A short description of the main differences between JSBML and libSBML

Andreas Dräger[*] Nicolas Rodriguez[†] Marine Dumousseau[†]
Alexander Dörr[*] Clemens Wrzodek[*]

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

December 14, 2011

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

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

In addition, JSBML can be used as a communication layer between the widespread application CellDesigner and any application that works with JSBML as its internal data structure. An example is given, that demonstrates how to convert between CellDesigner's plug-in data structures and JSBML objects.

In the same way, it is possible to inter-convert between data structures obtained from libSBML and JSBML. We provide an example of how to read SBML files with libSBML, turn them into JSBML data structures, manipulate them and turn them back to libSBML for writing.

Furthermore, JSBML will provides a compatibility module, whose member classes show an identical API as defined in libSBML. In this way, the compatibility module will facilitate a switch from libSBML to JSBML and vice versa by simply exchanging the included JAR file in the project.

Contents

1 Introduction

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

2 An extended type hierarchy

Whenever multiple elements defined in at least one of the SBML specifications share some attributes, JSBML provides a common superclass or at least a common interface that gathers methods for the manipulation of the shared properties. In this way, the type hierarchy of JSBML has become quite complex (see Figs. 1 to 5 on pages 6–10). Just as in libSBML, all elements extend

Figure 1: The type hierarchy of the main SBML constructs in JSBML. With letting `SBase` extend the interface `TreeNodeWithChangeSupport` that in turn extends the Java interfaces `Cloneable`, `Serializable`, and `TreeNode`, all derived elements of `SBase` also implement these types. In this way, derivatives of `SBase` can be used wherever an instance of `TreeNode` is requested. Furthermore, SBML elements that do not extend `SBase` are also derived from the identical base type `TreeNodeWithChangeSupport`, hence sharing several common methods and attributes. Elements colored in blue have been introduced as additional, in most cases abstract, data types in JSBML but do not have a corresponding element in libSBML. The yellow types `Creator` and `History` correspond to `ModelCreator` and `ModelHistory` in libSBML.

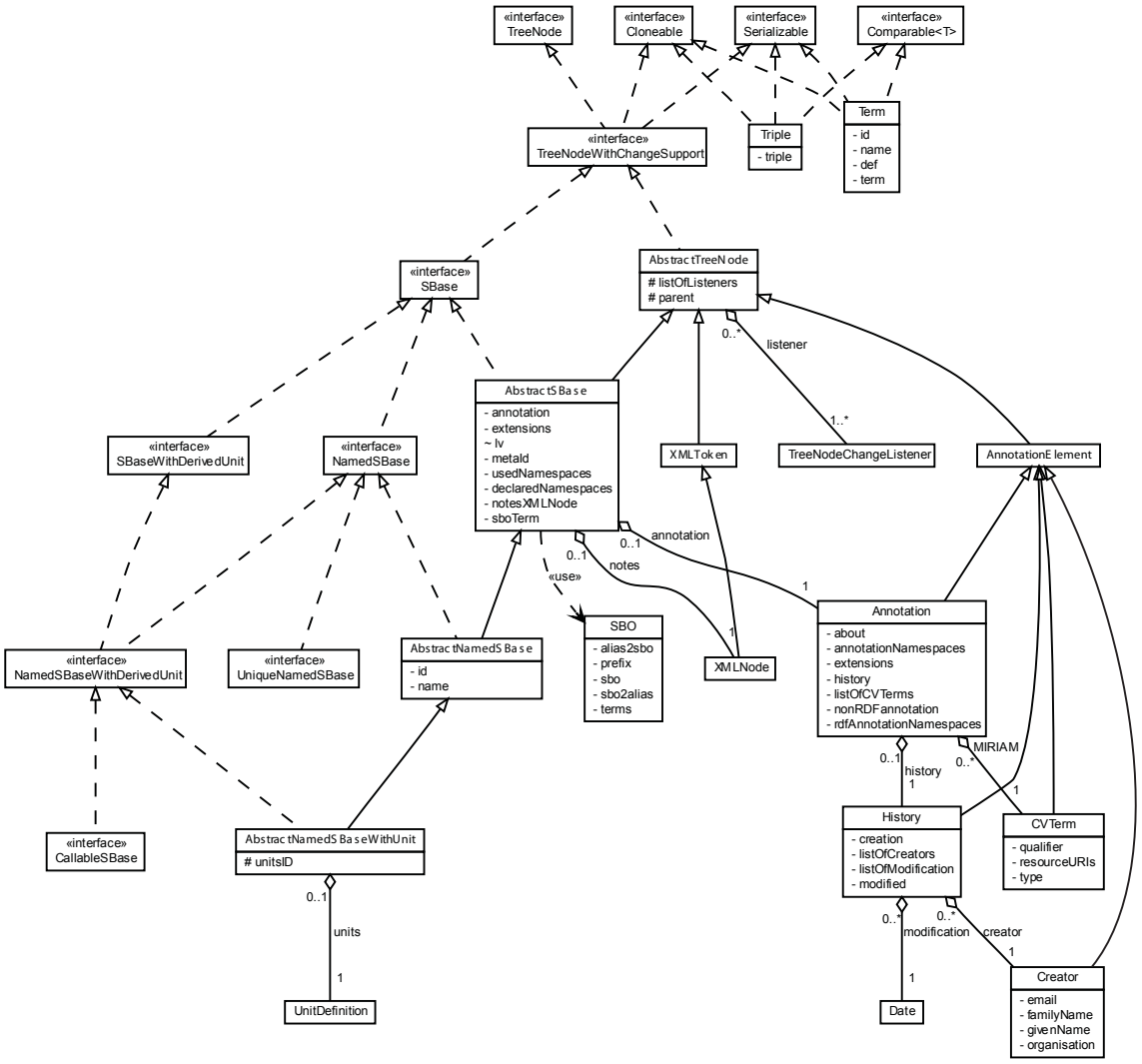

Figure 2: The interface `SBase`. This figure displays the most important top-level data structures of JSBML with main focus on the differences to libSBML. All data types that represent SBML constructs in JSBML extend `AbstractTreeNode`. Derivatives of `SBase` extend either one of the two abstract classes `AbstractSBase` or `AbstractNamedSBase`, which in turn also extend `AbstractTreeNode`. The class `SBO` parses the ontology file provided on the SBO web site (`http://www.ebi.ac.uk/sbo/main/`) in OBO format (Open Biomedical Ontologies) using a parser provided by the BioJava project (Holland *et al.*, 2008). For the sake of a clear arrangement, this figure omits all methods in the UML diagram. `SBO` stores its ontology in the classes `Term` that are interrelated in `Triples` consisting of subject, predicate, and object (each being an instance of `Term`).

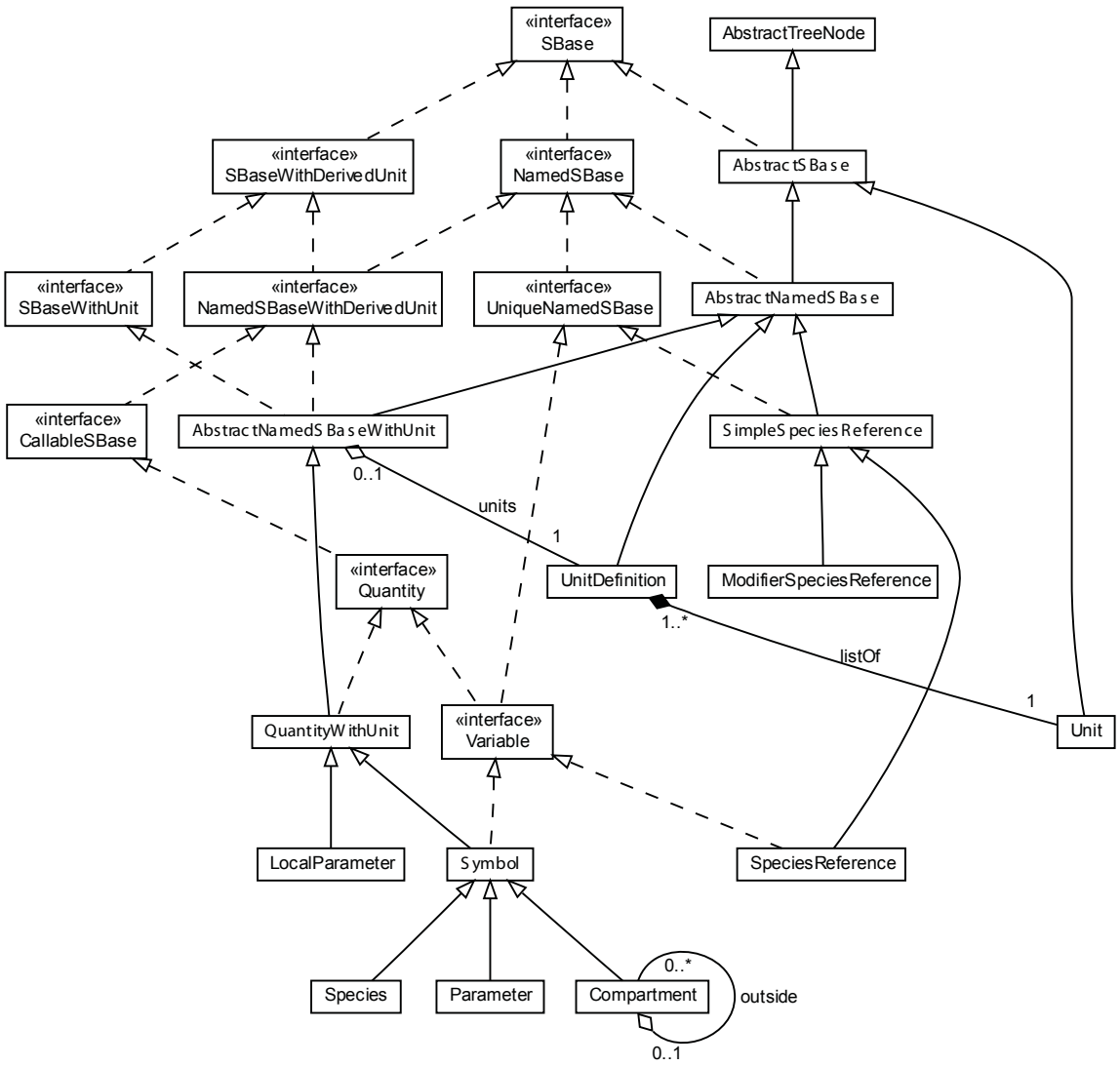

Figure 3: The interface `Variable`. JSBML refers to those components of a model that may change their value during a simulation as `Variables`. The class `Symbol` serves as the abstract superclass for variables that can also be equipped with a unit. Instances of `Parameter` do not contain any additional field. In `Species`, a Boolean switch decides whether its value is to be interpreted as an initial amount or as an initial concentration. In contrast to `Variables`, `LocalParameters` represent constant unit-value pairs that can only be accessed within their declaring `KineticLaw`.

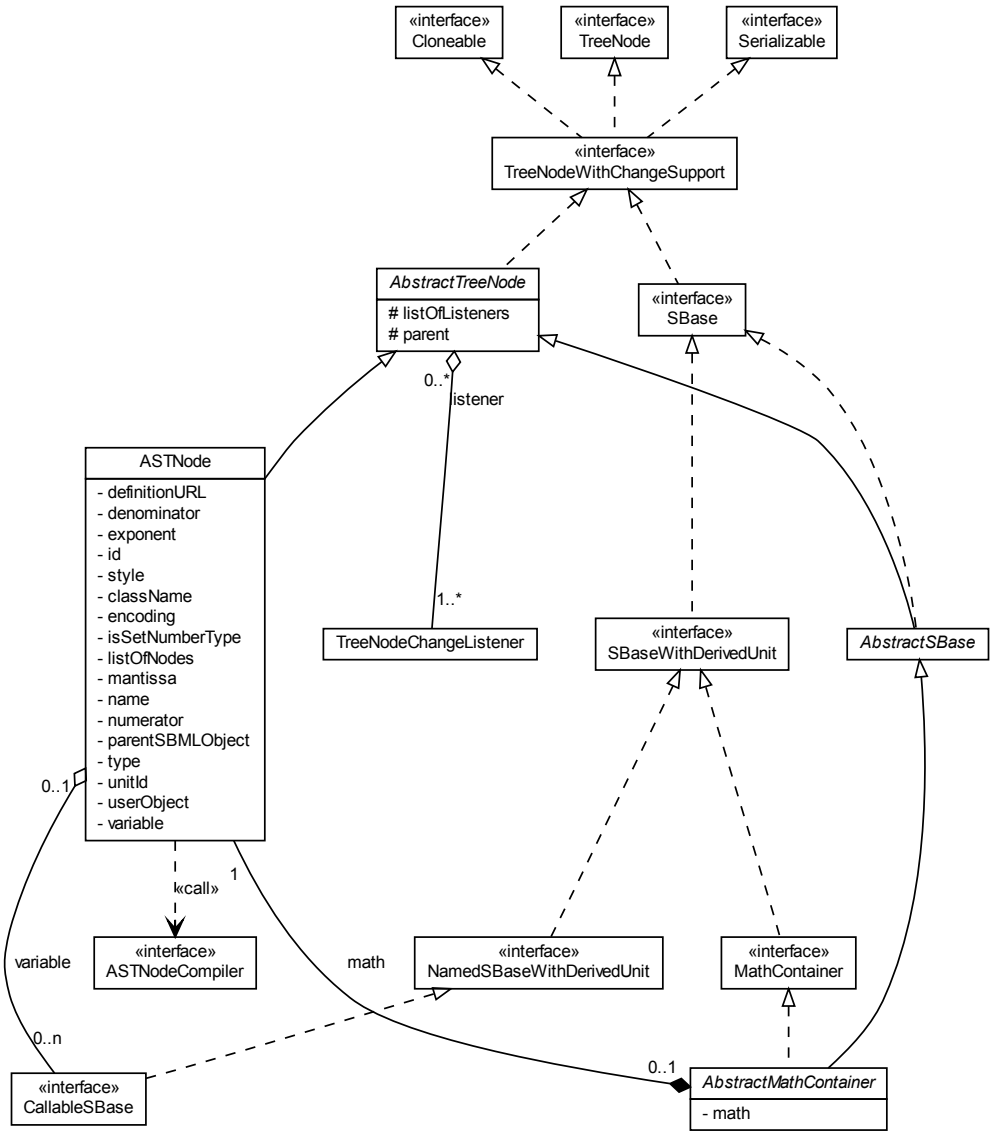

Figure 4: Abstract syntax trees. The class `AbstractMathContainer` serves as the superclass for several model components in JSBML. It provides methods to manipulate and access an instance of `ASTNode`, which can be converted to or read from `C`-like formula `Strings`. Internally, `AbstractMathContainers` only deal with instances of `ASTNode`. It should be noted that these abstract syntax trees do not implement the `SBase` interface, but extend `AbstractTreeNode`.

Figure 5: Containers for mathematical expressions. The interface `MathContainer`, particularly its directly derived class `AbstractMathContainer`, constitutes the superclass for all elements that store and manipulate mathematical formulas in JSBML, which is done in form of `ASTNode` objects. These can be evaluated using an implementation of `ASTNodeCompiler`. Note that some classes that extend `AbstractMathContainer` do not contain any own fields or methods: `Delay`, `Priority`, `StoichiometryMath`, or `AlgebraicRule`.

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

```
1  package org.sbml.gui;
2
3  import javax.swing.*;
4  import org.sbml.jsbml.*;
5
6  /** Displays the content of an SBML file in a {@link JTree} */
7  public class JSBMLvisualizer extends JFrame {
8
9    public JSBMLvisualizer(SBase sbase) {
10     super("SBML Visualizer");
11     getContentPane().add(new JScrollPane(new JTree(sbase)));
12     setDefaultCloseOperation(EXIT_ON_CLOSE);
13     pack();
14     setVisible(true);
15   }
16   /** @param args Expects a valid path to an SBML file. */
17   public static void main(String[] args) throws Exception {
18     UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
19     new JSBMLvisualizer(SBMLReader.read(new java.io.File(args[0])));
20   }
21
22 }
```

the abstract type `SBase`, but in JSBML, `SBase` has become an interface. This allows more complex relations between derived data types. In contrast to libSBML, `SBase` in JSBML extends the interface `TreeNodeWithChangeSupport` that in turn extends three other interfaces: `Cloneable`, `Serializable`, override the `clone()` method from the class `java.lang.Object`, all JSBML elements can be deeply copied and are therefore *clone-able*. By extending the interface `Serializable`, it is possible to store JSBML elements in binary form without explicitly writing them to an SBML file. In this way, programs can easily load and save their in-memory objects or send complex data structures through a network connection without the need of additional file encoding and subsequent parsing. The third interface, `TreeNode` is actually defined in Java's `swing` package. `TreeNode` is a type that is independent of any graphical information. It basically defines recursive methods on hierarchically structured data types, such as iteration over all of its successors. In this way, all instances of JSBML's`SBase` interface can be directly passed to the `swing` class `JTree` and can hence be easily visualized. Listing 1 demonstrates in a simple code example how to parse an SBML file and to immediately display its content on a `JFrame`. Fig. 6 on the next page shows an example output when applying the program to an SBML test model. The `ASTNode` class in JSBML is also derived from all these three interfaces and can hence be cloned, serialized, and visualized in the same way.

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

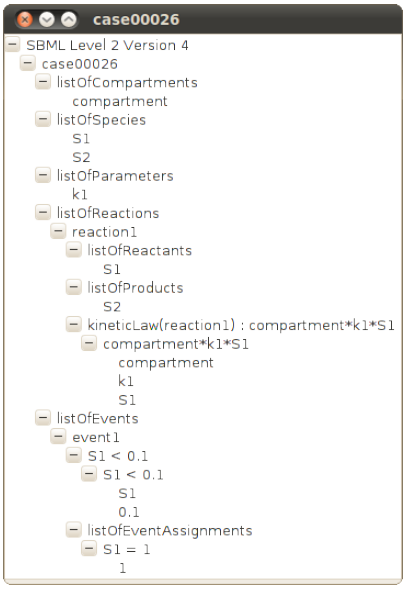

Figure 6: A tree representation of the content of SBML test model `case00026`. In JSBML, the hierarchically structured `SBMLDocument` can be traversed recursively because all instances of `SBase` implement the interface `TreeNode`.

`Node`.

2.1 `AbstractTreeNode`

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

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

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

2.2 Characteristic features of SBases

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

In contrast to libSBML, the Level and Version of such an AbstractSBase is stored in a special generic object, a ValuePair. The class ValuePair takes two values of any type that both implement the interface Comparable. Storing the Level/Version combination in such a ValuePair, which itself implements the Comparable interface, allows users to perform checks for an expected Level/Version combination of an element more easily, as the example in Listing 2 demonstrates. The method getLevelAndVersion() in AbstractSBase delivers an

Listing 2: Check for a minimal expected Level/Version combination

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

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

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

Many SBML elements represent some quantitative value, which is associated with a unit. However, the value does not necessarily have to be defined explicitly. In many cases, it needs to be computed from a formula contained in the instance of SBase in form of an abstract syntax tree, i.e., ASTNode. Therefore, also the associated unit may not be set explicitly but can be derived when evaluating the formula. In JSBML, the interface SBaseWithDerivedUnit unifies all those elements that either explicitly or implicitly contain some unit. If these elements can also be addressed using an identifier, they also implement the interface NamedSBaseWithDerivedUnit. Within formulas, i.e., ASTNodes, references can only be made to instances of CallableSBase, which is a special case of NamedSBaseWithDerivedUnit. Fig. 3 on page 8 shows this part of JSBML's type hierarchy in more detail.

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

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

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

2.3 The `MathContainer` interface

This interface gathers all those elements that may contain mathematical expressions encoded in abstract syntax trees (instances of `ASTNode`). The abstract class `AbstractMathContainer` serves as actual superclass for the majority of the derived types. Figs. 4 to 5 on pages 9–10 give a better overview of how this data structure is intended to function.

2.4 The `Assignment` interface

JSBML unifies all those elements that may change the value of some *variable* in SBML under the interface `Assignment`. This interface uses the term *variable* for the element whose value is to be changed depending on some mathematical expression that is also present in the `Assignment` (because `Assignment` extends the interface `MathContainer`). Therefore, an `Assignment` contains methods such as `set-/getVariable(Variable v)` and also `isSetVariable()` as well as `unsetVariable()`. In addition to that, JSBML also provides the methods `set-/getSymbol(String symbol)` in the `InitialAssignment` class to make sure that switching from libSBML to JSBML is quite smoothly. However, the preferred way in JSBML is to apply the methods `setVariable` either with `String` or `Variable` instances as arguments. Fig. 5 on page 10 displays the type hierarchy of the `Assignment` interface in more detail.

3 Differences in the abstract programming interface

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

3.1 Abstract syntax trees

Both libraries define a class ASTNode for in-memory manipulation and evaluation of abstract syntax trees that represent mathematical formulas and equations. These can either be parsed from a representation in C language-like Strings, or from a MathML representation. The JSBML ASTNode provides various methods to transform these trees to other formats, for instance, LaTeX Strings. In JSBML, several static methods allow easy creation of new syntax trees, for instance, the following code

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

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

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

Getter and setter:

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

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

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

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

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

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

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

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

Some examples for convenient constructors:

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

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

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

3.2 The `ASTNodeCompiler` class

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

3.3 Cloning when adding child nodes to instances of `SBase`

When adding elements such as a `Species` to a `Model`, libSBML will clone the object and add the clone to the `Model`. In contrast, JSBML does not automatically perform cloning. The advantage is that modifications on the object belonging to the original pointer will also propagate to the element added to the `Model`. Furthermore, this is more efficient with respect to the run time and also more intuitive for Java programmers. If cloning is necessary, users should call the `clone()` method manually. Since all instances of `SBase` and also `Annotation`, `ASTNode`, `CVTerm`, and `History` extend `AbstractTreeNode`, which in turn implements the interface `Cloneable` (see Fig. 1 on page 6), all these elements can be naturally cloned. However, when cloning an object in JSBML, such as an `AbstractNamedSBase`, define a tree, in which each element has exactly one parental node.

3.4 Deprecation

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

3.5 Compartments

In SBML Level 3 (Hucka *et al.*, 2010), the domain of the `spatialDimensions` attribute in `Compartments` was changed from $\{0,1,2,3\}$, which can be represented with a `short` value in Java, to a value in $\mathbb{R}$, i.e., a `double` value. For this reason, the method `getSpatialDimensions()` in JSBML always returns a `double` value. For consistency with libSBML, the `Compartment` class in JSBML also provides the redundant method `getSpatialDimensionsAsDouble()` that returns the identical value, but that is marked as a deprecated method.

3.6 Exceptions

In case of an error, JSBML throws often an exception while libSBML methods return some error codes instead. This behavior helps programmers and users to avoid creating invalid SBML data structures already when dealing with these in memory. Furthermore, exception handling is very well implemented in Java and it is therefore a better programming style in this language. Methods can already declare that these may potentially throw exceptions. In this way, programmers can be aware of potential sources of problems already at the time of writing the source code. Examples are the `ParseException` that may be thrown if a given formula cannot be parsed properly into an `ASTNode` data structure, or `InvalidArgumentExceptions` if inappropriate values are passed to methods. For instance,

- An object representing a constant such as a `Parameter` whose `constant` attribute has been set to `true` cannot be used as the `Variable` element in an `Assignment`.
- An instance of `Priority` can only be assigned to an `Events` if its `level` attribute has at least been set to three.
- Another example is the `InvalidArgumentException` that is thrown when trying to set an invalid identifier `String` for an instance of `AbstractNamedSBase`.

- JSBML keeps track of all identifiers within a model. For each namespace it contains a separate set of identifiers within the `Model`. It is therefore not possible to assign duplicate identifiers in case of elements that implement the interface `UniqueNamedSBase`. For `UnitDefinitions` and `LocalParameters` separate sets are maintained. Since local parameters are only visible within the `KineticLaw` that contain these, JSBML will only prohibit having more than one local parameter within the same list that has the identical identifier. All these sets are updated upon any changes within the model. When adding an element with an already existing identifier for its namespace, or changing some identifier to a value that is already defined within this namespace, JSBML will throw an exception.

- Meta identifiers must be unique through the entire SBML file. To ensure that no duplicate meta identifiers are created, JSBML keeps a set of all meta identifiers on the level of the `SBMLDocument`, which is updated upon any change of elements within the data structure. In this way, it is not possible to set the meta identifier of some element to an already existing value or to add nodes to the SBML tree that contain a meta identifier defined somewhere else within the tree. In both cases, JSBML will throw an exception. Since meta identifiers can be generated in a fully automatic way (method `nextMetaId()` on `SBMLDocument`), users of JSBML should not care about these identifiers at all. JSBML will automatically create meta identifiers where missing upon writing an SBML file.

Hence, you have to be aware of potential exceptions and errors when using JSBML, on the other hand this will prevent you from doing obvious mistakes. The class `SBMLReader` in JSBML catches those errors and exceptions. With the help of the logging utility, JSBML notifies users about syntactical problems in SBML files. JSBML follows the rule that illegal or invalid properties are not set.

3.7 Model history

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

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

You will not find an implementation corresponding to the interface `libSBMLConstants` in JSBML. The reason is that the JSBML team decided to encode constants using the Java construct `enum`. For instance, all the fields starting with the prefix `AST_TYPE_*` have a corresponding field

in the ASTNode class itself. There you can find the enum Type. Instead of typing libSBML-Constants.AST_TYPE_PLUS, you would therefore type ASTNode.Type.PLUS.

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

3.9 The classes libSBML and JSBML

There is no class libSBML because this library is called JSBML. You can therefore only find a class JSBML. This class provides some similar methods as the libSBML class in libSBML, such as getJSBMLDottedVersion() to obtain the current version of the JSBML library, which is 0.8.* at the time of writing this document. However, many other methods that you might expect to find there, if you are used to libSBML, are located in the actual classes that are related with the function. For instance, the method to convert between a String and a corresponding Unit.Kind can be done by using the method

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

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

3.10 Various types of ListOf* classes

In JSBML, there is not a specific ListOf* class for each type of SBase elements. We used a generic implementation ListOf<? extends SBase> that allows us to use the same class for each of the different ListOf* classes defined in libSBML while keeping a type-safe class. We defined several methods that use the Filter interface to search or filter a ListOf object. For example, to query an instance of ListOf in JSBML for names or identifiers or both, you can apply the following filter:

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

This will give you the first element in the list with the given identifier. Various filters are already implemented, but you can easily add your customized filter. To this end, you only have to implement the Filter interface in org.sbml.jsbml.util.filters. There you can also find an OrFilter and an AndFilter, which take as arguments multiple other filters. With the SBOFilter you can query for certain SBO annotations (Le Novère, 2006; Le Novère *et al.*, 2006) in your list, whereas the CVTermFilter helps you to identify SBase instances with a desired MIRIAM (Minimal Information Required In the Annotation of Models) annotation (Le Novère *et al.*, 2005). For instances of ListOf<Species> you can apply the BoundaryCondition-Filter to look for those species that operate on the boundary of the reaction system.

3.11 Units and unit definitions

3.11.1 The exponent attribute of units

Since SBML Level 3 (Hucka *et al.*, 2010) the data type of the exponent attribute in the `Unit` class has been changed from `int` to `double` values. JSBML reflects this in the method `getExponent()` by returning `double` values only. For a better compatibility with libSBML, whose corresponding method still returns `int` values, JSBML also provides the method `getExponentAsDouble()`. This method returns the value from the `getExponent()` method and is therefore absolutely redundant.

3.11.2 Predefined unit definitions

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

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

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

3.11.3 Access to the units of an element

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

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

Fig. 1 on page 6 provides a better overview about the relationships between all the classes explained here. Note that `AbstractNamedSBaseWithUnit` serves as the abstract superclass for `Event` and `QuantityWithUnit`. In the class `Event`, all methods to deal with units are deprecated

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

because the timeUnits attribute was removed in SBML Level 2 Version 2. The same holds true for instances of ExplicitRule and KineticLaw, which both can only be explicitly populated with units in SBML Level 1 for ExplicitRule and before SBML in Level 2, Version 3 for KineticLaw. In contrast, QuantityWithUnit serves as the abstract superclass for LocalParameter and Symbol, which is then again the super type of Compartment, Species, and (global) Parameter.

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

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

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

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

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

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

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

4 Additional features of JSBML

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

4.1 Change listeners

JSBML introduces the possibility to listen to change events in the life of an SBML document. To benefit from this advantage, simply let your class implement the interface `TreeNodeChangeListener` and add it to the list of listeners in your instance of `SBMLDocument`. You only have to implement three methods

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

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

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

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

Note that the class `TreeNodeChangeEvent` extends the class `java.beans.PropertyChangeEvent`, which is derived from `java.util.EventObject`. It should also be pointed out that the interface `TreeNodeChangeListener` extends the interface `java.beans.PropertyChangeListener` which in turn extends the interface `EventListener` in the package `java.util`. In this way, the event and listener data structures fit into the common Java™ API (Application Programming Interface) and allow users also to make use of, e.g., `EventHandlers` to deal with changes in a model.

Since in JSBML all major data objects implement the interface `TreeNode`, these listeners are notified about any kind of change in any implementing data structure. The interface `TreeNodeWithChangeSupport` extends Java's standard `TreeNode` interface by adding methods that maintain a list of `TreeNodeChangeListeners` and notify these whenever some property changes or nodes are added/deleted from the tree. In this way, the `TreeNodeChangeLis-`

teners do not only keep track of changes in instances of SBase. This means that changes inside of, e.g., CVTerm or History may can also be traced with this implementation.

4.2 Determination of the variable in AlgebraicRules

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

4.3 find* methods

JSBML provides users with several find* methods on a Model to quickly query for elements, based on their identifier or name. Developers can search for various instances of SBase (for instance, CallableSBase, NamedSBase, NamedSBaseWithDerivedUnit) or use the methods findLocalParameters, findQuantity, findQuantityWithUnit, findQuantityWithUnit, findSymbol, and findVariable to search for the corresponding element in the model. This enables a quick and easy way to work with SBML models, without having to iterate through the elements of a Model again and again.

4.4 Utility classes provided by JSBML

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

4.4.1 Pre-implemented mathematical functions and constants

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

4.4.2 Some tools for String manipulation

The class StringTools provides several methods for convenient String manipulation. These methods are particularly useful when parsing or displaying double numbers in a Locale-de-

pendent 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.

4.5 Logging functionality

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

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

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

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

4.5.1 Some example configurations

Listing 3 gives a short overview about how to customize the configuration file to log all the changes that happen to the SBML elements by putting the threshold of all the loggers in the `org.sbml.jsbml.util` package to `DEBUG`. The class `SimpleTreeNodeChangeListener` will then output the old value and the new value whenever a setter methods is used on the SBML elements.

Listing 3: A simple log4j example.

```
1  # All logging output sent to the console
2  log4j.rootCategory=INFO, console
3
4  #
5  # Console Display
6  #
7  log4j.appender.console=org.apache.log4j.ConsoleAppender
8  log4j.appender.console.layout=org.apache.log4j.PatternLayout
9
10 # Pattern to output the caller's file name and line number.
```

[2] http://logging.apache.org/log4j/
[3] http://logging.apache.org/log4j/1.2/manual.html

```
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
```

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

If you are deploying your application in an application server such as Tomcat, you could define an appender that would send some messages by e-mail. Listing 4 gives an example of that, were any messages from the error level are send by mail. All the messages are also written to a rolling log file.

Listing 4: SMTPAppender log4j example.

```
# 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
#defines how often emails 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
```

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

[4] `http://en.wikipedia.org/wiki/Log4j#Log_Viewers`

4.6 JSBML modules

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

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

The capabilities of the SBML validator constitute the major strength of libSBML (Bornstein *et al.*, 2008) in comparison to JSBML, which does not yet contain a stand-alone validator for SBML, but makes use of the online validation provided at `http://sbml.org`. Furthermore, if the platform-dependency of libSBML does not hamper your application, or you want to slowly switch from libSBML to JSBML, you may want to be able to still read and write SBML models using libSBML. To this end, the JSBML module `libSBMLio` provides the classes `LibSBMLReader` and `LibSBMLWriter`. Listing 5 on the next page gives a small example of how to use the `LibSBMLReader`. For this example to run, please make sure to have libSBML installed correctly on your system. The current version of the libSBML/JSBML interface at the time of writing this document requires libSBML version 4.2.0. To this end, you may have to set environment variables, e.g., the `LD_LIBRARY_PATH` under Linux operating system, appropriately. For details, see the documentation of libSBML[5]. Writing SBML works similarly. Example 5 on the facing page will display the content of an SBML file in a `JTree`, similar as shown in Fig. 6 on page 12.

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

Once an application has been implemented based on JSBML, it can easily be accessed from CellDesigner's plugin menu (Funahashi *et al.*, 2003). To this end, it is necessary to extend two classes that are defined in CellDesigner's plugin API (Application Programming Interface). The Listings 6 to 7 on pages 28–29 show a very simple example of how to pass CellDesigner plugin model data structures to the translator in JSBML, which creates then a JSBML `Model` data structure.The examples described by Listings 6 to 7 on pages 28–29 create a plugin for CellDesigner, which displays the SBML data structure in a tree, like the example in Fig. 6 on page 12. This example only shows how to translate a plugin data structure from CellDesigner into a corresponding JSBML data structure. With the help of the class `PluginSBMLWriter` it is possible to notify CellDesigner about changes in the model data structure. Note that Listing 7 on page 29 is only completed by implementing the methods from the superclass, `CellDesignerPlugin`. In this example it is sufficient to leave the implementation of several methods empty.

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

Listing 5: A simple example for converting libSBML data structures into JSBML data objects

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

4.6.3 `libSBMLcompat`, the JSBML compatibility module for libSBML

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

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

4.6.4 `android`, a compatibility module for Android systems

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

A Frequently Asked Questions (FAQ)

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

Why does the class `LocalParameter` not inherit from `Parameter`?

The reason is the Boolean attribute `constant`, which is present in `Parameter` and can be set to `false`. A parameter in the meaning of SBML is not a constant, it might be some

Listing 6: A simple implementation of CellDesigner's abstract class `PluginAction`

```
1  package org.sbml.jsbml.cdplugin;
2
3  import java.awt.event.ActionEvent;
4  import javax.swing.JMenuItem;
5  import jp.sbi.celldesigner.plugin.PluginAction;
6
7  /** A simple implementation of an action for a CellDesigner plug-in,
8   *  which invokes the actual plug-in program. */
9  public class SimpleCellDesignerPluginAction extends PluginAction {
10
11   /** Memorizes a pointer to the actual plug-in program. */
12   private SimpleCellDesignerPlugin plugin;
13
14   /** Constructor memorizes the plug-in data structure. */
15   public SimpleCellDesignerPluginAction(SimpleCellDesignerPlugin plugin) {
16     this.plugin = plugin;
17   }
18
19   /** Executes an action if the given commant occurs. */
20   public void myActionPerformed(ActionEvent ae) {
21     if (ae.getSource() instanceof JMenuItem) {
22       String itemText = ((JMenuItem) ae.getSource()).getText();
23       if (itemText.equals(SimpleCellDesignerPlugin.ACTION)) {
24         plugin.startPlugin();
25       }
26     } else {
27       System.err.printf("Unsupported source of action %s\n", ae
28           .getSource().getClass().getName());
29     }
30   }
31
32 }
```

Listing 7: A simple example for a CellDesigner plugin using JSBML as a communication layer

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

system variable and can therefore be the subject of Rules, Events, InitialAssignments and so on, i.e., all instances of Assignment, whereas a LocalParameter is defined as a constant quantity that never changes its value during the evaluation of a model. It would therefore only be possible to let Parameter inherit from LocalParameter but this could lead to a semantic misinterpretation.

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

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

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

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

Does JSBML support SBML extension packages?

In version 0.8, JSBML does not provide an abstract programming interface for extension packages. However, the JSBML community is actively developing extension packages for the following SBML extensions: fbc, groups, layout, multi, qual, and spatial. The release 1.0 of JSBML will support at least these packages. If you want to make use of SBML extensions in JSBML before an official release of version 1.0, please checkout the latest copy from the trunk of JSBML. To this end, please follow the instructions at http://sourceforge.net/projects/jsbml/develop.

B Acknowledgments and funding

The authors are grateful to Sebastian Fröhlich, Roland Keller, Sarah Rachel Müller vom Hagen, Simon Schäfer who all contributed to the JSBML project (alphabetic order).

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

References

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

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

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

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

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

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

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

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

Le Novère, N., Finney, A., Hucka, M., Bhalla, U. S., Campagne, F., Collado-Vides, J., Crampin, E. J., Halstead, M., Klipp, E., Mendes, P., Nielsen, P., Sauro, H., Shapiro, B. E., Snoep, J. L., Spence, H. D., and Wanner, B. L. (2005). Minimum information requested in the annotation of biochemical models (MIRIAM). *Nature Biotechnology*, **23**(12), 1509–1515.

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

Index