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‡

March 9, 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^{TM} 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 No need to mention the specific versions in the abstract, I think. We provides 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 specifications. However, it is important to keep in mind that SBML is a language that defines how to store and to exchange models of biological processes 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, such as CellDesigner (Funahashi *et al.*, 2003). The following sections will not only give a detailled 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 the abstract type `SBase`, but in JSBML, `SBase` has become an interface. This allows more complex relations between derived data types. In contrast to libSBML, `SBase` in JSBML extends three other interfaces: `Cloneable`, `Serializable`, and `TreeNode`. As all elements defined in JSBML override the `clone()` method from the class `java.lang.Object`, all JSBML elements can be deeply copied and are therefore *cloneable*. By extending the interface `Serializable`, it is possible to store JSBML elements in binary form without explicitly writing them to an SBML file. In this way, programs can easily load and save their in-memory objects or send complex data struc-

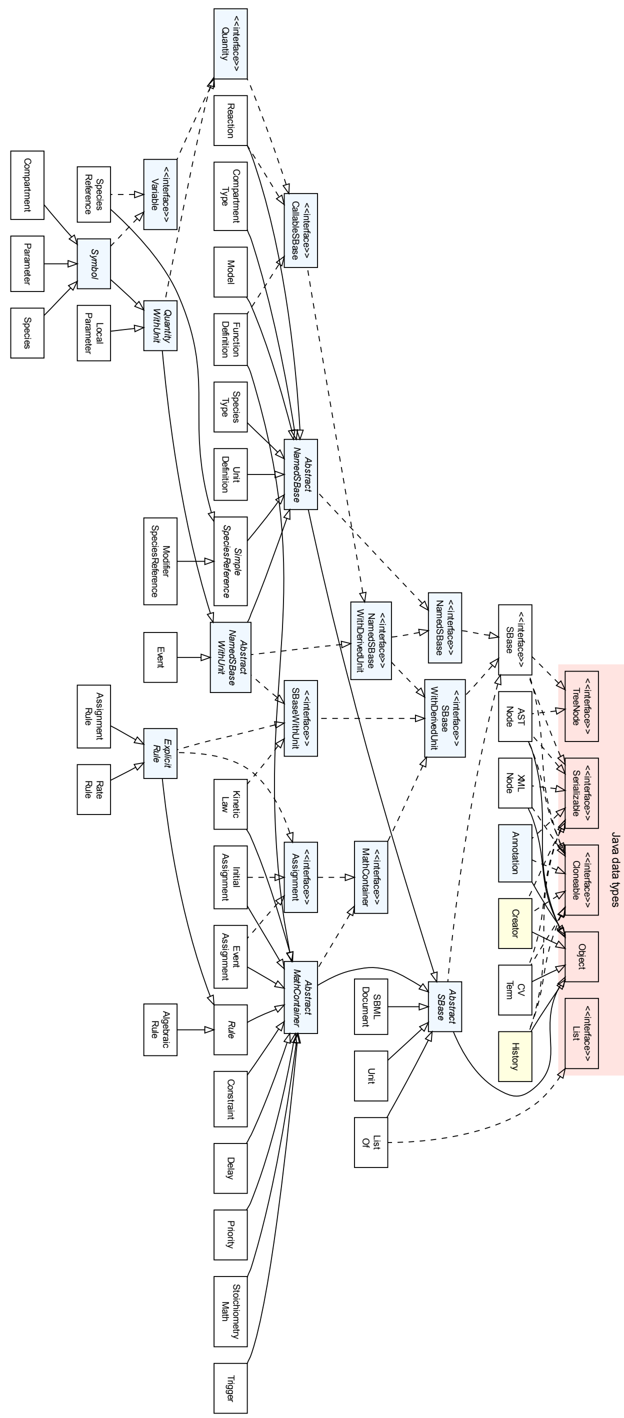

Figure 1: The type hierarchy of the main SBML constructs in JSBML. With letting SBase implement the Java interfaces Cloneable, Serializable, and TreeNode, all derived elements also implement these types. Elements colored in blue have been introduced as additional, in most cases abstract, data types in JSBML but do not have a corresponding element in libSBML. The yellow types Creator and History correspond to ModelCreator and ModelHistory in libSBML.

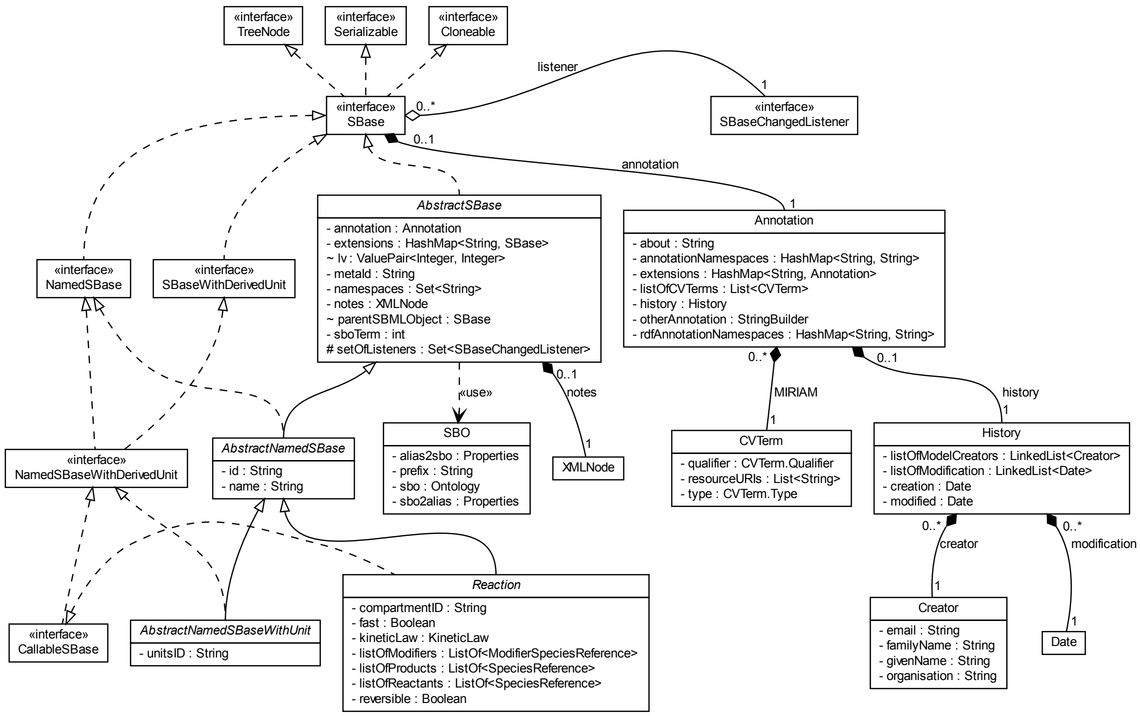

Figure 2: The interface `SBase`, adapted from (Dräger, 2011). This figure displays the most important top-level data structures of JSBML with main focus on the differences to libSBML. All other data types that represent SBML constructs in JSBML extend either one of the two abstract classes `AbstractSBase`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

tures 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 hence be easily visualized. Listing **??** on page ?? demonstrates in a simple code example how to parse an SBML file and to immediately display its content on a `JFrame`. The `ASTNode` class in JSBML also implements all these three interfaces and can hence be cloned, serialized, and visualized in the same way.

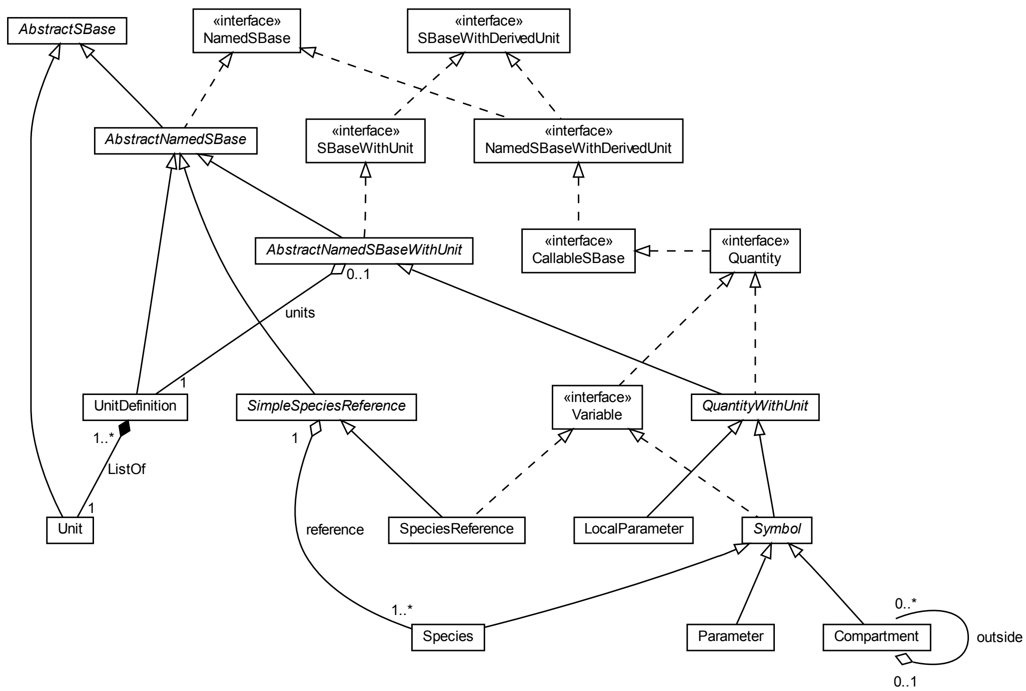

Figure 3: The interface `Variable`, adapted from (Dräger, 2011). JSBML refers to those components of a model that may change their value during a simulation as `Variables`. The class `Symbol` serves as the abstract superclass for variables that can also be equipped with a unit. Instances of `Parameter` 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`.

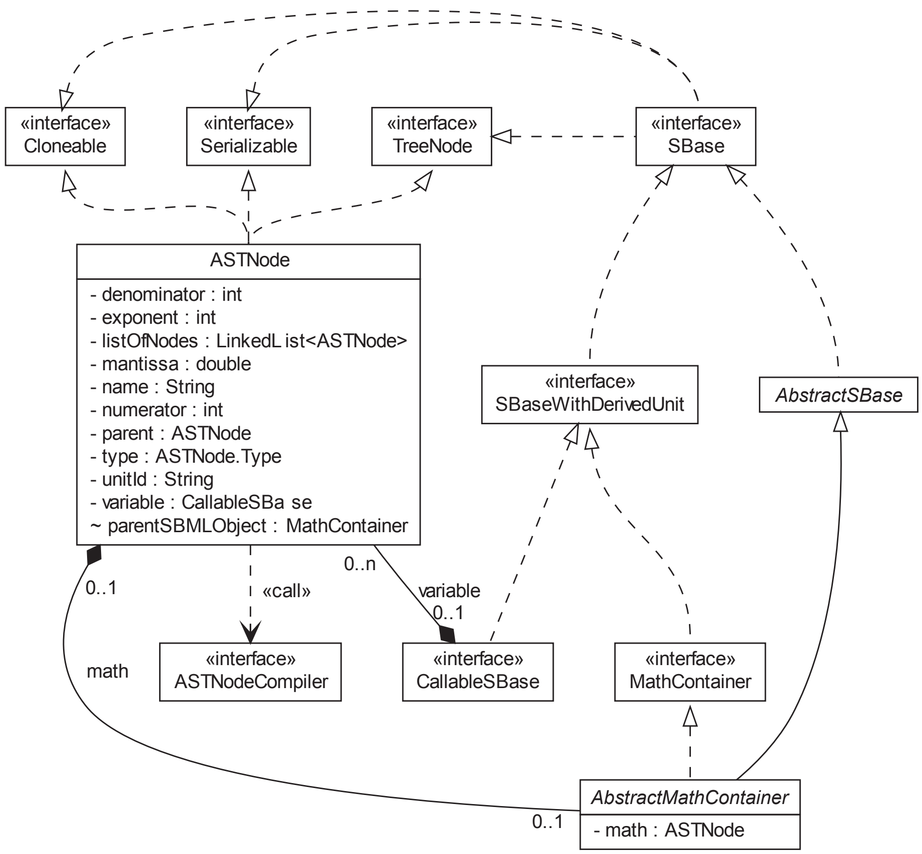

Figure 4: Abstract syntax trees, adapted from (Dräger, 2011). The class `AbstractMathContainer` serves as the superclass for several model components in JSBML. It provides methods to manipulate and access an instance of `ASTNode`, which can be converted to or read from C-like formula `String`s. Internally, `AbstractMathContainer`s only deal with instances of `ASTNode`. It should be noted that these abstract syntax trees do not implement the `SBase` interface, but implement the Java interfaces `Cloneable`, `Serializable`, and `TreeNode`.

2.1 `SBase`s with names, values and units

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`.

Some types derived from `SBase` contain a unique identifier, an `id`. JSBML gathers all these elements under the common interface `NamedSBase`. The class `AbstractNamedSBase`, which extends `AbstractSBase`, implements this interface.

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

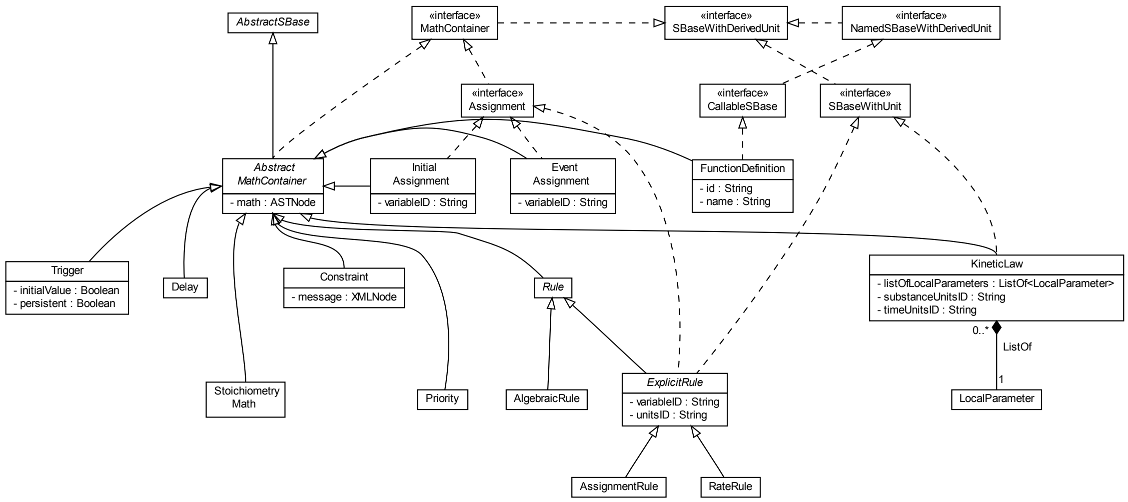

Figure 5: MathContainer, adapted from (Dräger, 2011). Instances of the interface MathContainer, particularly its directly derived class AbstractMathContainer, constitute the superclass for all elements that store and manipulate mathematical formulas in JSBML, which is done in form of ASTNode objects. 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.

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 a unique 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 unique 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.

2.2 The `MathContainer` interface

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

2.3 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 method `set-/getSymbol(String symbol)` in the `InitialAssignment` class to make sure that switching from libSBML to JSBML is quite smoothly. However, the preferred way in JSBML is to apply the methods `setVariable` either with `String` or `Variable` instances as arguments. Fig. 5 on the facing page 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 and named and behave like classes and methods in other Java libraries. In this section, we will discuss the most important differences in the APIs of JSBML and libSBML.

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.

There is more change than this, we have methods that take/return 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

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 implement 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, all children of this element will recursively be cloned before adding them to the new element. This is necessary, because the data structures specified in SBML define a tree, in which each element has exactly one parental node.

3.4 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 deprected method.

3.5 Deprecation

The intention of JSBML is to provide a Java library that support the latest specifications of SBML. But we also want to support earlier specifications. So JSBML provides methods and classes to cover those as well, but these are often marked as being deprecated to avoid creating models that refer to these elements.

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`.

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. One potential problem of detecting errors quickly and throwing exceptions is that it might prevent JSBML to be able to read properly invalid SBML models.

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 libSBMConstants by Java enums

You won't find a corresponding implementation of the interface libSBMConstants 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 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.

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 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 allow 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 uses 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 `BoundaryConditionFilter` to look for those species that operate on the boundary of the reaction system.

3.11 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.

3.12 Unit definitions

3.12.1 Predefined unit definitions

A model in JSBML always also contains all predefined units in the model if there are any, i.e., for models encoded 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 to generate MIRIAM URIs accordingly. In this way, JSBML facilitates the annotation of user-defined units and unit definitions with MIRIAM-compliant information.

3.12.2 Access to the units of an element

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

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

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

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

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

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

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

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

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

4.2 Determination of the variable in `AlgebraicRule`s

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 identify elements, based on their identifier or name. Developers can search for different instances of SBase (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 Logging functionality

JSBML provides logging.

4.5 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 provides 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.

4.5.1 Example: 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, whose SBML validation is not yet fully implemented. 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 provies the classes LibSBMLReader and LibSBMLWriter. Listing 1 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[2]. Writing SBML works similarly. This example will display the content of an SBML file in a JTree, similar as shown in Fig. **??** on page ??.

[2] http://sbml.org/Software/libSBML

```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();
  }
}
```

Listing 1: A simple example for converting libSBML data structures into JSBML data objects

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

Once an application has been implemented based on JSBML, it can easily be accessed from CellDesigner's plugin menu (Funahashi *et al.*, 2003). To this end, it is necessary to extend two classes that are defined in CellDesigner's plugin API (Application Programming Interface). The Listings 2 to 3 on pages 20–21 show a very simple example of how to pass CellDesigner plugin model data structures to the translator in JSBML, which creates then a JSBML `Model` data structure. The examples described by Listings 2 to 3 on pages 20–21 create a plugin for CellDesigner, which displays the SBML data structure in a tree, like the example in Fig. **??** on page ??. 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 3 on page 21 is only completed by implementing the methods from the superclass. In this example it is sufficient to leave the implementation open.

4.5.3 JSBML's compatibility module for libSBML

To be described here.

```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 */
public class SimpleCellDesignerPluginAction extends PluginAction {

  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());
    }
  }

}
```

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

```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);
      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());
    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) { }
}
```

Listing 3: A simple example for a CellDesigner plugin using JSBML as a communication layer

A Frequently Asked Questions (FAQ)

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

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

- JSBML does not yet provide a complete validator for SBML.
- The support for SBML Level 3 should be completed, particularly extension packages.
- The `toSBML()` methods in `SBase` are still missing.
- Constructors and methods with namespaces are not yet provided.
- The libSBML compatibility module need to be implemented.

References

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

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

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

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

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