

libsbml Developer's Manual

Ben Bornstein

`bornstei@cds.caltech.edu`

Systems Biology Workbench Development Group

ERATO Kitano Symbiotic Systems Project

Control and Dynamical Systems, MC 107-81

California Institute of Technology, Pasadena, CA 91125, USA

<http://sbw-sbml.org/>

Principal Investigators: John Doyle and Hiroaki Kitano

DRAFT

May 8, 2003



Contents

1 Quickstart	3
1.1 Linux, Cygwin or MacOS X	3
1.2 Windows	3
2 Introduction	3
3 Installation	4
4 SBML Classes in C	6
4.1 Primitive Types	6
4.2 Object Creation and Destruction	7
4.3 Accessing Fields	8
4.4 Lists	12
4.5 Enumerations	13
4.6 Abstract Classes	14
5 Reading SBML Files	16
5.1 A Simple Example	19
5.2 XML Schema Validation	20
6 Writing SBML Files	21
A Lists	22

1 Quickstart

LIBSBML depends on Apache's Xerces-C++ XML library for low-level XML tokenizing and Unicode support. Xerces is supported on Unix (Linux), Windows and MacOS X. Many popular Linux systems provide the Xerces library either as part of their standard distribution or as an optional RPM or Debian package. Apache provides a Windows binary distribution which includes both a DLL and a LIB file. For more information, see: <http://xml.apache.org/xerces-c/>. The instructions below assume Apache is already installed on your system.

1.1 Linux, Cygwin or MacOS X

At the Unix, Cygwin or OS X command prompt, `untar` the distribution, `cd` into it (e.g., `libsbml-1.0.1/`), and type:

```
% ./configure
% make
% make install
```

To compile programs that use `libsbml` (e.g., see Section 5.1) with GCC:

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

1.2 Windows

Unzip the distribution and open the resulting folder (e.g. `libsbml-1.0.1`). There are debug (`libsbml_d`) and release (`libsbml`) versions of LIBSBML, with `.dll` and `.lib` files for both versions in the `Win32` subdirectory. LIBSBML header files are located in `src/sbml`.

Visual C++ projects should link with `libsbml.lib` or `libsbml_d.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 API for reading, writing and manipulating the Systems Biology Markup Language (SBML). Currently, the library supports SBML level 1, versions 1 and 2. Support for SBML level 2 is nearing completion. RDF and writing L2 documents are only features left to be implemented.

Since the library provides a C API, familiarity with the C programming language is assumed. Some parts of the library were written in C++, however, experience with C++ is not required; its use is "hidden" behind C functions.

LIBSBML is entirely open-source and all specifications and source code are freely and publicly available. For more information about SBML, please see the references section or visit <http://www.sbw-sbml.org/>.

Some of the 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**

The Gepasi generated 100 Yeast file (2Mb; 2000 reactions <http://www.gepasi.org/gep3sbml.html>) loads in 1.18s on a 1 GHz AMD Athalon XP and uses 1.4Mb of memory.

- **Portable**

The C source code is pure ANSI 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. ANSI 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 a precompiled Windows distribution is available.

- **Full SBML Support**

The parser supports both spellings of *species* and *annotation* (i.e. with and without the last *s*) for any level and version. When writing SBML documents, the appropriate specification-sanctioned spelling is used.

The full-text (including any namespace declarations) of `<notes>` and `<annotation>` elements may be retrieved from any SBML object. For compatibility with some technically incorrect but popular SBML documents, the parser recognizes and stores notes and annotations defined for the top-level `<sbml>` element (though a warning is logged).

- **Full XML Schema Validation**

The library uses 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 schema file to use 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 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 there are 1681 individual assertions in 311 functional unit tests. Four test cases are responsible for reading entire SBML files (three are examples from the L1 document) 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. Currently all unit tests produce 1782 memory allocations and matched frees with no leaks.

3 Installation

LIBSBML depends on Apache's Xerces-C++ XML library for low-level XML tokenizing and Unicode support. Xerces is supported on Unix (Linux), Windows and MacOS X. Many popular

Linux systems provide the Xerces library either as part of their standard distribution or as an optional RPM or Debian package. Apache provides a Windows binary distribution which includes both a DLL and a LIB file.

For more information, see:

<http://xml.apache.org/xerces-c/>

A good way to determine whether or not Xerces-C is installed is to run the build script (see below); it will halt if it cannot find the Xerces-C library.

LIBSBML is designed to be extremely portable. It is written in 100% pure ANSI-C and the build system uses the GNU Autotools (Autoconf, Automake and Libtool). In most cases, building should be as easy unpacking the sources and running:

```
% ./configure
% make
% make check (optional)
% make install
```

However, if Xerces-C is installed in a non-standard place (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 library. Set the CPPFLAGS and LDFLAGS environment variables to include the custom directories containing Xerces-C header and library files. For example:

```
% export CPPFLAGS=-I/home/bornstei/software/xerces-c/2.2.0/include
% export LDFLAGS=-L/home/bornstei/software/xerces-c/2.2.0/lib
% ./configure
% # etc.
```

`make install` copies header files to `/usr/local/include/sbml` and (shared and static) library files to `/usr/local/lib`. To specify a different install location use:

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

`make check` 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.

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 eXtrememe Programming (all of LIBSBML was written using the test first approach). 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).

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 at <http://www.sourceforge.net/projects/sbml>.

3.0.1 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 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 bug report at <http://www.sourceforge.net/projects/sbml>.

For performance reasons, memory tracing should be disabled in production environments:

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

4 SBML Classes in C

The SBML specification, with its UML diagrams, is suggestive of an object-oriented (OO) design. An API which attempts to represent 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 of its features. 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 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). The two SBML enumeration types, `UnitKind` and `RuleType` are represented as C enumerations, but deviate slightly from the above rules (see Section 4.5 for more information). Finally, there is one class, `SBMLDocument_t`, that exists in the LIBSBML API, but not strictly in SBML. It serves primarily as a top-level container for models and stores any warning or error messages encountered when an SBML document is 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`.

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, UML attribute names, including capitalization, are preserved (e.g., `initialAmount`) when mapped to C struct fields.

SBML Class	C Class (typedef struct)
<i>SBase</i>	SBase_t
Model	Model_t
UnitDefinition	UnitDefinition_t
Unit	Unit_t
Compartment	Compartment_t
Parameter	Parameter_t
Species	Species_t
Reaction	Reaction_t
SpeciesReference	SpeciesReference_t
KineticLaw	KineticLaw_t
<i>Rule</i>	Rule_t
<i>AssignmentRule</i>	AssignmentRule_t
AlgebraicRule	AlgebraicRule_t
CompartmentVolumeRule	CompartmentVolumeRule_t
ParameterRule	ParameterRule_t
SpeciesConcentrationRule	SpeciesConcentrationRule_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.*

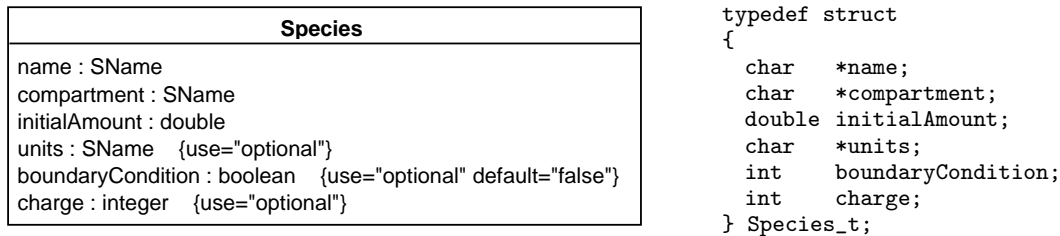


Figure 1: Definition of SBML Species in UML (left) and the corresponding Species_t C struct (right).

2. SName maps to a standard C string (pointer to an array of characters terminated by a NULL (0) character; for e.g. `char *name`). Note that SName syntax is not yet enforced in the API.
3. double and integer map to C `double` and `int` respectively.
4. boolean maps 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.

To instantiate (create) an object use either the `XXX_create()` or `XXX_createWith()` constructor. To destroy (free) an object use `XXX_free()`. The constructors and destructors for `Species` are:

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 *units, int boundaryCondition, int charge)

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

```
Species_t *s = Species_create();
Species_setName(s, name); 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 their attributes in a single operation. If `XXX_create()` is used instead, only attributes with default values (as defined by the specification) will be set. All other attributes will default to zero in the case of integers and doubles, false in the case of booleans or a NULL pointer in the case of SNames strings.

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

4.3 Accessing Fields

4.3.1 Getters

There are two ways to get the value of a field within a struct: i) direct reference and ii) via a method call. Direct reference should be preferred in almost all cases; it avoids the overhead of a function call and thus is significantly faster¹. The getter methods are provided for languages that do not allow direct access to the underlying C structs. As an example of direct access, Figure 2 lists a function (which is not part of the API) to print the fields of a `Species`.

The getter methods follow the naming convention `XXX_getYYY()`. The getters for `Species` are:

const char *Species_getName (const Species_t *s)

Returns the name of this Species.

const char *Species_getCompartment (const Species_t *s)

Returns the compartment of this Species.

double Species_getInitialAmount (const Species_t *s)

Returns the initialAmount of this Species.

const char *Species_getUnits (const Species_t *s)

Returns the units of this Species.

¹The getters are *not* `#define` macros that are inlined at compilation time. They are true functions.

```

1  /**
2   * Prints Species to stream (good for debugging).
3   */
4  void
5  myPrintSpecies (Species_t *s, FILE *stream)
6  {
7      char none[]          = "(none)";
8      char *compartment   = (s->compartment == NULL) ? none : s->compartment;
9      char *units         = (s->units      == NULL) ? none : s->units;
10     char *boundary      = (s->boundaryCondition == 0) ? "false" : "true";
11
12
13     fprintf(stream, "Species %s:\n", s->name);
14     fprintf(stream, "          Compartment: %s\n", compartment   );
15     fprintf(stream, "          Initial Amount: %f\n", s->initialAmount );
16     fprintf(stream, "          Units: %s\n", units             );
17     fprintf(stream, "          Boundary Condition: %s\n", boundary   );
18     fprintf(stream, "          Charge: %d\n", s->charge           );
19 }

```

Figure 2: Demonstrates direct access to the fields of an SBML object, in this case *Species.t*.

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

Returns the boundaryCondition of this Species.

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

Returns the charge of this Species.

Notice the `Species_t` passed to each getter is constant. The purpose of this constness is twofold: i) it reinforces the notion that a getter simply returns a value and does not modify the state of the passed-in object and ii) 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 disastrous (most likely resulting in a segmentation or general protection fault). Memory management issues are elaborated when discussing setters.

4.3.2 Setters

A value is assigned to a field via a set method². Requiring all assignments to take place through a setter 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 make a distinction between state and value is critical and is discussed further in Section 4.3.3.

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

```
void Species_setName (Species_t *s, const char *sname)
```

Sets the name of this Species to a copy of sname.

```
void Species_setCompartment (Species_t *s, const char *sname)
```

Sets the compartment of this Species to a copy of sname.

²Earlier versions of LIBSBML allowed primitive types to be set directly. Such direct access, however, made it impossible to distinguish between set and unset states of a field, since all possible values are valid (i.e. no sentinel value exists to indicate an unset state).

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

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

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

Sets the units of this Species to a copy of sname.

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

Sets the boundaryCondition of this Species to value (boolean) and marks the field as set.

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

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

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 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 Species `s` to the string “cell”:

```
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).

There is nothing in C language specification or the compiler that prevents string fields (or any other field type for that matter) from being set directly, as in:

```
s->name = "s2";
```

But to do so would likely cause a memory leak if `s->name` was already assigned to another string. Using the setter methods is much safer than assigning the memory directly and enables the set versus unset state of the field to be tracked.

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`. `Species` has an optional charge attribute, which means it need not ever be read-in (specified), written or manipulated. For this reason, LIBSBML must be able to determine the set or unset state of the `charge` field to either output or omit the field as appropriate.

The easiest way to determine whether a particular field is set or unset is to use the `XXX_isSetYYY()` class of methods. For `Species`, the following are available:

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

Return 1 if the name of this Species has been set, 0 otherwise. In SBML L1, a Species name is required and therefore **should always be set**. In L2, name is optional and as such may or may not be set.

int Species.isSetCompartment (const Species.t *s)

Returns 1 if the compartment 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 L1, a Species initialAmount is required and therefore **should always be set**. In L2, initialAmount is optional and as such may or may not be set.

int Species.isSetUnits (const Species.t *s)

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

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 an `isSetYYY()`. 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 from SBML Level 1 to Level 2, some fields changed from required to optional. If this is the case for a particular field the method documentation will state it (as above). That said, this second point is more relevant when it comes to unsetting fields.

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

void Species.unsetName (Species.t *s)

Unsets the name of this Species. This is equivalent to:

```
safe_free(s->name); s->name = NULL;
```

In SBML L1, a Species name is required and therefore **should always be set**. In L2, name is optional and as such may or may not be set.

void Species.unsetInitialAmount (Species.t *s)

Unsets the initialAmount of this Species. In SBML L1, a Species initialAmount is required and therefore **should always be set**. In L2, initialAmount is optional and as such may or may not be set.

void Species.unsetUnits (Species.t *s)

Unsets the units of this Species. This is equivalent to:

```
safe_free(s->units); s->units = NULL;
```

void Species.unsetCharge (Species.t *s)

Unsets the charge 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()` (a compartment is required for a Species in both SBML Level 1 and Level 2).

It is also possible to check the set status of a field directly. For SName strings this is simple, unset fields will contain a NULL pointer:

```
s->units != NULL /* Equivalent to Species_isSetUnits(s). */
```

For primitive types things are only slightly more complicated. A nested `isSet` struct of one bitfield per primitive type is used to keep track of set states. The value of a bitfield field is one if corresponding field is set and zero otherwise. The `isSet` struct was omitted from the definition of `Species_t` in Figure 1 to simplify the explanation of getter methods. A more complete definition of `Species_t`, with its `isSet` nested struct is shown in Figure 3.

So, to determine if a given species `s` has a charge:

```
s->isSet.charge == 1 /* Equivalent to Species_isSetCharge(s) */
```

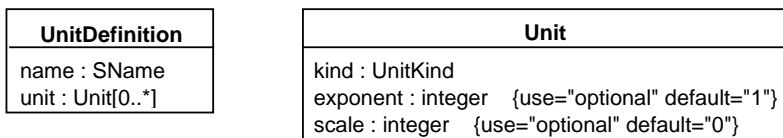
```

1 typedef struct
2 {
3     char    *name;
4     char    *compartment;
5     double  initialAmount;
6     char    *units;
7     int     boundaryCondition;
8     int     charge;
9
10    struct
11    {
12        unsigned int initialAmount:1;
13        unsigned int charge      :1;
14    } isSet;
15
16 } Species_t;
```

Figure 3: Definition of `Species_t` C struct, including its nested `isSet` struct of bitfields, used to track the state of primitive types.

4.4 Lists

`Species` contains only SNames and primitive types, but many SBML classes also contain lists of other objects. For example, a `UnitDefinition` contains a list of `Units`:



To help manage this containment relationship, three standard functions are provided `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 $mmoll^{-1}s^1$ named “mmls”, corresponding to the SBML:

```
<listOfUnitDefinitions>
  <unitDefinition name="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) );
```

In this case, `UnitDefinition_getNumUnits(ud)` returns 3 and `UnitDefinition_getUnit(ud, 1)` returns the *second* Unit³. Notice that items are numbered starting at zero.

Related to lists is a set of convenience methods for creating and adding SBML objects to a `Model` 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`, but this time inside a `Model`, 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` is added to the last `UnitDefinition` created. One caveat to be aware of with these methods is the case where no intermediate container exists; e.g., if no `UnitDefinition` were created above. In that case, the call to `Model_createUnit()` does nothing. More specifically, no `Unit` is created and (obviously) nothing is added to the model.

For more detailed information on Lists, see [Appendix A](#).

4.5 Enumerations

SBML has two enumeration types, `UnitKind` and `RuleType`. These translate directly to C enums with a few support functions for equality testing and converting to and from strings. From `UnitKind.h`:

```
typedef enum
{
    UNIT_KIND_AMPERE
    , UNIT_KIND_BECQUEREL
```

³(The `UNIT_KIND_XXX` enumerations are discussed later.)

```

, UNIT_KIND_CANDELA

/* Omitted for space */

, UNIT_KIND_WATT
, UNIT_KIND_WEBER
, UNIT_KIND_INVALID
} UnitKind_t;

```

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.

4.5.1 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:

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

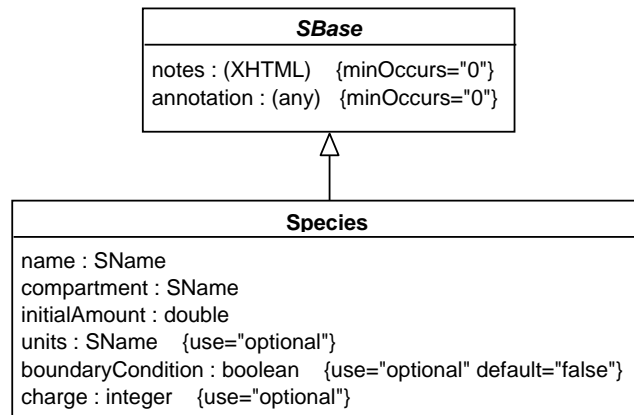
Since abstract classes cannot be created or destroyed directly, they have no XXX_create() or XXX_free() methods. Instead they have XXX_init() and XXX_clear() methods which sub-

SBML Class	C Class (typedef struct)
<i>SBase</i>	SBase_t
<i>Rule</i>	Rule_t
<i>AssignmentRule</i>	AssignmentRule_t

Table 2: Abstract SBML classes their corresponding C class.

classes use to initialize and free their memory, respectively. Users of the API do not need to worry about these functions.

Fields of the abstract class are **#defined** to a symbol in the class header file. This symbol is used in subclasses *in the order of inheritance*. For example, Species inherits from SBase:



SBase.h defines:

```

/**
 * As shown below, put SBASE_FIELDS as the *first* item of any struct
 * which "is a(n)" SBML object.
 */
#define SBASE_FIELDS \
    SBMLTypeCode_t typecode; \
    char *notes; \
    char *annotation
  
```

and recall Species_t from earlier:

```

typedef struct
{
    SBASE_FIELDS;
    char *name;
    char *compartment;
    double initialAmount;
    char *units;
    int boundaryCondition;
    int charge;
} Species_t;
  
```

The effect is that when the library source is compiled, the first three fields of Species are typecode, notes, and annotation. In fact, every class that inherits from SBase, i.e. all SBML classes, have these same first three fields. Accessing the notes or annotation field of a Species_t *s, or any other SBML object, is the same as for other fields. For example:

```

if (s->notes != NULL)
{
  
```

```

    printf("Notes for Species %s:\n", (s->name == NULL) ? "(null)" : s->name);
    printf("%s", s->notes);
}

```

Setting string fields requires special care to guard against memory leaks. The `XXX_setYYY()` methods must be used. But, `Species` does not define either the notes or annotation fields and as such there are *no* `Species_setNotes()` or `Species_setAnnotation()` methods. Instead, `SBase` defines them:

```
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.

The first argument to these functions is, of course, an object of type `SBase`. Since `Species` inherits from `SBase`, i.e. *Species is an SBase*, it can be used as the first argument to these functions. A slight caveat is a cast is required. To set the notes field of `Species` `s` then:

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

The same applies to all other SBML objects.

Finally, each SBML class has a `typecode` which is initialized when an object is instantiated. The `typecode` is a simple C enumeration, defined in `SBMLTypeCodes.h`:

```

/**
 * An enumeration of SBML 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_KINETIC_LAW
    , SBML_MODEL
    , SBML_PARAMETER
    , SBML_REACTION
    , SBML_SPECIES
    , SBML_SPECIES_REFERENCE
    , SBML_UNIT_DEFINITION
    , SBML_UNIT
    , SBML_ALGEBRAIC_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` contains a list of rules, but a `Rule` in SBML may be of one of four specific types: `AlgebraicRule`, `SpeciesConcentrationRule`, `CompartmentVolumeRule` and `ParameterRule`.

5 Reading SBML Files

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

SBMLDocument_t *readSBML (const char *filename)

Reads the SBML document from the given file 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 e.g.,:

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

These functions return a pointer to an SBMLDocument:

```
/**
 * The SBMLDocument
 */
typedef struct
{
    unsigned int level;
    unsigned int version;

    List_t *error;
    List_t *fatal;
    List_t *warning;

    Model_t *model;
} SBMLDocument_t;
```

The level and version of the SBML document are stored in the first two fields. The last field is the SBML model itself. The three lists record warnings and errors encountered during the XML parse. Each warning or error is a ParseMessage (again from SBMLDocument.h):

```
/**
 * SBMLDocuments contain three Lists of ParseMessages, one for each class
 * of messages that could be triggered during an XML parse: Warnings,
 * Errors and Fatal Errors.
 *
 * Each ParseMessage contains the message itself and the line and column
 * numbers of the XML entity that triggered the message. If line or column
 * information is unavailable, -1 is used.
 */
typedef struct
{
    char *message;
    int line;
    int column;
} ParseMessage_t;
```

While its possible to access these lists directly, convenience functions are provided:

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 $n > \text{getNumErrors}() - 1$.

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. $\text{SBMLDocument_getNumWarnings}(d) == 0$, 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, i.e. $\text{SBMLDocument_getNumErrors}(d) == 0$, 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, i.e. $\text{SBMLDocument_getNumFatals}(d) == 0$, 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)`.

5.1 A Simple Example

The following example is included in the distribution as `readSBML.c`. It is compiled as part of the build process, but is not installed. The 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: warnings or errors (if any are reported) as well as the total read time (in milliseconds).

To run the example, go to the top-level directory where LIBSBML was unpacked and:

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

The exact procedure for compile the example (and linking-in LIBSBML in general) varies from one platform and compiler to another. On Linux or Solaris with GCC the following should work:

```
% gcc -o readSBML readSBML.c -lsbml
```

The example is listed below:

```
1 #include <stdio.h>
2 #include <sys/time.h>
3
4 #include "sbml/SBMLReader.h"
5 #include "sbml/SBMLTypes.h"
6
7 /**
8  * Function Prototypes
9  */
10 unsigned long getCurrentMillis (void);
11 void          reportErrors     (SBMLDocument_t *d);
12
13
14 int
15 main (int argc, char *argv [])
16 {
17     SBMLDocument_t *d;
18     Model_t *m;
19
20     unsigned long start, stop;
21
22
23     if (argc != 2)
24     {
25         printf("usage: readSBML <filename>\n");
26         return 1;
27     }
28
29     start = getCurrentMillis();
30     d     = readSBML(argv[1]);
31     stop  = getCurrentMillis();
32
33     m = d->model;
34
35     printf( "File: %s\n", argv[1]);
36     printf( "      model name: %s\n", m->name );
37     printf( "      unitDefinitions: %d\n", Model_getNumUnitDefinitions(m) );
38     printf( "      compartments: %d\n", Model_getNumCompartments(m) );
39     printf( "      species: %d\n", Model_getNumSpecies(m) );
```

```

40     printf( "           parameters: %d\n", Model_getNumParameters(m) );
41     printf( "           reactions: %d\n", Model_getNumReactions(m) );
42     printf( "           rules: %d\n", Model_getNumRules(m) );
43     printf( "\n");
44
45     printf( "Total Read Time (ms): %lu\n", stop - start);
46
47     SBMLDocument_printWarnings(d, stdout);
48     SBMLDocument_printErrors (d, stdout);
49     SBMLDocument_printFatalS (d, stdout);
50
51     SBMLDocument_free(d);
52     return 0;
53 }
54
55
56 /**
57  * @return the number of milliseconds elapsed since the Epoch.
58  */
59 unsigned long
60 getCurrentMillis (void)
61 {
62     struct timeval tv;
63     unsigned long result = 0;
64
65
66     if (gettimeofday(&tv, NULL) == 0)
67     {
68         result = (tv.tv_sec * 1000) + (tv.tv_usec * .001);
69     }
70
71     return result;
72 }

```

5.2 XML Schema Validation

To validate an SBML document against an SBML (XML) schema as it is being read-in 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_setSchemaFilename (SBMLReader_t *sr, const char *filename)

Sets the schema filename used by this `SBMLReader`. The filename should be either i) an absolute path or ii) relative to the directory contain the SBML file(s) to be read.

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

Sets the schema validation level used by this SBMLReader.

The levels 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.

Once an SBMLReader has been created, there are two variants of `readSBML()` and `readSBMLFromString()` functions described above. These variants can be thought of as methods of the SBMLReader class.

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

Reads the SBML document from the given file and returns a pointer to it.

```
SBMLDocument_t *SBMLReader_readSBMLFromString (SBMLReader_t *sr, 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 e.g.,:

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

Schema violations are reported in the SBMLDocument's list of ParseMessages.

6 Writing SBML Files

The SBMLWriter class is similar to the SBMLReader class. The difference with an SBMLWriter is instead of selecting a schema filename and validation level, an output encoding is chosen instead. The supported output encodings are:

```
/**
 * The character encodings supported by SBMLWriter
 */
typedef enum
{
    CHARACTER_ENCODING_ASCII
    , CHARACTER_ENCODING_UTF_8
    , CHARACTER_ENCODING_UTF_16
    , CHARACTER_ENCODING_ISO_8859_1
    , CHARACTER_ENCODING_INVALID
} CharacterEncoding_t;
```

SBMLWriter.h defines the following functions:

SBMLWriter_t *SBMLWriter_create (void)

Creates a new SBMLWriter and returns a pointer to it. By default the character encoding is ISO 8859-1 (CHARACTER_ENCODING_ISO_8859_1).

void SBMLWriter_free (SBMLWriter_t *sw)

Frees the given SBMLWriter.

void SBMLWriter_setEncoding (SBMLWriter_t *sw, CharacterEncoding_t encoding)

Sets the character encoding for this SBMLWriter to the given CharacterEncoding type.

int SBMLWriter_writeSBML (SBMLWriter_t *sw, SBMLDocument_t *d, const char *filename)

Writes the given SBML document to filename (with the settings provided by this SBMLWriter).

Returns 1 on success and 0 on failure (e.g., if filename could not be opened for writing or the SBMLWriter character encoding is invalid).

char *SBMLWriter_writeSBMLToString (SBMLWriter_t *sw, SBMLDocument_t *d)

Writes the given SBML document to an in-memory string (with the settings provided by this SBMLWriter) 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 (e.g., if the SBMLWriter character encoding is invalid).

Finally, if the default character encoding of ISO 8859-1 is acceptable, there are two simpler equivalents to the above functions:

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

Writes the given SBML document to filename with the settings provided by this SBMLWriter. This convenience function is functionally equivalent to:

```
SBMLWriter_writeSBML(SBMLWriter_create(), d, filename);
```

Returns 1 on success and 0 on failure (e.g., if filename could not be opened for writing or the SBMLWriter character encoding is invalid).

char *writeSBMLToString (SBMLDocument_t *d)

Writes the given SBML document to an in-memory string (with the settings provided by this SBMLWriter) and returns a pointer to it. The string is owned by the caller and should be freed (with `free()`) when no longer needed. This convenience function is functionally equivalent to:

```
SBMLWriter_writeSBMLToString(SBMLWriter_create(), d);
```

Returns NULL on failure (e.g., if the SBMLWriter character encoding is invalid).

A Lists

While List 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 `List_t`. The full set of list methods are:

```
void List_add (List_t *list, void *item)
```

Adds item to this List.

```
void *List_get (List_t *list, unsigned int n)
```

Returns the nth item in this List. If `n > List_size(list) - 1` returns NULL.

```
void *List_remove (List_t *list, unsigned int n)
```

Removes the nth item from this List and returns a pointer to it. If `n > List_size(list) - 1` returns NULL.

```
unsigned int List_size (List_t *list)
```

Returns the number of elements in this List.

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

```
UnitDefinition_t *ud = UnitDefinition_createWith("mmls");  
  
List_add(ud->unit, Unit_createWith(UNIT_KIND_MOLE , 1, -3) );  
List_add(ud->unit, Unit_createWith(UNIT_KIND_LITRE , -1, 0) );  
List_add(ud->unit, Unit_createWith(UNIT_KIND_SECOND, -1, 0) );
```

However, this approach is not preferred. The best reason to use specific `XXX_getYYY()` methods over the List API is that the former are typed to specific items, whereas `List_get()` returns a void pointer that must be cast to a specific type. For example, compare:

```
Unit_t *u = UnitDefinition_getUnit(u, 1);
```

to:

```
Unit_t *u = (Unit_t *) List_get(ud->unit, 1);
```

Further, the first is more readable. The List API is mentioned i) for the sake of completeness and ii) currently the only way to remove an item from a list is to use the API directly:

```
List_remove(ud->unit, 2);
```

has no analog in UnitDefinition. “Remove” convenience methods can be added to the API. This feature was skipped because list item removal seemed like an uncommon operation.