is a QuantityWithUnit, but not a Variable, because it is always constant.

2.2.3 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 superclass for the majority of the derived types. Figs. 2.4 to 2.5 on pages 17–18 give a better overview of how this data structure is intended to function.

2.2.4 The Assignment interface

JSBML unifies all those elements that may change the value of some *variable* 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 isSetVari-

able() as well as unsetVariable(). In addition to that, JSBML also provides the methods 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. Fig. 2.5 on page 18 displays the type hierarchy of the Assignment interface in more detail.

2.3 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++ and C code to Java would not be very elegant. JSBML wants to provide a Java API, whose classes and methods are structured, 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.

2.3.1 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 Strings, 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

```
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.

In SBML, abstract syntax trees may refer to the following elements: Parameters, LocalParameters, FunctionDefinitions, Reactions, Compartments, Species, and, since Level 3, also SpeciesReferences. JSBML gathers all these elements under the common interface CallableSBase, which extends the interface NamedSBaseWithDerivedUnit. In this way, JSBML ensures that only identifiers of those elements can be set in instances of ASTNode. JSBML provides a set of convenient constructors and methods to work with instances of CallableSBase, of which we here give a short overview. The set method allows users to change

Getter and setter:

```
public void setVariable(CallableSBase variable) { ... }

public CallableSBase getVariable() { ... }
```

the type of an ASTNode to ASTNode.Type.NAME and to directly set the name to the identifier of the given CallableSBase. The get method directly looks for the corresponding element in the Modeland returns this element. If no such element can be found or the type of the ASTNode is something different from ASTNode.Type.NAME, an exception will be thrown.

Some examples for convenient manipulation methods, of which some are static:

```
public static ASTNode frac(MathContainer container,
      CallableSBase numerator, CallableSBase denominator) {...}

public static ASTNode pow(MathContainer container,
      CallableSBase basis, CallableSBase exponent) { ... }

public ASTNode plus(CallableSBase nsb) { ... }
```

Methods like these above facilitate creating or manipulating complex abstract syntax trees. Several static methods are available that directly create small trees from given elements in memory, whereas some methods such as the plus method changes the structure of existing syntax trees.

Some examples for convenient constructors:

```
public ASTNode(CallableSBase nsb) { ... }

public ASTNode(CallableSBase nsb, MathContainer parent) { ... }
```

With these constructors, dedicated single nodes can be created whose type (from the enumeration ASTNode.Type) will be NAME and whose name will be set to the identifier of the given CallableSBase.

2.3.2 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 C language-like Strings, LaTeX, or MathML for further processing. Furthermore, the class UnitsCompiler, which JSBML uses to derive the unit of an abstract syntax tree, also implements this interface.

2.3.3 Cloning when adding child nodes to instances of SBase

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 for Java programmers. If cloning is necessary, users should call the `clone()` method manually. Since all instances of `SBase` and also `Annotation`, `ASTNode`, `CVTerm`, and `History` extend `AbstractTreeNode`, which in turn implements the interface `Cloneable` (see Fig. 2.1 on page 14), all these elements can be naturally cloned. However, 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. It is important to note that some properties of the elements must not be copied when cloning:

1. The pointer to the parent node of the top level element that is recursively cloned is not copied and is left as `null` because the cloned object will get a parent set as soon as it is added or linked again to an existing tree. Note that only the top-level element of the cloned subtree will have a `null` value as its parent. All sub-element will point to their correct parent element.

2. The list of `TreeNodeChangeListeners` is needed in all other `setXX()` methods. Copying pointers to these might lead to strange and unexpected behavior, because when doing a deep cloning, the listeners of the old object would suddenly be informed about all value changes within this new object. Since we are cloning, all values of all child elements have to be touched, i.e., all listeners would be informed many times, but each time receive the identical value as it was before. Since it is totally unclear of which type listeners are, a deep cloning of these is not possible.

2.3.4 Deprecation

The intention of JSBML is to provide a Java library that supports the latest specifications of SBML. But we also want to support earlier specifications. So JSBML provides methods and classes to cover elements and properties from earlier SBML specifications as well, but these are often marked as being deprecated to avoid creating models that refer to these elements. Furthermore, JSBML contains many methods just for compatibility with libSBML, for instance, a method such as `getNumXyz()` is not considered to be very Java-like, but very common `C++` programming style. Usually, Java programmers would expect the method being called `getXyzCount()` instead. In cases like this, JSBML provides alternative methods and marks these methods that originate from libSBML as deprecated.

2.3.5 Compartments

In SBML Level 3 (Hucka *et al.*, 2010), the domain of the `spatialDimensions` attribute in `Compartments` was changed from $\{0,1,2,3\}$, which can be represented with a `short` value

in Java, to a value in $\mathbb{R}$, i.e., a `double` value. For this reason, the method `getSpatialDimensions()` in JSBML always returns a `double` value. For consistency with libSBML, the `Compartment` class in JSBML also provides the redundant method `getSpatialDimensionsAsDouble()` that returns the identical value, but that is marked as a deprecated method.

2.3.6 Exceptions

In case of an error, JSBML throws often an exception while libSBML methods return some error codes instead. This behavior helps programmers and users to avoid creating invalid SBML data structures already when dealing with these in memory. Furthermore, exception handling is very well implemented in Java and it is therefore a better programming style in this language. Methods can already declare that these may potentially throw exceptions. In this way, programmers can be aware of potential sources of problems already at the time of writing the source code. 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`.
- An instance of `Priority` can only be assigned to an `Events` if its `level` attribute has at least been set to three.
- Another example is the `InvalidArgumentException` that is thrown when trying to set an invalid identifier `String` for an instance of `AbstractNamedSBase`.
- JSBML keeps track of all identifiers within a model. For each namespace it contains a separate set of identifiers within the `Model`. It is therefore not possible to assign duplicate identifiers in case of elements that implement the interface `UniqueNamedSBase`. For `UnitDefinitions` and `LocalParameters` separate sets are maintained. Since local parameters are only visible within the `KineticLaw` that contain these, JSBML will only prohibit having more than one local parameter within the same list that has the identical identifier. All these sets are updated upon any changes within the model. When adding an element with an already existing identifier for its namespace, or changing some identifier to a value that is already defined within this namespace, JSBML will throw an exception.
- Meta identifiers must be unique through the entire SBML file. To ensure that no duplicate meta identifiers are created, JSBML keeps a set of all meta identifiers on the level of the `SBMLDocument`, which is updated upon any change of elements within the data structure. In this way, it is not possible to set the meta identifier of some element to an already existing value or to add nodes to the SBML tree that contain a meta identifier defined somewhere else within the tree. In both cases, JSBML will throw an exception. Since meta identifiers can be

generated in a fully automatic way (method `nextMetaId()` on `SBMLDocument`), users of JSBML should not care about these identifiers at all. JSBML will automatically create meta identifiers where missing upon writing an SBML file.

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. The class `SBMLReader` in JSBML catches those errors and exceptions. With the help of the logging utility, JSBML notifies users about syntactical problems in SBML files. JSBML follows the rule that illegal or invalid properties are not set.

2.3.7 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`.

2.3.8 Replacement of the interface `libSBMLConstants` by Java `enums`

You will not find an implementation corresponding to the interface `libSBMLConstants` 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 `enum Type`. Instead of typing `libSBML-Constants.AST_TYPE_PLUS`, you would therefore type `ASTNode.Type.PLUS`.

The same holds true for `Unit.Kind.*` corresponding to the `libSBMLConstants.UNIT_-KIND_*` fields.

2.3.9 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, which is 1.0.* at the time of writing this document. 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 myKind = Unit.Kind.valueOf(myString);
```

In a similar way, the ASTNode class provides a method to parse C-like infix formula Strings according to the specification of SBML Level 1 (Hucka *et al.*, 2003) into an abstract syntax tree. Therefore, in contrast to the libSBML class, the class JSBML contains only a few methods.

2.3.10 Various types of ListOf* classes

In JSBML, there is not a specific ListOf* class for each type of SBase elements. We used a generic implementation ListOf<? extends SBase> that allows us to use the same class for each of the different ListOf* classes defined in libSBML while keeping a type-safe class. We defined several methods that use the Filter interface to search or filter a ListOf object. For example, 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 (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 BoundaryCondition-Filter to look for those species that operate on the boundary of the reaction system.

2.3.11 Units and unit definitions

The exponent attribute of 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.

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 with 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.

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 contains methods to generate MIRIAM URIs accordingly. In this way, JSBML facilitates the annotation of user-defined units and unit definitions with MIRIAM-compliant information.

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 to directly access an object representing their unit. Currently, the following elements implement this interface:

- `AbstractNamedSBaseWithUnit`
- `ExplicitRule`
- `KineticLaw`

Fig. 2.1 on page 14 provides a better overview about the relationships between all the classes explained here. Note that `AbstractNamedSBaseWithUnit` serves as the abstract superclass for `Event` and `QuantityWithUnit`. In the class `Event`, all methods to deal with units are deprecated because the `timeUnits` attribute was removed in SBML Level 2 Version 2. The same holds true for instances of `ExplicitRule` and `KineticLaw`, which both can only be explicitly populated with units in SBML Level 1 for `ExplicitRule` and before SBML in Level 2, Version 3 for `KineticLaw`. In contrast, `QuantityWithUnit` serves as the abstract superclass 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`.

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

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.

2.4 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.

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

`nodeAdded(TreeNode node)` This method notifies the listener that the given `TreeNode` has just been added to the `SBMLDocument`. When this method is called, the given node is already fully linked to the `SBMLDocument`, i.e., it has a valid parent that in turn points to the given node.

`nodeRemoved(TreeNode node)` The `TreeNode` instance passed to this method is no longer part of the `SBMLDocument` as it has just been removed. This means that the entire `SBMLDocument` does not contain any pointers to this node anymore, but the node itself still contains a pointer to its former parent. In this way, it is possible to recognize where in the tree this node was located and even to revert the deletion of the node.

`propertyChange(PropertyChangeEvent node)` This method provides detailed information about some value change within the `SBMLDocument`. The object passed to this method is an `TreeNodeChangeEvent`, which provides information about the `TreeNode` 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. To this end, JSBML already provides the implementation `SimpleTreeNodeChangeListener`, which notifies a logger about each change.

Note that the class `TreeNodeChangeEvent` extends the class `java.beans.PropertyChangeEvent`, which is derived from `java.util.EventObject`. It should also be pointed out that the interface `TreeNodeChangeListener` extends the interface `java.beans.PropertyChangeListener` which in turn extends the interface `EventListener` in the package `java.util`. 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.

Since in JSBML all major data objects implement the interface `TreeNode`, these listeners are notified about any kind of change in any implementing data structure. The interface `TreeNodeWithChangeSupport` extends Java's standard `TreeNode` interface by adding methods that maintain a list of `TreeNodeChangeListeners` and notify these whenever some property changes or nodes are added/deleted from the tree. In this way, the `TreeNodeChangeListeners` do not only keep track of changes in instances of `SBase`. This means that changes inside of, e.g., `CVTerm` or `History` may can also be traced with this implementation.

2.4.2 Determination of the variable in `AlgebraicRules`

The class `OverdeterminationValidator` in JSBML provides methods to determine if a model is over determined. This is done using the algorithm of Hopcroft and Karp (1973). While doing that, it also determines the variable element for each `AlgebraicRule` if possible. In JSBML, `AlgebraicRule` even provides a method `getDerivedVariable()` to directly obtain a pointer to its free variable.

2.4.3 `find*` methods

JSBML provides users with several `find*` methods on a `Model` to quickly query for elements, based on their identifier or name. Developers can search for various instances of `SBase` (for instance, `CallableSBase`, `NamedSBase`, `NamedSBaseWithDerivedUnit`) or use the methods `findLocalParameters`, `findQuantity`, `findQuantityWithUnit`, `findQuantityWithUnit`, `findSymbol`, and `findVariable` to search for the corresponding

element in the model. This enables a quick and easy way to work with SBML models, without having to iterate through the elements of a `Model` again and again.

2.4.4 Utility classes provided by JSBML

JSBML also provides some convenient additional utility classes. We here discuss some of these classes in more detail, which are all gathered in the package `org.sbml.jsbml.util`. There you can also find a growing number of additional helpful classes.

Pre-implemented mathematical functions and constants

The class `org.sbml.jsbml.util.Maths` contains several `static` methods for mathematics operations not provided by the standard Java class `java.lang.Math`. Most of these methods are basic operations, for instance, `cot(double x)` or `ln(double x)`. The class `Maths` also provides some less commonly used methods, such as `csc(double x)` or `sech(double x)` as well as `double` constants representing Avogadro's number ($6.02214199 \cdot 10^{23}$ mol^{-1}) and the universal gas constant $R = 8.314472$ J $\cdot$ mol$^{-1} \cdot$ K^{-1}. In this way, the functions and constants implemented in class `Maths` complement standard Java with methods and numbers required by the SBML specifications (Hucka *et al.*, 2003, 2008, 2010).

Some tools for `String` manipulation

The class `StringTools` provides several methods for convenient `String` manipulation. These methods are particularly useful when parsing or displaying `double` numbers in a `Locale`-dependent way. To this end, this class predefines a selection of useful number formats. It can also wrap `String` elements into HTML code, mask non-ASCII characters using corresponding HTML codes, efficiently concatenate `Strings`, or deliver the operating system-dependent new line character.

2.4.5 Logging functionality

JSBML makes use of the logger provided by the log4j project[2]. Log4j allows us to use six levels of logging (`TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, and `FATAL`) but inside JSBML we mainly use `ERROR`, `WARN`, and `DEBUG`. The default configuration of log4j used in JSBML can be found in the folder `resources` with the name `log4j.properties`. In this file, you will found some documentation of which JSBML classes do some logging and at which levels.

If you do not change anything, all the log messages, starting at the info level (meaning info, warn, error and fatal), will be printed on the console. Some of these messages might be useful to warn the end-users that something goes wrong.

[2] `http://logging.apache.org/log4j/`

If you want to modify the default log4j behavior, you will need to create a customized log4j configuration file. The best way of doing this, according to the log4j manual[3], is to define and use the `log4j.configuration` environment variable to point to the log4j configuration file to use. One way of doing this is to add the following option to your `java` command:

```
-Dlog4j.configuration=/home/user/myLog4j.properties
```

Some example configurations

Listing 2.2 gives a short overview about how to customize the configuration file to log all the changes that happen to the SBML elements by putting the threshold of all the loggers in the `org.sbml.jsbml.util` package to `DEBUG`. The class `SimpleTreeNodeChangeListener` will then output the old value and the new value whenever a setter methods is used on the SBML elements.

Listing 2.2: A simple log4j example.

```
1  # All logging output sent to the console
2  log4j.rootCategory=INFO, console
3
4  #
5  # Console Display
6  #
7  log4j.appender.console=org.apache.log4j.ConsoleAppender
8  log4j.appender.console.layout=org.apache.log4j.PatternLayout
9
10 # Pattern to output the caller's file name and line number.
11 log4j.appender.console.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} - %5p
       (%F:%L) - %m%n
12
13 # Log the messages from the SimpleTreeNodeChangeListener at the DEBUG Level
14 # Allow to see all the changes that happened to the SBML elements
15 log4j.logger.org.sbml.jsbml.util=DEBUG
```

When you enable the debug level on some loggers, the output can became quite large and the help of some log viewers software[4] can become handy to filter the log output.

If you are deploying your application in an application server such as Tomcat, you could define an appender that would send some messages by e-mail, Listing 2.3 gives an example of that, were any messages from the error level are send by mail. All the messages are also written to a rolling log file.

Listing 2.3: SMTPAppender log4j example.

```
1  # Logging is sent to a file and by email from the info level.
2  log4j.rootLogger=info, file, mail
```

[3] http://logging.apache.org/log4j/1.2/manual.html
[4] http://en.wikipedia.org/wiki/Log4j#Log_Viewers

```
3
4  #
5  # email appender definition
6  # it will send by email all messages from the error level.
7  #
8  log4j.appender.mail=org.apache.log4j.net.SMTPAppender
9  #defines how often emails are send
10 log4j.appender.mail.BufferSize=1
11 log4j.appender.mail.SMTPHost="smtp.myservername.xx"
12 log4j.appender.mail.From=fromemail@myservername.xx
13 log4j.appender.mail.To=toemail@myservername.xx
14 log4j.appender.mail.Subject=Log ...
15 log4j.appender.mail.threshold=error
16 log4j.appender.mail.layout=org.apache.log4j.PatternLayout
17 log4j.appender.mail.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
18
19 ### file appender
20 log4j.appender.file=org.apache.log4j.RollingFileAppender
21 log4j.appender.file.maxFileSize=100KB
22 log4j.appender.file.maxBackupIndex=5
23 log4j.appender.file.File=test.log
24 log4j.appender.file.threshold=info
25 log4j.appender.file.layout=org.apache.log4j.PatternLayout
26 log4j.appender.file.layout.ConversionPattern=%d{ISO8601} %5p %c{1}:%L - %m%n
```

Using XML instead of a properties file to define the log4j configuration, you can even send some log levels to one appender and others to an other appender, using the `LevelRange` filter. In this way, you could output the `DEBUG` messages only to a separate file.

2.4.6 JSBML modules

JSBML modules extend the functionality of JSBML and are provided as separate libraries (JAR files). With the help of the current JSBML modules, JSBML can be used as a communication layer between your application and libSBML (Bornstein *et al.*, 2008) or between your program and the program known as CellDesigner (Funahashi *et al.*, 2003). Furthermore, a compatibility module will try to provide the same package structure and API as in the libSBML Java bindings. In this section, we will give small code examples of how to make use of these modules.

How to use libSBML for parsing SBML into JSBML data structures?

The capabilities of the SBML validator constitute the major strength of libSBML (Bornstein *et al.*, 2008) in comparison to JSBML, which does not yet contain a stand-alone validator for SBML, but makes use of the online validation provided at `http://sbml.org`. Furthermore, if the platform-dependency of libSBML does not hamper your application, or you want to slowly switch from libSBML to JSBML, you may want to be able to still read and write SBML models using libSBML. To this end, the JSBML module `libSBMLio` provides the classes `LibSBMLReader`

Listing 2.4: A simple example for converting libSBML data structures into JSBML data objects

```
1  /** @param args the path to a valid SBML file. */
2  public static void main(String[] args) {
3    try {
4      // Load libSBML:
5      System.loadLibrary("sbmlj");
6      // Extra check to be sure we have access to libSBML:
7      Class.forName("org.sbml.libsbml.libsbml");
8
9      // Read SBML file using libSBML and convert it to JSBML:
10     LibSBMLReader reader = new LibSBMLReader();
11     SBMLDocument doc = reader.convertSBMLDocument(args[0]);
12
13     // Run some application:
14     new JSBMLvisualizer(doc);
15
16   } catch (Throwable e) {
17     e.printStackTrace();
18   }
19 }
```

and `LibSBMLWriter`. Listing 2.4 gives a small example of how to use the `LibSBMLReader`. For this example to run, please make sure to have libSBML installed correctly on your system. The current version of the libSBML/JSBML interface at the time of writing this document requires libSBML version 4.2.0. To this end, you may have to set environment variables, e.g., the `LD_LIBRARY_PATH` under Linux operating system, appropriately. For details, see the documentation of libSBML[5]. Writing SBML works similarly. Example 2.4 will display the content of an SBML file in a `JTree`, similar as shown in Fig. 1.1 on page 11.

How to turn a JSBML-based application into a CellDesigner plugin?

Once an application has been implemented based on JSBML, it can easily be accessed from CellDesigner's plugin menu (Funahashi *et al.*, 2003). To this end, it is necessary to extend two classes that are defined in CellDesigner's plugin API (Application Programming Interface). The Listings 2.5 to 2.6 on pages 34–35 show a very simple example of how to pass CellDesigner plugin model data structures to the translator in JSBML, which creates then a JSBML `Model` data structure. The examples described by Listings 2.5 to 2.6 on pages 34–35 create a plugin for CellDesigner, which displays the SBML data structure in a tree, like the example in Fig. 1.1 on page 11. This example only shows how to translate a plugin 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 2.6 on page 35 is only completed

[5] `http://sbml.org/Software/libSBML`

Listing 2.5: A simple implementation of CellDesigner's abstract class `PluginAction`

```
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   *  which invokes the actual plug-in program. */
9  public class SimpleCellDesignerPluginAction extends PluginAction {
10
11   /** Memorizes a pointer to the actual plug-in program. */
12   private SimpleCellDesignerPlugin plugin;
13
14   /** Constructor memorizes the plug-in data structure. */
15   public SimpleCellDesignerPluginAction(SimpleCellDesignerPlugin plugin) {
16     this.plugin = plugin;
17   }
18
19   /** Executes an action if the given commant occurs. */
20   public void myActionPerformed(ActionEvent ae) {
21     if (ae.getSource() instanceof JMenuItem) {
22       String itemText = ((JMenuItem) ae.getSource()).getText();
23       if (itemText.equals(SimpleCellDesignerPlugin.ACTION)) {
24         plugin.startPlugin();
25       }
26     } else {
27       System.err.printf("Unsupported␣source␣of␣action␣%s\n", ae
28           .getSource().getClass().getName());
29     }
30   }
31
32 }
```

Listing 2.6: A simple example for a CellDesigner plugin using JSBML as a communication layer

```
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 plugin 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 plugin 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 superclass, 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 }
```

by implementing the methods from the superclass, `CellDesignerPlugin`. In this example it is sufficient to leave the implementation empty.

`libSBMLcompat`, the JSBML compatibility module for libSBML

The compatibility module of JSBML will use the same package structure as the libSBML java bindings and provides identically named classes and API. Using the module, it will be possible to switch an existing application from libSBML to JSBML or the other way around without changing any code.

This module is in development and will be available with the version 1.0 of JSBML.

`android`, a compatibility module for Android systems

This module is intended to provide all those classes from the Java™ standard distribution that are required for JSBML, but might be missing on Android systems. Since this module is currently under development, it can be expected to be available with the release of JSBML version 1.0.

3 Howto write extensions

3.1 How to implement extensions in JSBML

This section presents an example for implementing SBML extensions in JSBML. For this, we define the *Example* extension specification and use it to explain the necessary steps to implement it in JSBML.

3.1.1 Extending an SBase

In most cases, you probably want to extend a model. Listing 3.1 shows the beginning of the class `ExampleModel` that is an extension to the standard Model of the SBML core.

Listing 3.1: Extending `AbstractSBasePlugin`

```
1 public class ExampleModel extends AbstractSBasePlugin {
2
3   public ExampleModel(Model model) {
4     super(model);
5   }
6
7 }
```

Technically, an extension needs to implement the `SBasePlugin` interface, but since the abstract class `AbstractSBasePlugin` already implements some important methods, extending that one should be preferred.

In this example, the constructor accepts an object that is a `Model`, because that is what we want to extend. The call to the super constructor will save the given model as the `SBase` that is being extended in the `extendedSBase` attribute. For convenience, a getModel() method to retrieve the extended model should also be added

Listing 3.2: Convenience method to retrieve the extended model

```
1   public Model getModel() {
2     return (Model) getExtendedSBase();
3   }
```

3.1.2 Adding new classes

In almost all cases, extensions introduce new classes that have no counterpart in the SBML core. Since those new classes are no extensions to existing ones, no extension-specific work has to be

Listing 3.3: Five necessary methods that should be created for each `Foo` class attribute

```
1  public int getBar();
2  public boolean isSetBar();
3  public void setBar(int value);
4  public boolean unsetBar();
5
6  // if the attribute is a boolean type, additionally
7  public boolean isBar();
```

done here. In the *Example* extension, there is the new `Foo` class that is an `SBase` and extends `AbstractNamedSBase`. It has the three attributes *id*, *name*, and *bar*. For each attribute, there need to be the following five methods, shown here for the *bar* attribute, which is an integer: Note, that if *bar* would be a boolean type, we should also provide the method `isBar()`, which delegates to `getBar()`.

In this special case, *id* and *name* should be unique, so it also implements the `UniqueNamedSBase` interface. Because of that, you will be required to implement the above mentioned methods for *id* anyway, those for *name* are already present in the abstract super class. Listing 3.4 show how those methods should be implemented in general. It is very important to call the `FirePropertyChange` listener in the set and unset methods and to throw the `PropertyUndefineError` in the `method` if the attribute is not set.

Listing 3.4: Five necessary methods that should be created for each `Foo` class attribute in detail

```
1  // use Integer, so we can denote unset values as null
2  public Integer bar;
3
4  public int getBar() {
5    if (isSetBar()) {
6      return bar.intValue();
7    }
8    throw new PropertyUndefinedError(ExampleConstant.bar, this);
9  }
10
11 public boolean isSetBar() {
12   return this.bar != null;
13 }
14
15 public void setBar(int value) {
16   Integer oldBar = this.bar;
17   this.bar = bar;
18   firePropertyChange(ExampleConstant.bar, oldBar, this.bar);
19 }
20
21 public boolean unsetBar() {
22   if (isSetBar()) {
23     Integer oldBar = this.bar;
```

```
24        this.bar = null;
25        firePropertyChange(ExampleConstant.bar, oldBar, this.bar);
26        return true;
27      }
28      return false;
29    }
```

Even though some or all of the attributes of a class are mandatory, the default constructor without arguments needs to be defined. This is due to the internal working of parsers that read SBML files and create the data structure in memory. All attributes can be set after the object has been created.

Nevertheless, some cases are more frequent than others and one can define constructors for those cases. On the other hand, creating a separate constructor for each combination of possible passed argument will probably create too many lines of code that are confusing and more difficult to maintain.

You should at least have the constructors listed in Listing 3.5. As you can see, constructors for id only, level and version only, and all together are implemented. If you delegate the constructor call to the super class, you have to take care of the initialization of your custom attributes yourself (by calling a method like `initDefaults()`). If you delegate to another constructor in your class, you only have to do that at the last one in the delegation chain. Also, as you can see, this class requires at minimum SBML Level 3, Version 1.

Listing 3.5: Constructors for `Foo`

```
1   public Foo() {
2     super();
3     initDefaults();
4   }
5
6   public Foo(String id) {
7     super(id);
8     initDefaults();
9   }
10
11  public Foo(int level, int version){
12    this(null, null, level, version);
13  }
14
15  public Foo(String id, int level, int version) {
16    this(id, null, level, version);
17  }
18
19  public Foo(String id, String name, int level, int version) {
20    super(id, name, level, version);
21    if (getLevelAndVersion().compareTo(Integer.valueOf(3), Integer.valueOf(1))
          < 0) {
22      throw new LevelVersionError(getElementName(), level, version);
23    }
24    initDefaults();
```

```
25    }
26
27    /**
28     * Clone constructor
29     */
30    public Foo(Foo foo) {
31      super(foo);
32
33      bar = foo.bar;
34    }
35
36    public void initDefaults() {
37      addNamespace(ExampleConstant.namespaceURI);
38      bar = null;
39    }
```

As stated above, you may also have additional constructors like this one:

Listing 3.6: Additional constructor for `Foo`

```
1   public Foo(String id, int bar) {
2     this(id);
3     setBar(bar);
4   }
```

3.1.3 `ListOfs`

The *Example* extension adds no new attributes to the extended model, but it introduces a new child in form of a list, in this case a `ListOfFoos`, for the new class `Foo`. Instances of `Foo` can be children of the extended model via a newly defined `ListOfFoos`. For this, the methods `isSetListOfFoos()`, `getListOfFoos()`, `setListOfFoos(ListOf<Foo>)`, and `unsetListOfFoos()` need to be implemented (see Listing 3.7).

Listing 3.7: Implementation of the ListOf methods: `isSetListOfFoos()`, `getListOfFoos()`, `setListOfFoos()`

```
1   public boolean isSetListOfFoos() {
2       if ((listOfFoos == null) || listOfFoos.isEmpty()) {
3         return false;
4       }
5       return true;
6   }
7
8   public ListOf<Foo> getListOfFoos() {
9     if (!isSetListOfFoos()) {
10      Model m = getModel();
11      listOfFoos = new ListOf<Foo>(m.getLevel(), m.getVersion());
12      listOfFoos.addNamespace(ExampleConstants.namespaceURI);
13      listOfFoos.setSBaseListType(ListOf.Type.other);
```

```
14        m.registerChild(listOfFoos);
15      }
16      return ListOfFoos;
17    }
18
19    public void setListOfFoos(ListOf<Foo> listOfFoos) {
20      unsetListOfFoos();
21      this.listOfFoos = listOfFoos;
22      getModel().registerChild(this.listOfFoos);
23    }
24
25    public boolean unsetListOfFoos() {
26      if(isSetListOfFoos()) {
27        ListOf<Foos> oldFoos = this.listOfFoos;
28        this.listOfFoos = null;
29        oldFoos.fireNodeRemovedEvent();
30        return true;
31      }
32      return false;
33    }
```

When adding and removing Foo objects to the model, direct access to the ListOfs should not be necessary. Therefore, convenience methods for adding and removing an object should be added to the model, which will also do additional consistency checking (Listing 3.8).

Listing 3.8: Implementation of ListOf methods `addFoo(Foo foo)`, `removeFoo(Foo foo)`, `removeFoo(int i)`

```
1   public boolean addFoo(Foo foo) {
2       return getListOfFoos().add(foo);
3   }
4
5   public boolean removeFoo(Foo foo) {
6     if (isSetListOfFoos()) {
7       return getListOfFoos().remove(foo);
8     }
9     return false;
10  }
11
12  public void removeFoo(int i) {
13    if (!isSetListOfFoos()) {
14      throw new IndexOutOfBoundsException(Integer.toString(i));
15    }
16    listOfFoos.remove(i);
17  }
18
19  // if the ID is mandatory for Foo objects, one should also add
20  public void removeFoo(String id) {
21    return getListOfFoos().removeFirst(new NameFilter(id));
22  }
```

To let the additional `ListOfFoo` appear as a child of the standard model, the important methods for the `TreeNode` need to be implemented (see Listing 3.9). `getAllowsChildren()` should return `true` in this case, since this extension obviously allows children. The child count and the indices of the children is a bit more complicated, because it varies with the number of `ListOf`s that actually contain elements. So, for every non-empty `ListOf` child of our model extension, we increase the counter by one. If a child is queried by its index, the possibility of an index shift needs to be taken into account.

Listing 3.9: Methods which need to be implemented to make the children available in the extended model

```
1  public boolean getAllowsChildren() {
2    return true;
3  }
4
5  public int getChildCount() {
6    int count = 0;
7
8    if (isSetListOfFoos()) {
9      count++;
10   }
11   // same for each additional ListOf* in this extension
12   return count;
13 }
14
15 public SBase getChildAt(int childIndex) {
16   if (childIndex < 0) {
17     throw new IndexOutOfBoundsException(childIndex + " < 0");
18   }
19
20   int pos = 0;
21   if (isSetListOfFoos()) {
22     if (pos == childIndex) {
23       return getListOfFoos();
24     }
25     pos++;
26   }
27   // same for each additional ListOf* in this extension
28   throw new IndexOutOfBoundsException(MessageFormat.format("Index
         {0,number,integer} >= {1,number,integer}", childIndex, +((int)
         Math.min(pos, 0))));
29 }
```

3.1.4 Create methods

Because a newly created instance of type Foo is not part of the model unless it is added to it, `create*` methods should be provided that take care of all that (see Listing 3.10). These create

methods should be part of the model to which the Foo instance should be added, in this case ExampleModel.

Listing 3.10: Convenience method to create elements

```
1  public class ExampleModel extends AbstractSBasePlugin {
2
3    ...
4
5    // only, if ID is not mandatory in Foo
6    public Foo createFoo() {
7      return createFoo(null);
8    }
9
10   public Foo createFoo(String id) {
11     Foo foo = new Foo(id, getModel().getLevel(), getModel().getVersion());
12     addFoo(foo);
13     return foo;
14   }
15
16   public Foo createFoo(String id, int bar) {
17     Foo foo = createFoo(id);
18     foo.setBar(bar);
19     return foo;
20   }
21 }
```

3.1.5 `equals`, `hashCode`, and `clone`

There are three further methods which should be implemented in an extension class: `equals`, `hashCode` and `clone`. This is no different than in any other Java class, but since mistakes here can lead to bugs that are very hard to find, we will go through the basics anyway.

Whenever two objects `o1` and `o2` should be regarded as equal, i.e., all their attributes are equal, the `o1.equals(o2)` and the symmetric case `o2.equals(o1)` must return `true`, and otherwise `false`. The `hashCode` method has two purposes here, namely allowing for a quick check if two objects might be equal or not, and providing hash values for hash maps or hash sets and such. The relationship between `equals` and `hashCode` is that whenever `o1` is equal to `o2`, their hash codes must be the same. Vice versa, whenever their hash codes are different, they cannot be equal.

Listing 3.11 and 3.12 are examples how to write these methods for the class `Foo` with the attribute `bar`. Since `equals` accepts general objects, it first needs to check if the passed object is of the same class as the object it is called on. Luckily, this has been implemented in `AbstractTreeNode`, the super class of `AbstractSBase`. Each class only checks the attributes it adds to the super class when extending it, but not the `ListOfs`, because they are automatically checked in the `AbstractTreeNode` class, the super class of `AbstractSBase`.

Listing 3.11: Example of the `equals` method

```
@Override
  public boolean equals(Object object) {
    boolean equals = super.equals(object);    // recursively checks all
        children
    if (equals) {
      Foo foo = (Foo) object;
      equals &= foo.isSetBar() == isSetBar();
      if (equals && isSetBar()) {
        equals &= (foo.getBar().equals(getBar()));
      }
      // ...
      // further attributes
    }
    return equals;
  }
```

Listing 3.12: Example of the `hashCode` method. The variable `prime` should be a big prime number to prevent collisions

```
  @Override
  public int hashCode() {
    final int prime = 491;
    int hashCode = super.hashCode();    // recursively checks all children
    if (isSetBar()) {
      hashCode += prime * getBar().hashCode();
    }
    // ...
    // further attributes

    return hashCode;
  }
```

To clone an object, the call to the `clone()` method is delegated to a constructor of that class that takes an object of itself as argument. There, all the elements of the class must be copied, which may require recursive cloning.

Listing 3.13: Example of the `clone` method for the `ExampleModel` class

```
  public ExampleModel clone() {
    return new ExampleModel(this);
  }

  public ExampleModel(ExampleModel model) {
    super();

    if (model.isSetListOfFoos()) {
      listOfFoos = model.listOfFoos.clone();
    }
  }
```

Listing 3.14: Example of the `clone` method for the `Foo` class

```
1   public Foo clone() {
2     return new Foo(this);
3   }
4
5   public Foo(Foo f) {
6     super();
7
8     // Integer objects are immutable, so they can be copied
9     bar = f.bar;
10  }
```

3.1.6 Writing the parser/writer

One last thing is missing to be able to properly read and write SBML files using the new extension: a parser and a writer. An easy way to do that is to extend the `AbstractReaderWriter` and implement the required methods. To implement the parser, in this case the `ExamplePaser`, one should start with two members and two simple methods, as shown in Listing 3.15.

As can be seen from this code snippet, an additional class `ExampleConstant` and an enum `ExampleList` are used.

TODO

Listing 3.15: The first part of the parser for the extension

```
1   public class ExampleParser extends AbstractReaderWriter {
2
3     /**
4      * The logger for this parser
5      */
6     private Logger logger = Logger.getLogger(ExampleParser.class);
7
8     /**
9      * The ExampleList enum which represents the name of the list this parser is
10    * currently reading.
11    */
12    private ExampleList groupList = ExampleList.none;
13
14    /* (non-Javadoc)
15     * @see org.sbml.jsbml.xml.parsers.AbstractReaderWriter#getShortLabel()
16     */
17    public String getShortLabel() {
18      return ExampleConstant.shortLabel;
19    }
20
21    /* (non-Javadoc)
22     * @see org.sbml.jsbml.xml.parsers.AbstractReaderWriter#getNamespaceURI()
23     */
24    public String getNamespaceURI() {
```

```
25      return ExampleConstant.namespaceURI;
26    }
27
28  }
```

Writing

The method `getListOfSBMLElementsToWrite()` (see 3.16) has to return a list of all objects that have to be written because of the passed object. This way, the writer can traverse the XML tree to write all nodes. Basically, there are three classes of objects that need to be distinguished:

- `SBMLDocument`
- extended classes
- `TreeNode`

TODO: SBMLDocument. After that we need to check if the current object is extendable using our extension. In our example extension, a model can be extended using ExampleModel to also have a list of Foos as children. In Java, this ListOfFoos is not a children of the original model, but of the example model. The example model, on the other hand, is just an SBasePlugin, which is not an SBase and also not a children of the original model. To "inject" the ListOfFoos in the right place, all children of the example model in Java become direct children of the original model in XML.

All other objects that implement SBase also implement TreeNode, so we just add all of their children to the list of elements to write.

Listing 3.16: Extension parser: `getListOfSBMLElementsToWrite()`

```
1   public ArrayList<Object> getListOfSBMLElementsToWrite(Object sbase) {
2
3     if (logger.isDebugEnabled()) {
4       logger.debug("getListOfSBMLElementsToWrite␣:␣" +
            sbase.getClass().getCanonicalName());
5     }
6
7     ArrayList<Object> listOfElementsToWrite = new ArrayList<Object>();
8
9     if (sbase instanceof SBMLDocument) {
10      // nothing to do
11      // TODO : the 'required' attribute is written even if there is no plugin
            class for the SBMLDocument, so I am not totally sure how this is
            done.
12    }
13    else if (sbase instanceof Model) {
14      ExampleModel modelGE = (ExampleModel) ((Model)
            sbase).getExtension(ExampleConstant.namespaceURI);
15
```

```
16        Enumeration<TreeNode> children = modelGE.children();
17
18        while (children.hasMoreElements()) {
19          listOfElementsToWrite.add(children.nextElement());
20        }
21      }
22      else if (sbase instanceof TreeNode) {
23        Enumeration<TreeNode> children = ((TreeNode) sbase).children();
24
25        while (children.hasMoreElements()) {
26          listOfElementsToWrite.add(children.nextElement());
27        }
28      }
29
30      if (listOfElementsToWrite.isEmpty()) {
31        listOfElementsToWrite = null;
32      } else if (logger.isDebugEnabled()) {
33        logger.debug("getListOfSBMLElementsToWrite size = " +
              listOfElementsToWrite.size());
34      }
35
36      return listOfElementsToWrite;
37    }
```

In some cases it may be necessary to modify the `writeElement()` method. For example, this can happen when the same Java class is mapped to different XML tags, e.g., a default element and multiple additional tags. If this would be represented not via an attribute, but by using different tags, one could alter the name of the XML object in this method.

The actual writing of XML attributes must be implemented in each of the classes in the `writeXMLAttributes()`. An example is shown in Listing 3.17 for the class `Foo`.

Listing 3.17: Method to write the XML attributes

```
1  public class Foo extends AbstractNamedSBase {
2    ...
3
4    public Map<String, String> writeXMLAttributes() {
5      Map<String, String> attributes = super.writeXMLAttributes();
6      if (isSetBar()) {
7        attributes.remove("bar");
8        attributes.put(Foo.shortLabel + ":bar", getBar());
9      }
10
11     // ...
12     // further class attributes
13   }
14 }
```

Parsing

The `processStartElement()` method is responsible for handling start elements, such as `<listOfFoos>`, and creating the appropriate objects. The `contextObject` is the object representing the parent node of the tag the parser just encountered. First, you need to check for every class that may be a parent node of the classes in your extension. In this case, those are objects of the classes `Model`, `Foo` and `ListOf`. Note, that the `ExampleModel` has no corresponding XML tag and the core model is already handled by the core parser. This also means that the context object of a ListOfFoos is not of the type `ExampleModel`, but of type `Model`. But since the `ListOfFoos` can only be added to an `ExampleModel`, the extension is retrieved or created on the fly.

The `groupList` variable is used to keep track of where we are in nested structures. If the `listOfFoos` starting tag is encountered, the corresponding enum value is assigned to that variable. Due to Java's type erasure, the context object inside a listOfFoos tag is of type `ListOf<?>` and a correctly set `groupList` variable is the only way of knowing where we are. If we have checked that we are, in fact, inside a `listOfFoos` node and encounter a `foo` tag, we create `Foo` object and add it to the example model. Technically, it is added to the `ListOfFoos` of the example model, but since `ExampleModel` provides convenience methods for managing its lists, it is easier to call the `addFoo()` method on it.

Listing 3.18: Extension parser: `processStartElement()`

```
1  // Create the proper object and link it to his parent.
2  public Object processStartElement(String elementName, String prefix,
3      boolean hasAttributes, boolean hasNamespaces, Object contextObject)
4  {
5
6    if (contextObject instanceof Model) {
7      Model model = (Model) contextObject;
8      ExampleModel exModel = null;
9
10     if (model.getExtension(ExampleConstant.namespaceURI) != null) {
11       exModel = (ExampleModel)
             model.getExtension(ExampleConstant.namespaceURI);
12     } else {
13       exModel = new ExampleModel(model);
14       model.addExtension(ExampleConstant.namespaceURI, exModel);
15     }
16
17     if (elementName.equals("listOfFoos")) {
18
19       ListOf<Foos> listOfFoos = exModel.getListOfFoos();
20       this.groupList = QualList.listOfFoos;
21       return listOfFoos;
22     }
23   } else if (contextObject instanceof Foo) {
24     Foo foo = (Foo) contextObject;
```

```
25
26        // if Foo would have children, that would go here
27
28      }
29      else if (contextObject instanceof ListOf<?>)
30      {
31        ListOf<SBase> listOf = (ListOf<SBase>) contextObject;
32
33        if (elementName.equals("foo") &&
              this.groupList.equals(QualList.listOfFoos)) {
34          Model model = (Model) listOf.getParentSBMLObject();
35          ExampleModel exModel = (ExampleModel)
                model.getExtension(ExampleConstant.namespaceURI);
36
37          Foo foo = new Foo();
38          exModel.addFoo(foo);
39          return foo;
40        }
41      }
42      return contextObject;
43    }
```

The processEndElement() method is called whenever a closing tag is encountered. The groupList attribute needs to be updated to reflect the step up in the tree of nested elements. In this example, if the end of `</listOfFoos>` is reached, we certainly are inside the model tags again, which is denoted by *none*. Of course, more complicated extensions with lots of nested lists need a more elaborate handling here, but it should still be straight-forward.

Listing 3.19: Extension parser: `processEndElement()`

```
1   /* (non-Javadoc)
2    * @see
         org.sbml.jsbml.xml.parsers.AbstractReaderWriter#processEndElement(java.lang.String,
         java.lang.String, boolean, java.lang.Object)
3    */
4   public boolean processEndElement(String elementName, String prefix,
5       boolean isNested, Object contextObject)
6   {
7
8     if (elementName.equals("listOfFoos")
9     {
10      this.groupList = QualList.none;
11    }
12
13    return true;
14  }
```

Attributes of a tag are read into the corresponding object via the `readAttributes()` method that must be implemented for each class. An example is shown in Listing 3.20 for the class `Foo`.

Listing 3.20: Method to read the XML attributes

```
1  @Override
2  public boolean readAttribute(String attributeName, String prefix, String
       value) {
3
4    boolean isAttributeRead = super.readAttribute(attributeName, prefix,
         value);
5
6    if (!isAttributeRead) {
7      isAttributeRead = true;
8
9      if (attributeName.equals(ExampleConstant.bar)) {
10       setBar(StringTools.parseSBMLInt(value));
11     } else {
12       isAttributeRead = false;
13     }
14   }
15
16   return isAttributeRead;
17 }
```

3.2 Implementation checklist

- ☐ Add the extension to an existing model (see Listing 3.1)
- ☐ Add the five necessary methods for each class attribute (see Listing 3.3, 3.4):
  - ☐ `getBar()`
  - ☐ `isBarMandatory()`
  - ☐ `isSetBarFoo()` (only required if the attribute is an id)
  - ☐ `setBar(int value)`
  - ☐ `unsetBar()`
- ☐ Add the default constructors (see Listing 3.5)
- ☐ If the class has children, check if all list methods are implemented (see Listing 3.9, 3.7, 3.8, 3.9):
  - ☐ `isSetListOfFoos()`
  - ☐ `getListOfFoos()`
  - ☐ `setListOfFoos(ListOf<Foo> listOfFoos)`
  - ☐ `addFoo(Foo foo)`
  - ☐ `removeFoo(Foo foo)`

- ☐ `removeFoo(int i)`
- ☐ `getAllowsChildren()`
- ☐ `getChildCount()`

☐ All necessary create methods are implemented (see Listing 3.10)

☐ Implement the `equals()` method (see Listing 3.11)

☐ Implement the `hashCode()` method (see Listing 3.12)

☐ Implement the `clone()` method (see Listing 3.13 and 3.14)

☐ Implement the `toString()` method

☐ Implement the `writeXMLAttribute()` method (see Listing 3.17)

☐ Implement the parser/writer method (see Listing 3.15, 3.16, 3.18, 3.19):

4 Open tasks in JSBML

- JSBML does not yet provide a stand-alone validator for SBML. It currently uses the online validator for SBML.
- The support for SBML Level 3 should be completed by implementing the extension packages.
- The `toSBML()` methods in `SBase` are missing.
- Constructors and methods with namespaces are not yet provided.
- The libSBML compatibility module needs to be fully implemented.
- Also the `android` module is not ready yet.
- A more general implementation for ontology access and manipulation in order to access other ontologies than just the SBO. See, for instance, the work of Courtot *et al.* (2011) for details.

Appendix A

Frequently Asked Questions (FAQ)

For questions regarding SBML, please see the SBML FAQ at `http://sbml.org/Documents/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.

Does JSBML depend on SWING or any particular graphical user interface implementation?

Although all classes in JSBML implement the `TreeNode` interface, which is located in the package `javax.swing.tree`, all classes in JSBML are entirely independent from any graphical user interface, such as the SWING implementation. When loading the `TreeNode` interface, no other class from SWING will be initialized or loaded; hence JSBML can also be used on computers that do not provide any graphical system without the necessity of catching a `HeadlessException`. The `TreeNode` interface only defines methods and properties that all recursive tree data structures have to implement anyway. Letting JSBML classes extend this interface makes JSBML compatible with many other Java classes and methods that make use of the standard `TreeNode` interface, hence ensuring a high compatibility with other Java libraries. Since the SWING package belongs to the standard Java™ distribution, the `TreeNode` interface should always be localized by the Java Virtual Machine, independent from the specific hardware or system. Android systems might be an exceptional case, which do not provide any parts from the SWING package of Java. Therefore, the JSBML team is currently developing a specialized `android` compatibility module for JSBML. You can obtain this module by checking out the repository `https://jsbml.svn.sourceforge.net/svnroot/jsbml/modules/android` or by downloading this as a binary from the download page of JSBML.

Does the usage of the the `java.beans` package for the `TreeNodeChangeListener` lead to an incompatibility with light-weight Java installations?

With the `java.beans` package being part of the standard Java distribution, such an incompatibility will not occur. Extending existing standard Java classes leads to a higher compatibility with other libraries and should therefore be the preferred way to go in the development of JSBML.

Does JSBML support SBML extension packages?

In version 0.8, JSBML did not provide an abstract programming interface for extension packages. Since version 1.0 the JSBML community has actively developed extension packages for the following SBML extensions: `fba`, `groups`, `layout`, `multi`, `qual`, and `spatial`. These packages can be used with the latest release of JSBML.

Appendix B

Acknowledgments and funding

The authors are grateful to Meike Aichele , Finja Bc̈hel, Sebastian Fröhlich, Roland Keller, Florian Mittag, Sarah Rachel Müller vom Hagen, Alexander Peltzer, and Simon Schäfer who all contributed to the JSBML project (alphabetic order).

The development of JSBML is funded by the National Institute of General Medical Sciences (NIGMS, USA); funds from EMBL-EBI (Germany, UK); Federal Ministry of Education and Research (BMBF, Germany) in the projects Virtual Liver and Spher4Sys (grant numbers 0315756 and 0315384C). The grant number for the NIH grant that was, among others, used for the JSBML article reads 2R01GM070923.

Bibliography

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

Courtot, M., Juty, N., Knüpfer, C., Waltemath, D., Zhukova, A., Dräger, A., Dumontier, M., Finney, A., Golebiewski, M., Hastings, J., Hoops, S., Keating, S., Kell, D. B., Kerrien, S., Lawson, J., Lister, A., Lu, J., Machne, R., Mendes, P., Pocock, M., Rodriguez, N., Villeger, A., Wilkinson, D. J., Wimalaratne, S., Laibe, C., Hucka, M., and Le Novère, N. (2011). Controlled vocabularies and semantics in systems biology. *Molecular Systems Biology*, **7**(1), 543.

Dräger, A. (2011). *Computational Modeling of Biochemical Networks*. Ph.D. thesis, University of Tuebingen, Tübingen, Germany.

Dräger, A., Rodriguez, N., Dumousseau, M., Dörr, A., Wrzodek, C., Le Novère, N., Zell, A., and Hucka, M. (2011). JSBML: a flexible Java library for working with SBML. *Bioinformatics*, **27**(15), 2167–2168.

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.

Hopcroft, J. E. and Karp, R. M. (1973). An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, **2**, 225.

Hucka, M., Finney, A., Sauro, H., and Bolouri, H. (2003). 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., 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