# libsbml Developer's Manual

Ben Bornstein

bornstei@cds.caltech.edu

The SBML Team
Control and Dynamical Systems, MC 107-81
California Institute of Technology, Pasadena, CA 91125, USA

http://www.sbml.org/

DRAFT

April 26, 2004

libsbml Version 2.0.3

This and other projects of the SBML Team are supported by the following organizations: the National Institutes of Health (USA); the International Joint Research Program of NEDO (Japan); the JST ERATO-SORST Program (Japan); the Japanese Ministry of Agriculture; the Japanese Ministry of Education, Culture, Sports, Science and Technology; the BBSRC e-Science Initiative (UK); the DARPA IPTO Bio-Computation Program (USA); the Air Force Office of Scientific Research (USA); the California Institute of Technology (USA); the University of Hertfordshire (UK); the Molecular Sciences Institute (USA); and the Systems Biology Institute (Japan).

SBML Systems Biology Markup Language

# Contents


**1 Quick Start** **3**
1.1 Linux, Solaris, Cygwin or MacOS X . . . 3
1.2 Windows . . . 3
**2 Introduction** **3**
**3 Detailed Installation Instructions** **5**
3.1 Unpacking and Configuring LIBSBML . . . 5
3.2 Building and Installing LIBSBML . . . 6
3.3 Additional Options for Configuring LIBSBML . . . 6
**4 SBML Classes in C** **7**
4.1 Primitive Types . . . 8
4.2 Object Creation and Destruction . . . 8
4.3 Accessing Fields . . . 10
4.4 Lists . . . 15
4.5 Enumerations . . . 16
4.6 Abstract Classes . . . 17
4.7 Fields Inherited from SBase . . . 17
4.8 Typecodes . . . 19
**5 Reading and Writing SBML Files** **20**
5.1 A Simple Example of Reading SBML . . . 22
5.2 XML Schema Validation . . . 22
5.3 Writing SBML Files . . . 24
**6 Handling of Mathematical Formulas and MathML** **25**
6.1 Reading and Writing Formulas in Text-String Form . . . 25
6.2 Reading Formulas in MathML Form: `MathMLDocument_t` and ASTs . . . 26
6.3 Differences between SBML Level 1 Formulas and MathML . . . 28
6.4 Additional Notes about the Handling of Mathematical Formulas . . . 28
**7 Levels of SBML** **30**
**8 Validation of SBML Models** **30**
**9 Special Considerations and Known Issues** **31**
9.1 Conformance to SBML . . . 31
9.2 Issues Related to XML Parsers . . . 32
**10 Acknowledgments** **32**
**A Lists and `ListOf_t`** **33**
**B Abstract Syntax Trees and `ASTNode_t`** **33**
B.1 Methods for Manipulating AST Nodes . . . 34
B.2 Notes about ASTNode . . . 38
**References** **39**

# 1 Quick Start

LIBSBML requires a separate XML library for low-level XML tokenizing and Unicode support. It currently supports the Xerces-C++ and Expat XML libraries on Linux, Solaris, Windows and MacOS X. You will first need to make sure one of these libraries is installed on your system. Many Linux systems provide one or both of these libraries either as part of their standard distribution or as an optional RPM or Debian package. For more information, see `http://xml.apache.org/xerces-c/` for Xerces and `http://expat.sf.net` for Expat.

## 1.1 Linux, Solaris, Cygwin or MacOS X

If you have obtained the source code distribution, then at your Unix, Cygwin or MacOS X command prompt, unpack the distribution, `cd` into the directory created (e.g., `libsbml-2.0.3/`), and type the following command to configure LIBSBML for your system:

```
./configure
```

To specify Expat explicitly rather than the LIBSBML default of Xerces, use a command such as the following instead (and make sure to read about the limitations surrounding the use of Expat explained in Section 3):

```
./configure --with-expat
```

Next, compile and install the LIBSBML library using the following command:

```
make
make install
```

To compile programs that use libsbml with GCC (for an example, see Section 5.1):

```
gcc -o myapp.c myapp.c -lsbml
```

## 1.2 Windows

Unzip the LIBSBML distribution and open the resulting folder (which will have a name such as `libsbml-2.0.3-expat` or `libsbml-2.0.3-xerces`). There are debug (`libsbmld`) and release (`libsbml`) versions of LIBSBML, with `.dll` and `.lib` files for both versions in the `win32` subdirectory of the LIBSBML distribution. Header files are located in the subdirectory `src/sbml`.

Users of Visual C++ should make their Visual C++ projects link with the files `libsbml.lib` or `libsbmld.lib` and generate code for the Multithreaded DLL or Debug Multithreaded DLL version of the VC++ runtime, respectively.

# 2 Introduction

This manual describes LIBSBML, a C application programming interface (API) library for reading, writing and manipulating the Systems Biology Markup Language (SBML; Hucka et al., 2001, 2003; Finney and Hucka, 2003). Currently, the library supports all of SBML Level 1 Version 1 and Version 2, and nearly all of SBML Level 2 Version 1. (The still-unimplemented parts of Level 2 are: support for RDF, and support for MathML's `semantics`, `annotation` and `annotation-xml` elements. These will be implemented in the near future.) For more information about SBML, please see the references or visit `http://www.sbml.org/` on the Internet. LIBSBML is entirely open-source under the terms of the GNU LGPL, and all source code and other materials are freely and publicly available.

Because the library provides a C API, the rest of this manual assumes familiarity with the C programming language. Some parts of the library were written in C++, however, experience with C++ is not required; its use is "hidden" behind C functions. A companion document (Bornstein, 2004) provides a detailed reference manual for the API.

Some of the technical features of LIBSBML include:

- ***Small Memory Footprint***: The parser is event-based (SAX2) and loads SBML data into C structures that mirror the classes in the SBML specification. No intermediate DOM is used, which greatly reduces runtime memory usage.
- ***Fast Runtime***: A 2000 reaction test file (`http://www.gepasi.org/gep3sbml.html`) loads in 1.18s on a 1 GHz AMD Athlon XP and uses 1.4Mb of memory.
- ***Portable***: The C source code is pure ISO standard C for maximum portability. It produces no errors or warnings when compiled with either GCC or Microsoft Visual C++. GCC compilation flags are: `-ansi -pedantic-errors -Wall`, i.e., ISO violations are compilation errors instead of warnings and all warnings are reported. The build system uses the GNU Autotools (Autoconf, Automake, and Libtool) to build shared and static libraries on a variety of Unix-based platforms. Microsoft Windows is supported either through the Cygwin environment or as a native Win32 Dynamic Link Library (DLL). A Microsoft Visual C++ 6.0 project file is included in the source distribution and precompiled Windows distributions are available.
- ***Full SBML Support***: All constructs in SBML Level 1 (Versions 1 and 2) and SBML Level 2 are supported, with the exceptions noted above (i.e., RDF, and three rarely-used MathML constructs). The exceptions will be removed in the near future. LIBSBML handles such SBML differences as the alternate spellings of *species* and *annotation* between the SBML specifications.

  The full-text of `<notes>` and `<annotation>` elements (the latter including namespace declarations) may be retrieved from any SBML object. For compatibility with some technically incorrect but popular Level 1 documents, the parser recognizes and stores notes and annotations defined for the top-level `<sbml>` element (logging a warning).
- ***Full XML Schema Validation***: The library can use the Apache Xerces-C++ XML library, which supports full XML Schema validation. All XML and Schema warning, error and fatal error messages are logged with line and column number information and may be retrieved and manipulated pragmatically. The XML Schema file used by the parser for validation is configurable.
- ***Full Unicode Support***: SBML Documents are parsed and manipulated in the Unicode codepage for efficiency (this is Xerces-C++ native format); however, strings are transcoded to the local code page for the SBML C interface data structures.
- ***Well Tested***: The entire library was written using the test-first approach popularized by Kent Beck and eXtreme Programming, where it's one of the 12 principles. Currently the LIBSBML code base features 3340 individual assertions in 636 functional unit tests. Four test cases are responsible for reading entire SBML files (three are examples from the SBML Level 1 specification) into memory and verifying every field of the resulting structures.
- ***Conscientious Use of Memory***: All memory for and contained in SBML structures is C memory, so `realloc()` and `free()` may be used. This is an important distinction when working with mixed C and C++ code. C's `malloc()` and `free()` are not compatible with C++'s `new` and `delete` operators.

  A custom memory trace facility can be used to track all memory allocated and freed in both the library and all test suites. This facility must be enabled at build time with `./configure --enable-memory-tracing`. For performance reasons memory tracing should be turned off in production environments.

# 3 Detailed Installation Instructions

The LIBSBML distributions can be downloaded from the *SBML* project area on SourceForge.net, at http://sf.net/projects/sbml. A link to the download area is also provided on the SBML project home page, http://www.sbml.org.

LIBSBML depends on a separate XML library for low-level XML tokenizing and Unicode support. Currently, LIBSBML can use either Apache's Xerces-C++ XML library or James Clark's Expat XML library on Linux, Solaris, Windows and MacOS X. You will first need to make sure one of these libraries is installed on your system. Many systems provide one or both of these libraries either as part of their standard distribution or as an optional RPM or Debian package. DLL and LIB file distributions are available for both Xerces and Expat. For more information, see the following resources:

- Xerces: http://xml.apache.org/xerces-c/
- Expat: http://expat.sf.net

Note that if you use Expat instead of Xerces, LIBSBML will not be able to perform certain kinds of validation tests on SBML models, because Expat is not a validating XML parser. A validating parser can check XML input against an XML Schema. Xerces is currently the only validating parser library usable with LIBSBML. Using Expat with LIBSBML effectively means it will be unable to check SBML input against the SBML XML Schema. The tradeoff is that Expat can be noticeably faster at parsing large models than Xerces.

Unless instructed otherwise, the LIBSBML build process will **default to using Xerces**. A good way to determine whether Xerces-C++ is installed on your system is to run the configuration command (see below); it will halt if it cannot find the Xerces-C++ library. You can provide the configuration command with a flag telling it to use Expat instead of Xerces and you can indicate where the libraries are located if they are not in standard locations on your system.

LIBSBML is designed to be extremely portable. It is written in 100% pure ISO C and the build system uses the GNU Autotools (Autoconf, Automake and Libtool).

## 3.1 Unpacking and Configuring libsbml

First, uncompress and unpack the LIBSBML sources. Then, configure the package by typing the following command at your Unix, Cygwin or MacOS X command prompt:

```
./configure
```

To specify Expat explicitly rather than the LIBSBML default of Xerces, use a command such as the following instead:

```
./configure --with-expat
```

If either Expat or Xerces is installed in a non-standard location on your computer system (e.g., a home directory), `configure` will not be able to detect it. In this case, `configure` needs to be told explicitly where to find the libraries. Use the following forms:

```
./configure --with-expat=dir
```

or

```
./configure --with-xerces=dir
```

where *`dir`* is the **parent** directory of where the `include` and `lib` directories of Xerces or Expat (whichever one you are trying to use) is located. For example, on MacOS X, if you used Fink to install Expat in Fink's default software tree, you would configure LIBSBML using the following:

```
./configure --with-xerces=/sw
```

During the installation phase (i.e., during `make install`, discussed below), the LIBSBML installation commands will copy header files to `/usr/local/include/sbml` and (shared and static) library files to `/usr/local/lib`, by default. To specify a different installation location, use the `--prefix` argument to `configure`. For example,

```
./configure --prefix=/my/favorite/path
```

Of course, you can combine the flags to `configure`, giving both `--prefix` and `--with-expat` or `--with-xerces` to set both options. Other options are described below.

## 3.2 Building and Installing libsbml

Once configured, building should be very easy. Simply execute the following commands at your Unix, Cygwin or MacOS X command prompt:

```
make
make install
```

If all went as it should, LIBSBML should end up compiled and installed on your system, in either the default location (`/usr/local/`) or in the location you indicated during the configuration step as explained above. And that that's all that should be required to build and install LIBSBML normally!

## 3.3 Additional Options for Configuring libsbml

In addition to the `--prefix`, `--with-expat` and `--with-xerces` options already described, the LIBSBML configuration command supports the options described below.

### 3.3.1 Unit Testing

LIBSBML provides built-in facilities for testing itself. To run the unit tests, a second library is required, `libcheck`. Check is a very lightweight C unit test framework based on the xUnit framework popularized by Kent Beck and eXtreme Programming. Check is quite small and once installed, it consists of only two files: `libcheck.a` and `check.h`. To download Check, visit:

- http://check.sf.net/

*Note:* Debian users can find Check as a standard add-on package (.deb). MacOS X users can find and install Check using the Fink system.

To enable the unit testing facilities in LIBSBML, add the `--with-check` flag to the configure command:

```
./configure --with-check
```

Following this, you must build LIBSBML and then you can run the tests:

```
make
make check
```

The `make check` step is optional and will build and run an extensive suite of unit tests to verify all facets of the library. These tests are meant primarily for developers of LIBSBML and running them is not required for the library to function properly. All tests should pass with no failures or errors. If for some reason this is not the case on your system, please submit a bug report using the facilities at http://www.sf.net/projects/sbml.

### 3.3.2 Memory Tracing

In addition to the unit tests, a custom memory tracing facility is available. It is disabled by default and must be enabled explicitly at build time, either as an argument to configure:

```
./configure --enable-memory-tracing
```

or, in your own projects, by defining the C preprocessor symbol `TRACE_MEMORY`:

```
#define TRACE_MEMORY
```

With memory tracing turned on, every piece of memory in both the library and all test suites is tracked. At the end of the test run, statistics are printed on total memory allocations, deallocations and leaks. The memory statistics for the test suites should report zero leaks. If for some reason this is not the case, please submit a report at http://www.sf.net/projects/sbml.

For performance reasons, memory tracing should be disabled in production environments. It is disabled by default in LIBSBML, but if enabled it, you can reconfigure and disable it as follows:

```
./configure --disable-memory-tracing
```

# 4 SBML Classes in C

The SBML specification, with its UML diagrams, suggests an object-oriented (OO) design. An API for interacting with SBML would do well to use an object-oriented programming (OOP) style to lower the inevitable impedance mismatch between specification and implementation. Unfortunately, the C programming language was not designed with OOP in mind and therefore does not support many object-oriented concepts. It is possible, however, to construct a minimal object-like system in C with few, if any, drawbacks. For these reasons, the LIBSBML API mimics an object-oriented programming style.

The particular OOP-like style used by LIBSBML is not revolutionary. In fact, it is quite common and comprised of only a few simple stylistic conventions:

1. SBML classes are represented as C structs with a typedef shorthand. The shorthand form is derived by appending `_t` to the name of the SBML class, e.g., `Model` becomes `Model_t`.

2. C "objects" are nothing more than pointers to specific C structs in memory. These pointers, instead of the structs themselves, are passed to and returned from "methods" (functions).

3. Functions meant to represent methods of (or messages to) an object are named beginning with the SBML class, followed by an underscore and ending in the method name. The functions take the object (pointer to struct) receiving the method as their first argument. For example, the function prototype for the `addCompartment()` method of a `Model` is:

   ```
   void Model_addCompartment(Model_t *m, Compartment_t *c);
   ```

4. Constructor and destructor names are similar to method names, but end in `_create()` and `_free()`, respectively.

Every SBML class defined in the specification has a corresponding C class (see Table 1 on the following page). The two SBML enumeration types, `UnitKind` and `RuleType` are represented as C enumerations, but deviate slightly from the rules above (see Section 4.5). Finally, there is one class, `SBMLDocument_t`, that exists in the LIBSBML API, but not in SBML Level 1 (though the equivalent exists in Level 2). It serves as a top-level container for models and stores warnings and error messages encountered when an SBML document was read (see Section 5).

The methods for SBML classes are declared in header files that correspond to the class name (e.g., `Model.h`). To include all methods for all classes in one fell swoop, `#include SBMLTypes.h`.

| SBML Class | C Class (typedef struct) |
|---|---|
| *SBase* | `SBase_t` |
| Model | `Model_t` |
| FunctionDefinition | `FunctionDefinition_t` |
| UnitDefinition | `UnitDefinition_t` |
| Unit | `Unit_t` |
| Compartment | `Compartment_t` |
| Parameter | `Parameter_t` |
| Species | `Species_t` |
| Reaction | `Reaction_t` |
| SpeciesReference | `SpeciesReference_t` |
| ModifierSpeciesReference | `ModifierSpeciesReference_t` |
| SimpleSpeciesReference | `SimpleSpeciesReference_t` |
| KineticLaw | `KineticLaw_t` |
| *Rule* | `Rule_t` |
| *AssignmentRule* | `AssignmentRule_t` |
| RateRule | `RateRule_t` |
| AlgebraicRule | `AlgebraicRule_t` |
| CompartmentVolumeRule | `CompartmentVolumeRule_t` |
| ParameterRule | `ParameterRule_t` |
| SpeciesConcentrationRule | `SpeciesConcentrationRule_t` |
| Event | `Event_t` |
| EventAssignment | `EventAssignment_t` |

| SBML Enumeration | C Enumeration (typedef enum) |
|---|---|
| UnitKind | `UnitKind_t` |
| RuleType | `RuleType_t` |

**Table 1:** *SBML classes and enumerations and their corresponding C class. Italicized classes are abstract, which sets their C implementation slightly apart from the others. See Section 4.6 for more information.*

## 4.1 Primitive Types

The mapping from SBML primitive types to C is straightforward, as an example will help illustrate. A Species has at least one attribute of every primitive type defined by SBML. Figure 1 shows the UML definition for Species and the corresponding C struct side-by-side. The similarity between the two demonstrates the mapping rules for primitive types:

1. In all cases, the form of UML attribute names, including their capitalization, are preserved (e.g., `initialAmount`) when mapped to C struct fields. The names of getters and setters (see below) reflect these names.

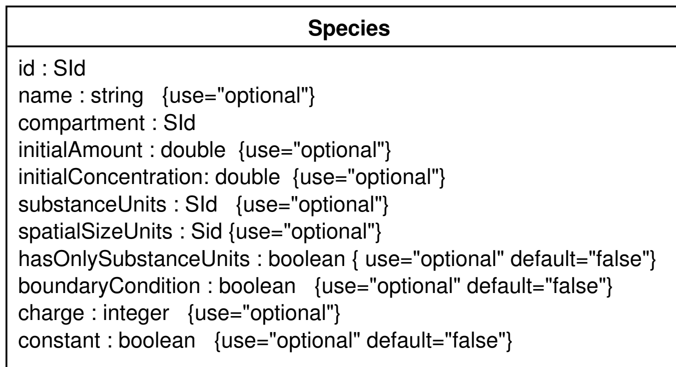
2. SName (in SBML Level 1) and SId (in SBML Level 2) are mapped to standard C strings (pointers to arrays of `char` terminated by a `NULL` or `0` character; e.g., `char *name`). Note that the syntax of SName and SId is not yet enforced in the API.

3. SBML types double and integer are mapped to C `double` and `int` respectively.

4. Boolean is mapped to C `int`, where zero represents false and non-zero represents true.

## 4.2 Object Creation and Destruction

This section and subsequent ones focus on functions or methods to create, destroy and otherwise manipulate SBML C objects. Since all functions and methods follow the same naming convention, when discussing them generically, `XXX` will be used to stand for some class name and `YYY` some class attribute.




```
typedef struct
{
  SBASE_FIELDS;
  char   *id;
  char   *name;
  char   *compartment;
  union
  {
    double Amount;
    double Concentration;
  } initial;
  char   *substanceUnits;
  char   *spatialSizeUnits;
  int     hasOnlySubstanceUnits;
  int     boundaryCondition;
  int     charge;
  int     constant;
} Species_t;
```

**Figure 1:** *Example: the definition of SBML's Species in UML (left) and the corresponding* `Species_t` *C struct (right) in* libsbml. *SBASE_FIELDS is part of the OOP-like style used to implement objects in C; it is a macro that expands into the fields defined by SBase. The use of a union for amount and concentration reflects that these two fields are mutually exclusive in the SBML Species definition.*

To instantiate (create) an object use either the `XXX_create()` or `XXX_createWith()` constructor. To destroy (free) an object use `XXX_free()`.

To give a concrete example, the following are the constructors and destructors for SBML's Species objects. (The complete list of API methods for `Species_t` and other data objects in libsbml is available in the libsbml API Reference Manual.)

**Species_t *Species_create (void)**

Creates a new Species and returns a pointer to it.

**Species_t *Species_createWith (const char *name, const char *compartment, double initialAmount, const char *substanceUnits, int boundaryCondition, int charge)**

Creates a new Species object with the given `name`, `compartment`, `initialAmount`, `substanceUnits`, `boundaryCondition` and `charge` and returns a pointer to it. This convenience function is functionally equivalent to the following:

```
Species_t *s = Species_create();
Species_setId(s, id); Species_setCompartment(s, compartment); ...;
```

**void Species_free (Species_t *s)**

Frees the given Species.

The `XXX_createWith()` constructors are a convenient way both to create SBML objects and initialize many of their attributes in a single operation. If `XXX_create()` is used instead, only attributes with default values (as defined by the SBML specification) will be set. All other attributes will be marked as not having been set.

When an SBML object is destroyed with `XXX_free()`, all of its strings are freed (see Section 4.3 for more information) and all of its contained objects are freed (see Section 4.4 for more information).

## 4.3 Accessing Fields

Accessing fields in data structures is accomplished using functions that offer interfaces to getting and setting the values of the fields. The generic form of these is discussed in this section. To give concrete examples, we repeatedly use the SBML Species class of objects.

### 4.3.1 Getters

The getter methods follow the naming convention `XXX_getYYY()`. To give a concrete example, here are the getters for `Species_t`:

**const char * Species_getId (const Species_t *s)**

Returns the `id` field of this Species.

**const char * Species_getName (const Species_t *s)**

Returns the `name` field of this Species.

**const char * Species_getCompartment (const Species_t *s)**

Returns the `compartment` field of this Species.

**double Species_getInitialAmount (const Species_t *s)**

Returns the `initialAmount` field of this Species.

**double Species_getInitialConcentration (const Species_t *s)**

Returns the `initialConcentration` field of this Species.

**const char * Species_getSubstanceUnits (const Species_t *s)**

Returns the `substanceUnits` field of this Species.

**const char * Species_getSpatialSizeUnits (const Species_t *s)**

Returns the `spatialSizeUnits` field of this Species.

**const char * Species_getUnits (const Species_t *s)**

Returns the `units` field of this Species (SBML Level 1 only).

**int Species_getHasOnlySubstanceUnits (const Species_t *s)**

Returns true if this Species' `hasOnlySubstanceUnits` field is true, false (`0`) otherwise.

**int Species_getBoundaryCondition (const Species_t *s)**

Returns the `boundaryCondition` field of this Species.

**int Species_getCharge (const Species_t *s)**

Returns the `charge` field of this Species.

**int Species_getConstant (const Species_t *s)**

Returns true (non-zero) if this Species is constant, false (0) otherwise.

Notice the `Species_t` passed to each getter is `constant`. The purpose of this `const`ness is twofold: (1) it reinforces the notion that a getter simply returns a value and does not modify the state of the passed-in object and (2) as a result, in certain contexts a compiler may be able to use this information to perform certain optimizations. Notice also, whenever a getter returns a string, it is constant (`const char *`); i.e., it cannot be modified or freed. The reason for this is each struct tracks and owns all of its internal memory. To modify (or especially free) this memory without using one of the sanctioned access methods could be particularly disasterous (most likely resulting in a segmentation or general protection fault). Memory management issues are elaborated in the discussion of setters in the next section.

Figure 2 provides an example of using getters.

```
1  /**
2   * Prints some basic information about an SBML Level 2 Species.
3   */
4  void
5  myPrintSpecies (Species_t *s, FILE *stream)
6  {
7    if (s == NULL)
8    {
9      fprintf(stream, "Null species pointer\n");
10     return;
11   }
12
13   const char none[] = "(none)";
14   const char *id = Species_getId(s);
15   const char *name = Species_getName(s);
16   const char *comp = Species_getCompartment(s);
17
18   fprintf(stream, "    Species id: %s\n", id   != NULL ? id : none);
19   fprintf(stream, "          name: %s\n", name != NULL ? name : none);
20   fprintf(stream, "compartment id: %s\n", comp != NULL ? comp : none);
21 }
```

**Figure 2:** *Demonstrates accessing the fields of an SBML Level 2 object (in this case,* `Species_t`*) using getter methods.*

### 4.3.2 Setters

A value is assigned to a field via a *set* method. Requiring all assignments to be done using setter methods allows LIBSBML to track (and the developer to query) the set or unset *state* of a field apart from its actual *value*. The need to distinguish between state and value is critical and is discussed further in Section 4.3.3. (Earlier versions of LIBSBML allowed primitive types to be set directly; however, direct access made it impossible to distinguish between set and unset states of a field, since all possible values are valid—no sentinel value exists to indicate an unset state.)

The setter methods follow the naming convention `XXX_setYYY()`. The setters for `Species_t` are:

**void Species_setId (Species_t *s, const char *sid)**

Sets the `id` field of this Species to a copy of `sid`.

**void Species_setName (Species_t *s, const char *string)**

Sets the `name` field of this Species to a copy of `string` (which must be conform to `SName` syntax).

**void Species_setCompartment (Species_t *s, const char *sid)**

Sets the `compartment` field of this Species to a copy of `sid`.

**void Species_setInitialAmount (Species_t *s, double value)**

Sets the `initialAmount` field of this Species object to `value` and marks the field as set. This method also unsets the `initialConentration` field of the Species object.

**void Species_setInitialConcentration (Species_t *s, double value)**

Sets the `initialConcentration` field of this Species to `value` and marks the field as set. This method also unsets the `initialAmount` field.

**void Species_setSubstanceUnits (Species_t *s, const char *sid)**

Sets the `substanceUnits` field of this Species to a copy of `sid`.

**void Species_setSpatialSizeUnits (Species_t *s, const char *sid)**

Sets the `spatialSizeUnits` field of this Species to a copy of `sid`.

**void Species_setUnits (Species_t *s, const char *sname)**

Sets the `units` field of this Species to a copy of `sname` (L1 only).

**void Species_setHasOnlySubstanceUnits (Species_t *s, int value)**

Sets the `hasOnlySubstanceUnits` field of this Species to `value` (boolean).

**void Species_setBoundaryCondition (Species_t *s, int value)**

Sets the `boundaryCondition` field of this Species to `value` (boolean).

**void Species_setCharge (Species_t *s, int value)**

Sets the `charge` field of this Species to `value` and marks the field as set.

**void Species_setConstant (Species_t *s, int value)**

Sets the `constant` field of this Species to `value` (boolean).

In the case of strings, requiring setter methods also enables clean and simple memory semantics. The rule is: every SBML object is responsible for its own memory, including SId and SName strings. Whenever a set method is called, the passed-in string is copied and stored. If the field being set previously contained a string, it is freed. When `XXX_free()` is called, all strings are freed.

For example, to set the compartment of a Species object stored in variable `s` to the string `"cell"`, you could do the following:

```
Species_setCompartment(s, "cell");
```

The effect of passing a `NULL` pointer as the string argument is to free the previously stored string and mark the field as unset. The preferred method for doing this, however, is to use the `XXX_unsetYYY()` class of methods (see Section 4.3.3).

### 4.3.3 Field States

For each optional field **without a default value**, LIBSBML tracks both its state and value. The state of a field indicates whether the field is set (contains a valid value) or unset (contains no value at all). As mentioned before, the distinction between a set and unset field is critical for both LIBSBML and applications that depend upon it to function correctly (in accordance with the SBML specifications).

Take, for example, the case of outputting SBML for a species. The SBML Species object has an optional field named `charge` with no defined default value. Because it's optional, it need not ever be read in (specified), written or manipulated. It may not have a value for a given species in a given model. Upon writing out the definition of the species in a model, LIBSBML must be able to determine whether the field has ever been set in order to know whether to output or omit the field while writing the model.

To determine whether a particular field in a structure is set or unset, calling programs should use LIBSBML's `XXX_isSetYYY()` class of methods. For `Species_t`, the following are available:

**int Species_isSetId (const Species_t *s)**

Returns `1` if the `id` field of this Species has been set, `0` otherwise.

**int Species_isSetName (const Species_t *s)**

Returns `1` if the `name` of this Species has been set, `0` otherwise.
In SBML Level 1, a Species name is required and therefore **should always be set**. In Level 2, the name is optional and as such may or may not be set.

**int Species_isSetCompartment (const Species_t *s)**

Returns `1` if the `compartment` field of this Species has been set, `0` otherwise.

**int Species_isSetInitialAmount (const Species_t *s)**

Returns `1` if the `initialAmount` of this Species has been set, `0` otherwise.
In SBML Level 1, a Species `initialAmount` is required and therefore **should always be set**. In Level 2, the `initialAmount` field value is optional and as such may or may not be set.

**int Species_isSetInitialConcentration (const Species_t *s)**

Returns `1` if the `initialConcentration` of this Species has been set, `0` otherwise.

**int Species_isSetSubstanceUnits (const Species_t *s)**

Returns `1` if the `substanceUnits` of this Species has been set, `0` otherwise.

**int Species_isSetSpatialSizeUnits (const Species_t *s)**

Returns `1` if the `spatialSizeUnits` of this Species has been set, `0` otherwise.

**int Species_isSetUnits (const Species_t *s)**

Returns `1` if the `units` of this Species has been set, `0` otherwise (SBML Level 1 only).

**int Species_isSetCharge (const Species_t *s)**

Returns `1` if the `charge` of this Species has been set, `0` otherwise.

Fields with default values do not have a `isSetYYY()` method. If the value for such a field is never supplied by an SBML document or user, the default is used. Therefore, if an `isSetYYY()` method did exist, it would always return true (`1`).

Required fields, on the other hand, *do* have `isSetYYY()` methods. There are two points worth mentioning here. First, it is possible that a value for a required field is not given and a program may want to check for and handle this case (especially if the program is an SBML validator). Second, please be aware that in the transition from SBML Level 1 to Level 2, some fields changed from being required to being optional. If this is the case for a particular field, the documentation for the corresponding `isSetYYY()` will state it (as above).

Just as fields may be set and their set state queried, they may also be unset. Unset methods are named (predictably) `XXX_unsetYYY()`. The methods for unsetting fields in Species are:

**void Species_unsetName (Species_t *s)**

Unsets the `name` field of this Species.
In SBML Level 1, a Species `name` is required and therefore **should always be set**. In Level 2, `name` is optional and as such may or may not be set.

**void Species_unsetInitialAmount (Species_t *s)**

Unsets the `initialAmount` field of this Species.
In SBML Level 1, a Species `initialAmount` is required and therefore **should always be set**. In Level 2, `initialAmount` is optional and as such may or may not be set.

**void Species_unsetInitialConcentration (Species_t *s)**

Unsets the `initialConcentration` field of this Species.

**void Species_unsetSubstanceUnits (Species_t *s)**

Unsets the `substanceUnits` field of this Species.

**void Species_unsetSpatialSizeUnits (Species_t *s)**

Unsets the `spatialSizeUnits` field of this Species.

**void Species_unsetUnits (Species_t *s)**

Unsets the `units` field of this Species (Level 1 only).

**void Species_unsetCharge (Species_t *s)**

Unsets the `charge` field of this Species.

Again, for the reason mentioned above, fields with default values do not have `unsetYYY()` methods. Similarly, required fields have `unsetYYY()` methods only if they are declared optional in at least one of SBML Level 1 and Level 2. Notice, for example, there is an `isSetCompartment()` method but no corresponding `unsetCompartment()` (because a compartment is required for a Species in both SBML Level 1 and Level 2).

## 4.4 Lists

Species only contains fields having types SId, SName and primitive types, but many SBML classes also contain lists of other objects. For example, a UnitDefinition contains a list of Units, as shown in Figure 3.

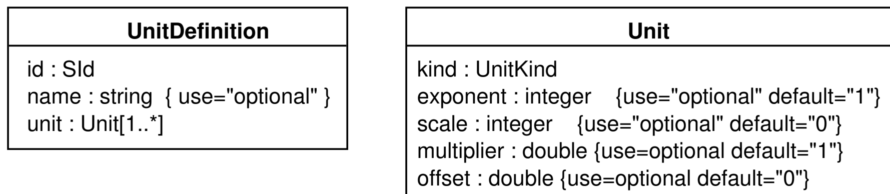



**Figure 3:** *SBML Level 2's UnitDefinition and Unit.*

To help manage this containment relationship, three standard functions are provided by LIBSBML: `XXX_addYYY()`, `XXX_getYYY()` and `XXX_getNumYYY()`. For example, the methods for UnitDefinition are:

**void UnitDefinition_addUnit (UnitDefinition_t *ud, Unit_t *u)**

Adds the given Unit to this UnitDefinition.

**Unit_t *UnitDefinition_getUnit (const UnitDefinition_t *ud, unsigned int n)**

Returns the nth Unit of this UnitDefinition.

**unsigned int UnitDefinition_getNumUnits (const UnitDefinition_t *ud)**

Return the number of Units in this UnitDefinition.

Furthering the example, creating the `UnitDefinition` $mmoll^-1s^1$ with an identifer of "`mmls`", corresponding to the following SBML,

```
<listOfUnitDefinitions>
  <unitDefinition id="mmls">
    <listOfUnits>
      <unit kind="mole" scale="-3"/>
      <unit kind="litre" exponent="-1"/>
      <unit kind="second" exponent="-1"/>
    </listOfUnits>
  </unitDefinition>
</listOfUnitDefinitions>
```

could be accomplished with the following C:

```
UnitDefinition_t *ud = UnitDefinition_createWith("mmls");

UnitDefinition_addUnit(ud, Unit_createWith(UNIT_KIND_MOLE , 1, -3) );
UnitDefinition_addUnit(ud, Unit_createWith(UNIT_KIND_LITRE , -1, 0) );
UnitDefinition_addUnit(ud, Unit_createWith(UNIT_KIND_SECOND, -1, 0) );
```

List items are numbered starting at zero. In the case above, `UnitDefinition_getNumUnits(ud)` returns 3 and `UnitDefinition_getUnit(ud, 1)` returns the *second* Unit structure. (The `UNIT_KIND_XXX` enumerations are discussed later.)

Related to lists is a set of convenience methods for creating and adding SBML objects to a Model object in a single operation. The rationale is that since a Model is the top-level container for all other SBML objects, programmers are likely to have handles to them. Another way to construct the above `UnitDefinition_t` object, but this time inside a `Model_t`, is:

```
Model_t          *m  = Model_createWith("MyModel");
UnitDefinition_t *ud = Model_createUnitDefinition(m);

UnitDefinition_setName(ud, "mmls");

Model_createUnit(m, Unit_createWith(UNIT_KIND_MOLE , 1, -3) );
Model_createUnit(m, Unit_createWith(UNIT_KIND_LITRE , -1, 0) );
Model_createUnit(m, Unit_createWith(UNIT_KIND_SECOND, -1, 0) );
```

`Model_createUnit()` creates a new Unit inside the Model `m` and returns a pointer to it (in this case the result is discarded). The `Unit_t` is added to the last `UnitDefinition_t` created. One caveat to be aware of with these methods is the case where no intermediate container exists; e.g., if no `UnitDefinition_t` were created above. In that case, the call to `Model_createUnit()` does nothing. More specifically, no `Unit_t` is created, nothing is added to the model, and `NULL` is returned.

For more detailed information on lists in LIBSBML and the `ListOf_t` utility type provided in the library, see Appendix A.

## 4.5 Enumerations

SBML has two enumeration types, UnitKind and RuleType (the latter only for SBML Level 1). These translate directly to C `enums` with a few support functions for equality testing and converting to and from strings.

```
typedef enum
{
    UNIT_KIND_AMPERE
  , UNIT_KIND_BECQUEREL

   /* Omitted for space */

  , UNIT_KIND_WEBER
  , UNIT_KIND_INVALID
} UnitKind_t;
```

The following are the methods available for UnitKind:

**int UnitKind_equals (UnitKind_t uk1, UnitKind_t uk2)**

Tests for logical equality between two UnitKinds. This function behaves exactly like C's `==` operator, except for the following two cases:

- `UNIT_KIND_LITER == UNIT_KIND_LITRE`
- `UNIT_KIND_METER == UNIT_KIND_METRE`

where C would yield false (since each of the above is a distinct enumeration value), `UnitKind_equals(...)` yields true. Returns true (`!0`) if uk1 is logically equivalent to uk2, false (`0`) otherwise.

**UnitKind_t UnitKind_forName (const char *name)**

Returns the UnitKind with the given name (case-insensitive).

**const char *UnitKind_toString (UnitKind_t uk)**

Returns the name of the given UnitKind. The caller does not own the returned string and is therefore not allowed to modify it.

The last item in the enumeration, `UNIT_KIND_INVALID`, is used whenever, as the name implies, the `UnitKind` is invalid or unknown. The corresponding string representation is "(Invalid UnitKind)". When a Unit is created, its kind field is initialized to `UNIT_KIND_INVALID`. Also, `UnitKind_forName()` will return `UNIT_KIND_INVALID` if the passed-in name does not match any known `UnitKind`.

The same ideas apply to RuleType, except there is no need for `RuleType_equals()`. See `RuleType.h` for more information.

*Implementation Note*: The internal `UNIT_KIND_STRINGS` table is sorted alphabetically and `UnitKind_t` matches this sort order. Because of this, `UnitKind_forName()` is able to perform a binary search to find a matching name, making its complexity $O(log(n))$. That is, `UnitKind_forName()` is implemented efficiently.

## 4.6 Abstract Classes

The SBML specification defines three classes that have no representation apart from subclasses that specialize (inherit from) them. In OOP parlance, these types are termed abstract. The abstract SBML classes are listed in Table 2.

| SBML Class | C Class (typedef struct) | SBML Level |
|---|---|---|
| *SBase* | `SBase_t` | all |
| *Rule* | `Rule_t` | all |
| *AssignmentRule* | `AssignmentRule_t` | Level 1 |
| *SimpleSpeciesReference* | `SimpleSpeciesReference_t` | Level 2 |

**Table 2:** *Abstract SBML classes their corresponding C class. Although all classes are present in* LIBSBML *at the same time, some of the classes only have meaning for certain levels of SBML.*

The conventions for abstract classes in the LIBSBML API are similar to that of other classes with a few modifications and additions.

Since abstract classes cannot be created or destroyed directly, they have no `XXX_create()` or XXX_free() methods. Instead they have `XXX_init()` and `XXX_clear()` methods which subclasses use to initialize and free their memory, respectively. Users of the API do not need to worry about the create and free operations on these classes.

## 4.7 Fields Inherited from SBase

Every major structure in SBML is derived from an abstract base type called SBase. Figure 4 shows the pseudo-UML definition of SBase itself, while Figure 5 on the next page shows the overall inheritance hierarchy of SBML. In addition to the relationships shown in Figure 5, all substructures such as `trigger` on `Event` and the `listOf______` lists in SBML are also derived from `SBase`.

| ***SBase*** |
|---|
| metaid : ID {use="optional"}<br>notes : (ANY : {namespace="http://www.w3.org/1999/xhtml"}) {minOccurs="0"}<br>annotation : (ANY) {minOccurs="0"} |

**Figure 4:** *The definition of* `SBase` *in SBML Level 2. See the SBML specifications for an explanation of the notation.*

The practical implication is that every class has methods for working with the `metaid`, `notes` and `annotation` fields. However, the methods to work with these fields in LIBSBML are generic:

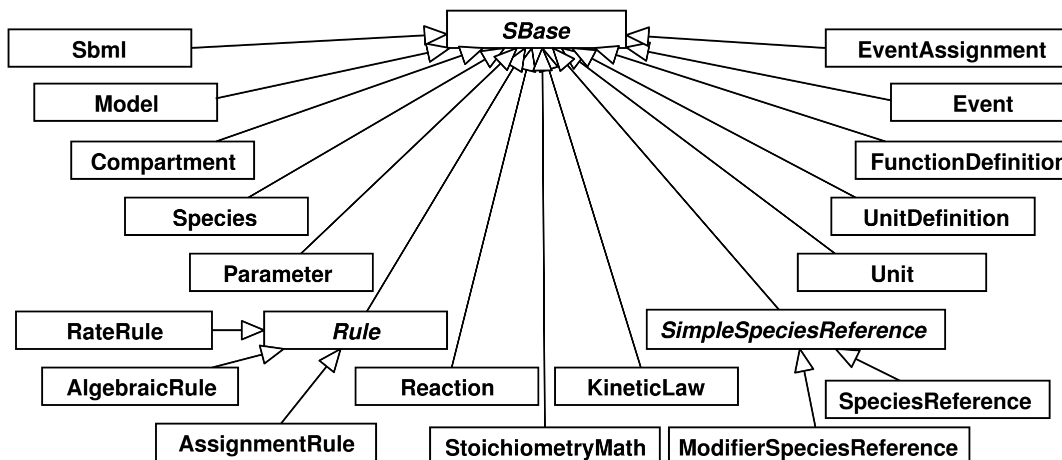



**Figure 5:** *A UML diagram of the inheritance hierarchy of major data types in SBML. Open arrows indicate inheritance, pointing from inheritors to their parents. In addition to these types, all substructures in SBML (including, for example, all the `listOf` lists) are also derived from `SBase`.*

**const char * SBase_getMetaId (const SBase_t *sb)**

Returns the `metaid` field for this SBML object.

**const char * SBase_getNotes (const SBase_t *sb)**

Returns the `notes` field for this SBML object.

**const char * SBase_getAnnotation (const SBase_t *sb)**

Returns the `annotation` field for this SBML object.

**unsigned int SBase_getColumn (const SBase_t *sb)**

Returns the column number for this SBML object.

**unsigned int SBase_getLine (const SBase_t *sb)**

Returns the line number for this SBML object.

**int SBase_isSetMetaId (const SBase_t *sb)**

Returns `1` if the `metaid` for this SBML object has been set, `0` otherwise.

**int SBase_isSetNotes (const SBase_t *sb)**

Returns `1` if the `notes` for this SBML object has been set, `0` otherwise.

**int SBase_isSetAnnotation (const SBase_t *sb)**

Returns `1` if the `annotation` for this SBML object has been set, `0` otherwise.

**void SBase_setMetaId (SBase_t *sb, const char *metaid)**

Sets the `metaid` field of the given SBML object to a copy of metaid. If object already has a metaid, the existing string is freed before the new one is copied.

**void SBase_setNotes (SBase_t *sb, const char *notes)**

Sets the notes field of the given SBML object to a copy of notes. If object already has notes, the existing string is freed before the new one is copied.

**void SBase_setAnnotation (SBase_t *sb, const char *annotation)**

Sets the annotation field of the given SBML object to a copy of annotations. If object already has an annotation, the existing string is freed before the new one is copied.

**void SBase_unsetMetaId (SBase_t *sb)**

Unsets the metaid for this SBML object.

**void SBase_unsetNotes (SBase_t *sb)**

Unsets the notes for this SBML object.

**void SBase_unsetAnnotation (SBase_t *sb)**

Unsets the annotation for this SBML object.

The first argument to these functions is, of course, an object of type SBase. Since Species inherits from SBase, i.e., `Species_t` *is an* SBase, it can be used as the first argument to these functions. Beware, however, that a cast is required. For example, to set the notes field of some `Species` held in variable `s`, cast the variable to type `SBase_t`:

```
SBase_setNotes( (SBase_t *) s, "My Favorite Species" );
```

The same applies to all other SBML objects.

## 4.8 Typecodes

Finally, each SBML class has a `typecode` that is initialized when an object is instantiated. The `typecode` is a simple C enumeration, defined in `SBMLTypeCodes.h` (which is included by the main LIBSBML include file, `SBMLTypes.h`, so client code does not need to include it separately):

```
/**
 * An enumeration of SBML Level 2 types to help identify SBML objects at runtime.
 * Abstract types do not have a typecode since they cannot be instantiated.
 */
typedef enum
{
    SBML_COMPARTMENT
  , SBML_DOCUMENT
  , SBML_EVENT
  , SBML_EVENT_ASSIGNMENT
  , SBML_FUNCTION_DEFINITION
  , SBML_KINETIC_LAW
  , SBML_LIST_OF
  , SBML_MODEL
  , SBML_PARAMETER
  , SBML_REACTION
  , SBML_SPECIES
  , SBML_SPECIES_REFERENCE
  , SBML_MODIFIER_SPECIES_REFERENCE
  , SBML_UNIT_DEFINITION
  , SBML_UNIT
  , SBML_ALGEBRAIC_RULE
```

```
    , SBML_ASSIGNMENT_RULE
    , SBML_RATE_RULE
    , SBML_SPECIES_CONCENTRATION_RULE
    , SBML_COMPARTMENT_VOLUME_RULE
    , SBML_PARAMETER_RULE
  } SBMLTypeCode_t;
```

The primary reason for the `typecode` is distinguish specific types of rules in a Model. A `Model_t` contains a list of rules, but a `Rule_t` in SBML Level 1 may be of one of four specific types: `AlgebraicRule`, `SpeciesConcentrationRule`, `CompartmentVolumeRule` and `ParameterRule`. Having a type code associated with each object allows calling programs to distinguish its type. Without type codes, it would be impossible.

# 5 Reading and Writing SBML Files

SBML may be read from a file or an in memory string into an SBMLDocument. LIBSBML defines two basic read functions:

**SBMLDocument_t *readSBML (const char *filename)**

Reads the SBML document from the file named by `filename` and returns a pointer to it.

**SBMLDocument_t *readSBMLFromString (const char *xml)**

Reads the SBML document from the given XML string and returns a pointer to it. The XML string must be complete and legal XML document. Among other things, it must start with an XML processing instruction. For example,

```
<?xml version='1.0' encoding='UTF-8'?>
```

These functions return a pointer to an `SBMLDocument_t` object. This object represents the whole SBML model; it corresponds to the `Sbml` class object in the SBML Level 2 specification, but does not have a direct correspondence in SBML Level 1. (But, it is created by LIBSBML no matter whether the model is Level 1 or Level 2.)

`SBMLDocument_t` in Level 2 is derived from SBase, so that it contains the usual SBase fields of `metaid`, `notes` and `annotation`, as well as two other fields defined by `Sbml`: `level` and `version`. The following methods provide access to information about the level and version of the SBML input:

**unsigned int SBMLDocument_getLevel (const SBMLDocument_t *d)**

Returns the SBML level of this SBML document.

**unsigned int SBMLDocument_getVersion (const SBMLDocument_t *d)**

Returns the SBML version of this SBML document.

Of course, the whole point of reading an SBML file or data stream is to get at the SBML model it contains. The following method allows access to the Model object within an SBML document:

**Model_t * SBMLDocument_getModel (const SBMLDocument_t *d)**

Returns the Model associated with this `SBMLDocument_t` object.

LIBSBML stores warnings and error messages that may be encountered while parsing the XML input. Each warning or error is a `ParseMessage_t` object. To access the lists of diagnostic messages in an `SBMLDocument_t` object, use the following methods:

**ParseMessage_t *SBMLDocument_getWarning (SBMLDocument_t *d, unsigned int n)**

Returns the nth warning encountered during the parse of this SBMLDocument or `NULL` if `n > getNumWarnings() - 1`.

**ParseMessage_t *SBMLDocument_getError (SBMLDocument_t *d, unsigned int n);**

Returns the nth error encountered during the parse of this SBMLDocument or `NULL` if `n > getNumErrors() - 1`.

**ParseMessage_t *SBMLDocument_getFatal (SBMLDocument_t *d, unsigned int n)**

Returns the nth fatal error encountered during the parse of this SBMLDocument or `NULL` if `n > getNumErrors() - 1`.

**unsigned int SBMLDocument_getNumWarnings (SBMLDocument_t *d)**

Returns the number of warnings encountered during the parse of this SBMLDocument.

**unsigned int SBMLDocument_getNumErrors (SBMLDocument_t *d)**

Returns the number of errors encountered during the parse of this SBMLDocument.

**unsigned int SBMLDocument_getNumFatals (SBMLDocument_t *d)**

Returns the number of fatal errors encountered during the parse of this SBMLDocument.

**void SBMLDocument_printWarnings (SBMLDocument_t *d, FILE *stream)**

Prints all warnings encountered during the parse of this SBMLDocument to the given `stream`. If no warnings have occurred, i.e. `SBMLDocument_getNumWarnings(d) == 0`, no output will be sent to `stream`. The format of the output is:

```
%d Warning(s):
  Line %d, Col %d: %s
  ...
```

This is a convenience function to aid in debugging. For example: `SBMLDocument_printWarnings(d, stdout)`.

**void SBMLDocument_printErrors (SBMLDocument_t *d, FILE *stream)**

Prints all errors encountered during the parse of this SBMLDocument to the given `stream`. If no errors have occurred, that is, if `SBMLDocument_getNumErrors(d) == 0`, no output will be sent to `stream`. The format of the output is:

```
%d Error(s):
  Line %d, Col %d: %s
  ...
```

This is a convenience function to aid in debugging. For example: `SBMLDocument_printErrors(d, stdout)`.

**void SBMLDocument_printFatals (SBMLDocument_t *d, FILE *stream)**

Prints all fatals encountered during the parse of this SBMLDocument to the given `stream`. If no fatals have occurred, that is, if `SBMLDocument_getNumFatals(d) == 0`, no output will be sent to `stream`. The format of the output is:

```
%d Fatal(s):
  Line %d, Col %d: %s
  ...
```

This is a convenience function to aid in debugging. For example: `SBMLDocument_printFatals(d, stdout)`.

There are a few other methods defined by `SBMLDocument_t`, but their discussion is left to Section 7

## 5.1 A Simple Example of Reading SBML

The following example is included in the LIBSBML distribution as `readSBML.c` in the subdirectory `examples`. It is not compiled as part of the normal build process, but a Makefile is provided in the `examples` subdirectory that can be used to build `readSBML.c` and other examples. Once LIBSBML itself is installed, you should be able to compile the examples by simply typing the following command in the `examples` directory:

```
make
```

The `readSBML` program takes a single command-line argument, the name of an SBML file, reads it into memory and reports some basic information about the file and any warnings or errors generated by LIBSBML while parsing the file. Here is an example of using it on some of the sample SBML files provided in the `src/test-data` subdirectory of the LIBSBML distribution. In this example, the current directory is assumed to be `examples`.

```
./readSBML ../src/test-data/l1v1-branch.xml
./readSBML ../src/test-data/l1v1-minimal.xml
./readSBML ../src/test-data/l1v1-rules.xml
# etc...
```

The complete text of `readSBML` is shown in Figure 6 on the following page.

## 5.2 XML Schema Validation

To have LIBSBML validate an SBML document against an SBML (XML) Schema when using a Schema-aware parser such as Xerces requires creating an `SBMLReader` object and setting the appropriate schema filename and validation level. The functions for doing this are:

**SBMLReader_t *SBMLReader_create (void)**

Creates a new SBMLReader and returns a pointer to it. By default schema validation is off (`XML_SCHEMA_VALIDATION_NONE`) and `schemaFilename` is `NULL`.

**void SBMLReader_free (SBMLReader_t *sr)**

Frees the given SBMLReader.

**void SBMLReader_setSchemaFilenameL1v1 (SBMLReader_t *sr, const char *filename)**

Sets the file containing the XML Schema used by this SBMLReader to validate SBML Level 1 Version 1 documents. The `filename` should be either (1) an absolute path or (2) a path relative to the directory containing the SBML file(s) to be read.

```c
#include <stdio.h>

#include "sbml/SBMLTypes.h"

int
main (int argc, char *argv[])
{
  SBMLDocument_t *d;
  Model_t        *m;

  unsigned int level, version;


  if (argc != 2)
  {
    printf("\n  usage: printSBML <filename>\n\n");
    return 1;
  }

  d = readSBML(argv[1]);
  m = SBMLDocument_getModel(d);

  level   = SBMLDocument_getLevel  (d);
  version = SBMLDocument_getVersion(d);

  printf("\n");
  printf("File: %s (Level %u, version %u)\n", argv[1], level, version);

  printf("         ");
  if (level == 1)
  {
    printf("model name: %s\n", Model_getName(m));
  }
  else
  {
    printf("  model id: %s\n",
           Model_isSetId(m) ? Model_getId(m) : "(empty)");
  }

  printf("functionDefinitions: %d\n", Model_getNumFunctionDefinitions(m));
  printf("    unitDefinitions: %d\n", Model_getNumUnitDefinitions(m)    );
  printf("       compartments: %d\n", Model_getNumCompartments(m)       );
  printf("            species: %d\n", Model_getNumSpecies(m)            );
  printf("         parameters: %d\n", Model_getNumParameters(m)         );
  printf("          reactions: %d\n", Model_getNumReactions(m)          );
  printf("              rules: %d\n", Model_getNumRules(m)              );
  printf("             events: %d\n", Model_getNumEvents(m)             );
  printf("\n");

  SBMLDocument_printWarnings(d, stdout);
  SBMLDocument_printErrors  (d, stdout);
  SBMLDocument_printFatals  (d, stdout);

  SBMLDocument_free(d);
  return 0;
}
```

**Figure 6:** *The text of the program `printSBML.c` provided in the `examples` subdirectory of the* LIBSBML *distribution.*

**void SBMLReader_setSchemaFilenameL1v2 (SBMLReader_t *sr, const char *filename)**

Sets the file containing the XML Schema used by this SBMLReader to validate SBML Level 1 Version 2 documents. The `filename` should be either (1) an absolute path or (2) a path relative to the directory containing the SBML file(s) to be read.

**void SBMLReader_setSchemaFilenameL2v1 (SBMLReader_t *sr, const char *filename)**

Sets the file containing the XML Schema used by this SBMLReader to validate SBML Level 2 Version 1 documents. The `filename` should be either (1) an absolute path or (2) a path relative to the directory containing the SBML file(s) to be read.

**void SBMLReader_setSchemaValidationLevel ( SBMLReader_t *sr, XMLSchemaValidation_t level )**

Sets the level of schema validation used by this SBMLReader. The possible values for `level` are:

- `XML_SCHEMA_VALIDATION_NONE` (0) turns schema validation off.
- `XML_SCHEMA_VALIDATION_BASIC` (1) validates an XML instance document against an XML Schema. Those who wish to perform schema checking on SBML documents should use this option.
- `XML_SCHEMA_VALIDATION_FULL` (2) validates both the instance document itself *and* the XML Schema document. The XML Schema document is checked for violation of particle unique attribution constraints and particle derivation restrictions, which is both time-consuming and memory intensive. Few users will be interested in this.

Note that the `SBMLReader_setXYZ` methods above have no effect when using a parser such as Expat, because it is not a validating XML parser and the settings have no meaning for it.

Once an `SBMLReader_t` object has been created, two variants of the functions `readSBML()` and `readSBMLFromString()` previously discussed in Section 5 become available. These variants can be thought of as methods of the `SBMLReader_t` class:

**SBMLDocument_t *SBMLReader_readSBML (SBMLReader_t *sr, const char *filename)**

Reads the SBML document using the `SBMLReader_t` object passed in argument `sr` from the given `filename` and returns a pointer to it.

**SBMLDocument_t *SBMLReader_readSBMLFromString (SBMLReader_t *sr, const char *xml)**

Reads the SBML document using the `SBMLReader_t` object passed in argument `sr` from the character string passed in variable `xml` and returns a pointer to it. The XML string in `xml` must be complete and legal XML document. Among other things, it must start with an XML processing instruction, i.e.,

```
<?xml version='1.0' encoding='UTF-8'?>
```

Schema violations are reported in the `SBMLDocument_t`'s list of `ParseMessages_t`, according to the principles discussed in Section 5.

## 5.3 Writing SBML Files

Writing SBML is, in the end, a very simple matter in LIBSBML. The library provides the following two methods for this purposes.

**int writeSBML (SBMLDocument_t *d,
const char *filename)**

Writes the given SBML document to the given `filename`. Returns `1` on success and `0` on failure (e.g., if the file named by `filename` could not be opened for writing).

**char *writeSBMLToString (SBMLDocument_t *d)**

Writes the given SBML document to an in-memory string and returns a pointer to it. The string is owned by the caller and should be freed (with `free()`) when no longer needed. Returns `NULL` on failure.

# 6 Handling of Mathematical Formulas and MathML

LIBSBML can read and write MathML 2.0 (Ausbrooks et al., 2001) content in SBML documents and data streams, as well as translate between MathML and the text-string formulas used in SBML Level 1. This section describes the library's capabilities for handling MathML and mathematics.

## 6.1 Reading and Writing Formulas in Text-String Form

In SBML Level 1, mathematical formulas are expressed as text strings using a simple C-like syntax. In SBML Level 2, mathematical formulas are expressed in MathML syntax. LIBSBML helps calling programs smooth over this difference by providing an API that allows working with formulas in both text-string and MathML form, and to interconvert mathematical expressions between these forms (to the extent possible by the differences between SBML Levels 1 and 2.)

Formulas in LIBSBML are represented internally using Abstract Syntax Trees (ASTs). ASTs are described in detail in Appendix B. When LIBSBML reads an SBML model, it converts the expressions into ASTs and stores the ASTs in the corresponding data structures that have mathematical formulas (such as in an SBML KineticLaw). Thus, the `KineticLaw_getMath()` method, for example, returns a pointer to the root of an AST corresponding to the formula stored there.

Many software packages provide users with the ability to express formulas for such things as reaction rate expressions, and these packages' interfaces often let users type in the formulas directly as strings. LIBSBML provides two high-level functions for working with mathematical expressions in the form of strings: `SBML_parseFormula()` and `SBML_formulaToString()`.

**ASTNode_t *SBML_parseFormula (const char *formula)**

Parses the given string as a mathematical formula in SBML Level 1 syntax form, and returns a representation of it as an Abstract Syntax Tree (AST). This function returns the root node of the AST. If the formula contains a syntax error, this function returns `NULL` instead.

**char *SBML_formulaToString (ASTNode_t *tree)**

Returns a text-string mathematical expression corresponding to the Abstract Syntax Tree given as the argument. The caller owns the memory allocated for the returned string and is responsible for freeing it when it is no longer needed.

Using these methods is easy. The following is a code fragment that illustrates calling the parser function repeatedly with different formula strings, taking the ASTs returned each time and handing them back to the formula generator and comparing the strings to make sure they matched. (This is not something a real application would ever need to do, but it does simply illustrate the use of these two methods.)

```
const char *formulae[] =
{
  "1",
  "2.1",
  "2.1e+10",
  "foo",
  "1 + foo",
  "1 + 2",
  "1 + 2 * 3",
  "(1 - 2) * 3",
  "1 + -2 / 3",
  "1 + -2e-100 / 3",
  "1 - -foo / 3",
  "2 * foo^bar + 3.1",
  "foo()",
  "foo(1)",
  "foo(1, bar)",
  "foo(1, bar, 2^-3)",
  ""
};


ASTNode_t *n;
char     *s;
int       i;

for (i = 0; i < *formulae[i]; i++)
{
  n = SBML_parseFormula( formulae[i] ); /* Convert string to AST */
  s = SBML_formulaToString(n);       /* Convert AST back to string */

  if ( strcmp(s, formulae[i]) != 0 )
  {
    printf("Formula '%s' parsed incorrectly\n", formulae[i] );
  }

  ASTNode_free(n);
  free(s);
}
```

Section 6.4 describes some additional points that are worth knowing about the mathematical formula handling in LIBSBML. For example, Level 1 formula strings and Level 2 MathML expressions can be interconverted.

## 6.2 Reading Formulas in MathML Form: `MathMLDocument_t` and ASTs

There may arise situations in which an application needs to convert MathML directly into an AST. LIBSBML provides the utility function `readMathMLFromString()` for this purpose:

**MathMLDocument_t *readMathMLFromString (const char *xml)**

Reads a string containing an XML MathML expression, constructs the corresponding Abstract Syntax Tree and returns a pointer to a `MathMLDocument_t` object holding the tree structure.

The object returned by `readMathMLFromString()` is a simple container for an AST. The class of this object, MathMLDocument, is not defined by the SBML language standard but is provided in LIBSBML as a utility class. MathMLDocument serves as a top-level container for XML documents containing only MathML; in some ways it mirrors the SBMLDocument class, which acts as a container for XML documents containing SBML. The definition of `MathMLDocument_t` is as follows:

```
/**
 * The MathMLDocument
 */
typedef struct
{
  ASTNode_t *math;
} MathMLDocument_t;
```

The following are the functions defined for the MathMLDocument class:

**MathMLDocument_t *MathMLDocument_create ()**

Creates a `MathMLDocument_t` object.

**void MathMLDocument_free (MathMLDocument_t *d)**

Frees the given `MathMLDocument_t` object.

**ASTNode_t *MathMLDocument_getMath (const MathMLDocument_t *d)**

Returns the Abstract Syntax Tree representation of the mathematical formula stored in this `MathMLDocument_t` object.

**int MathMLDocument_isSetMath (const MathMLDocument_t *d)**

Returns `1` if the math of this MathMLDocument has been set, `0` otherwise.

**void MathMLDocument_setMath (MathMLDocument_t *d, ASTNode_t *math)**

Sets the math of this MathMLDocument to the given AST node. The node **is not copied** and this MathMLDocument **takes ownership** of it; i.e., subsequent calls to this function or a call to `MathMLDocument_free()` will free the AST node (and any child nodes attached to it).

Note that because the content passed to `readMathMLFromString()` is handed to an XML parser, the string given as argument must be a complete XML (though not necessarily SBML) document. The following example illustrates the use of this function with a valid MathML input.

```
MathMLDocument_t *doc;
ASTNode_t        *ast;
char             *result;

const char* s = "<?xml version='1.0' encoding='UTF-8'?>"
                "<math xmlns='http://www.w3.org/1998/Math/MathML'>"
                  "<apply><arccos/><ci> x </ci></apply>"
                "</math>";

doc   = readMathMLFromString(s);
ast   = MathMLDocument_getMath(doc);
```

The code above would create an AST structure stored in the variable `ast`. This tree structure could then be inspected using the AST node methods described in Appendix B.

Finally, LIBSBML provides two utility methods for writing out MathML represented in ASTs. Both of the following take a `MathMLDocument_t` class object, convert the expression tree stored there, and write out the appropriate text in MathML syntax.

**int writeMathML (MathMLDocument_t *d, const char *filename)**

Writes the given MathML document to filename. Returns `1` on success and `0` on failure (e.g., if `filename` could not be opened for writing or the MathMLWriter character encoding is invalid).

**char * writeMathMLToString (MathMLDocument_t *d)**

Writes the given MathML document to an in-memory string and returns a pointer to it. The string returned is owned by the caller and should be freed (with `free()`) when no longer needed. Returns `NULL` on failure

## 6.3 Differences between SBML Level 1 Formulas and MathML

The text-string based mathematical formula syntax of SBML Level 1 is *mostly* compatible with the representation of formulas in MathML. A few differences exist in the names of predefined functions such as `arccos`. Table 3 gives the mapping between SBML Level 1 and Level 2 function names.

| SBML Level 1 | SBML Level 2 |
|---|---|
| `abs` | `abs` |
| `acos` | `arccos` |
| `asin` | `arcsin` |
| `atan` | `arctan` |
| `ceil` | `ceiling` |
| `cos` | `cos` |
| `exp` | `exp` |
| `floor` | `floor` |
| `log` | `ln` |
| `log10(x)` | `log(10, x)` |
| `pow(x, y)` | `power(x, y)` |
| `sqr(x)` | `power(x, 2)` |
| `sqrt(x)` | `root(2, x)` |
| `sin` | `sin` |
| `tan` | `tan` |

**Table 3:** *Basic mathematical functions defined in SBML Levels 1 and 2. The underlined functions are different between the two levels of SBML.*

## 6.4 Additional Notes about the Handling of Mathematical Formulas

The LIBSBML formula parser has been carefully engineered so that transformations from MathML to infix string notation *and back* is possible with a minimum of disruption to the structure of the mathematical expression.

Figure 7 on the following page shows a simple program that, when run, takes a MathML string compiled into the program, converts it to an AST, converts *that* to an infix representation of the formula, compares it to the expected form of that formula, and finally translates that formula back to MathML and displays it. The output displayed on the terminal should have the same structure as the MathML it started with. The program is a simple example of using the various MathML and AST reading and writing methods, and shows that LIBSBML preserves the ordering and structure of the mathematical expressions.

The string form produced by `SBML_formulaToString()` and written by `writeMathMLToString()` is in SBML Level 1 formula string syntax, a simple C-inspired infix notation defined in the SBML Level 1 specification Hucka et al. (2001). It can therefore be handed to a program that

understands SBML Level 1 mathematical expressions, or used as part of a translation system. The LIBSBML distribution comes with an example program in the `examples` subdirectory called `translateMath` that implements an interactive command-line demonstration of translating infix formulas into MathML and vice-versa.

LIBSBML offers the ability to translate entire SBML Level 1 models to SBML Level 2, as explained below, and hopefully in the future will also provide the ability to translate a subset of Level 2 models to Level 1 (though this latter capability is not yet implemented).

```
1  #include <stdio.h>
2  #include <string.h>
3  #include "sbml/SBMLTypes.h"
4
5  int
6  main (int argc, char *argv[])
7  {
8      MathMLDocument_t *doc;
9      ASTNode_t        *ast;
10     char             *result;
11     MathMLDocument_t *new_doc;
12     ASTNode_t        *new_mathml;
13     char             *new_s;
14
15     const char* expected = "1 + f(x)";
16
17     const char* s = "<?xml version='1.0' encoding='UTF-8'?>"
18         "<math xmlns='http://www.w3.org/1998/Math/MathML'>"
19         "  <apply> <plus/> <cn> 1 </cn>"
20         "                  <apply> <ci> f </ci> <ci> x </ci> </apply>"
21         "  </apply>"
22         "</math>";
23
24     doc    = readMathMLFromString(s);
25     ast    = MathMLDocument_getMath(doc);
26     result = SBML_formulaToString(ast);
27
28     if ( strcmp(result, expected) == 0 )
29     {
30         printf("Got expected result\n");
31     }
32     else
33     {
34         printf("Mismatch after readMathMLFromString()\n");
35     }
36
37     new_mathml = SBML_parseFormula(result);
38     new_doc    = MathMLDocument_create();
39     MathMLDocument_setMath(new_doc, new_mathml);
40     new_s      = writeMathMLToString(new_doc);
41
42     printf("Result of writing AST:\n");
43     printf(new_s);
44
45     return 0;
46 }
```

**Figure 7:** *Short program to translate MathML into a formula string and back.*

# 7 Levels of SBML

At the time of this writing, there exist 3 flavors of SBML: Level 1 Versions 1 and 2, and SBML Level 2 Version 1. A software application may need to read and/or write any of these versions, depending on its purpose. LIBSBML provides support for all three definitions of SBML.

Along with the methods discussed in Section 5, the `SBMLDocument_t` object class also defines the following methods that impact how a model is written out:

**void SBMLDocument_setModel (SBMLDocument_t *d, Model_t *m)**

Sets the Model of this SBML document to the given `Model_t` object. Any previously defined model in `d` is unset and freed.

**void SBMLDocument_setLevel (SBMLDocument_t *d, unsigned int level)**

Sets the level of this SBML document to `level`. Valid levels are currently 1 and 2.

**void SBMLDocument_setVersion (SBMLDocument_t *d, unsigned int version)**

Sets the version of this SBML document to the given `version` number. Valid versions are currently 1 and 2 for SBML Level 1 and 1 for SBML Level 2.

Setting the level using `SBMLDocument_setLevel()` affects the possible fields and values available when setting and reading fields. Certain translations take place immediately upon changing levels. For example, if one starts with a Level 1 model and then calls `SBMLDocument_setLevel()` to set the level to 2, the model structure at that moment is translated internally so that such things as object names are converted to `id`'s (which do not exist in Level 1).

The C program listed in Figure 8 is provided in the LIBSBML distribution in the `examples` subdirectory. This command-line program takes two arguments: the name of an input file and the name of an output file. It then translates the SBML in the input file into SBML Level 2 and writes it out to the named output file. It may be surprising to see how short this program is.

# 8 Validation of SBML Models

LIBSBML performs a certain amount of validation of SBML inputs at the time of parsing files and data streams. However, the checks performed are mostly syntactic in nature, based on the XML Schema for SBML (and as noted elsewhere, getting the most of this validation capability requires the using of an XML Schema-aware parser such as Xerces).

LIBSBML implements more extensive semantic validation rules internally. At the time of this writing, over 30 tests are implemented. Examples of these rules include: compartments' `spatialSizeUnits` fields must be consistent with their `spatialDimensions`; species with `hasOnlySubstanceUnits` set to true must not have an `initialConcentration`; and others.

Semantic validation rules in LIBSBML (and indeed, in SBML in general) are still somewhat experimental; for this reason, the library does not perform them automatically. Callers must request semantic validation to be invoked explicitly by calling the following method.

**unsigned int SBMLDocument_validate (const SBMLDocument_t *d)**

Performs semantic validation on the document. Calling programs can query the results by calling `SBMLDocument_getNumWarnings()`, `SBMLDocument_getNumErrors()`, `SBMLDocument_getNumFatals()`, and related methods. This method returns (as an integer) the number of semantic validation errors encountered

```
#include <stdio.h>
#include "sbml/SBMLTypes.h"

int
main (int argc, char *argv[])
{
  unsigned int errors = 0;
  SBMLDocument_t *d;

  if (argc != 3)
  {
    printf("\nusage: convertSBML <input-filename> <output-filename>\n\n");
    return 1;
  }

  d = readSBML(argv[1]);

  errors = SBMLDocument_getNumWarnings(d) + SBMLDocument_getNumErrors(d) +
           SBMLDocument_getNumFatals(d);

  if (errors > 0)
  {
    printf("Error(s):\n");

    SBMLDocument_printWarnings(d, stdout);
    SBMLDocument_printErrors  (d, stdout);
    SBMLDocument_printFatals  (d, stdout);

    printf("Conversion skipped.  Please correct the above and re-run.\n");
  }
  else
  {
    SBMLDocument_setLevel(d, 2);
    writeSBML(d, argv[2]);
  }

  SBMLDocument_free(d);
  return errors;
}
```

**Figure 8:** *The text of the example C program `convertSBML.c`.*

# 9 Special Considerations and Known Issues

This section summarizes special considerations, known issues and caveats surrounding the use and behavior of LIBSBML.

## 9.1 Conformance to SBML

Currently, LIBSBML supports all of SBML Level 1 Version 1 and Version 2, and nearly all of SBML Level 2 Version 1. The still-unsupported parts of the Level 2 specification are:

- Support for RDF
- Support for MathML's `semantics` elements
- Support for MathML's `annotation` elements
- Support for MathML's `annotation-xml` elements

## 9.2 Issues Related to XML Parsers

Using Expat prevents libSBML from performing XML Schema-based validation of SBML input. This removes a number of verification checks from the parsing stage and may cause unexpected behavior in the face of malformed or invalid SBML content. Here are some implications of not performing XML Schema validation:

- The syntax of identifiers (i.e., conformance to SId syntax) will not be verified. This means that identifiers that are not in conformance to SBML SId specifications will be passed through without being flagged as invalid.
- Data types of values assigned to fields in a model will not be verified for conformance to the SBML specification. In some cases this means that those values will not be assigned to the corresponding object structures created by libSBML. For example, reading a model containing a compartment definition having a volume of `"mumble"` (a string instead of a number) will result in libSBML simply ignoring the value and treating the input as if no value was supplied.
- Elements present in an SBML input file or data stream, but that are not actually defined by the SBML specification, will not be noticed. (Such SBML input should be flagged as invalid, but will not be.)
- XML entity references (e.g. XML's ` `), which are most likely to occur in XHTML `<notes>` sections, will be output as their UTF-8 byte sequence instead of the more human readable entity reference. This is a bug in the Expat support in libSBML, stemming from a limitation in the API of Expat. (While Expat reads and writes UTF-8 by default, it comes with no APIs to manipulate or translate Unicode encodings. Writing such a conversion routine and ensuring it is cross-platform is non-trivial.)
- The methods discussed in Section 5.2, namely the `SBMLReader_setSchemaFilename****()` methods and `SBMLReader_setSchemaValidationLevel()`, have no effect.

Although it is poorly documented, SBML XML documents **must use only** the `UTF-8` encoding. Parsing a non-UTF-8 document may fail unpredictably and this particular error may be difficult to diagnose, because it will happen in the underlying XML parser and not libSBML itself.

# 10 Acknowledgments

Thanks to Mike Hucka for updating and editing this manual for version 2.0 of libSBML.

# A Lists and ListOf_t

While list-based convenience methods (e.g., XXX_getNumYYY()) are provided for every class, it is possible to access and manipulate each list directly. All lists are themselves objects of type ListOf_t. The full set of list methods are:

**unsigned int ListOf_getNumItems (const ListOf_t *lo)**

Returns the number of items in this list.

**void ListOf_append (ListOf_t *lo, void *item)**

Adds item to the end of this list.

**void * ListOf_get (const ListOf_t *lo, unsigned int n)**

Returns the nth item in this list. If n > ListOf_getNumItems(list), it returns NULL.

**void ListOf_prepend (ListOf_t *lo, void *item)**

Adds item to the beginning of this ListOf_t object.

**void * ListOf_remove (ListOf_t *lo, unsigned int n)**

Removes the nth item from this list and returns a pointer to it. If n > ListOf_getNumItems(list), it returns NULL.

Since UnitDefinitions maintains a list of Units, the UnitDefinition example presented in Section 4.4 could also be written as:

```
UnitDefinition_t *ud = UnitDefinition_createWith("mmls");

UnitDefinition_addUnit(ud, Unit_createWith(UNIT_KIND_MOLE , 1, -3) );
UnitDefinition_addUnit(ud, Unit_createWith(UNIT_KIND_LITRE , -1, 0) );
UnitDefinition_addUnit(ud, Unit_createWith(UNIT_KIND_SECOND, -1, 0) );
```

However, this approach is not the preferred one. The best reason to use specific XXX_getYYY() methods over the list API in libSBML is that the former are typed to specific items, whereas ListOf_get() returns a void pointer that must be cast to a specific type. Moreover, the code resulting from using the XXX_getYYY() methods is arguably more readable.

Although many specialized methods are available for accessing various data objects in SBML, the list API is necessary for accessing such things as the content of notes and annotation elements on SBML's listOf*** elements. In addition, the only way to remove an item from a list is to use the API directly.

# B Abstract Syntax Trees and ASTNode_t

Abstract Syntax Trees (ASTs) in libSBML are a simple data structure for storing mathematical expressions. For many applications, the details of ASTs are irrelevant because the applications can use the text-string based translation functions described in Sections 6.1 and 6.2. However, other applications do need to read and manipulate ASTs directly. This section describes libSBML's AST in detail so software authors can write code to work with them.

An AST *node* is a recursive structure containing a pointer to the node's value (e.g., a number or a symbol) and a list of children nodes. LIBSBML provides a number of methods for manipulating `ASTNode_t` objects, the full set of which is documented in the LIBSBML API Reference Manual. The following discussion only covers a subset of all the possible methods.

## B.1 Methods for Manipulating AST Nodes

First, there is a set of methods for creating and manipulating LIBSBML AST nodes and their children structures:

**ASTNode_t * ASTNode_create (void)**

Creates a new `ASTNode_t` object and returns a pointer to it. The returned node will have a type of `AST_UNKNOWN` and should be set to something else as soon as possible.

**void ASTNode_free (ASTNode_t *node)**

Frees the given `ASTNode_t` including any child nodes.

**unsigned int ASTNode_getNumChildren (const ASTNode_t *node)**

Returns the number of children of this AST node or 0 is this node has no children.

**void ASTNode_addChild (ASTNode_t *node, ASTNode_t *child)**

Adds the given node as a child of this AST node. Child nodes are added in left-to-right order.

**void ASTNode_prependChild (ASTNode_t *node, ASTNode_t *child)**

Adds the given node as a child of this AST node. This method adds child nodes in right-to-left order.

**ASTNode_t * ASTNode_getChild (const ASTNode_t *node, unsigned int n)**

Returns the nth child of this AST node or NULL if this node has no nth child (n > `ASTNode_getNumChildren()` - 1).

**ASTNode_t * ASTNode_getLeftChild (const ASTNode_t *node)**

Returns the left child of this AST node. This is equivalent to `ASTNode_getChild(node, 0);`

**ASTNode_t * ASTNode_getRightChild (const ASTNode_t *node)**

Returns the right child of this AST node or NULL if this node has no right child. If `ASTNode_getNumChildren(node)` > 1, then this is equivalent to:

```
ASTNode_getChild(node, ASTNode_getNumChildren(node) - 1);
```

AST nodes are typed. The list of possible types is quite long, because it covers all the mathematical functions that are permitted in SBML. Table 4 on the next page shows the list of type names which are part of the enumeration `ASTNodeType_t`. Most of the names are hopefully fairly-self explanatory; e.g., `AST_PLUS` stands for the "+" operator, `AST_REAL` signifies a real number, etc. The following methods can be used to interrogate the type of a given AST node:

| | | |
|---|---|---|
| AST_PLUS | AST_FUNCTION_ARCCOTH | AST_FUNCTION_POWER |
| AST_MINUS | AST_FUNCTION_ARCCSC | AST_FUNCTION_ROOT |
| AST_TIMES | AST_FUNCTION_ARCCSCH | AST_FUNCTION_SEC |
| AST_DIVIDE | AST_FUNCTION_ARCSEC | AST_FUNCTION_SECH |
| AST_POWER | AST_FUNCTION_ARCSECH | AST_FUNCTION_SIN |
| AST_INTEGER | AST_FUNCTION_ARCSIN | AST_FUNCTION_SINH |
| AST_REAL | AST_FUNCTION_ARCSINH | AST_FUNCTION_TAN |
| AST_REAL_E | AST_FUNCTION_ARCTAN | AST_FUNCTION_TANH |
| AST_RATIONAL | AST_FUNCTION_ARCTANH | AST_LOGICAL_AND |
| AST_NAME | AST_FUNCTION_CEILING | AST_LOGICAL_NOT |
| AST_NAME_DELAY | AST_FUNCTION_COS | AST_LOGICAL_OR |
| AST_NAME_TIME | AST_FUNCTION_COSH | AST_LOGICAL_XOR |
| AST_CONSTANT_E | AST_FUNCTION_COT | AST_RELATIONAL_EQ |
| AST_CONSTANT_FALSE | AST_FUNCTION_COTH | AST_RELATIONAL_GEQ |
| AST_CONSTANT_PI | AST_FUNCTION_CSC | AST_RELATIONAL_GT |
| AST_CONSTANT_TRUE | AST_FUNCTION_CSCH | AST_RELATIONAL_LEQ |
| AST_LAMBDA | AST_FUNCTION_EXP | AST_RELATIONAL_LT |
| AST_FUNCTION | AST_FUNCTION_FACTORIAL | AST_RELATIONAL_NEQ |
| AST_FUNCTION_ABS | AST_FUNCTION_FLOOR | AST_UNKNOWN |
| AST_FUNCTION_ARCCOS | AST_FUNCTION_LN | |
| AST_FUNCTION_ARCCOSH | AST_FUNCTION_LOG | |
| AST_FUNCTION_ARCCOT | AST_FUNCTION_PIECEWISE | |

**Table 4:** *The list of AST node types in the enumeration `ASTNodeType_t`.*

**ASTNodeType_t ASTNode_getType (const ASTNode_t *node)**

Returns the type of this AST node.

**int ASTNode_isConstant (const ASTNode_t *node)**

Returns true (non-zero) if this AST node is a MathML constant (true, false, pi, exponentiale), false (0) otherwise.

**int ASTNode_isFunction (const ASTNode_t *node)**

Returns true (non-zero) if this AST node is a function in SBML L1, L2 (MathML) (everything from `abs()` to `tanh()`) or user-defined, false (0) otherwise.

**int ASTNode_isInteger (const ASTNode_t *node)**

Returns true (non-zero) if this AST node is of type `AST_INTEGER`, false (0) otherwise.

**int ASTNode_isLambda (const ASTNode_t *node)**

Returns true (non-zero) if this AST node is of type `AST_LAMBDA`, false (0) otherwise.

**int ASTNode_isLog10 (const ASTNode_t *node)**

Returns true (non-zero) if the given AST node represents a `log10()` function, false (0) otherwise.
More precisley, the node type is `AST_FUNCTION_LOG` with two children the first of which is an `AST_INTEGER` equal to 10.

**int ASTNode_isLogical (const ASTNode_t *node)**

Returns true (non-zero) if this AST node is a MathML logical operator (and, or, not, xor), false (0) otherwise.

**int ASTNode_isName (const ASTNode_t *node)**

Returns true (non-zero) if this AST node is a user-defined variable name in SBML L1, L2 (MathML) or the special symbols delay or time, false (0) otherwise.

**int ASTNode_isNumber (const ASTNode_t *node)**

Returns true (non-zero) if this AST node is a number, false (0) otherwise. This is functionally equivalent to:

```
ASTNode_isInteger(node) || ASTNode_isReal(node)
```

**int ASTNode_isOperator (const ASTNode_t *node)**

Returns true (non-zero) if this AST node is an operator, false (0) otherwise. Operators are: `+`, `-`, `*`, `/` and `^` (power).

**int ASTNode_isRational (const ASTNode_t *node)**

Returns true (non-zero) if this AST node is of type `AST_RATIONAL`, false (0) otherwise.

**int ASTNode_isReal (const ASTNode_t *node)**

Returns true (non-zero) if the value of this AST node can represented as a real number, false (0) otherwise. To be a represented as a real number, this node must be of one of the following types: `AST_REAL`, `AST_REAL_E` or `AST_RATIONAL`.

**int ASTNode_isRelational (const ASTNode_t *node)**

Returns true (non-zero) if this AST node is a MathML relational operator (`==`, `>=`, `>`, `<=`, `<`, `!=`), false (0) otherwise.

**int ASTNode_isSqrt (const ASTNode_t *node)**

Returns true (non-zero) if the given AST node represents a `sqrt()` function, false (0) otherwise. More precisely, the node type is `AST_FUNCTION_ROOT` with two children the first of which is an `AST_INTEGER` equal to 2.

**int ASTNode_isUMinus (const ASTNode_t *node)**

Returns true (non-zero) if this AST node is a unary minus, false (0) otherwise. For numbers, unary minus nodes can be "collapsed" by negating the number. In fact, `SBML_parseFormula()` does this during its parse. However, unary minus nodes for symbols (`AST_NAMES`) cannot be "collapsed", so this predicate function is necessary. A node is defined as a unary minus node if it is of type `AST_MINUS` and has exactly one child.

**int ASTNode_isUnknown (const ASTNode_t *node)**

Returns true (non-zero) if this AST node is of type `AST_UNKNOWN`, false (0) otherwise.

Programs manipulating AST node structures should check the type of a given node before calling methods that return a value from the node. The following meethods are available for returning values from nodes:

**char ASTNode_getCharacter (const ASTNode_t *node)**

Returns the value of this AST node as a single character. This function should be called only when `ASTNode_getType()` is one of `AST_PLUS`, `AST_MINUS`, `AST_TIMES`, `AST_DIVIDE` or `AST_POWER`.

**long ASTNode_getInteger (const ASTNode_t *node)**

Returns the value of this AST node as a (long) integer. This function should be called only when `ASTNode_getType() == AST_INTEGER`.

**const char * ASTNode_getName (const ASTNode_t *node)**

Returns the value of this AST node as a string. This function may be called on nodes that are not operators (`ASTNode_isOperator(node) == 0`) or numbers (`ASTNode_isNumber(node) == 0`).

**long ASTNode_getNumerator (const ASTNode_t *node)**

Returns the value of the numerator of this AST node. This function should be called only when `ASTNode_getType() == AST_RATIONAL`.

**long ASTNode_getDenominator (const ASTNode_t *node)**

Returns the value of the denominator of this AST node. This function should be called only when `ASTNode_getType() == AST_RATIONAL`.

**double ASTNode_getReal (const ASTNode_t *node)**

Returns the value of this AST node as a real (double). This function should be called only when `ASTNode_isReal(node) != 0`. This function performs the necessary arithmetic if the node type is `AST_REAL_E` (mantissa 10^exponent) or `AST_RATIONAL` (numerator / denominator).

**double ASTNode_getMantissa (const ASTNode_t *node)**

Returns the value of the mantissa of this AST node. This function should be called only when `ASTNode_getType()` is `AST_REAL_E` or `AST_REAL`. If `AST_REAL`, this method is identical to `ASTNode_getReal()`.

**long ASTNode_getExponent (const ASTNode_t *node)**

Returns the value of the exponent of this AST node. This function should be called only when `ASTNode_getType()` is `AST_REAL_E` or `AST_REAL`.

**int ASTNode_getPrecedence (const ASTNode_t *node)**

Returns the precedence of this AST node (as defined in the SBML L1 specification).

Finally (and rather predictably), libsbml provides methods for setting the values of AST nodes.

**void ASTNode_setCharacter (ASTNode_t *node, char value)**

Sets the value of this AST node to the given character. If character is one of `+`, `-`, `*`, `/` or `^`, the node type will be set accordingly. For all other characters, the node type will be set to `AST_UNKNOWN`.

**void ASTNode_setName (ASTNode_t *node, const char *name)**

Sets the value of this AST node to the given name. The node type will be set (to `AST_NAME`) **only if** the AST node was previously an operator (`ASTNode_isOperator(node) != 0`) or number (`ASTNode_isNumber(node) != 0`). This allows names to be set for `AST_FUNCTION`s and the like.

**void ASTNode_setInteger (ASTNode_t *node, long value)**

Sets the value of this AST node to the given (long) integer and sets the node type to `AST_INTEGER`.

**void ASTNode_setRational (ASTNode_t *node, long numerator, long denominator)**

Sets the value of this AST node to the given rational in two parts: the numerator and denominator. The node type is set to `AST_RATIONAL`.

**void ASTNode_setReal (ASTNode_t *node, double value)**

Sets the value of this AST node to the given real (double) and sets the node type to `AST_REAL`. This is functionally equivalent to:

```
ASTNode_setRealWithExponent(node, value, 0);
```

**void ASTNode_setRealWithExponent (ASTNode_t *node, double mantissa, long exponent)**

Sets the value of this AST node to the given real (double) in two parts: the mantissa and the exponent. The node type is set to `AST_REAL_E`.

**void ASTNode_setType (ASTNode_t *node, ASTNodeType_t type)**

Sets the type of this AST node to the given AST node type.

## B.2 Notes about ASTNode

The following are noteworthy about the AST node representation in libsbml:

- A numerical value represented in MathML as a real number with an exponent is preserved as such in the AST node representation, even if the number could be stored in a C `double` data type. This is done so that when an SBML model is read in and then written out again, the amount of change introduced by libsbml to the SBML during the round-trip activity is minimized.
- Rational numbers are represented in an AST node using separate numerator and denominator values. These can be retrieved using the `ASTNode_t` methods `ASTNode_getNumerator()` and `ASTNode_getDenominator()`.
- The `children` field of `ASTNode_t` is a list of pointers to other `ASTNode_t` objects. This list is empty for AST nodes that are leaf elements, such as numbers. For AST nodes that are actually roots of expression subtrees, the list of children points to the parsed objects that make up the rest of the expression.

# References


Ausbrooks, R., Buswell, S., Dalmas, S., Devitt, S., Diaz, A., Hunter, R., Smith, B., Soiffer, N., Sutor, R., and Watt, S. (2001). Mathematical markup language (MathML) version 2.0 (second edition) W3C recommendation 21 October 2003.

Bornstein, B. J. (2004). LibSBML API reference manual. Available on the Internet at http://www.sbml.org/software/libsbml.

Finney, A. M. and Hucka, M. (2003). Systems biology markup language: Level 2 and beyond. *Biochemical Society Transactions*, 31:1472–1473.

Hucka, M., Finney, A., Sauro, H. M., and Bolouri, H. (2001). Systems biology markup language (sbml) level 1: Structures and facilities for basic model definitions. Technical report. Available on the Internet at http://www.sbml.org/.

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