User Guide for JSBML

Version: 1.0.0 (preview)

Authors:

Andreas Dräger [a]
Alexander Dörr [a]
Florian Mittag [a]
Nicolas Rodriguez [b]
Clemens Wrzodek [a]
Michael Hucka [c]
Marine Dumousseau [b]
Finja Büchel [a]

Principal investigators:

Nicolas Le Novère [b]
Andreas Zell [a]
Michael Hucka [c]

Institutional affiliations:

[a] Center for Bioinformatics Tuebingen
University of Tuebingen
Tübingen, Germany

[b] European Bioinformatics Institute
Wellcome Trust Genome Campus
Hinxton, Cambridge, UK

[c] Computing and Mathematical Sciences
California Institute of Technology
Pasadena, CA, USA

SBML (the Systems Biology Markup Language) is an XML-based format for storing and exchanging computational descriptions of biological processes. To read, write, manipulate, and perform higher-level operations on SBML files and data streams, software applications need to map SBML entities to suitable software objects. JSBML provides a pure Java library for this purpose. It supports all Levels and Versions of SBML, and provides many powerful features, including facilities to help migrate from the use of libSBML (a popular library for SBML that is not written in Java).

This document provides an introduction to JSBML and its use. It is aimed at both developers writing new Java-based applications as well as those who want to adapt libSBML-based applications to using JSBML. This user guide is a companion to the JSBML API documentation.

The JSBML home page is http://sbml.org/Software/JSBML/.

JSBML

Contents

1 Getting started with JSBML

JSBML is a Java™ library that will help you to read, write and manipulate SBML files [8, 9]. This chapter provides information for quickly getting started with using JSBML.

Before you can use JSBML, you will need to obtain a copy of the library. Section 1.1 below describes different ways of doing this, and explains which additional libraries you may need. JSBML also requires the use of a Java Runtime Environment (JRE) version 1.5 or later [23]. In the rest of this document, we assume that you have already installed a suitable JRE or Java Development Kit (JDK), and know how to configure the Java class path on your system.

It is also essential to *understand SBML* in order to be able to use it (and JSBML) properly. If you are not already familiar with SBML, a good starting point for learning about it is the latest SBML specification [25]. You can find answers to many questions in the SBML FAQ [24] and optionally by asking on one of the SBML discussion lists [26].

1.1 Obtaining and using JSBML

We provide four options for obtaining a copy of JSBML: (1) download the JAR file distribution for JSBML complete with dependencies, that is, packaged with third-party Java libraries needed by JSBML; (2) download the JAR file distribution for JSBML *excluding* dependencies; (3) download the source code distribution; and (4) obtain the source code directly from the project's Subversion repository. These four options are described below.

1.1.1 The JSBML archive with dependencies

The version of the JSBML archive that includes dependencies is a merged JAR file that contains all of JSBML's required third-party libraries. You can download it from the JSBML area on SourceForge [17]. Once you have installed the JAR file on your computer, it is sufficient to add it to your Java build and/or class path in order to use JSBML.

1.1.2 The JSBML archive without dependencies

The version of the JSBML archive that excludes dependencies is a JAR file that contains only JSBML classes. You can download it from the JSBML area on SourceForge [17]. Since it does not include the third-party libraries needed by JSBML to operate, you will need to obtain and download those libraries separately. Table 1.1 lists what they are. Once you have installed the JSBML JAR file *and* these third-party libraries on your computer, you will need to add them *all* to your Java build and/or class path in order to use JSBML.

Table 1.1: *List of other, third-party libraries needed by JSBML.*

Library name	Purpose	Source URL
`biojava-1.7-ontology.jar`	A stripped-down version of biojava-1.7 containing mostly ontology-related classes [12].	biojava.org
`junit-4.8.jar`	Unit-test support library; only needed if you intend to run the tests in the `tests` folder.	www.junit.org
`stax2-api-3.0.3.jar`	Used for reading and writing XML.	docs.codehaus.org/display/WSTX/StAX2
`stax-api-1.0.1.jar`	Used for reading and writing XML.	stax.codehaus.org
`woodstox-core-lgpl-4.0.9.jar`	Used for reading and writing XML.	woodstox.codehaus.org
`staxmate-2.0.0.jar`	Used for reading and writing XML. Provides a more user-friendly StAX interface.	staxmate.codehaus.org
`xstream-1.3.1.jar`	Used for reading and writing XML, specifically parsing results from the SBML validator.	xstream.codehaus.org
`jigsaw-dateParser.jar`	Portion of the *Jigsaw* library (version from Dec. 2010), containing classes for date manipulation.	jigsaw.w3.org
`log4j-1.2.8.jar`	Library for logging errors and other diagnostics.	logging.apache.org/log4j

1.1.3 The JSBML source archive

The source distribution for JSBML is similar to the JAR distribution that excludes third-party dependency libraries, except that the JSBML files are not compiled into class files; you must compile them yourself. As with the other options described above, the source distribution is available from the JSBML area on SourceForge [17], as an archive file in both ZIP and compressed TAR archive formats.

Download whichever format is more convenient for you and unpack the archive on your computer somewhere. The act of unpacking the archive will create a folder on your computer named after the distribution version; for example, this may be "`jsbml-1.0`". Next, you will need to compile the Java source code. JSBML comes with a *build file* (i.e., scripted instructions in a specialized format) for Apache Ant [2]; you can use other approaches for compiling the JSBML classes and performing other tasks, but Ant provides an especially convenient approach. For the rest of the instructions below, we use Ant. Here is an example of how to compile the JSBML class files after you have unpacked the source code archive:

```
cd jsbml-1.0
ant compile
```

Compiling JSBML with Ant; this example uses Bash shell syntax.

Next, if you wish to run the self-tests included with JSBML, you can do so by running the following command:

```
ant test
```

Running the unit tests provided with JSBML.

Finally, if you want to produce a JAR file containing all the JSBML compiled class files, run the following command:

```
ant jar
```

Creating a JAR file.

1.1.4 The JSBML source code repository

The fourth approach to obtaining a copy of JSBML is to retrieve it directly from the Subversion repository [18]. Here is an example of how to retrieve the latest version of the core JSBML sources:

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

Downloading the latest JSBML sources from the JSBML project's Subversion repository.

(The name you give to the copy on your computer is up to you. We used "*`jsbml`*" in this example, but you could name the folder something else if you wish.) Once you have retrieved the folder from the Subversion repository, you can compile the source files and create a JAR file. Please refer to the instructions in Section 1.1.3.

The Subversion repository contains copies of all the third-party libraries listed in Table 1.1 on the previous page and needed by JSBML. They are located in the folder "*`jsbml`*`/lib`".

1.1.5 Optional extensions, modules and examples available for JSBML

JSBML provides a number of additional extensions, modules and example programs that you may find useful in your work. The *extensions* are optional add-ons that implement support for SBML Level 3 Packages; these packages extend SBML syntax to support, for example, storing the layout of a model's graphical diagram directly in the SBML file. The JSBML *modules* provide additional features and interfaces, for example, to allow CellDesigner [10] plugins to use JSBML. Finally, the JSBML *examples* are full-fledged applications that demonstrate the use of JSBML in actual running software. Each of these optional components of JSBML are available from the project's code repository (and in some cases, from the download area on SourceForge [17]). In the subsections below, we explain how to obtain copies of them from the repository.

JSBML Extensions

The JSBML repository's `extensions` folder contains a separate subfolder for each currently-implemented JSBML extension. You can either retrieve a copy of each extension separately, or obtain the complete `extensions` portion of the repository. Here we explain the latter.

First, find a suitable location on your computer where you would like to place the JSBML extensions folder. (We suggest placing it side-by-side at the same level as your JSBML core folder, e.g., next to the folder "*jsbml*" discussed above.) Then, perform the following step:

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

Downloading the latest JSBML extensions source folder from the project's Subversion repository.

Each of the extensions has its own Ant build script, located in a file named (as per Ant conventions) "`build.xml`" within the extension's subfolder. To build, for example, the `layout` extension, you could do the following:

```
cd extensions/layout
ant compile
```

Compiling the "`layout`" extension.

JSBML Modules

JSBML currently provides five additional modules. Each provides features for task-specific purposes. Binary versions of the modules can be found at the download site of JSBML [17]; here we explain how you can obtain the most recent versions of the modules directly from the source code repository. (Note: at the time of this writing, only the `CellDesigner` and the `libSBMLio` module have been extensively tested.)

First, find a suitable location on your computer where you would like to place the JSBML extensions folder. We suggest creating a folder named "`modules`" placed side-by-side at the same level as your JSBML core folder, e.g., next to the folder "*jsbml*" discussed above.

```
mkdir modules
cd modules
```

Creating a folder for the modules.

Next, perform the following operation, once for each of the modules you would like to obtain, where the variable *modulename* is one of the names listed in the first column of Table 1.2:

```
svn co https://jsbml.svn.sourceforge.net/svnroot/jsbml/trunk/modules/modulename  modulename
```

Obtaining a JSBML module.

(In other words, if you would like to obtain both the Android and libSBML modules, execute the command above twice, once with `android` in place of *modulename* and a second time with `libSBMLcompat` in place of *modulename*.) Once they're downloaded, please check inside each module directory for information about how to use them.

Table 1.2: *JSBML modules available today.*

Module name	Purpose
`android`	Support for writing JSBML-based programs for Android OS.
`celldesigner`	A bridge module that supports writing JSBML-based plugins for CellDesigner [10]
`compare`	Facilities for drawing comparisons between libSBML and JSBML
`libSBMLcompat`	A module that allows easier switching between libSBML and JSBML by providing wrapper classes replicating much of libSBML's API in JSBML
`libSBMLio`	A libSBML communications layer.

You can find more information and explanation about JSBML's modules in Section 3.6 on page 28.

JSBML Examples

The JSBML repository's `examples` folder contains a separate subfolder for each sample application. To obtain them, first, find a suitable location on your computer where you would like to place the JSBML examples folder. We suggest creating a folder named "`examples`" placed side-by-side at the same level as your JSBML core folder.

```
mkdir examples
cd examples
```

Creating a folder for the examples.

Next, retrieve the examples you would like to obtain. At the time of this writing, there is only one example available:

```
svn co https://jsbml.svn.sourceforge.net/svnroot/jsbml/trunk/examples/sbmlbargraph sbmlbargraph
```

Retrieving the SBML Bar Graph *example application.*

Finally, please read the "`README.txt`" file in the freshly-obtained `sbmlbargraph` folder to learn more about how to get started with the example application.

1.2 *Hello World*: writing your first JSBML applications

In this section, we present two examples of using JSBML. The first is a program that reads a file containing an SBML document and displays its components in a Java `JTree` graphical object. The second example illustrates the creation of an object representing an SBML document (which, in JSBML, is represented programmatically using an object of class `SBMLDocument`), as well as writing that object to a file. These basic examples should help serve as a foundation for writing your own, more elaborate programs.

1.2.1 Reading and visualizing an `SBMLDocument` *object*

Figure 1.1 shows the listing of a very simple program called "`JSBMLvisualizer`". When it is run, it expects to be given the path name of a valid SBML file as its sole argument. The program uses the static method `read()` defined by the JSBML object class `SBMLReader` to read the file; `SBMLReader` returns an object of class `SBMLDocument`, the main SBML document container in JSBML. Next, the program constructs a new `JSBMLvisualizer` object, which is

```java
import java.io.File;
import javax.swing.*;
import org.sbml.jsbml.*;

/** Displays the content of an SBML file in a {@link JTree} */
public class JSBMLvisualizer extends JFrame {
    /** @param document The SBML root node of an SBML file */
    public JSBMLvisualizer(SBMLDocument document) {
        super(document.getModel().getId());
        getContentPane().add(new JScrollPane(new JTree(document)));
        pack();
        setVisible(true);
    }

    /**
     * Main routine. Note: this does not perform any error checking, but should. It is an illustration only.
     *
     * @param args Expects a valid path to an SBML file.
     */
    public static void main(String[] args) throws Exception {
        UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        new JSBMLvisualizer(SBMLReader.read(new File(args[0])));
    }
}
```

Figure 1.1: *Parsing and visualizing the content of an SBML file.*

derived from the standard Java `JFrame` class. It invokes the class constructor (line 9) with the identifier of the model in the SBML file, obtained by calling `getModel().getId()` on the `SBMLDocument` object; this sets the `JFrame`'s title to the identifier of the model. Since JSBML's `SBase` object (and all objects derived from it) implement the `TreeNode` interface, it is possible to create a `JTree` directly from the information in an `SBMLDocument` object instance. (To keep our examples short and focused on the essentials of using JSBML, we have omitted error checking steps. A real application program should guard against various situations, such as `getModel()` or `getId()` returning `null`, and take steps to deal with them appropriately. You might also like to read SBML files in a separate thread and monitor the progress of reading the file in some progress bar.)

To compile and execute "`JSBMLvisualizer`", you would need to do the following sequence of commands:

```
javac -classpath classpath JSBMLvisualizer.java
java -classpath classpath JSBMLvisualizer
```

Compiling and executing the example program.

In the example commands above, replace the placeholder text *`classpath`* with the actual Java class path for the JSBML libraries and its dependencies on your particular computer; we do not show an exact value here because it depends on where you have installed the JAR files for JSBML and the third-party libraries.

Figure 1.2 shows the example output when applying the program to an SBML test model. Each element in the model shows up as an item in the hierarchy displayed by the Java `JTree` object. In the working application, the user can click on the control boxes (i.e., the boxed "+" and "-" symbols next to the element names) to collapse or expand the views of the substructures of an SBML model.

We hasten to add that this simple program lacks many features that a proper application should possess. We kept this example purposefully as simple as possible so that it is easier to focus on the main point of the example (which is, how read to an SBML file). Perhaps the most important missing aspect is checking for and handling errors that may be encountered when trying to read and parse the file given as argument to the program. Not all SBML files are valid, owing to the unfortunate reality that *not all software tools in the world produce syntactically and semantically correct SBML.* The JSBML library is flexible and attempts to carry on in the face of problems, because it is the responsibility of the calling application to decide when and how problems should be handled. A realistic application should be coded defensively: it should be prepared for the possibility of receiving badly-formed input, check for any warnings and errors reported by `SBMLReader` when it attempts to read the SBML file, and deal with them appropriately. Elsewhere in this document, we provide examples of checking for errors.

Reading a file is nice, but what about writing an SBML file? That is the topic of the next example.

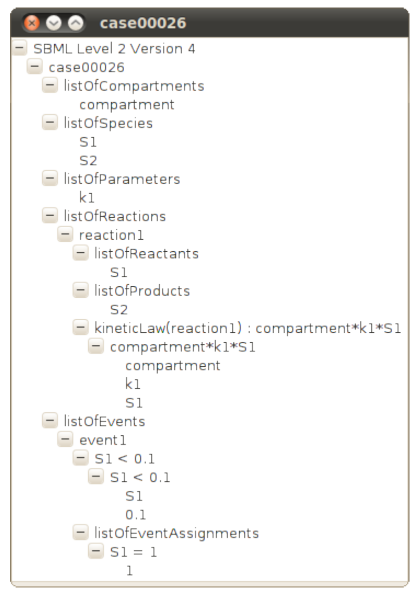

Figure 1.2: *Tree representation of the contents of the SBML test file "`case00026`.xml". In JSBML, the hierarchically structured `SBMLDocument` can be traversed recursively because all instances of `SBase`, the parent class, implement the interface `TreeNode`.*

1.2.2 Creating and writing an `SBMLDocument` object

Our next example, shown in Figure 1.3 on the following page, illustrates how to construct an in-memory representation of an SBML model and write it to a file. The program first creates an `SBMLDocument` object, then attaches a `Model` object to it, and then to the `Model` adds one `Compartment`, two `Species`, and one `Reaction` objects. To write the contents to a file named "`test.xml`", the program uses a static method on the JSBML class `SBMLWriter`.

This program illustrate also the preferred way to create JSBML object instances. The only constructor you should need to use is the constructor of the `SBMLDocument`, specifying the SBML level and version you want to use. Then each JSBML class should have some `createXXX` methods where 'XXX' is the children class name. For example, `model.createSpecies(String)`, `model.createReaction(String)` or `reaction.createReactant()`. These methods will guaranty that you are creating a proper representation of the SBML model.

```java
import java.beans.PropertyChangeEvent;
import javax.swing.tree.TreeNode;
import org.sbml.jsbml.*;
import org.sbml.jsbml.util.TreeNodeChangeListener;
import org.sbml.jsbml.util.TreeNodeRemovedEvent;

/** Creates an {@link SBMLDocument} and writes its contents to a file. **/
public class JSBMLexample implements TreeNodeChangeListener {

    public JSBMLexample() throws Exception {
        // Create a new SBMLDocument object, using SBML Level 2 Version 4.
        SBMLDocument doc = new SBMLDocument(2, 4);
        doc.addTreeNodeChangeListener(this);

        // Create a new SBML model, and add a compartment to it.
        Model model = doc.createModel("test_model");
        Compartment compartment = model.createCompartment("default");
        compartment.setSize(1d);

        // Create a model history object and add author information to it.
        History hist = model.getHistory(); // Will create the History, if it does not exist
        Creator creator = new Creator("Given␣Name", "Family␣Name", "Organisation", "My@EMail.com");
        hist.addCreator(creator);

        // Create some sample content in the SBML model.
        Species specOne = model.createSpecies("test_spec1", compartment);
        Species specTwo = model.createSpecies("test_spec2", compartment);
        Reaction sbReaction = model.createReaction("reaction_id");

        // Add a substrate (SBO:0000015) and product (SBO:0000011) to the reaction.
        SpeciesReference subs = sbReaction.createReactant(specOne);
        subs.setSBOTerm(15);
        SpeciesReference prod = sbReaction.createProduct(specTwo);
        prod.setSBOTerm(11);

        // For brevity, WE DO NOT PERFORM ERROR CHECKING, but you should,
        // using the method doc.checkConsistency() and then checking the error log.

        // Write the SBML document to a file.
        SBMLWriter.write(doc, "test.xml", "JSBMLexample", "1.0");
    }

    /** Main routine.  This does not take any arguments. */
    public static void main(String[] args) throws Exception {
        new JSBMLexample();
    }

    /* Methods for TreeNodeChangeListener, to respond to events from SBaseChangedListener. */
    public void nodeAdded(TreeNode sb) {
        System.out.println("[ADD]␣" + sb);
    }

  public void nodeRemoved(TreeNodeRemovedEvent evt) {
        System.out.println("[RMV]␣" + evt.getSource());
  }

    public void propertyChange(PropertyChangeEvent ev) {
        System.out.println("[CHG]␣" + ev);
    }

}
```

Figure 1.3: *Creating a new* `SBMLDocument` *object and writing its content into a file.*

1.3 More examples

Figure 3.3 on page 28 shows how to convert libSBML data structures into JSBML data objects. Figure 3.4 on page 29 demonstrates the implementation of CellDesigner's abstract class `PluginAction` and Figure 3.5 on page 30 gives a complete example for writing CellDesigner plugins with JSBML (See Section 3.6 on page 28 to learn more on JSBML Modules in general). Some more complex examples are available from the JSBML SourceForge repository, see Section 1.1.5 on page 7 to know how to obtain them.

2 Differences between JSBML and libSBML

Prior to the availability of JSBML, the most widely-used API library for SBML offering a Java interface has been libSBML [6]. As a result, many Java application developers working with SBML are already accustomed to the classes, methods and general approach provided by libSBML. This chapter discusses the main differences between these two libraries, and is aimed at current libSBML users who want to transition to using JSBML. But we also provide some programming examples and hints about how to use and work with JSBML. You will also have an overview about the type hierarchy and API of JSBML, not necessarily only important for people coming from libSBML.

2.1 Why are there differences?

In developing a pure Java Application Programming Interface (API) for working with SBML, our intention was not to simply re-implement the Java API already provided by libSBML [6]. We took the opportunity to rethink the API from the ground up to produce something more natural for Java programmers; moreover, we benefited from being able to take a fresh look at today's entire set of SBML specifications [14, 15, 16] and redesign, for example, JSBML's type hierarchy without the constraints of backwards compatibility that libSBML faces.

JSBML has also been developed as a library that provides more than only facilities for reading, manipulating, and writing SBML files and data streams. Although SBML only defines the structure of representations of biological processes in files and does not prescribe how its components should be stored *in computer memory*, many software developers nevertheless find it convenient to follow similar representational structures in their programs. With this in mind, we designed JSBML with the intention that it be directly usable as a flexible internal data structure for numerical computation, visualization, and 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 CellDesigner [10], a popular software application for modeling and simulation in systems biology. Finally, JSBML (like libSBML before it) hides some of the differences and inconsistencies in SBML that grew into the language over the years as it evolved from Level to Level and Version to Version; this makes it considerably easier for developers to support multiple Levels/Versions of SBML transparently.

Where possible, we maintained many of libSBML's naming conventions for methods and variables. Owing to the very different backgrounds of the two libraries, and the fact that libSBML is implemented in C and C++ , some differences are unavoidable. To help libSBML developers transition more easily to using JSBML, we provide a compatibility module that implements many libSBML methods as adaptors around the corresponding JSBML methods.

2.2 Differences between the class hierarchies

Wherever multiple SBML elements defined in at least one SBML Level/Version combination share attributes, JSBML provides a common superclass, or at least a common interface, that gathers methods for manipulating the shared properties. Consequently, JSBML's type hierarchy is richer than libSBML's (see Figure 2.1 to Figure 2.5 on pages 12–18).

Just as in libSBML, all SBML objects derived from SBML's `SBase` extend the JSBML abstract class `SBase`, but in JSBML, `SBase` is an interface rather than an object class. This allows more complex relations to be defined between derived data types. In contrast to libSBML, JSBML's `SBase` extends the interface `TreeNodeWithChangeSupport`, which in turn extends three other interfaces: `Cloneable`, `Serializable`, and `TreeNode` (Figure 2.2 on page 13). This brings with it various advantages. One is that, because all elements defined in JSBML override the `clone()` method from the class `java.lang.Object`, all JSBML elements can be deeply copied and are therefore *cloneable*. Further, extending the interface `Serializable` makes it possible for JSBML objects to be stored in binary form without having to write them explicitly to an SBML file. In this way, programs can easily load and save their in-memory objects or send data structures across a network connection without the need of additional file encoding and subsequent parsing.

The third interface extended by `SBase`, `TreeNode` is defined in Java's *Swing* package; however, `TreeNode` is actually independent of any graphical information. (We hasten to add that JSBML does *not* depend on any particular graphical user interface, and no other classes are initialized when loading `TreeNode` from Java Swing.) `TreeNode` defines recursive methods on hierarchically structured data types, such as iteration over all successors. This means

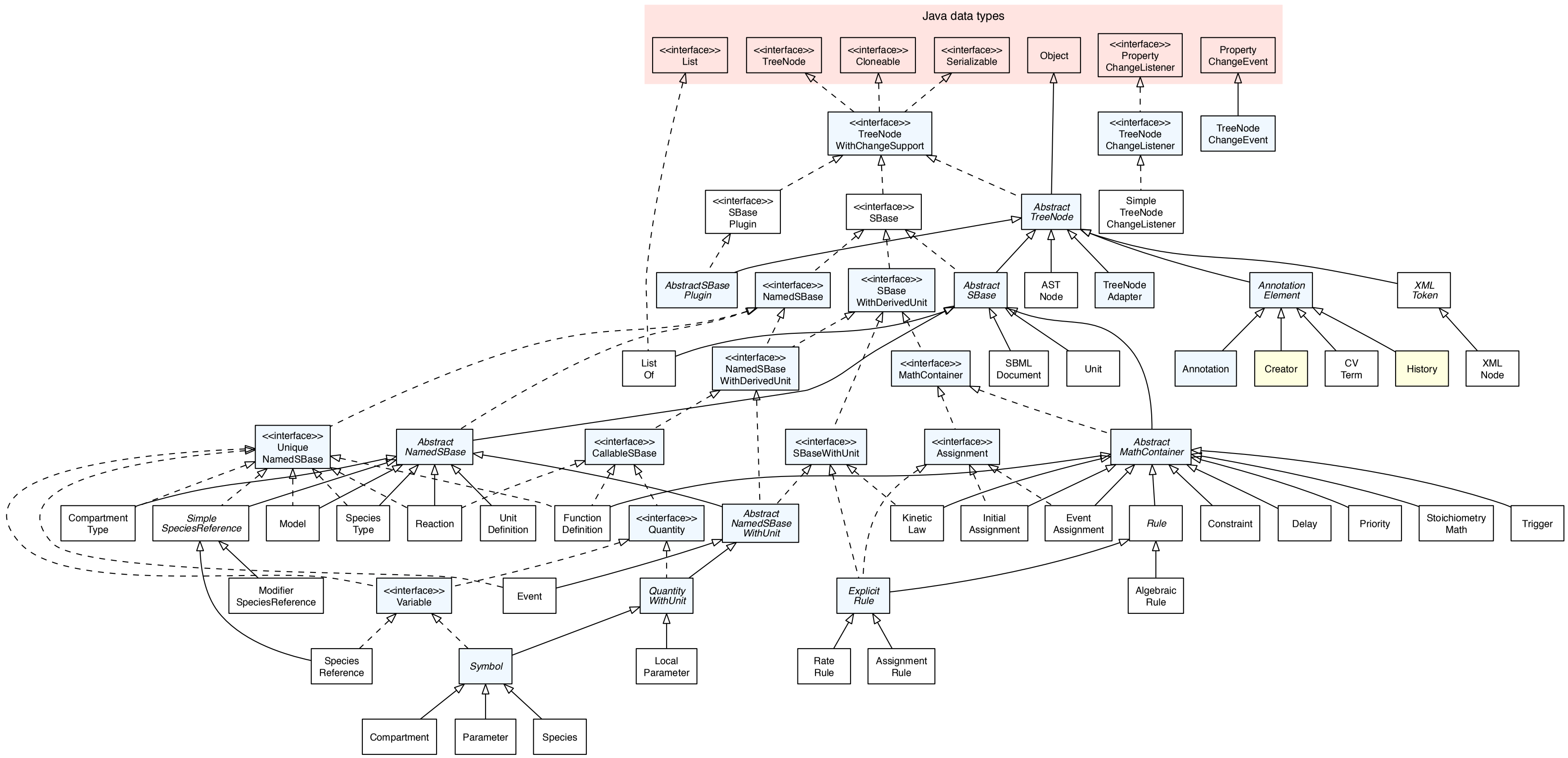

Figure 2.1: *The type hierarchy of the main SBML constructs in JSBML. The elements colored in yellow,* `Creator` *and* `History`*, correspond to* `ModelCreator` *and* `ModelHistory` *in libSBML. Elements colored in blue are additional, in most cases abstract, data types in JSBML that do not have corresponding features in libSBML. Many other classes and interfaces in this diagram have no equivalent in libSBML, but offer more powerful capabilities for Java programmers. By making* `SBase` *extend the interface* `TreeNodeWithChangeSupport` *(another class defined by JSBML), which in turn extends the Java interfaces* `Cloneable`*,* `Serializable`*, and* `TreeNode`*. All subclasses of* `SBase` *also provide the functionality of these classes and interfaces. In JSBML, even SBML components that are not defined by SBML as actually being derived from* `SBase` *are nevertheless derived from* `TreeNodeWithChangeSupport`*; thus, they (and all their subclasses) share many common methods and attributes, which makes them easy to use when wherever instances of* `TreeNode` *or operations across hierarchies of objects are needed.*

that, if a developer so desires, all instances of JSBML's `SBase` interface can be passed directly to the Java Swing class `JTree` for easy visualization. The program shown in Figure 1.1 on page 7 (and whose output is presented in Figure 1.2 on page 8) demonstrates the simple code needed to parse an SBML file and immediately display its contents in 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.

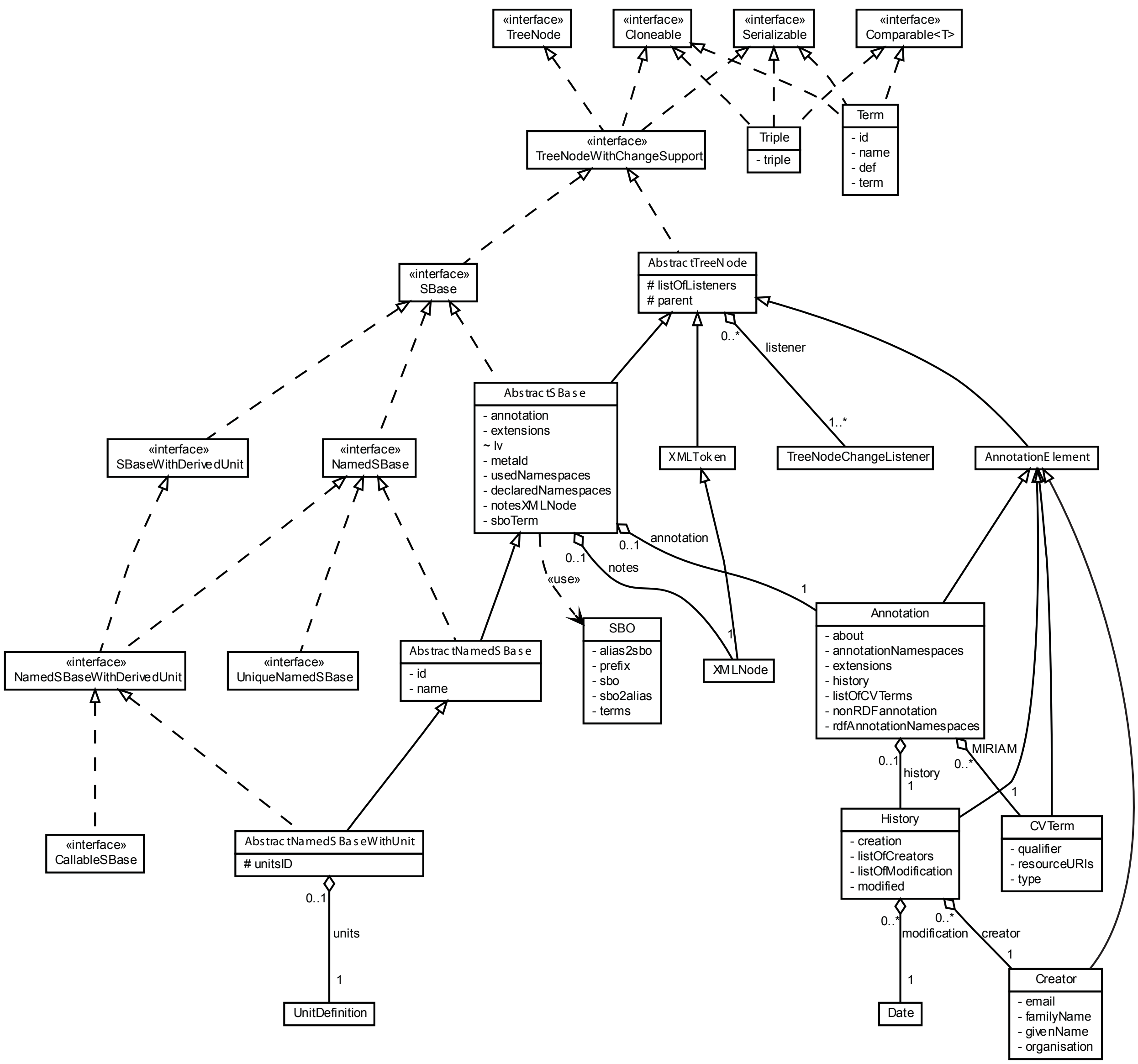

Figure 2.2: *The interface* `SBase`*. This figure shows the most important top-level data structures in JSBML, with a focus on the differences compared to libSBML. For the sake of clarity, we have omitted all the methods on the classes shown here. As can be seen in this diagram, all data types that represent 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` *implements facilities for parsing 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 [12].* `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`*).*

2.2.1 Common interface for hierarchical structures: `AbstractTreeNode`

When reading the SBML specifications [14, 15, 16], it quickly becomes apparent that an SBML model has a tree-shaped, hierarchical structure, with `SBase` being the superclass of nearly all other SBML components. In JSBML, other kinds of objects besides `SBase` are also organized hierarchically within an `SBMLDocument`. To unify the programming interfaces for all of these kinds of objects, 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).

As mentioned above, the interface `TreeNodeWithChangeSupport` defines a cloneable and serializable version of `TreeNode`. (See the diagram in Figure 2.2 on the previous page.) In addition, it also provides methods to notify dedicated `TreeNodeChangeListener` class objects about any changes within the data structure. Its abstract implementation, `AbstractTreeNode`, implements many of the methods inherited from `TreeNodeWithChangeSupport` and also maintains a list of change listeners (implemented as `TreeNodeChangeListener`s). Furthermore, this class contains a basic implementation of the methods `equals` and `hashCode`, which both make use of a recursive call over all descendants within the hierarchical SBML data structure. By basing the object definitions on this class, the implementation of all derived classes has become much simpler.

2.2.2 Common root of SBML components: `AbstractSBase`

With `SBase` being an interface rather than an object class, most SBML-related object classes in JSBML extend the abstract implementation `AbstractSBase`, as shown in Figure 2.2 on the preceding page. One of the features of this abstract class is that it tracks the SBML Level and Version of every concrete object implementing it. The need for tracking each object's Level+Version combination individually (a feature shared with libSBML) may seem odd at first. The need arises because a software system may need to work with more than one combination at a given time; it may also need to create individual SBML components before they are hooked into `SBMLDocument`, which again requires that individual objects know the SBML Level and Version for which they were created.

2.2.3 Interface for SBML components with identifiers: `NamedSBase`

Some classes of objects derived from `SBase` in SBML contain an identifier, colloquially often simply called the *id* after the attribute name used in the SBML specifications. JSBML gathers all elements that have SBML identifiers under the common interface `NamedSBase`. The class `AbstractNamedSBase`, which extends `AbstractSBase`, implements this interface. The interface `UniqueNamedSBase` is shared by those elements whose identifier must be unique within the model, i.e., for which no other element within the model may have the same identifier. The identifiers of all instances of `UniqueNamedSBase` of the same group, such as all `UnitDefinition` instances, 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 non-unique identifiers are `UnitDefinition`, whose identifiers exist in a separate namespace, and `LocalParameter`, whose identifiers may shadow the identifiers of global elements. (However, within a given list of `UnitDefinition` objects or list of `LocalParameter` objects, duplicate identifiers are not allowed.)

2.2.4 Interface for SBML components with units: `SBaseWithDerivedUnit`

Many SBML components represent some quantitative value with which a unit of measurement is associated. However, the numerical value of an SBML component does not necessarily have to be defined explicitly in the model; it may instead be determined by a mathematical formula contained in a given `SBase` object in the model. This implies that the unit associated with the value may be derivable. In JSBML, the interface `SBaseWithDerivedUnit` is used to represent all components that either explicitly or implicitly contain some unit. Figure 2.3 on the next page shows this part of JSBML's type hierarchy in more detail.

If the SBML component can be addressed with an identifier (which means that it has an `id` field in SBML), it will also implement the JSBML interface `NamedSBaseWithDerivedUnit`, and if it can appear within a formula (which in JSBML, is represented using `ASTNode`, discussed further below), the entity will further implement the interface `CallableSBase`, a special case of `NamedSBaseWithDerivedUnit`. When a component can be assigned a unit explicitly, in JSBML the `SBaseWithUnit` serves as its superclass. JSBML further defines the convenience class `AbstractNamedSBaseWithUnit`; it 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 SBML identifier.

In JSBML, the interface `Quantity` describes an element that is associated with a value, has at least a derived unit, and can be addressed using its unambiguous identifier. JSBML uses the abstract class `QuantityWithUnit` for a `Quantity` that explicitly declares its unit. If the corresponding SBML component includes a Boolean flag to indicate whether it is a constant or a variable, JSBML represents such a type using the interface `Variable`.

SBML variables that have a defined unit are represented as `Symbol` objects. (See Figure 2.3.) Thus, the SBML elements `Compartment`, `Parameter`, and `Species` are all special cases of `Symbol` in JSBML. The specification of SBML Level 3 introduced another type of `Variable`, which does not explicitly declare its unit: `SpeciesReference`. Level 3 also introduced `LocalParameter`, which is a `QuantityWithUnit` but not a `Variable` because it is always

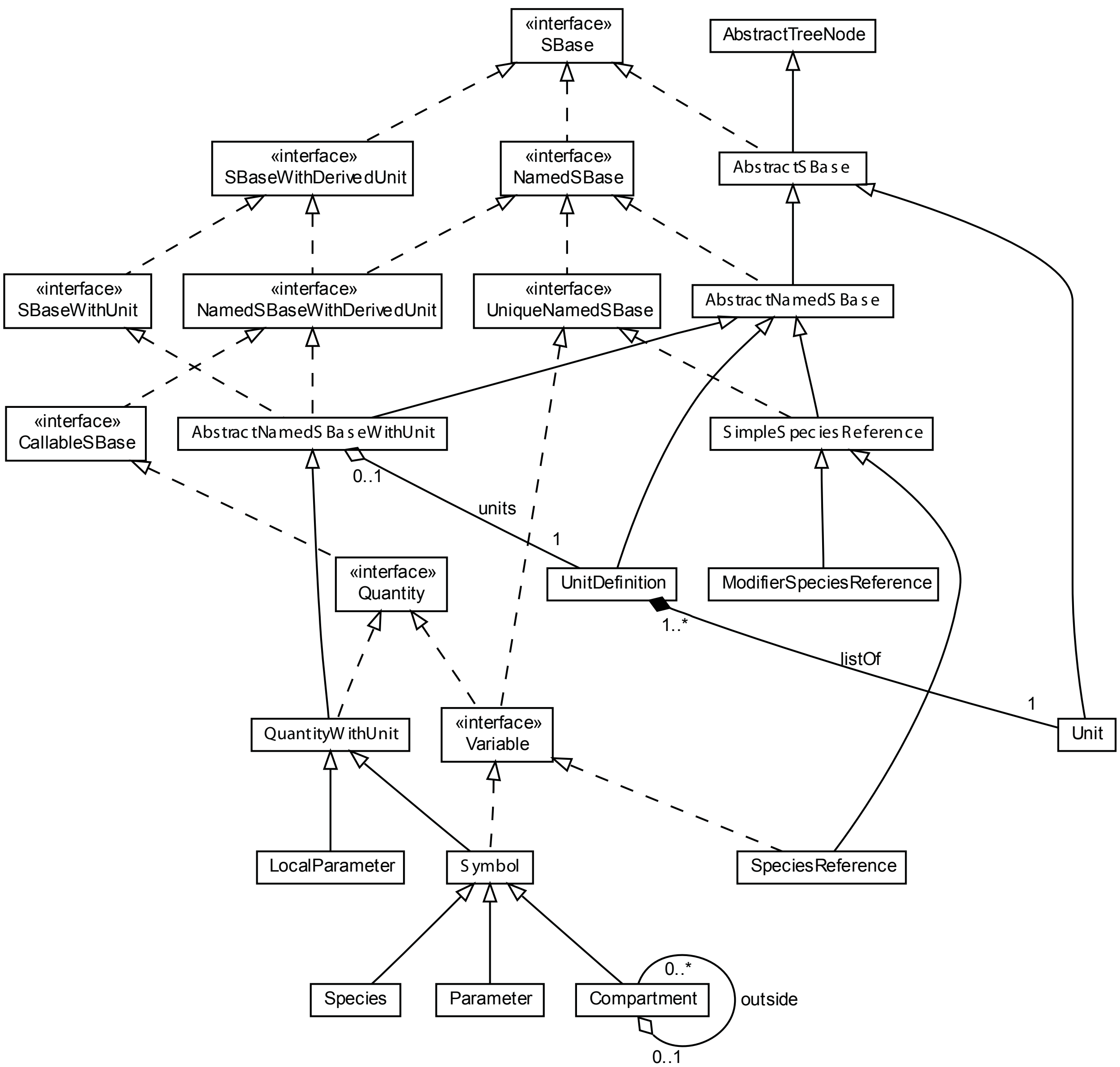

Figure 2.3: *Part of JSBML's type hierarchy focusing on the interface* `Variable`*. In JSBML, those components of a model that may change their value during a simulation are referred to as* variables*. The class* `Symbol` *serves as the abstract superclass for variables that have units of measurement associated with them. Instances of* `Parameter` *do not contain any additional fields. 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`*.*

constant. Section 2.2.6 explains the interfaces used for changing the values of `Variable`s.

2.2.5 Interface for SBML components containing a mathematical formula: `MathContainer`

The interface `MathContainer` in JSBML gathers all those elements that may contain mathematical expressions encoded in abstract syntax trees (i.e., instances of `ASTNode`). The abstract class `AbstractMathContainer` serves as actual superclass for the majority of the derived types. Figure 2.4 to Figure 2.5 on pages 17–18 give a better overview of how these data structures are organized and how they relate to each other and other ones in JSBML.

2.2.6 Interface for SBML components that may change the value of a variable: `Assignment`

JSBML provides a unified interface, `Assignment`, for all objects that may change the value of some variable in SBML. This interface uses the term *variable* for the element whose value can be changed depending on some mathematical expression that is also present in the `Assignment` (because the interface `Assignment` extends the interface `MathContainer`). Therefore, an `Assignment` contains methods such as `set-/getVariable(Variable v)` and also `isSetVariable()` as well as `unsetVariable()`.

In addition, JSBML also provides the methods `set-/getSymbol(String symbol)` in the `InitialAssignment` class to make it easier to switch from libSBML to JSBML. However, in JSBML, the preferred way is to apply the methods `setVariable()`, either with `String` or `Variable` instances as arguments. Figure 2.5 on page 18 shows the class hierarchy surrounding the `Assignment` interface in more detail.

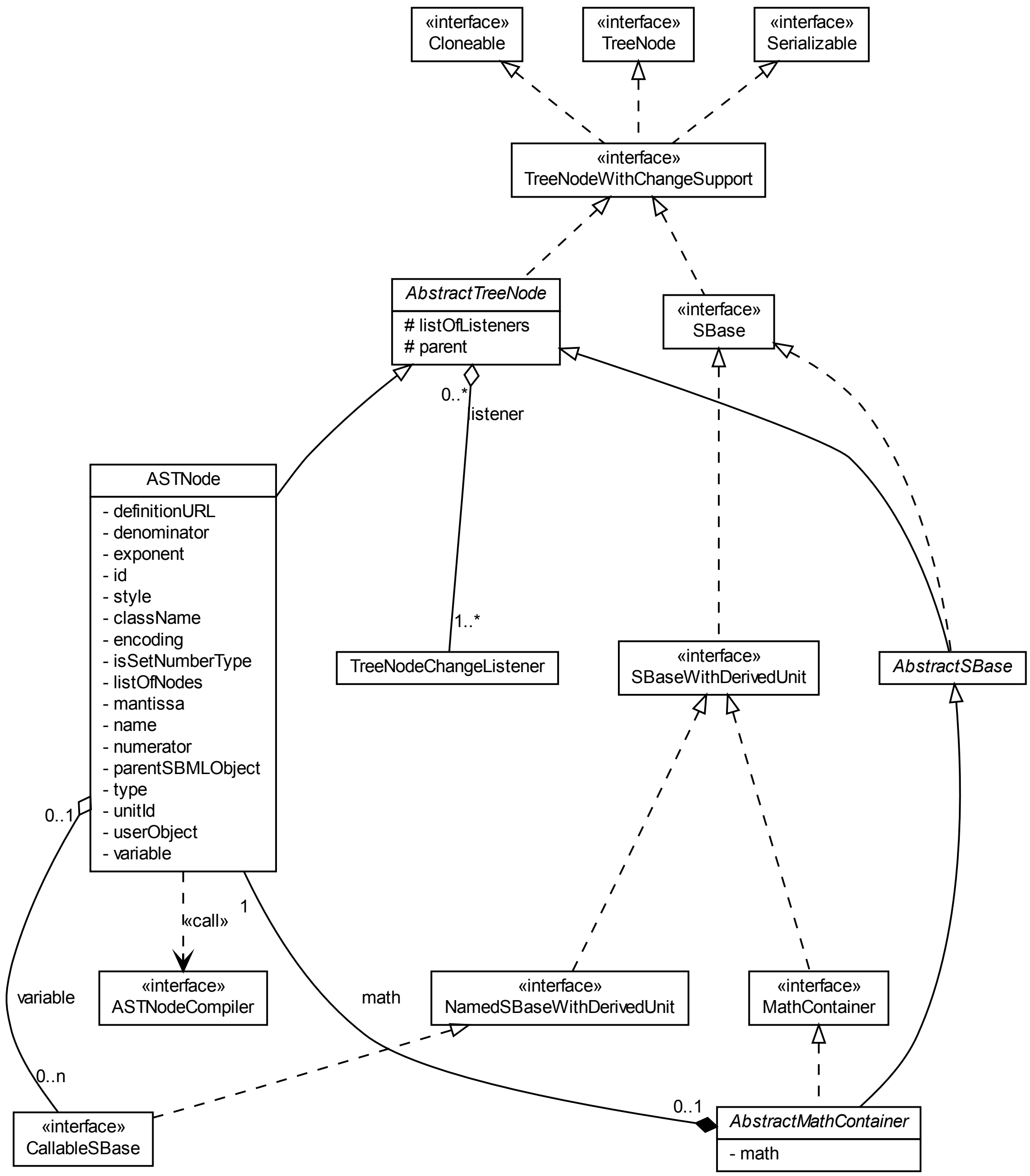

Figure 2.4: *Abstract syntax trees (ASTs). The class* `AbstractMathContainer` *serves as the superclass for several model components in JSBML. It provides methods to manipulate and access an instance of* `ASTNode`*, which can be converted to or read from text strings containing formulas in a C-like infix syntax. 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` *instead.*

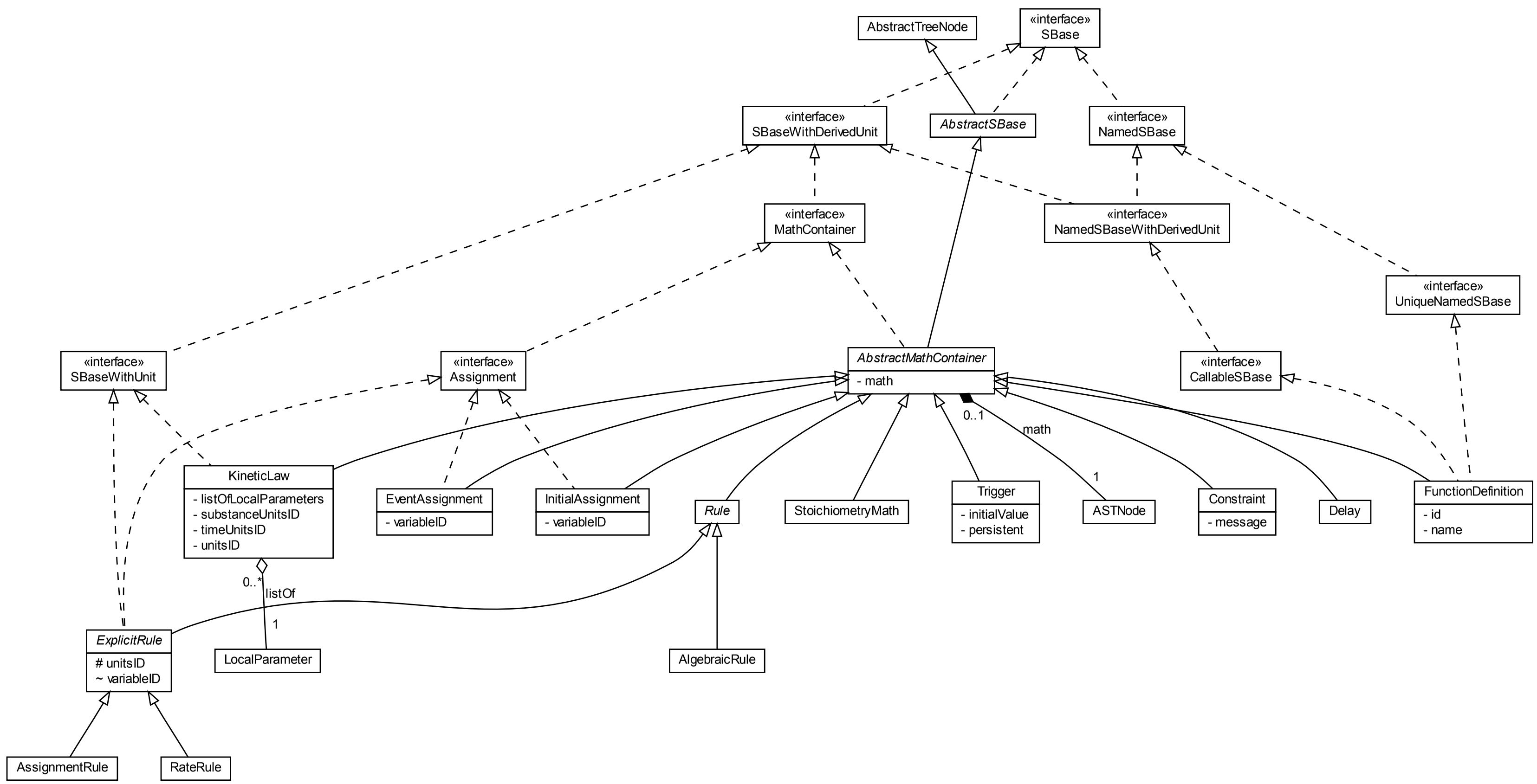

Figure 2.5: *Containers for mathematical expressions. The interface* `MathContainer`*, particularly its directly derived class* `AbstractMathContainer`*, constitutes the superclass for all elements that store and manipulate mathematical formulas in JSBML. The formulas themselves are stored in the form of* `ASTNode` *objects. These can be evaluated using an implementation of* `ASTNodeCompiler`*. Note that some classes that extend* `AbstractMathContainer` *do not contain any of their own additional fields or methods. This is the case for* `Delay`*,* `Priority`*,* `StoichiometryMath`*, and* `AlgebraicRule`*.*

2.3 Differences between the APIs of JSBML and libSBML

We have striven to make JSBML be closely compatible with libSBML. However, because of the differences in the programming languages used to implement these two libraries, some differences are simply impossible to overcome. In other cases, a direct translation from libSBML's C and C++ code to Java would be inelegant and too unnatural for Java users; this would conflict with another important goal of JSBML, which is is to provide a Java API whose classes and methods behave, and are organized like, those in other Java libraries.

In this section, we discuss the most important differences in the APIs of JSBML and libSBML. We also provide some examples of how the classes and methods in JSBML may be used.

2.3.1 *Level and Version* `ValuePair`

In libSBML, the Level and Version information is recorded as individual integers; by contrast, in JSBML it is stored in a generic object, `ValuePair`, stored within an `AbstractSBase` instance. The class `ValuePair` implements the Java interface `Comparable` and takes two values of any type that both also implement `Comparable`. Storing the information in this way allows users to check for a specific Level/Version combination more naturally, as the example in Figure 2.6 demonstrates. The method `getLevelAndVersion()` in `AbstractSBase` delivers an instance of `ValuePair` with the Level and Version combination for the respective element.

```
if (mySBase.getLevelAndVersion().compareTo(Integer.valueOf(2), Integer.valueOf(2)) < 0) {
  throw new IllegalArgumentException("Cannot create a " + mySBase.getElementName() +
      " with Level = " + getLevel() + " and Version = " + getVersion() + ".");
}
```

Figure 2.6: *Example program fragment showing how to check for a minimal expected SBML Level/Version combination.*

2.3.2 *Abstract syntax trees for mathematical formulas*

Both libSBML and JSBML define a class called `ASTNode` for in-memory storage and evaluation of abstract syntax trees (ASTs) that represent mathematical formulas. These can be parsed either from `String`s containing formulas in a C-like infix syntax, or from a MathML representation. JSBML's `ASTNode` class provides various methods to transform ASTs to other formats, for instance, `String`s in $\LaTeX$ syntax. Several static methods also make it easy to create syntax trees. The next example creates a new `ASTNode` which represents the sum of the two other nodes:

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

SBML specifies that mathematical formulas may contain references to the following kinds of components in a model: `Parameter`s, `LocalParameter`s, `FunctionDefinition`s, `Reaction`s, `Compartment`s, `Species`, and in SBML Level 3, `SpeciesReference`s. In JSBML, all of these object classes implement a common interface, `CallableSBase`, which extends the interface `NamedSBaseWithDerivedUnit`. This organization ensures that only identifiers of these particular SBML components can be set in instances of `ASTNode`.

Constructors and other methods for `CallableSBase`

JSBML provides useful constructors and methods to work with instances of `CallableSBase`. The `set` method changes the type of an `ASTNode` to `ASTNode.Type.NAME` and directly sets the name to the identifier of the given `CallableSBase`. The `get` method looks for the corresponding object in the `Model` and returns it. If no such object can be found or the type of the `ASTNode` is something different from `ASTNode.Type.NAME`, it throws an exception.

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

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

Getter and setter for `CallableSBase`.

The following are examples of methods for creating and manipulating complex ASTs. JSBML provides several static methods (such as `sum` shown above) that create small trees from objects in memory. Other methods, such as `plus`, `frac` and `pow`, change existing tree structures:

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

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

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

Some examples for convenience methods, some of them static methods, provided by JSBML for working with `ASTNode`s.

In contrast to the static `ASTNode.sum` function at the beginning of this section, the `frac` and the `pow` methods above take instances of `CallableSBase` as their arguments instead of `ASTNode` objects. Hence, the parent `MathContainer` must be passed to the methods in order to ensure that valid data structures are created. (In case of methods that take `ASTNode` objects as arguments, such as the static `ASTNode.sum`, the parent `MathContainer` can be taken from the first given node object.)

Finally, with the following `ASTNode` 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`.

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

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

The `ASTNodeCompiler` *class*

JSBML provides the interface `ASTNodeCompiler`; it allows users to create customized interpreters for the contents of mathematical formulas 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. As alluded to above, JSBML provides several implementations of this interface; for instance, `ASTNode` objects can be directly translated to C language-like `String`s, LaTeX, or MathML for further processing. In addition, the class `UnitsCompiler`, which JSBML uses to derive the unit of an abstract syntax tree, also implements this interface.

2.3.3 Compartments

In SBML Level 3 [14], the domain of the attribute `spatialDimensions` on `Compartment` is no longer $\{0, 1, 2, 3\}$, which can be represented with a `short` value in Java, and is instead a real-numbered value (i.e., a value in $\mathbb{R}$), which requires a `double` value in Java. 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; it is marked as a deprecated method.

2.3.4 Model history

Before SBML Level 3, only the `Model` object could have an associated history, that is, a description about the person(s) who build the model, including names, email addresses, modification and creation dates. In Level 3 of SBML, it is possible to annotate every construct with a history. This is reflected in JSBML by the name of the corresponding object—`History`—whereas it is named `ModelHistory` in libSBML. All instances of `SBase` in JSBML contain methods to access and manipulate its `History`. Also, JSBML does not have libSBML's classes `ModelCreator` and `ModelCreatorList` because JSBML gathers its `Creator` objects in a generic `List<Creator>` in the `History`.

2.3.5 Units and unit definitions

There are differences between libSBML and JSBML's interfaces for handling units. We describe them next.

The exponent attribute of units

In SBML Level 3 [14], the data type of the exponent attribute of a `Unit` object changed from `int` in previous Levels to `double` values. To provide a uniform interface no matter which Level of SBML is being dealt with, JSBML's method `getExponent()` only returns `double` values. In libSBML, `getExponent()` always returns `int`, and there is an additional method, `getExponentAsDouble()`, to handle the cases with `double` values. JSBML provides `getExponentAsDouble()` for compatibility with libSBML, but it is a redundant method in JSBML's case and therefore is marked as deprecated.

Predefined unit definitions

A model in JSBML always contains all predefined units defined by SBML. These can be accessed from an instance of `Model` by calling the method `getPredefinedUnit(String unit)`.

MIRIAM annotations [21] have been an integral part of SBML models since Level 2 Version 2. Recently, the Unit Ontology (UO) [11] has been included in the set of supported ontology and online resources of MIRIAM annotations [21]. 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 SBML Level/Version combination of the model supports MIRIAM annotations. In addition, 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 components whose value can be associated with a unit of measurement implement the interface `SBaseWithUnit`. This interface provides methods to access an object representing the unit. Currently, the interface is implemented by `AbstractNamedSBaseWithUnit`, `ExplicitRule`, and `KineticLaw`. Figure 2.1 on page 12 provides an overview about the relationships between these and other classes and interfaces.

`AbstractNamedSBaseWithUnit` is 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`. By contrast, the abstract class `QuantityWithUnit` serves as the superclass for `LocalParameter` and `Symbol`, which is then the superclass of `Compartment`, `Species`, and (global) `Parameter`. With `SBaseWithUnit` being a subclass of `SBaseWithDerivedUnit`, users can access the units of such an element in two different ways:

`getUnit()`: This method returns a `String` representation of the unit kind or the identifier of a 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, this method returns an empty `String`.

`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. If neither a user-defined nor a predefined unit is available, this method returns an empty `String`.

For convenience, JSBML also provides corresponding methods to the ones above for directly obtaining an instance of `UnitDefinition`. 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 for callers 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 provides the ability to set the substance unit and the time unit separately. 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 `UnitDefinition`s for both entries.

2.3.6 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. This has the advantage 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 at run-time and also more intuitive for Java programmers. If cloning is necessary, users should call the `clone()` method explicitly. Since all instances of `SBase`, and also `Annotation`, `ASTNode`, `CVTerm`, and `History`, extend `AbstractTreeNode` (which in turn implements the interface `Cloneable`—see Figure 2.1 on page 12), all these elements can be cloned naturally. 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 parent. 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 subelements will point to their correct parent element.

2. The list of `TreeNodeChangeListener` objects is used 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 they do not extends the `Cloneable` interface, we cannot clone them either and so the cloned object has no `TreeNodeChangeListener` attached to it. The user is responsible to add again some `TreeNodeChangeListener` on the cloned object if we want to be notify of any changes happening to it.

3. Since release 1.0, JSBML supports storing user objects in any object derived from `AbstractTreeNode`. These user objects are organized in a map data structure with object as key type, pointing to arbitrary user-defined objects. Note that generally no deep cloning of these user objects is possible, but JSBML keeps a pointer to these user objects in the cloned element.

2.3.7 Exceptions

In case of an error, JSBML methods will usually throw an exception, whereas libSBML methods return a numeric error code instead. The libSBML approach is rooted in the need to support C-like languages, while exception handling is more natural in Java. The JSBML approach of using exceptions helps programmers and users to avoid creating invalid SBML data structures already when dealing with these in memory.

As per usual Java practice, JSBML methods 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 of the kinds of exceptions that JSBML methods may throw include `ParseException`, which may be thrown if a given formula cannot be parsed properly into an `ASTNode` data structure, and `InvalidArgumentException`, which may be thrown if inappropriate values are passed to methods.

The following are some examples of situations that lead to exceptions:

- 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 `Event`s 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 map of identifiers within the `Model`. It is therefore not possible to assign duplicate identifiers in case of elements that implement the interface `UniqueNamedSBase`. For `UnitDefinition`s and `LocalParameter`s separate maps 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 maps 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 map 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 map 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. (See Section 3.3 on page 26.)
- In case that spatial dimension units of a `Species` are defined whose surrounding `Compartment` has zero dimensions or that has only substance units, JSBML also throws an exception.

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.

2.3.8 No interface `libSBMLConstants`

JSBML does not contain an equivalent to libSBML's `libSBMLConstants`. The reason is that in JSBML, constants are encoded in a more natural Java fashion, 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 enumeration `ASTNode.Type`. Instead of typing `libSBMLConstants.AST_TYPE_PLUS`, you would therefore type `ASTNode.Type.PLUS`.

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

2.3.9 No class `libSBML`

JSBML contains no class called `libSBML` simply because the library is called *JSBML*. In its place, there is a class named `JSBML`. This class provides some methods similar to the ones provided in libSBML's `libSBML`, such as `getJSBMLDottedVersion()` to obtain the current version of the JSBML library, which is 0.8 or 1.0-a* 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.

Here is an example of a method that is located on the relevant class. To convert between a `String` and a corresponding `Unit.Kind` you would use the following:

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

Converting a string to a unit kind in JSBML.

Analogous to the above, the `ASTNode` class provides a method to parse C-like infix formula `String`s according to the specification of SBML Level 1 [16] into an abstract syntax tree. Therefore, in contrast to the `libSBML` class, the class `JSBML` contains only a few methods.

2.3.10 No individual `ListOf*` classes, but a generic `ListOf`

JSBML does not have a specific `ListOf*` class for each type of `SBase` elements, which is unlike the case in libSBML. In JSBML, we use a generic implementation `ListOf<? extends SBase>` that enables the same class to be used for each of the different `ListOf*` classes defined in SBML while keeping a type-safe class.

To help developers work with `ListOf*` lists more conveniently, JSBML provides several methods that use the Java `Filter` interface to search and filter the lists. For example, to query an instance of a `ListOf*` list in JSBML for specific identifiers, or names, or both, you can apply the following filter:

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

Example of searching a list for an object with a particular identifier.

This will return the first element in the list with the given identifier. In SBML, a `ListOf*` list object usually must not contain multiple elements with the same identifier, so the element will usually be unique. The `firstHit` method stops after finding one element that satisfies the given `Filter`. The `ListOf<? extends SBase>` class also offers a `filter` method that takes a `Filter` object as argument and collects all elements accepted by that `Filter` object.

Various filters are already implemented in JSBML and made available for use in your programs, but you can easily add your own custom filter. You only need to implement the `Filter` interface defined in the JSBML package `org.sbml.jsbml.util.filters`. In that package, 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 [19, 20] in your list; similarly, the `CVTermFilter` helps you to identify `SBase` instances with a desired MIRIAM (Minimal Information Required In the Annotation of Models) annotation [21]. For instances of `ListOf<Species>`, you can apply the `BoundaryConditionFilter` to look for those species that operate on the boundary of the reaction system.

2.3.11 Use of 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 help users avoid creating models that refer to these elements.

JSBML also contains many methods added for greater compatibility with libSBML, but which programmers would probably not use unless they were transitioning existing software from libSBML. For instance, a method such as `getNumXyz()` is not considered to be very Java-like (but such methods are common for a C++ programming style). Usually, Java programmers would expect the method being called `getXyzCount()` instead. For cases like this, JSBML provides alternative methods and marks these methods that originate from libSBML as deprecated.

3 Additional features provided by JSBML

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

3.1 Change listeners

JSBML offers the ability to listen to change events in the life of an SBML document. To benefit from this facility, 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` instance has just been added to the `SBMLDocument` object. 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(TreeNodeRemoveEvent evt)`: This method notifies the listener that a `TreeNode` has just been removed, and therefore is no longer be a part of the `SBMLDocument`. The deleted element can be accessed using the `getSource()` method of the given event object. The entire `SBMLDocument` will not contain any pointers to this node anymore; however, the event object contains a pointer to its former parent that can be accessed by calling `getPreviousParent` on the event object. (This makes it possible to recognize where in the tree this node was located and even to revert the deletion of the node.)

`propertyChange(PropertyChangeEvent evt)`: This method provides detailed information about the change in a value within the `SBMLDocument`. The object passed to this method is a `TreeNodeChangeEvent`, which provides information about which `TreeNode` has been changed, which of its properties has been changed (as a `String` representation of the name of the property), the previous value, and the new value.

These methods can help software track what their `SBMLDocument` objects are doing at any given time. Furthermore, these features can be very useful in a graphical user interface, where, for example, the user might need to be asked if he or she really wants to delete some element or to approve changes before making these persistent. Another way this can be used is for writing 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 common Java API idioms and allow users also to make use of, e.g., `EventHandler`s to deal with changes in an SBML model.

As mentioned in Section 2.2.1 on page 14, all major objects implement the interface `TreeNode`, and its listeners are notified about all changes that occur in any implementing data structure. The use of `TreeNodeChangeListener`s allows a software application not only to keep track of changes in instances of `SBase`, but also changes inside of, e.g., `CVTerm` or `History`.

3.2 Determination of the variable in `AlgebraicRules`

JSBML's `OverdeterminationValidator` provides methods to determine if a given model is overdetermined; it uses the algorithm of Hopcroft and Karp [13].

`OverdeterminationValidator` simultaneously determines the free variable of each `AlgebraicRule` if possible. The class `AlgebraicRule` also provides a convenience method, `getDerivedVariable()`, to compute and return this free variable. However, we do not recommend calling this method except in limited circumstances, because each call invokes the matching algorithm—an operation that may be expensive for large models. JSBML does not store the results of applying the matching algorithm, because a change in the model's structure could also change these results and lead to an inconsistency. For models that contain multiple `AlgebraicRule` objects, it is instead more efficient to compute the matching once by invoking `OverdeterminationValidator`. Please see the documentation for `AlgebraicRule` for more details.

3.3 The find* methods

JSBML provides developers with a number of find* methods on a Model to help query for elements based on their identifiers or names. Software can search for various instances of SBase (for instance, CallableSBase, NamedSBase, and NamedSBaseWithDerivedUnit); using methods such as findLocalParameters, findQuantity, findQuantityWithUnit, findSymbol, and findVariable, software can also search for the corresponding model element. They enable software to work with SBML models more easily, without the need for explicit separate iteration loops for these common operations.

As of JSBML version 1.0, the find* methods do no longer query the model in an iterative way. Instead, the maps described in Section 2.3.7 on page 22 are used to access elements based on their id attribute. Similarly, the SBMLDocument can also directly access any of its subelements for a given metaid. Such a search can be performed in logarithmic runtime, i.e., $O(\log_2 n)$.

3.4 Other utility classes provided by JSBML

JSBML also provides additional utility classes besides those mentioned above. In the paragraphs below, we describe some of these classes in more detail. All of them are gathered in the package org.sbml.jsbml.util, where you can also find a growing number of additional helpful classes.

3.4.1 Pre-implemented mathematical functions and constants

The class org.sbml.jsbml.util.Maths contains several static methods for mathematical 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 JSBML 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 and the universal gas constant $R = 8.314472\ \mathrm{J{\cdot}mol^{-1}{\cdot}K^{-1}}$. In this way, the functions and constants implemented in class Maths complement standard Java with methods and numbers required by the SBML specifications [14, 15, 16].

3.4.2 Some tools for String manipulation

The JSBML 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.

3.5 Logging facilities

JSBML includes the logger provided by the log4j project [1]. Log4j allows us to use six levels of logging (TRACE, DEBUG, INFO, WARN, ERROR, and FATAL) but internally, JSBML mainly uses 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 find some documentation of which JSBML classes do some logging and at which levels.

If a software package using JSBML does not change the default settings, 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 end-users that something has gone wrong.

3.5.1 Changing the log4j configuration

If you want to modify the default log4j behavior, you will need to create a custom log4j configuration file. The best way to do this, as described in the log4j manual [1], is to use the environment variable log4j.configuration to point to the desired configuration file. One way to accomplish this is to add the following option to your java command (shown here for Unix/Linux and Mac OS X, but other operating systems have analogous facilities):

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

Command line option making log4j use a different configuration file. This syntax applies to Unix-like systems.

```
# All logging output sent to the console
log4j.rootCategory=INFO, console

# Console Display
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.layout=org.apache.log4j.PatternLayout

# Pattern to output the caller's file name and line number.
log4j.appender.console.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} - %5p (%F:%L) - %m%n

# Log the messages from the SimpleTreeNodeChangeListener at the DEBUG Level
# Allow to see all the changes that happened to the SBML elements
log4j.logger.org.sbml.jsbml.util=DEBUG
```

Figure 3.1: *A simple log4j configuration example. This sets the logging level of loggers in the* `org.sbml.jsbml.util` *to* `DEBUG`*, causing all changes to SBML elements to be logged.*

```
# Log4j configuration file.
# Logging is sent to a file and by email from the info level.
log4j.rootLogger=info, file, mail

# Email appender definition.
# It will send by email all messages from the error level.
log4j.appender.mail=org.apache.log4j.net.SMTPAppender

# The following set of properties defines how often email messages are send.
log4j.appender.mail.BufferSize=1
log4j.appender.mail.SMTPHost="smtp.myservername.xx"
log4j.appender.mail.From=fromemail@myservername.xx
log4j.appender.mail.To=toemail@myservername.xx
log4j.appender.mail.Subject=Log ...
log4j.appender.mail.threshold=error
log4j.appender.mail.layout=org.apache.log4j.PatternLayout
log4j.appender.mail.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

# File appender.
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.maxFileSize=100KB
log4j.appender.file.maxBackupIndex=5
log4j.appender.file.File=test.log
log4j.appender.file.threshold=info
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{ISO8601} %5p %c{1}:%L - %m%n
```

Figure 3.2: *Example of configuring log4j to send email messages for log events at the* `ERROR` *level.*

3.5.2 Some example configurations

Figure 3.1 gives a short example of a log4j configuration file. The effect of this particular configuration is to change the threshold of all loggers in the `org.sbml.jsbml.util` package to `DEBUG`, which results in all changes that happen to SBML elements to be logged. The class `SimpleTreeNodeChangeListener` will then output the old value and the new value whenever a setter methods is used on the SBML elements.

If your application is deployed in a server such as Tomcat [3], it may be useful to define a log4j "appender" that will send some messages by email. Figure 3.2 gives an example of doing this. It configures log4j so that any messages at the `ERROR` level are send by mail. All the messages are also written to a rolling log file.

Note that using log4j's alternative, XML-based approach to defining configurations instead of a properties file, you can configure log4j to direct some log messages to one appender and others to an other appender, using the `LevelRange` filter. In this way, it would be possible to cause `DEBUG` messages to be written to a separate file.

Finally, be warned that when you enable the debug level on some loggers, they may produce copious output. You may wish to investigate some of the freely-available software for log viewing [27] to work with the log files.

3.6 JSBML modules

JSBML modules extend the functionality of JSBML and are provided as separate libraries (packaged as JAR files). With the help of the current JSBML modules, JSBML can be used, for example, as a communication layer between your application and libSBML [6] or between your program and the program known as CellDesigner [10]. In addition, JSBML offers a compatibility module that helps provide the same package structure and API as libSBML's Java language interface. In the rest of this section, we provide examples of how to use these modules.

3.6.1 The `libSBMLio` module: using libSBML for parsing SBML into JSBML data structures

The capabilities of the SBML validator constitutes one of the major strengths of libSBML [6] in comparison to JSBML, which does not yet contain a standalone validator for SBML, but makes use of the online validation provided at `http://sbml.org`. However, if the platform-dependency of libSBML does not hamper your application, or you want to switch slowly from libSBML to JSBML, you may still read and write SBML models using libSBML in conjunction with JSBML.

To facilitate this, the module `libSBMLio` provides classes `LibSBMLReader` and `LibSBMLWriter`. Figure 3.3 provides a short code example illustrating the use of `LibSBMLReader`. The program displays the content of an SBML file in a `JTree`, similar to what is shown in Figure 1.2 on page 8.

As of version 1.0 of JSBML, the `libSBMLio` module also contains specialized `TreeNodeChangeListener`s that synchronizes any change in the JSBML data structure with corresponding libSBML data structures.

```java
/** @param args the path to a valid SBML file. */
public static void main(String[] args) {
    try {
        // Load libSBML:
        System.loadLibrary("sbmlj");
        // Extra check to be sure we have access to libSBML:
        Class.forName("org.sbml.libsbml.libsbml");

        // Read SBML file using libSBML and convert it to JSBML:
        LibSBMLReader reader = new LibSBMLReader();
        SBMLDocument doc = reader.convertSBMLDocument(args[0]);

        // Run some application:
        new JSBMLvisualizer(doc);
    } catch (Throwable e) {
        e.printStackTrace();
    }
}
```

Figure 3.3: *A simple example showing how to convert libSBML data structures into JSBML data objects. To run this example, you need libSBML installed on your system. You may need to set environment variables, e.g., the* `LD_LIBRARY_PATH` *under Linux, to values appropriate for your system. For details, please see the libSBML documentation [22].*

3.6.2 The `CellDesigner` module: turning 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 [10]. To support this, it is necessary to extend two classes that are defined in CellDesigner's plugin API. Figure 3.4 to Figure 3.5 on pages 29–30 show a simple example of (1) how to pass a model data structure in a CellDesigner plugin to the translator in JSBML, and (2) creating a plugin for CellDesigner which displays the SBML data structure in a tree, like the example in Figure 1.2 on page 8.

The examples in Figure 3.4 to Figure 3.5 on pages 29–30 only show how to translate a plugin's 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 data structure. Note that the program in Figure 3.5 on page 30 is only completed by implementing the methods from the superclass, `CellDesignerPlugin`. In this example it is sufficient to leave the implementation empty.

```java
package org.sbml.jsbml.cdplugin;

import java.awt.event.ActionEvent;
import javax.swing.JMenuItem;
import jp.sbi.celldesigner.plugin.PluginAction;

/** A simple implementation of an action for a CellDesigner plug-in,
 *  which invokes the actual plug-in program. */
public class SimpleCellDesignerPluginAction extends PluginAction {

  /** Memorizes a pointer to the actual plug-in program. */
  private SimpleCellDesignerPlugin plugin;

  /** Constructor memorizes the plug-in data structure. */
  public SimpleCellDesignerPluginAction(SimpleCellDesignerPlugin plugin) {
    this.plugin = plugin;
  }

  /** Executes an action if the given commant occurs. */
  public void myActionPerformed(ActionEvent ae) {
    if (ae.getSource() instanceof JMenuItem) {
      String itemText = ((JMenuItem) ae.getSource()).getText();
      if (itemText.equals(SimpleCellDesignerPlugin.ACTION)) {
        plugin.startPlugin();
      }
    } else {
      System.err.printf("Unsupported source of action %s\n", ae
          .getSource().getClass().getName());
    }
  }
}
```

Figure 3.4: *A simple implementation of CellDesigner's abstract class* `PluginAction`.

As of JSBML version 1.0, this module also contains a specialized implementation of the `TreeNodeChangeListener` interface for synchronization of changes in JSBML's data structures with CellDesigner.

3.6.3 The `libSBMLcompat` module: a JSBML compatibility module for libSBML

The goal of the libSBML compatibility module in JSBML is to provide the same package structure as libSBML's Java bindings, and provide identically-named classes and APIs. 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.

3.6.4 The `android` module: a compatibility module for Android systems

The JSBML *Android* 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.

```java
package org.sbml.jsbml.cdplugin;

import javax.swing.*;
import jp.sbi.celldesigner.plugin.*;
import org.sbml.jsbml.*;
import org.sbml.jsbml.gui.*;

/** A very simple implementation of a plugin for CellDesigner. */
public class SimpleCellDesignerPlugin extends CellDesignerPlugin {

    public static final String ACTION = "Display␣full␣model␣tree";
    public static final String APPLICATION_NAME = "Simple␣Plugin";

    /** Creates a new CellDesigner plugin with an entry in the menu bar. */
    public SimpleCellDesignerPlugin() {
        super();
        try {
            System.out.printf("\n\nLoading␣%s\n\n", APPLICATION_NAME);
            SimpleCellDesignerPluginAction action = new SimpleCellDesignerPluginAction(this);
            PluginMenu menu = new PluginMenu(APPLICATION_NAME);
            PluginMenuItem menuItem = new PluginMenuItem(ACTION, action);
            menuItem.setName("some_id");
            menu.add(menuItem);
            addCellDesignerPluginMenu(menu);
        } catch (Exception exc) {
            exc.printStackTrace();
        }
    }

    /** This method is to be called by our CellDesignerPluginAction. */
    public void startPlugin() {
        PluginSBMLReader reader
            = new PluginSBMLReader(getSelectedModel(), SBO.getDefaultPossibleEnzymes());

        // In CellDesigner, the SBMLDocument object is not accessible, so we must create a new one
        // after obtaining the model from the reader.

        Model model = reader.getModel();
        SBMLDocument doc = new SBMLDocument(model.getLevel(), model.getVersion());
        doc.setModel(model);
        new JSBMLvisualizer(doc);
    }

    // Include also methods from superclass, not needed in this example.
    public void addPluginMenu() { }
    public void modelClosed(PluginSBase psb) { }
    public void modelOpened(PluginSBase psb) { }
    public void modelSelectChanged(PluginSBase psb) { }
    public void SBaseAdded(PluginSBase psb) { }
    public void SBaseChanged(PluginSBase psb) { }
    public void SBaseDeleted(PluginSBase psb) { }
}
```

Figure 3.5: *A simple example for a CellDesigner plugin using JSBML as a communication layer.*

4 Implementing extensions in JSBML

In this chapter, we describe how to get started with writing an extension for JSBML to support an SBML Level 3 package. We use a concrete (though artificial) example to illustrate various points. This example extension is named, very cleverly, *Example*, and while it does not actually do anything significant, we hope it will help make the explanations more understandable. This chapter apply to JSBML version 1.0 only. The 0.8 branch does not support any extension packages.

4.1 Organizing the source code

In the JSBML SVN repository, all extensions are found in the subdirectory named `extensions` inside the `trunk` directory. (The process for checking out a local copy of the repository is described in Section 1.1.4 on page 5.) Each extension is named after the corresponding SBML short name for the SBML Level 3 package; for example, `fbc` for the Flux Balance Constraints package, `layout` for the Layout package, and so on. The source directories for the extensions follow some basic conventions for their organization and contents.

When creating a new extension for JSBML, please follow the conventions used in the existing extension directories. These conventions are illustrated in Figure 4.1. There should be a build script in a file named "`build.xml`" for use with Ant [2], and several subdirectories. The `doc` subdirectory should contain documentation about the extension, preferably with a subdirectory of its own, `img`, containing at least a UML diagram of the type hierarchy of the package. This can be in the form of, for instance, a Graphviz [5] file `type_hierarchy.dot`, so that the diagram can be generated in different image formats. The extension directory should also contain a `lib` subdirectory where any package-specific, third-party libraries are located; a `resources` subdirectory for any non-source files that may be required by the extension code; an `src` subdirectory for the Java source code comprising the extension; and finally, a `test` subdirectory containing tests for the extension code, preferably in JUnit [4] format.

```
/
├── build.xml
├── doc/
│   └── img/
├── lib/
├── resources/
├── src/
│   └── org/
│       └── sbml/
│           └── jsbml/
│               ├── ext/
│               │   └── NAME/
│               └── xml/
│                   └── parsers/
└── test/
```

Figure 4.1: *Typical structure of the source directory for a JSBML extension. The root of the tree shown here is the* `extensions/NAME` *subdirectory, which is located within the* `trunk` *subdirectory of the JSBML SVN repository.*

Note the structure of the `src` subdirectory. A JSBML extension needs to define at least two packages: `org.sbml.jsbml.ext.NAME`, for the data structures and code for defining and manipulating the SBML components specified by the extension, and `org.sbml.jsbml.xml.parsers`, for the parsing code for reading and writing SBML files with the extension constructs. As per Java conventions, these source subdirectories are organized hierarchically based on the package components, which leads to the nested structure shown in Figure 4.1.

4.2 Creating the object hierarchy

A JSBML extension may need to do different things depending on the details of the SBML Level 3 package that it implements. In this section, we discuss various common actions and how they can be written in JSBML.

4.2.1 Introducing new components and extending others

Most SBML Level 3 packages extend existing SBML core components or define entirely new components. A common need for packages is to extend the SBML `Model` object, so we begin by explaining how this can be achieved.

Figure 4.2 on the following page shows the beginnings of the definition for a class named `ExampleModel` that extends the plain SBML `Model`. Technically, an extension really only needs to implement the `SBasePlugin` interface, but because the abstract class `AbstractSBasePlugin` implements important and useful methods, it is generally preferable to extend that one instead. In this example, our constructor for `ExampleModel` 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` object that is being extended, and it will store it in an attribute named `extendedSBase`. Our example

```java
public class ExampleModel extends AbstractSBasePlugin {

  // Basic constructor.
  public ExampleModel(Model model) {
    super(model);
  }

  // Returns the model.
  public Model getModel() {
    return (Model) getExtendedSBase();
  }
}
```

Figure 4.2: *How to extending* `AbstractSBasePlugin` *to create an extended* `Model`*.*

`ExampleModel` class also adds a method, `getModel()`, to retrieve the extended model object.

In most cases, extensions will also introduce new components that have no counterpart in the SBML core. We illustrate this here by creating a component called `Foo` with three attributes: `id`, `name`, and an integer-valued attribute, `bar`. We assume that in the (hypothetical) package specification for *Example*, `Foo` is derived from `SBase`; let us also assume that *Example* provides the ability to attach a list of `Foo` objects to an extended version of `Model`. We show in Section 4.2.2 on page 34 how to create the list structure; here, we focus on the definition of `Foo`. We define the class `Foo` by extending `AbstractSBasePlugin`, and add methods for working with the attributes. In Figure 4.3, we list the code so far, focusing on just one of the attributes, *bar*.

```java
// Use Integer, so we can denote unset values as null public Integer bar;

public int getBar() {
  if (isSetBar()) {
    return bar.intValue();
  }
  // This is necessary because we cannot return null here.
  throw new PropertyUndefinedError(ExampleConstant.bar, this);
}

public boolean isSetBar() {
  return this.bar != null;
}

public void setBar(int value) {
  Integer oldBar = this.bar;
  this.bar = bar;
  firePropertyChange(ExampleConstant.bar, oldBar, this.bar);
}

public boolean unsetBar() {
  if (isSetBar()) {
    Integer oldBar = this.bar;
    this.bar = null;
    firePropertyChange(ExampleConstant.bar, oldBar, this.bar);
    return true;
  }
  return false;
}
```

Figure 4.3: *Implementation of the five necessary methods that should be created for every attribute on class* `Foo`*. Note: if attribute* bar *had been a boolean-valued attribute, we would also provide the method* `isBar()`*, whose implementation would delegate to* `getBar()`*.*

A few points about the code of Figure 4.3 are worth mentioning. The identifiers on SBML components are often required to be unique; for many components, the scope of uniqueness is the entire set of main SBML components (e.g., `Species`, `Compartment`s, etc.), but some have uniqueness requirements that are limited to some subset of entities (e.g., unit identifiers). For the purposes of this example, we assume that the identifiers of *Foo* objects in a model must be unique across all identifiers in the model. All entities that have such uniqueness constraints should implement the JSBML interface `UniqueNamedSBase`; in our example, this is taken care of by the abstract superclasses, so nothing needs to be done explicitly here.

The code in Figure 4.3 on the preceding page also illustrates another point, the need call to `firePropertyChange()` in set and unset methods. This is needed in order to ensure that all listeners are notified about changes to the objects. Finally, note that in cases that the return type is a Java base type, such as `int` or `boolean`, but the corresponding internal element (e.g., `Integer` or `Boolean`) is set to `null`, the program must throw a `PropertyUndefineError` in the get method to prevent incorrect results (see line 8).

The last basic matter that needs to be addressed is the definition of appropriate class constructors for our class `Foo`. The minimum we need to define is a constructor that takes no arguments. Even though some or all of the attributes of a class may be mandatory, default constructors that take no arguments still need to be defined in JSBML. 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.

Beyond this, the precise combination of constructor arguments defined for a class is a design issue that must be decided for each class individually. Attempting to define a separate constructor for every possible combination of arguments can lead to a combinatorial explosion, resulting in complex class definitions, confusing code, and

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

Figure 4.4: *Constructors for class* `Foo`*. Note the code testing for the SBML Level and Version, on lines 21–23; since this extension implements a hypothetical package for SBML Level 3, the code here rejects anything before Level 3 Version 1 by throwing the JSBML exception* `LevelVersionError`*.*

excessive maintenance costs, so it is better to decide which combinations of arguments are the most common and focus on them. In Figure 4.4 on the previous page, we show a recommended selection of constructors. They include a constructor that takes an identifier only, another that takes SBML Level and Version values only, and another that takes all arguments together. 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.

4.2.2 `ListOfs`

Our hypothetical *Example* extension adds no new attributes to the extended `Model` itself, but it does introduce the ability to have a list of `Foo` objects as a child of `Model`. In JSBML, this will be implemented using Java generics and the class `ListOf`, such that the type of the list will be `ListOf<Foo>`. (Unlike in libSBML, there will not be an actual separate `ListOfFoo` class.) In Figure 4.5, we show the basic implementation of the methods that would be added to `Model` to handle `ListOf<Foo>`: `isSetListOfFoos()`, `getListOfFoos()`, `setListOfFoos(ListOf<Foo>)`, and `unsetListOfFoos()`.

Typically, when adding and removing Foo objects to the `Model`, direct access to the actual `ListOf` object is not necessary. To add and remove `Foo` objects from a given SBML model, it is more convenient to have methods to add and remove on `Foo` object at a time. We show such methods in Figure 4.6 on the next page. The methods also do some additional consistency checking as part of their work.

To let a `ListOfFoo` appear as a child of the standard `Model`, some important methods for `TreeNode` need to be implemented (see Figure 4.7 on the following page). Method `getAllowsChildren()` should return `true` in this case, since this extension allows children. The child count and the indices of the children is a bit more complicated, because they vary with the number of non-empty `ListOf`s. So, for every non-empty `ListOf` child of our model extension, we increase the counter by one. (Note also that if callers access list entries by index number, they will need to take into account the possibility that a given object's index may shift.)

```java
public boolean isSetListOfFoos() {
  return (listOfFoos != null) && !listOfFoos.isEmpty();
}

public ListOf<Foo> getListOfFoos() {
  if (!isSetListOfFoos()) {
    Model m = getModel();
    listOfFoos = new ListOf<Foo>(m.getLevel(), m.getVersion());
    listOfFoos.addNamespace(ExampleConstants.namespaceURI);
    listOfFoos.setSBaseListType(ListOf.Type.other);
    m.registerChild(listOfFoos);
  }
  return this.listOfFoos;
}

public void setListOfFoos(ListOf<Foo> listOfFoos) {
  unsetListOfFoos();
  this.listOfFoos = listOfFoos;
  getModel().registerChild(this.listOfFoos);
}

public boolean unsetListOfFoos() {
  if (isSetListOfFoos()) {
    ListOf<Foos> oldFoos = this.listOfFoos;
    this.listOfFoos = null;
    oldFoos.fireNodeRemovedEvent();
    return true;
  }
  return false;
}
```

Figure 4.5: *Implementation of the methods* `isSetListOfFoos()`, `getListOfFoos()`, *and* `setListOfFoos()`.

```java
public boolean addFoo(Foo foo) {
    return getListOfFoos().add(foo);
}

public boolean removeFoo(Foo foo) {
  return isSetListOfFoos() ? getListOfFoos().remove(foo) : false;
}

public boolean removeFoo(int i) {
  if (!isSetListOfFoos()) {
    throw new IndexOutOfBoundsException(Integer.toString(i));
  }
  return listOfFoos.remove(i);
}

// If the object class has an id, one should also add the following:
public boolean removeFoo(String id) {
  return getListOfFoos().removeFirst(new NameFilter(id));
}
```

Figure 4.6: *Implementation of* `ListOf` *methods* `addFoo(Foo foo)`, `removeFoo(Foo foo)`, `removeFoo(int i)`.

4.2.3 Methods for creating new objects

Since 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 Figure 4.8 on the next page). These create methods should be part of the model to which the `Foo` instance is to be added, in this case `ExampleModel`.

```java
public boolean getAllowsChildren() {
  return true;
}

public int getChildCount() {
  int count = 0;
  if (isSetListOfFoos()) {
    count++;
  }
  return count;  // Same for each additional ListOf* or other subelement in this package.
}

public SBase getChildAt(int childIndex) {
  if (childIndex < 0) {
    throw new IndexOutOfBoundsException(childIndex + "␣<␣0");
  }

  // Important: there must be an index shift according to the number of child elements in the superclass.

  int pos = 0;
  if (isSetListOfFoos()) {
    if (pos == childIndex) {
      return getListOfFoos();
    }
    pos++;
  }
  // Same for each additional ListOf* or other subelements in this package.
  throw new IndexOutOfBoundsException(MessageFormat.format(
    "Index␣{0,number,integer}␣>=␣{1,number,integer}", childIndex, +((int) Math.min(pos, 0))));
}
```

Figure 4.7: *Methods which need to be implemented to make the children available in the extended model.*

```java
public class ExampleModel extends AbstractSBasePlugin {

  // ...

  // only, if ID is not mandatory in Foo
  public Foo createFoo() {
    return createFoo(null);
  }

  public Foo createFoo(String id) {
    Foo foo = new Foo(id, getLevel(), getVersion());
    addFoo(foo);
    return foo;
  }

  public Foo createFoo(String id, int bar) {
    Foo foo = createFoo(id);
    foo.setBar(bar);
    return foo;
  }
}
```

Figure 4.8: *Convenience method to create* `Foo` *objects.*

4.2.4 The methods `equals`*,* `hashCode`*, and* `clone`

There are three more methods which should be implemented in an extension class: `equals`, `hashCode` and `clone`. This is not different than when implementing any other Java class, but because mistakes here can lead to bugs that are very hard to find, we describe the process in detail.

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: allow a quick check if two objects might be equal, and provide 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.

Figure 4.9 and Figure 4.10 on the following page are examples of 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 `ListOf`s, because they are automatically checked in the `AbstractTreeNode` class, the super class of `AbstractSBase`.

Figure 4.11 and Figure 4.12 on the next page illustrates implementations of `clone()` methods. To clone an object, the call to the `clone()` method is delegated to a constructor of that class that takes an instance of itself as argument. There, all the elements of the class must be copied, which may require recursive cloning.

Although JSBML defines all SBML elements in a hierarchical data structure, it is still not possible to recursively clone child elements within the constructor of some abstract superclasses because these can be of various types and they cannot simply be organized as a list of children.

```java
@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()) {
      // Note: strictly speaking, this is only possible if the return type is some Object. For simple data types,
      // such as boolean, int, short, etc., the corresponding wrapper classes should be called instead
      // or a direct comparison should be performed.
      equals &= (foo.getBar().equals(getBar()));
    }
    // ...
    // further attributes
  }
  return equals;
}
```

Figure 4.9: *Example of the* `equals` *method.*

```java
@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;
}
```

Figure 4.10: *Example of the* `hashCode` *method. The variable* `prime` *should be a large prime number to prevent collisions.*

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

public ExampleModel(ExampleModel model) {
  super(model);  // This step is critical!
  // Deep cloning of all elements:
  if (model.isSetListOfFoos()) {
    listOfFoos = model.listOfFoos.clone();
  }
}
```

Figure 4.11: *Example of the* `clone` *method for the* `ExampleModel` *class.*

```java
@Override public Foo clone() {
  return new Foo(this);
}

public Foo(Foo f) {
  super(f);  // This step is critical!

  // Integer objects are immutable, so it is sufficient to copy the pointer
  bar = f.bar;
}
```

Figure 4.12: *Example of the* `clone` *method for the* `Foo` *class.*

4.3 Implementing the reader and writer for an SBML package

One last thing is missing to be able to properly read and write SBML files using the new extension: a `ReadingParser` and a `WritingParser`. An easy way to do that is to extend the `AbstractReaderWriter` that extends both interfaces and implement some of the required methods in a generic way. To implement the parser, in this case the `ExampleParser`, one should start with two members and two simple methods, as shown in Figure 4.13. As you can see from this code snippet, an additional class `ExampleConstants` and an enum `ExampleListType` are used. The class `ExampleConstants` is used to keep track of all the static `String`s used in the extension so that we are sure that the same value is used everywhere. The enum `ExampleListType` can be used to keep track of which `ListOf` we are in while reading an XML file.

```java
public class ExampleParser extends AbstractReaderWriter {

  /**
   * The logger for this parser
   */
  private Logger logger = Logger.getLogger(ExampleParser.class);

  /**
   * The ExampleListType enum which represents the name of the list this parser is
   * currently reading.
   */
  private ExampleListType groupList = ExampleListType.none;

  /* (non-Javadoc)
   * @see org.sbml.jsbml.xml.parsers.AbstractReaderWriter#getShortLabel()
   */
  public String getShortLabel() {
    return ExampleConstants.shortLabel;
  }

  /* (non-Javadoc)
   * @see org.sbml.jsbml.xml.parsers.AbstractReaderWriter#getNamespaceURI()
   */
  public String getNamespaceURI() {
    return ExampleConstants.namespaceURI;
  }

}
```

Figure 4.13: *The first part of the parser for the extension.*

4.3.1 Reading

The class `AbstractReaderWriter` provide more or less all the necessary code to read the XML file for your extension, you just need to implement one method from the `Reader` interface. In future version of JSBML, this method might be implemented in a generic way using the java reflection API.

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 a `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.

```java
// Create the proper object and link it to its parent.
public Object processStartElement(String elementName, String prefix,
    boolean hasAttributes, boolean hasNamespaces, Object contextObject) {

  if (contextObject instanceof Model) {
    Model model = (Model) contextObject;
    ExampleModel exModel = null;

    if (model.getExtension(ExampleConstants.namespaceURI) != null) {
      exModel = (ExampleModel) model.getExtension(ExampleConstants.namespaceURI);
    } else {
      exModel = new ExampleModel(model);
      model.addExtension(ExampleConstants.namespaceURI, exModel);
    }

    if (elementName.equals("listOfFoos")) {

      ListOf<Foos> listOfFoos = exModel.getListOfFoos();
      this.groupList = ExampleListType.listOfFoos;
      return listOfFoos;
    }
  } else if (contextObject instanceof Foo) {
    Foo foo = (Foo) contextObject;

    // if Foo would have children, that would go here

  }
  else if (contextObject instanceof ListOf<?>) {
    ListOf<SBase> listOf = (ListOf<SBase>) contextObject;

    if (elementName.equals("foo") && this.groupList.equals(ExampleListType.listOfFoos)) {
      Model model = (Model) listOf.getParentSBMLObject();
      ExampleModel exModel = (ExampleModel) model.getExtension(ExampleConstants.namespaceURI);

      Foo foo = new Foo();
      exModel.addFoo(foo);
      return foo;
    }
  }
  return contextObject;
}
```

Figure 4.14: *Extension parser:* `processStartElement()`.

The `processEndElement()` (see Figure 4.15 on the next page) 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 nested lists will require more elaborate handling, but it should still be straightforward. If you don't use an enum or something else to keep track of which `ListOf` we are in and if you don't need to do other things when a closing XML element is encountered, you don't need to implement this method.

The attributes of an XML element are read into the corresponding object via the `readAttributes()` method that must be implemented for each class. An example is shown in Figure 4.16 on the following page for the class `Foo`. The `AbstractReaderWriter` will use these methods to set the attribute values into the java objects.

4.3.2 Writing

The method `getListOfSBMLElementsToWrite()` has to return a list of all objects that have to be written because of the passed object. In this way, the writer can traverse the XML tree to write all nodes. If the classes of the

```java
public boolean processEndElement(String elementName, String prefix,
  boolean isNested, Object contextObject) {

  if (elementName.equals("listOfFoos") {
    this.groupList = ExampleListType.none;
  }

  return true;
}
```

Figure 4.15: *Extension parser:* `processEndElement()`.

```java
@Override
public boolean readAttribute(String attributeName, String prefix, String value) {

  boolean isAttributeRead = super.readAttribute(attributeName, prefix, value);

  if (!isAttributeRead) {
    isAttributeRead = true;

    if (attributeName.equals(ExampleConstants.bar)) {
      setBar(StringTools.parseSBMLInt(value));
    } else {
      isAttributeRead = false;
    }
  }

  return isAttributeRead;
}
```

Figure 4.16: *Method to read the XML attributes.*

extension follow the structured advice in Section 4.2 on page 31 this method does not need to be implement and the basic implementation from `AbstractReaderWriter` can be used. This basic implementation make use of the `TreeNode.children()` method to find the list of children to write.

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 Figure 4.17 on the following page for the class `Foo`. Then the `AbstractReaderWriter` class will use these methods to write the attributes.

```java
public class Foo extends AbstractNamedSBase {
  ...

  public Map<String, String> writeXMLAttributes() {
    Map<String, String> attributes = super.writeXMLAttributes();
    if (isSetBar()) {
      attributes.remove("bar");
      attributes.put(Foo.shortLabel + ":bar", getBar());
    }

    // ...
    // further class attributes
  }
}
```

Figure 4.17: *Method to write the XML attributes.*

4.4 Implementation checklist

Figure 4.18 on the next page presents a checklist summarizing the different aspects of an extension that need to be implemented.

- ☐ Add the extension to an existing model (see Figure 4.2 on page 32).
- ☐ Add the five necessary methods for each class attribute:
  - ☐ `getBar()`
  - ☐ `isSetBar`
  - ☐ `setBar(int value)`
  - ☐ `unsetBar()`
- ☐ Add the default constructors (see Figure 4.4 on page 33).
- ☐ If the class has children, check if all list methods are implemented (see the program fragments in Figure 4.7, Figure 4.5, Figure 4.6, Figure 4.7):
  - ☐ `isSetListOfFoos()`
  - ☐ `getListOfFoos()`
  - ☐ `setListOfFoos(ListOf<Foo> listOfFoos)`
  - ☐ `createFoo()`
  - ☐ `addFoo(Foo foo)`
  - ☐ `removeFoo(Foo foo)`
  - ☐ `removeFoo(int i)`
  - ☐ `getAllowsChildren()`
  - ☐ `getChildCount()`
  - ☐ `getChildAt(int i)`
- ☐ All necessary create methods are implemented (see Figure 4.8 on page 36).
- ☐ Implement the `equals()` method (see Figure 4.9 on page 37).
- ☐ Implement the `hashCode()` method (see Figure 4.10 on page 37).
- ☐ Implement the `clone()` method (see Figure 4.11 on page 37 and Figure 4.12 on page 37).
- ☐ Implement the `toString()` method.
- ☐ Implement the `writeXMLAttribute()` method (see Figure 4.17 on the previous page).
- ☐ Implement the `readAttribute(String, String, String)` method (see Figure 4.16 on page 40).
- ☐ Implement the reader/writer method (see Figure 4.13 on page 38, Figure 4.14 on page 39, and Figure 4.15 on page 40).

Figure 4.18: *Implementation checklist for JSBML extension authors.*

4.5 Eclipse code templates

We created a set of Eclipse templates that would ease a lot the creation of all the methods described in the previous section of this chapter. These templates can be downloaded from the JSBML sources repository at `https://jsbml.svn.sourceforge.net/svnroot/jsbml/trunk/dev/eclipse/`.

The file JSBML_templates.xml define some code templates to autogenerate some code, following the checklist define in the previous section. It can be included in "Java" -> "Editor" -> "Templates".

To use these templates while programming write "JSBML" and press "CTR + tab". Then all available JSBML code templates are listed. Then select the desired template by pressing "enter". If you have several fields to rename use "tab" to rename them all in one go.

5 Acknowledgments

The development and support of JSBML is a substantial undertaking and many people have put in time and effort on this project. The authors especially thank the following individuals for their many contributions to JSBML (in alphabetical order):

- Students from the University of Tuebingen: Meike Aichele, Alexander Diamantikos, Jakob Matthes, Sarah Rachel Müller vom Hagen, Eugen Netz, Jan Rudolph, Alexander Peltzer, and Simon Schäfer
- PhD students at the University of Tuebingen: Roland Keller and Johannes Eichner.
- Students from the Leibnitz Institute for Plant Culture (IKP): Sebastian Fröhlich.

The development of JSBML is currently funded by the following organizations:

- The National Institute of General Medical Sciences (USA) via grant number R01 GM070923,
- The EMBL European Bioinformatics Institute (Germany and UK), and
- The Federal Ministry of Education and Research (BMBF, Germany) via grant numbers 0315756 and 0315384C for the *Virtual Liver Network* and the MedSys (Medical Systems Biology) project *Spher4Sys*.

Last but not least, JSBML is an open-source project, and we thank others who have helped in its progress, in the form of comments, bug reports, bug fixes, and other contributions.

Other interested people are welcome to join the team and to contribute to the project. The JSBML Team also explicitely encourages students who would like to participate in a large software project, to ask for current JSBML subprojects that are in need of doing.

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 always a constant, it might be some system variable `Variable` and can therefore be the subject of `Rule`s, `Event`s, `InitialAssignment`s 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 implement the `TreeNode` interface (defined 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. As discussed in Section 1.1.5 on page 6, you can obtain this module by checking out the repository https://jsbml.svn.sourceforge.net/svnroot/jsbml/trunk/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 then, the JSBML community has actively developed extension packages for the following SBML extensions: `fbc`, `groups`, `layout`, `multi`, `qual`, and `spatial`. These packages can be used with the version 1.0 or later of JSBML.

B Open tasks in JSBML development

The following is an incomplete list of tasks still remaining to be done to complete 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 all extension packages.
- The `toSBML()` methods on `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. [7] for details.

References

[1] Apache Software Foundation, *log4j*. Available on the World Wide Web at http://logging.apache.org/log4j/1.2/, 2010.

[2] ———, *The Apache Ant project*. Available on the World Wide Web at http://ant.apache.org/, 2012.

[3] ———, *Apache Tomcat*. Available on the World Wide Web at http://tomcat.apache.org/, 2012.

[4] K. Beck, E. Gamma, and D. Saff, *JUnit.org*. Available on the World Wide Web at http://www.junit.org, 2012.

[5] A. Bilgin, D. Caldwell, J. Ellson, E. Gansner, Y. Hu, S. North, Y. Koren, D. Dobkin, E. Koutsofios, B. Lilly, G. Low, J. Mocenigo, J. Scheerder, and G. Woodhull, *Graphviz – graph visualization software*. Available on the World Wide Web at http://www.graphviz.org/, 2012.

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

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

[8] A. Dräger, *Computational Modeling of Biochemical Networks*, PhD thesis, University of Tuebingen, Tübingen, Germany, Jan. 2011.

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

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

[11] G. Gkoutos, *The Ontology of Units of Measurement*. Available on the World Wide Web at http://www.obofoundry.org/cgi-bin/detail.cgi?id=unit, 2012.

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

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

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

[15] M. Hucka, A. Finney, S. Hoops, S. M. Keating, and N. Le Novère, *Systems biology markup language (SBML) Level 2: structures and facilities for model definitions*, tech. rep., Nature Precedings, December 2008.

[16] M. Hucka, A. Finney, H. Sauro, and H. Bolouri, *Systems Biology Markup Language (SBML) Level 1: Structures and Facilities for Basic Model Definitions*, Tech. Rep. 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, August 2003.

[17] JSBML Team, *The JSBML downloads*. Available on the World Wide Web at http://sourceforge.net/projects/jsbml/files/, 2012.

[18] ———, *The JSBML SVN repository*. Available on the World Wide Web at http://sourceforge.net/scm/?type=svn&group_id=279608, 2012.

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

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

[21] N. Le Novère, A. Finney, M. Hucka, U. S. Bhalla, F. Campagne, J. Collado-Vides, E. J. Crampin, M. Halstead, E. Klipp, P. Mendes, P. Nielsen, H. Sauro, B. E. Shapiro, J. L. Snoep, H. D. Spence, and B. L. Wanner, *Minimum information requested in the annotation of biochemical models (MIRIAM)*, Nature Biotechnology, 23 (2005), pp. 1509–1515.

[22] LibSBML Team, *libSBML*. Available on the World Wide Web at http://sbml.org/Software/libSBML, 2012.

[23] Oracle, Inc., *Java™*. Available on the World Wide Web at http://www.java.com/en/download/manual.jsp, 2012.

[24] SBML Editors, *The SBML frequently asked questions and answers*. Available on the World Wide Web at http://sbml.org/Documents/FAQ, 2012.

[25] ———, *The SBML specifications*. Available on the World Wide Web at http://sbml.org/Documents/Specifications/, 2012.

[26] SBML Team, *The SBML discussion forums and mailing lists*. Available on the World Wide Web at http://sbml.org/Forums/, 2012.

[27] Wikipedia Foundation, *Log4j log viewers*. Available on the World Wide Web at http://en.wikipedia.org/wiki/Log4j#Log_Viewers, 2012.

Index