

# libsbml API Reference 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

9 May 2005

LIBSBML Version 2.3.1

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>API Reference</b>	<b>3</b>
2.1	AlgebraicRule.h . . . . .	4
2.2	AssignmentRule.h . . . . .	5
2.3	Compartment.h . . . . .	6
2.4	CompartmentVolumeRule.h . . . . .	9
2.5	EventAssignment.h . . . . .	10
2.6	Event.h . . . . .	11
2.7	FunctionDefinition.h . . . . .	14
2.8	KineticLaw.h . . . . .	16
2.9	ListOf.h . . . . .	18
2.10	Model.h . . . . .	19
2.11	ModifierSpeciesReference.h . . . . .	27
2.12	ParameterRule.h . . . . .	28
2.13	Parameter.h . . . . .	29
2.14	RateRule.h . . . . .	32
2.15	Reaction.h . . . . .	33
2.16	Rule.h . . . . .	37
2.17	RuleType.h . . . . .	38
2.18	SBase.h . . . . .	39
2.19	SBMLDocument.h . . . . .	41
2.20	SBMLReader.h . . . . .	44
2.21	SBMLWriter.h . . . . .	46
2.22	SimpleSpeciesReference.h . . . . .	47
2.23	SpeciesConcentrationRule.h . . . . .	48
2.24	SpeciesReference.h . . . . .	49
2.25	Species.h . . . . .	51
2.26	UnitDefinition.h . . . . .	55
2.27	UnitKind.h . . . . .	58
2.28	Unit.h . . . . .	59
	<b>References</b>	<b>63</b>

---

## 1 Introduction

This manual is a reference for the LIBSBML application programming interface (API). LIBSBML provides C and C++ APIs for reading, writing and manipulating the Systems Biology Markup Language (SBML; Hucka et al., 2001, 2003; Finney and Hucka, 2003). Currently, the library supports 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 and all specifications and source code are freely and publicly available. This document explains the library API in detail, but does not provide general information about LIBSBML, its use or its installation. For that, please consult the LIBSBML *Developer's Manual* (Bornstein, 2004).

---

## 2 API Reference

## 2.1 AlgebraicRule.h

---

### **AlgebraicRule\_t \* AlgebraicRule\_create (void)**

Creates a new AlgebraicRule and returns a pointer to it.

---

### **AlgebraicRule\_t \* AlgebraicRule\_createWith (const char \*formula)**

Creates a new AlgebraicRule with the given formula and returns a pointer to it. This convenience function is functionally equivalent to:

```
AlgebraicRule_t ar = AlgebraicRule_create();    Rule_setFormula((Rule_t )  
ar, formula);
```

---

### **AlgebraicRule\_t \* AlgebraicRule\_createWithMath (ASTNode\_t \*math)**

Creates a new AlgebraicRule with the given math and returns a pointer to it. This convenience function is functionally equivalent to:

```
AlgebraicRule_t ar = AlgebraicRule_create();    Rule_setMath((Rule_t ) ar,  
math);
```

The node is **not copied** and this AlgebraicRule **takes ownership** of it; i.e. subsequent calls to this function or a call to AlgebraicRule.free() will free the ASTNode (and any child nodes).

---

### **void AlgebraicRule\_free (AlgebraicRule\_t \*ar)**

Frees the given AlgebraicRule.

## 2.2 AssignmentRule.h

### **AssignmentRule\_t \* AssignmentRule\_create (void)**

Creates a new AssignmentRule and returns a pointer to it.

In L1 AssignmentRule is an abstract class. It exists solely to provide fields to its subclasses: CompartmentVolumeRule, ParameterRule and SpeciesConcentrationRule.

In L2 the three subclasses are gone and AssignmentRule is concrete; i.e. it may be created, used and destroyed directly.

### **AssignmentRule\_t \* AssignmentRule\_createWith (const char \*variable, ASTNode\_t \*math)**

Creates a new AssignmentRule with the given variable and math and returns a pointer to it. This convenience function is functionally equivalent to:

```
ar = AssignmentRule_create();    AssignmentRule_setVariable(ar, variable);
Rule_setMath((Rule_t ) ar, math);
```

### **void AssignmentRule\_free (AssignmentRule\_t \*ar)**

Frees the given AssignmentRule.

### **void AssignmentRule\_initDefaults (AssignmentRule\_t \*ar)**

The function is kept for backward compatibility with the SBML L1 API.

Initializes the fields of this AssignmentRule to their defaults:

- type = RULE\_TYPE\_SCALAR

### **RuleType\_t AssignmentRule\_getType (const AssignmentRule\_t \*ar)**

Returns the type for this AssignmentRule.

### **const char \* AssignmentRule\_getVariable (const AssignmentRule\_t \*ar)**

Returns the variable for this AssignmentRule.

### **int AssignmentRule\_isSetVariable (const AssignmentRule\_t \*ar)**

Returns 1 if the variable of this AssignmentRule has been set, 0 otherwise.

### **void AssignmentRule\_setType (AssignmentRule\_t \*ar, RuleType\_t rt)**

Sets the type of this Rule to the given RuleType.

### **void AssignmentRule\_setVariable (AssignmentRule\_t \*ar, const char \*sid)**

Sets the variable of this AssignmentRule to a copy of sid.

## 2.3 Compartment.h

**Compartment\_t \* Compartment\_create (void)**

Creates a new Compartment and returns a pointer to it.

**Compartment\_t \* Compartment\_createWith ( const char \*sid, double size, const char \*units, const char \*outside )**

Creates a new Compartment with the given id, size (volume in L1), units and outside and returns a pointer to it. This convenience function is functionally equivalent to:

```
Compartment_t c = Compartment_create();          Compartment_setId(c, id);
Compartment_setSize(c, size); ... ;
```

**void Compartment\_free (Compartment\_t \*c)**

Frees the given Compartment.

**void Compartment\_initDefaults (Compartment\_t \*c)**

Initializes the fields of this Compartment to their defaults:

- volume = 1.0 (L1 only) - spatialDimensions = 3 (L2 only) - constant = 1 (true) (L2 only)

**const char \* Compartment\_getId (const Compartment\_t \*c)**

Returns the id of this Compartment.

**const char \* Compartment\_getName (const Compartment\_t \*c)**

Returns the name of this Compartment.

**unsigned int Compartment\_getSpatialDimensions (const Compartment\_t \*c)**

Returns the spatialDimensions of this Compartment.

**double Compartment\_getSize (const Compartment\_t \*c)**

Returns the size (volume in L1) of this Compartment.

**double Compartment\_getVolume (const Compartment\_t \*c)**

Returns the volume (size in L2) of this Compartment.

**const char \* Compartment\_getUnits (const Compartment\_t \*c)**

Returns the units of this Compartment.

**const char \* Compartment\_getOutside (const Compartment\_t \*c)**

Returns the outside of this Compartment.

**int Compartment\_getConstant (const Compartment\_t \*c)**

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

**int Compartment\_isSetId (const Compartment\_t \*c)**

Returns 1 if the id of this Compartment has been set, 0 otherwise.

**int Compartment\_isSetName (const Compartment\_t \*c)**

Returns 1 if the name of this Compartment has been set, 0 otherwise.

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

**int Compartment\_isSetSize (const Compartment\_t \*c)**

Returns 1 if the size (volume in L1) of this Compartment has been set, 0 otherwise.

**int Compartment\_isSetVolume (const Compartment\_t \*c)**

Returns 1 if the volume (size in L2) of this Compartment has been set, 0 otherwise.

In SBML L1, a Compartment volume has a default value (1.0) and therefore **should always be set**. In L2, volume (size) is optional with no default value and as such may or may not be set.

**int Compartment\_isSetUnits (const Compartment\_t \*c)**

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

**int Compartment\_isSetOutside (const Compartment\_t \*c)**

Returns 1 if the outside of this Compartment has been set, 0 otherwise.

**void Compartment\_moveldToName (Compartment\_t \*c)**

Moves the id field of this Compartment to its name field (iff name is not already set). This method is used for converting from L2 to L1.

**void Compartment\_moveNameToId (Compartment\_t \*c)**

Moves the name field of this Compartment to its id field (iff id is not already set). This method is used for converting from L1 to L2.

**void Compartment\_setId (Compartment\_t \*c, const char \*sid)**

Sets the id of this Compartment to a copy of sid.

**void Compartment\_setName (Compartment\_t \*c, const char \*string)**

Sets the name of this Compartment to a copy of string (SName in L1).

**void Compartment\_setSpatialDimensions (Compartment\_t \*c, unsigned int value)**

Sets the spatialDimensions of this Compartment to value.

If value is not one of [0, 1, 2, 3] the function will have no effect (i.e. spatialDimensions will not be set).

**void Compartment\_setSize (Compartment\_t \*c, double value)**

Sets the size (volume in L1) of this Compartment to value.

**void Compartment\_setVolume (Compartment\_t \*c, double value)**

Sets the volume (size in L2) of this Compartment to value.

**void Compartment\_setUnits (Compartment\_t \*c, const char \*sid)**

Sets the units of this Compartment to a copy of sid.

**void Compartment\_setOutside (Compartment\_t \*c, const char \*sid)**

Sets the outside of this Compartment to a copy of sid.

**void Compartment\_setConstant (Compartment\_t \*c, int value)**

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

**void Compartment\_unsetName (Compartment\_t \*c)**

Unsets the name of this Compartment. This is equivalent to: `safe_free(c->name); c->name = NULL;`

**void Compartment\_unsetSize (Compartment\_t \*c)**

Unsets the size (volume in L1) of this Compartment.

**void Compartment\_unsetVolume (Compartment\_t \*c)**

Unsets the volume (size in L2) of this Compartment.

In SBML L1, a Compartment volume has a default value (1.0) and therefore **should always be set**. In L2, volume (size) is optional with no default value and as such may or may not be set.

**void Compartment\_unsetUnits (Compartment\_t \*c)**

Unsets the units of this Compartment. This is equivalent to: `safe_free(c->units); c->units = NULL;`

**void Compartment\_unsetOutside (Compartment\_t \*c)**

Unsets the outside of this Compartment. This is equivalent to: `safe_free(c->outside); c->outside = NULL;`

**int CompartmentIdCmp (const char \*sid, const Compartment\_t \*c)**

The `CompartmentIdCmp` function compares the string `sid` to `c->id`.

Returns an integer less than, equal to, or greater than zero if `sid` is found to be, respectively, less than, to match or be greater than `c->id`. Returns -1 if either `sid` or `c->id` is NULL.

## 2.4 CompartmentVolumeRule.h

**CompartmentVolumeRule\_t \* CompartmentVolumeRule\_create (void)**

Creates a new CompartmentVolumeRule and returns a pointer to it.

**CompartmentVolumeRule\_t \* CompartmentVolumeRule\_createWith ( const char \*formula, RuleType\_t type, const char \*compartment )**

Creates a new CompartmentVolumeRule with the given formula, type and compartment and returns a pointer to it. This convenience function is functionally equivalent to:

```
CompartmentVolumeRule_t cvr = CompartmentVolumeRule_create();  
Rule_setFormula((Rule_t ) cvr, formula); AssignmentRule_setType((AssignmentRule_t ) cvr, type); ...;
```

**void CompartmentVolumeRule\_free (CompartmentVolumeRule\_t \*cvr)**

Frees the given CompartmentVolumeRule.

**const char \* CompartmentVolumeRule\_getCompartment (const CompartmentVolumeRule\_t \*cvr)**

Returns the compartment of this CompartmentVolumeRule.

**int CompartmentVolumeRule\_isSetCompartment (const CompartmentVolumeRule\_t \*cvr)**

Returns 1 if the compartment of this CompartmentVolumeRule has been set, 0 otherwise.

**void CompartmentVolumeRule\_setCompartment ( CompartmentVolumeRule\_t \*cvr, const char \*sname )**

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

## 2.5 EventAssignment.h

**EventAssignment\_t \* EventAssignment\_create (void)**

Creates a new EventAssignment and returns a pointer to it.

**EventAssignment\_t \* EventAssignment\_createWith (const char \*variable, ASTNode\_t \*math)**

Creates a new EventAssignment with the given variable and math and returns a pointer to it. This convenience function is functionally equivalent to:

```
ea = EventAssignment_create(); EventAssignment_setVariable(ea, variable);
EventAssignment_setMath(ea, math);
```

**void EventAssignment\_free (EventAssignment\_t \*ea)**

Frees the given EventAssignment.

**const char \* EventAssignment\_getVariable (const EventAssignment\_t \*ea)**

Returns the variable of this EventAssignment.

**const ASTNode\_t \* EventAssignment\_getMath (const EventAssignment\_t \*ea)**

Returns the math of this EventAssignment.

**int EventAssignment\_isSetVariable (const EventAssignment\_t \*ea)**

Returns 1 if the variable of this EventAssignment has been set, 0 otherwise.

**int EventAssignment\_isSetMath (const EventAssignment\_t \*ea)**

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

**void EventAssignment\_setVariable (EventAssignment\_t \*ea, const char \*sid)**

Sets the variable of this EventAssignment to a copy of sid.

**void EventAssignment\_setMath (EventAssignment\_t \*ea, ASTNode\_t \*math)**

Sets the math of this EventAssignment to the given ASTNode.

The node **is not copied** and this EventAssignment **takes ownership** of it; i.e. subsequent calls to this function or a call to EventAssignment\_free() will free the ASTNode (and any child nodes).

## 2.6 Event.h

**Event\_t \* Event\_create (void)**

Creates a new Event and returns a pointer to it.

**Event\_t \* Event\_createWith (const char \*sid, ASTNode\_t \*trigger)**

Creates a new Event with the given id and trigger and returns a pointer to it. This convenience function is functionally equivalent to:

```
e = Event_create(); Event_setId(e, id); Event_setTrigger(e, trigger);
```

**void Event\_free (Event\_t \*e)**

Frees the given Event.

**const char \* Event\_getId (const Event\_t \*e)**

Returns the id of this Event.

**const char \* Event\_getName (const Event\_t \*e)**

Returns the name of this Event.

**const ASTNode\_t \* Event\_getTrigger (const Event\_t \*e)**

Returns the trigger of this Event.

**const ASTNode\_t \* Event\_getDelay (const Event\_t \*e)**

Returns the delay of this Event.

**const char \* Event\_getTimeUnits (const Event\_t \*e)**

Returns the timeUnits of this Event.

**int Event\_isSetId (const Event\_t \*e)**

Returns 1 if the id of this Event has been set, 0 otherwise.

**int Event\_isSetName (const Event\_t \*e)**

Returns 1 if the name of this Event has been set, 0 otherwise.

**int Event\_isSetTrigger (const Event\_t \*e)**

Returns 1 if the trigger of this Event has been set, 0 otherwise.

**int Event\_isSetDelay (const Event\_t \*e)**

Returns 1 if the delay of this Event has been set, 0 otherwise.

**int Event\_isSetTimeUnits (const Event.t \*e)**

Returns 1 if the timeUnits of this Event has been set, 0 otherwise.

**void Event\_setId (Event.t \*e, const char \*sid)**

Sets the id of this Event to a copy of sid.

**void Event\_setName (Event.t \*e, const char \*string)**

Sets the name of this Event to a copy of string.

**void Event\_setTrigger (Event.t \*e, ASTNode.t \*math)**

Sets the trigger of this Event to the given ASTNode.

The node is **not copied** and this Event **takes ownership** of it; i.e. subsequent calls to this function or a call to Event\_free() will free the ASTNode (and any child nodes).

**void Event\_setDelay (Event.t \*e, ASTNode.t \*math)**

Sets the delay of this Event to the given ASTNode.

The node is **not copied** and this Event **takes ownership** of it; i.e. subsequent calls to this function or a call to Event\_free() will free the ASTNode (and any child nodes).

**void Event\_setTimeUnits (Event.t \*e, const char \*sid)**

Sets the timeUnits of this Event to a copy of sid.

**void Event\_unsetId (Event.t \*e)**

Unsets the id of this Event. This is equivalent to: safe\_free(e->id); e->id = NULL;

**void Event\_unsetName (Event.t \*e)**

Unsets the name of this Event. This is equivalent to: safe\_free(e->name); e->name = NULL;

**void Event\_unsetDelay (Event.t \*e)**

Unsets the delay of this Event. This is equivalent to: ASTNode\_free(e->delay); e->delay = NULL;

**void Event\_unsetTimeUnits (Event.t \*e)**

Unsets the timeUnits of this Event. This is equivalent to: safe\_free(e->timeUnits); e->timeUnits = NULL;

**void Event\_addEventAssignment (Event.t \*e, EventAssignment.t \*ea)**

Appends the given EventAssignment to this Event.

**ListOf.t \* Event\_getListOfEventAssignments (Event.t \*e)**

Returns the list of EventAssignments for this Event.

---

**EventAssignment\_t \* Event\_getEventAssignment (const Event\_t \*e, unsigned int n)**

Returns the nth EventAssignment of this Event.

---

**unsigned int Event\_getNumEventAssignments (const Event\_t \*e)**

Returns the number of EventAssignments in this Event.

---

**int EventIdCmp (const char \*sid, const Event\_t \*e)**

The EventIdCmp function compares the string sid to e->id.

Returns an integer less than, equal to, or greater than zero if sid is found to be, respectively, less than, to match or be greater than e->id. Returns -1 if either sid or e->id is NULL.

## 2.7 FunctionDefinition.h

**FunctionDefinition\_t \* FunctionDefinition\_create (void)**

Creates a new FunctionDefinition and returns a pointer to it.

**FunctionDefinition\_t \* FunctionDefinition\_createWith (const char \*sid, ASTNode\_t \*math)**

Creates a new FunctionDefinition with the given id and math and returns a pointer to it. This convenience function is functionally equivalent to:

```
fd = FunctionDefinition_create();      FunctionDefinition_setId(fd, id);  
FunctionDefinition_setMath(fd, math);
```

**void FunctionDefinition\_free (FunctionDefinition\_t \*fd)**

Frees the given FunctionDefinition.

**const char \* FunctionDefinition\_getId (const FunctionDefinition\_t \*fd)**

Returns the id of this FunctionDefinition.

**const char \* FunctionDefinition\_getName (const FunctionDefinition\_t \*fd)**

Returns the name of this FunctionDefinition.

**const ASTNode\_t \* FunctionDefinition\_getMath (const FunctionDefinition\_t \*fd)**

Returns the math of this FunctionDefinition.

**int FunctionDefinition\_isSetId (const FunctionDefinition\_t \*fd)**

Returns 1 if the id of this FunctionDefinition has been set, 0 otherwise.

**int FunctionDefinition\_isSetName (const FunctionDefinition\_t \*fd)**

Returns 1 if the name of this FunctionDefinition has been set, 0 otherwise.

**int FunctionDefinition\_isSetMath (const FunctionDefinition\_t \*fd)**

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

**void FunctionDefinition\_setId (FunctionDefinition\_t \*fd, const char \*sid)**

Sets the id of this FunctionDefinition to a copy of sid.

**void FunctionDefinition\_setName (FunctionDefinition\_t \*fd, const char \*string)**

Sets the name of this FunctionDefinition to a copy of string.

**void FunctionDefinition\_setMath (FunctionDefinition\_t \*fd, ASTNode\_t \*math)**

Sets the math of this FunctionDefinition to the given ASTNode.

The node **is not copied** and this FunctionDefinition **takes ownership** of it; i.e. subsequent calls to this function or a call to FunctionDefinition\_free() will free the ASTNode (and any child nodes).

---

**void FunctionDefinition\_unsetName (FunctionDefinition\_t \*fd)**

Unsets the name of this FunctionDefinition. This is equivalent to: `safe_free(fd->name);`  
`fd->name = NULL;`

---

**int FunctionDefinitionIdCmp (const char \*sid, const FunctionDefinition\_t \*fd)**

The FunctionDefinitionIdCmp function compares the string sid to fd->id.  
Returns an integer less than, equal to, or greater than zero if sid is found to be, respectively, less than, to match or be greater than fd->id. Returns -1 if either sid or fd->id is NULL.

## 2.8 KineticLaw.h

**KineticLaw\_t \* KineticLaw\_create (void)**

Creates a new KineticLaw and returns a pointer to it.

**KineticLaw\_t \* KineticLaw\_createWith ( const char \*formula, const char \*timeUnits, const char \*substanceUnits )**

Creates a new KineticLaw with the given formula, timeUnits and substanceUnits and returns a pointer to it. This convenience function is functionally equivalent to:

```
KineticLaw_t kl = KineticLaw_create();           KineticLaw_setFormula(kl,
formula); KineticLaw_setTimeUnits(kl, timeUnits); ...;
```

**void KineticLaw\_free (KineticLaw\_t \*kl)**

Frees the given KineticLaw.

**const char \* KineticLaw\_getFormula (const KineticLaw\_t \*kl)**

Returns the formula of this KineticLaw.

**const ASTNode\_t \* KineticLaw\_getMath (const KineticLaw\_t \*kl)**

Returns the math of this KineticLaw.

**ListOf\_t \* KineticLaw\_getListOfParameters (KineticLaw\_t \*kl)**

Returns the list of Parameters for this KineticLaw.

**const char \* KineticLaw\_getTimeUnits (const KineticLaw\_t \*kl)**

Returns the timeUnits of this KineticLaw.

**const char \* KineticLaw\_getSubstanceUnits (const KineticLaw\_t \*kl)**

Returns the substanceUnits of this KineticLaw.

**int KineticLaw\_isSetFormula (const KineticLaw\_t \*kl)**

Returns true (non-zero) if the formula (or equivalently the math) of this KineticLaw has been set, false (0) otherwise.

**int KineticLaw\_isSetMath (const KineticLaw\_t \*kl)**

Returns true if the math (or equivalently the formula) of this KineticLaw has been set, false otherwise.

**int KineticLaw\_isSetTimeUnits (const KineticLaw\_t \*kl)**

Returns 1 if the timeUnits of this KineticLaw has been set, 0 otherwise.

**int KineticLaw\_isSetSubstanceUnits (const KineticLaw\_t \*kl)**

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

**void KineticLaw\_setFormula (KineticLaw\_t \*kl, const char \*string)**

Sets the formula of this KineticLaw to a copy of string.

**void KineticLaw\_setMath (KineticLaw\_t \*kl, ASTNode\_t \*math)**

Sets the math of this KineticLaw to the given ASTNode.

The node is **not copied** and this KineticLaw **takes ownership** of it; i.e. subsequent calls to this function or a call to KineticLaw\_free() will free the ASTNode (and any child nodes).

**void KineticLaw\_setFormulaFromMath (const KineticLaw\_t \*kl)**

This function is no longer necessary. LibSBML now keeps formula strings and math ASTs synchronized automatically. The function is kept around for backward compatibility (and is used internally).

**void KineticLaw\_setMathFromFormula (const KineticLaw\_t \*kl)**

This function is no longer necessary. LibSBML now keeps formula strings and math ASTs synchronized automatically. The function is kept around for backward compatibility (and is used internally).

**void KineticLaw\_setTimeUnits (KineticLaw\_t \*kl, const char \*sname)**

Sets the timeUnits of this KineticLaw to a copy of sname.

**void KineticLaw\_setSubstanceUnits (KineticLaw\_t \*kl, const char \*sname)**

Sets the substanceUnits of this KineticLaw to a copy of sname.

**void KineticLaw\_addParameter (KineticLaw\_t \*kl, Parameter\_t \*p)**

Adds the given Parameter to this KineticLaw.

**Parameter\_t \* KineticLaw\_getParameter (const KineticLaw\_t \*kl, unsigned int n)**

Returns the nth Parameter of this KineticLaw.

**unsigned int KineticLaw\_getNumParameters (const KineticLaw\_t \*kl)**

Returns the number of Parameters in this KineticLaw.

**void KineticLaw\_unsetTimeUnits (KineticLaw\_t \*kl)**

Unsets the timeUnits of this KineticLaw. This is equivalent to: safe\_free(kl->timeUnits); kl->timeUnits = NULL;

**void KineticLaw\_unsetSubstanceUnits (KineticLaw\_t \*kl)**

Unsets the substanceUnits of this KineticLaw. This is equivalent to: safe\_free(kl->substanceUnits); kl->substanceUnits = NULL;

## 2.9 ListOf.h

**ListOf\_t \* ListOf\_create (void)**

Creates a new ListOf and returns a pointer to it.

**void ListOf\_free (ListOf\_t \*lo)**

Frees the given ListOf and its constituent items.  
This function assumes each item in the list is derived from SBase.

**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  $n \geq \text{ListOf\_getNumItems}(\text{list})$  returns NULL.

**unsigned int ListOf\_getNumItems (const ListOf\_t \*lo)**

Returns the number of items in this List.

**void ListOf\_prepend (ListOf\_t \*lo, void \*item)**

Adds item to the beginning of this ListOf.

**void \* ListOf\_remove (ListOf\_t \*lo, unsigned int n)**

Removes the nth item from this List and returns a pointer to it. If  $n \geq \text{ListOf\_getNumItems}(\text{list})$  returns NULL.

## 2.10 Model.h

### **Model.t \* Model\_create (void)**

Creates a new Model and returns a pointer to it.

### **Model.t \* Model\_createWith (const char \*sid)**

Creates a new Model with the given id and returns a pointer to it. This convenience function is functionally equivalent to:

```
Model_setId(Model_create(), sid);
```

### **Model.t \* Model\_createWithName (const char \*string)**

Creates a new Model with the given name and returns a pointer to it. This convenience function is functionally equivalent to:

```
Model_setName(Model_create(), string);
```

### **void Model\_free (Model.t \*m)**

Frees the given Model.

### **const char \* Model\_getId (const Model.t \*m)**

Returns the id of this Model.

### **const char \* Model\_getName (const Model.t \*m)**

Returns the name of this Model.

### **int Model\_isSetId (const Model.t \*m)**

Returns 1 if the id of this Model has been set, 0 otherwise.

### **int Model\_isSetName (const Model.t \*m)**

Returns 1 if the name of this Model has been set, 0 otherwise.

### **void Model\_moveAllIdsToNames (Model.t \*m)**

Moves the id field to the name field for this Model and all of its constituent UnitDefinitions, Compartments, Species, Parameters, and Reactions. This method is used for converting from L2 to L1.

NOTE: Any object with its name field already set will be skipped.

@see moveIdToName

### **void Model\_moveAllNamesToIds (Model.t \*m)**

Moves the name field to the id field for this Model and all of its constituent UnitDefinitions, Compartments, Species, Parameters, and Reactions. This method is used for converting from L1 to L2.

NOTE: Any object with its id field already set will be skipped.

@see moveNameToId

---

**void Model\_moveIdToName (Model.t \*m)**

Moves the id field of this Model to its name field (iff name is not already set). This method is used for converting from L2 to L1.

---

**void Model\_moveNameToId (Model.t \*m)**

Moves the name field of this Model to its id field (iff id is not already set). This method is used for converting from L1 to L2.

---

**void Model\_setId (Model.t \*m, const char \*sid)**

Sets the id of this Model to a copy of sid.

---

**void Model\_setName (Model.t \*m, const char \*string)**

Sets the name of this Model to a copy of string (SName in L1).

---

**void Model\_unsetId (Model.t \*m)**

Unsets the id of this Model. This is equivalent to: safe\_free(m->id); m->id = NULL;

---

**void Model\_unsetName (Model.t \*m)**

Unsets the name of this Model. This is equivalent to: safe\_free(m->name); m->name = NULL;

---

**FunctionDefinition.t \* Model\_createFunctionDefinition (Model.t \*m)**

Creates a new FunctionDefinition inside this Model and returns a pointer to it. This convenience function is functionally equivalent to:

```
Model_addFunctionDefinition(m, FunctionDefinition_create());
```

---

**UnitDefinition.t \* Model\_createUnitDefinition (Model.t \*m)**

Creates a new UnitDefinition inside this Model and returns a pointer to it. This convenience function is functionally equivalent to:

```
Model_addUnitDefinition(m, UnitDefinition_create());
```

---

**Unit.t \* Model\_createUnit (Model.t \*m)**

Creates a new Unit inside this Model and returns a pointer to it. The Unit is added to the last UnitDefinition created.

If a UnitDefinitions does not exist for this model, a new Unit is not created and NULL is returned.

---

**Compartment.t \* Model\_createCompartment (Model.t \*m)**

Creates a new Compartment inside this Model and returns a pointer to it. This convenience function is functionally equivalent to:

```
Model_addCompartment(m, Compartment_create());
```

---

**Species\_t \* Model\_createSpecies (Model\_t \*m)**

Creates a new Species inside this Model and returns a pointer to it. This convenience function is functionally equivalent to:

```
Model_addSpecies(m, Species_create());
```

---

**Parameter\_t \* Model\_createParameter (Model\_t \*m)**

Creates a new Parameter inside this Model and returns a pointer to it. This convenience function is functionally equivalent to:

```
Model_addParameter(m, Parameter_create());
```

---

**AssignmentRule\_t \* Model\_createAssignmentRule (Model\_t \*m)**

Creates a new AssignmentRule inside this Model and returns a pointer to it. This convenience function is functionally equivalent to:

```
Model_addRule(m, AssignmentRule_create());  
(L2 only)
```

---

**RateRule\_t \* Model\_createRateRule (Model\_t \*m)**

Creates a new RateRule inside this Model and returns a pointer to it. This convenience function is functionally equivalent to:

```
Model_addRule(m, RateRule_create());  
(L2 only)
```

---

**AlgebraicRule\_t \* Model\_createAlgebraicRule (Model\_t \*m)**

Creates a new AlgebraicRule inside this Model and returns a pointer to it. This convenience function is functionally equivalent to:

```
Model_addRule(m, AlgebraicRule_create());
```

---

**CompartmentVolumeRule\_t \* Model\_createCompartmentVolumeRule (Model\_t \*m)**

Creates a new CompartmentVolumeRule inside this Model and returns a pointer to it. This convenience function is functionally equivalent to:

```
Model_addRule(m, CompartmentVolumeRule_create());
```

---

**ParameterRule\_t \* Model\_createParameterRule (Model\_t \*m)**

Creates a new ParameterRule inside this Model and returns a pointer to it. This convenience function is functionally equivalent to:

```
Model_addRule(m, ParameterRule_create());
```

---

**SpeciesConcentrationRule\_t \* Model\_createSpeciesConcentrationRule (Model\_t \*m)**

Creates a new SpeciesConcentrationRule inside this Model and returns a pointer to it. This convenience function is functionally equivalent to:

```
Model_addRule(m, SpeciesConcentrationRule_create());
```

---

**Reaction\_t \* Model\_createReaction (Model\_t \*m)**

Creates a new Reaction inside this Model and returns a pointer to it. This convenience function is functionally equivalent to:

```
Model_addRule(m, Reaction_create());
```

---

**SpeciesReference\_t \* Model\_createReactant (Model\_t \*m)**

Creates a new Reactant (i.e. SpeciesReference) inside this Model and returns a pointer to it. The SpeciesReference is added to the reactants of the last Reaction created. If a Reaction does not exist for this model, a new SpeciesReference is not created and NULL is returned.

---

**SpeciesReference\_t \* Model\_createProduct (Model\_t \*m)**

Creates a new Product (i.e. SpeciesReference) inside this Model and returns a pointer to it. The SpeciesReference is added to the products of the last Reaction created. If a Reaction does not exist for this model, a new SpeciesReference is not created and NULL is returned.

---

**ModifierSpeciesReference\_t \* Model\_createModifier (Model\_t \*m)**

Creates a new Modifier (i.e. ModifierSpeciesReference) inside this Model and returns a pointer to it. The ModifierSpeciesReference is added to the modifiers of the last Reaction created. If a Reaction does not exist for this model, a new ModifierSpeciesReference is not created and NULL is returned.

---

**KineticLaw\_t \* Model\_createKineticLaw (Model\_t \*m)**

Creates a new KineticLaw inside this Model and returns a pointer to it. The KineticLaw is associated with the last Reaction created. If a Reaction does not exist for this model, or a Reaction does exist, but already has a KineticLaw, a new KineticLaw is not created and NULL is returned.

---

**Parameter\_t \* Model\_createKineticLawParameter (Model\_t \*m)**

Creates a new Parameter (of a KineticLaw) inside this Model and returns a pointer to it. The Parameter is associated with the KineticLaw of the last Reaction created. If a Reaction does not exist for this model, or a KineticLaw for the Reaction, a new Parameter is not created and NULL is returned.

---

**Event\_t \* Model\_createEvent (Model\_t \*m)**

Creates a new Event inside this Model and returns a pointer to it. This convenience function is functionally equivalent to:  
`Model_addEvent(m, Event_create());`

---

**EventAssignment\_t \* Model\_createEventAssignment (Model\_t \*m)**

Creates a new EventAssignment inside this Model and returns a pointer to it. The EventAssignment is added to the the last Event created. If an Event does not exist for this model, a new EventAssignment is not created and NULL is returned.

---

**void Model\_addFunctionDefinition (Model\_t \*m, FunctionDefinition\_t \*fd)**

Adds the given FunctionDefinition to this Model.

---

**void Model\_addUnitDefinition (Model\_t \*m, UnitDefinition\_t \*ud)**

Adds the given UnitDefinition to this Model.

**void Model.addCompartment (Model.t \*m, Compartment.t \*c)**

Adds the given Compartment to this Model.

**void Model.addSpecies (Model.t \*m, Species.t \*s)**

Adds the given Species to this Model.

**void Model.addParameter (Model.t \*m, Parameter.t \*p)**

Adds the given Parameter to this Model.

**void Model.addRule (Model.t \*m, Rule.t \*r)**

Adds the given Rule to this Model.

**void Model.addReaction (Model.t \*m, Reaction.t \*r)**

Adds the given Reaction to this Model.

**void Model.addEvent (Model.t \*m, Event.t \*e)**

Adds the given Event to this Model.

**ListOf.t \* Model.getListOfFunctionDefinitions (Model.t \*m)**

Returns the list of FunctionDefinitions for this Model.

**ListOf.t \* Model.getListOfUnitDefinitions (Model.t \*m)**

Returns the list of UnitDefinitions for this Model.

**ListOf.t \* Model.getListOfCompartments (Model.t \*m)**

Returns the list of Compartments for this Model.

**ListOf.t \* Model.getListOfSpecies (Model.t \*m)**

Returns the list of Species for this Model.

**ListOf.t \* Model.getListOfParameters (Model.t \*m)**

Returns the list of Parameters for this Model.

**ListOf.t \* Model.getListOfRules (Model.t \*m)**

Returns the list of Rules for this Model.

**ListOf.t \* Model.getListOfReactions (Model.t \*m)**

Returns the list of Rules for this Model.

**ListOf.t \* Model.getListOfEvents (Model.t \*m)**

Returns the list of Rules for this Model.

---

**FunctionDefinition\_t \* Model\_getFunctionDefinition (const Model\_t \*m, unsigned int n)**

Returns the nth FunctionDefinition of this Model.

---

**FunctionDefinition\_t \* Model\_getFunctionDefinitionById (const Model\_t \*m, const char \*sid)**

Returns the FunctionDefinition in this Model with the given id or NULL if no such FunctionDefinition exists.

---

**UnitDefinition\_t \* Model\_getUnitDefinition (const Model\_t \*m, unsigned int n)**

Returns the nth UnitDefinition of this Model.

---

**UnitDefinition\_t \* Model\_getUnitDefinitionById (const Model\_t \*m, const char \*sid)**

Returns the UnitDefinition in this Model with the given id or NULL if no such UnitDefinition exists.

---

**Compartment\_t \* Model\_getCompartment (const Model\_t \*m, unsigned int n)**

Returns the nth Compartment of this Model.

---

**Compartment\_t \* Model\_getCompartmentById (const Model\_t \*m, const char \*sid)**

Returns the Compartment in this Model with the given id or NULL if no such Compartment exists.

---

**Species\_t \* Model\_getSpecies (const Model\_t \*m, unsigned int n)**

Returns the nth Species of this Model.

---

**Species\_t \* Model\_getSpeciesById (const Model\_t \*m, const char \*sid)**

Returns the Species in this Model with the given id or NULL if no such Species exists.

---

**Parameter\_t \* Model\_getParameter (const Model\_t \*m, unsigned int n)**

Returns the nth Parameter of this Model.

---

**Parameter\_t \* Model\_getParameterById (const Model\_t \*m, const char \*sid)**

Returns the Parameter in this Model with the given id or NULL if no such Parameter exists.

---

**Rule\_t \* Model\_getRule (const Model\_t \*m, unsigned int n)**

Returns the nth Rule of this Model.

---

**Reaction\_t \* Model\_getReaction (const Model\_t \*m, unsigned int n)**

Returns the nth Reaction of this Model.

---

**Reaction\_t \* Model\_getReactionById (const Model\_t \*m, const char \*sid)**

Returns the Reaction in this Model with the given id or NULL if no such Reaction exists.

**Event\_t \* Model\_getEvent (const Model\_t \*m, unsigned int n)**

Returns the nth Event of this Model.

**Event\_t \* Model\_getEventById (const Model\_t \*m, const char \*sid)**

Returns the Event in this Model with the given id or NULL if no such Event exists.

**unsigned int Model\_getNumFunctionDefinitions (const Model\_t \*m)**

Returns the number of FunctionDefinitions in this Model.

**unsigned int Model\_getNumUnitDefinitions (const Model\_t \*m)**

Returns the number of UnitDefinitions in this Model.

**unsigned int Model\_getNumCompartments (const Model\_t \*m)**

Returns the number of Compartments in this Model.

**unsigned int Model\_getNumSpecies (const Model\_t \*m)**

Returns the number of Species in this Model.

**unsigned int Model\_getNumSpeciesWithBoundaryCondition (const Model\_t \*m)**

Returns the number of Species in this Model with boundaryCondition set to true.

**unsigned int Model\_getNumParameters (const Model\_t \*m)**

Returns the number of Parameters in this Model. Parameters defined in KineticLaws are not included.

**unsigned int Model\_getNumRules (const Model\_t \*m)**

Returns the number of Rules in this Model.

**unsigned int Model\_getNumReactions (const Model\_t \*m)**

Returns the number of Reactions in this Model.

**unsigned int Model\_getNumEvents (const Model\_t \*m)**

Returns the number of Events in this Model.

**ListOf\_t \* Model\_getListOfLayouts (Model\_t \*m)**

Returns a reference to the ListOf object that holds the layouts.

**Layout\_t \* Model\_getLayout (Model\_t \*m, unsigned int index)**

Returns the layout object that belongs to the given index. If the index is invalid, NULL is returned.

---

**void Model\_addLayout (Model\_t \*m, Layout\_t \*layout)**

Adds a copy of the layout object to the list of layouts.

---

**Layout\_t \* Model\_createLayout (Model\_t \*m)**

Creates a new layout object and adds it to the list of layout objects. A reference to the newly created object is returned.

## 2.11 ModifierSpeciesReference.h

**ModifierSpeciesReference\_t \* ModifierSpeciesReference\_create (void)**

Creates a new ModifierSpeciesReference and returns a pointer to it.

**ModifierSpeciesReference\_t \* ModifierSpeciesReference\_createWith (const char \*species)**

Creates a new ModifierSpeciesReference with the given species and returns a pointer to it. This convenience function is functionally equivalent to:

```
ModifierSpeciesReference_t msr = ModifierSpeciesReference_create();  
ModifierSpeciesReference_setSpecies(msr, species);
```

**void ModifierSpeciesReference\_free (ModifierSpeciesReference\_t \*msr)**

Frees the given ModifierSpeciesReference.

**const char \* ModifierSpeciesReference\_getSpecies (const ModifierSpeciesReference\_t \*msr)**

Returns the species for this ModifierSpeciesReference.

**int ModifierSpeciesReference\_isSetSpecies (const ModifierSpeciesReference\_t \*msr)**

Returns 1 if the species for this ModifierSpeciesReference has been set, 0 otherwise.

**void ModifierSpeciesReference\_setSpecies ( ModifierSpeciesReference\_t \*msr, const char \*sid )**

Sets the species of this ModifierSpeciesReference to a copy of sid.

## 2.12 ParameterRule.h

**ParameterRule\_t \* ParameterRule.create (void)**

Creates a new ParameterRule and returns a pointer to it.

**ParameterRule\_t \* ParameterRule.createWith ( const char \*formula, RuleType\_t type, const char \*name )**

Creates a new ParameterRule with the given formula, type, and name and returns a pointer to it. This convenience function is functionally equivalent to:

```
ParameterRule_t pr = ParameterRule.create();      Rule_setFormula((Rule_t )
pr, formula); scr->type = type; ...;
```

**void ParameterRule.free (ParameterRule\_t \*pr)**

Frees the given ParameterRule.

**const char \* ParameterRule.getName (const ParameterRule\_t \*pr)**

Returns the (Parameter) name for this ParameterRule.

**const char \* ParameterRule.getUnits (const ParameterRule\_t \*pr)**

Returns the units for this ParameterRule.

**int ParameterRule.isSetName (const ParameterRule\_t \*pr)**

Returns 1 if the (Parameter) name for this ParameterRule has been set, 0 otherwise.

**int ParameterRule.isSetUnits (const ParameterRule\_t \*pr)**

Returns 1 if the units for this ParameterRule has been set, 0 otherwise.

**void ParameterRule.setName (ParameterRule\_t \*pr, const char \*sname)**

Sets the (Parameter) name for this ParameterRule to a copy of sname.

**void ParameterRule.setUnits (ParameterRule\_t \*pr, const char \*sname)**

Sets the units for this ParameterRule to a copy of sname.

**void ParameterRule.unsetUnits (ParameterRule\_t \*pr)**

Unsets the units for this ParameterRule. This is equivalent to: safe\_free(pr->units); pr->units = NULL;

## 2.13 Parameter.h

**Parameter\_t \* Parameter\_create (void)**

Creates a new Parameter and returns a pointer to it.

**Parameter\_t \* Parameter\_createWith (const char \*sid, double value, const char \*units)**

Creates a new Parameter with the given id, value and units and returns a pointer to it. This convenience function is functionally equivalent to:

```
Parameter_t p = Parameter_create();           Parameter_setId(p, id);
Parameter_setValue(p, value); ... ;
```

**void Parameter\_free (Parameter\_t \*p)**

Frees the given Parameter.

**void Parameter\_initDefaults (Parameter\_t \*p)**

Initializes the fields of this Parameter to their defaults:  
- constant = 1 (true) (L2 only)

**const char \* Parameter\_getId (const Parameter\_t \*p)**

Returns the id of this Parameter.

**const char \* Parameter\_getName (const Parameter\_t \*p)**

Returns the name of this Parameter.

**double Parameter\_getValue (const Parameter\_t \*p)**

Returns the value of this Parameter.

**const char \* Parameter\_getUnits (const Parameter\_t \*p)**

Returns the units of this Parameter.

**int Parameter\_getConstant (const Parameter\_t \*p)**

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

**int Parameter\_isSetId (const Parameter\_t \*p)**

Returns 1 if the id of this Parameter has been set, 0 otherwise.

**int Parameter\_isSetName (const Parameter\_t \*p)**

Returns 1 if the name of this Parameter has been set, 0 otherwise.

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

---

**int Parameter\_isSetValue (const Parameter\_t \*p)**

Returns 1 if the value of this Parameter has been set, 0 otherwise.

In SBML L1v1, a Parameter value is required and therefore **should always be set**. In L1v2 and beyond, a value is optional and as such may or may not be set.

---

**int Parameter\_isSetUnits (const Parameter\_t \*p)**

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

---

**void Parameter\_moveIdToName (Parameter\_t \*p)**

Moves the id field of this Parameter to its name field (iff name is not already set). This method is used for converting from L2 to L1.

---

**void Parameter\_moveNameToId (Parameter\_t \*p)**

Moves the id field of this Parameter to its name field (iff name is not already set). This method is used for converting from L2 to L1.

---

**void Parameter\_setId (Parameter\_t \*p, const char \*sid)**

Sets the id of this Parameter to a copy of sid.

---

**void Parameter\_setName (Parameter\_t \*p, const char \*string)**

Sets the name of this Parameter to a copy of string (SName in L1).

---

**void Parameter\_setValue (Parameter\_t \*p, double value)**

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

---

**void Parameter\_setUnits (Parameter\_t \*p, const char \*sid)**

Sets the units of this Parameter to a copy of sid.

---

**void Parameter\_setConstant (Parameter\_t \*p, int value)**

Sets the constant of this Parameter to value (boolean).

---

**void Parameter\_unsetName (Parameter\_t \*p)**

Unsets the name of this Parameter. This is equivalent to: `safe_free(p->name); p->name = NULL;`

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

---

**void Parameter\_unsetValue (Parameter\_t \*p)**

Unsets the value of this Parameter.

In SBML L1v1, a Parameter value is required and therefore **should always be set**. In L1v2 and beyond, a value is optional and as such may or may not be set.

---

**void Parameter.unsetUnits (Parameter.t \*p)**

Unsets the units of this Parameter. This is equivalent to: `safe.free(p->units); p->units = NULL;`

---

**int ParameterIdCmp (const char \*sid, const Parameter.t \*p)**

The `ParameterIdCmp` function compares the string `sid` to `p->id`.

Returns an integer less than, equal to, or greater than zero if `sid` is found to be, respectively, less than, to match or be greater than `p->id`. Returns -1 if either `sid` or `p->id` is `NULL`.

## 2.14 RateRule.h

**RateRule\_t \* RateRule\_create (void)**

Creates a new RateRule and returns a pointer to it.

**RateRule\_t \* RateRule\_createWith (const char \*variable, ASTNode\_t \*math)**

Creates a new RateRule with the given variable and math and returns a pointer to it. This convenience function is functionally equivalent to:

```
rr = RateRule_create();           RateRule_setVariable(rr, variable);
Rule_setMath((Rule_t ) rr, math);
```

**void RateRule\_free (RateRule\_t \*rr)**

Frees the given RateRule.

**const char \* RateRule\_getVariable (const RateRule\_t \*rr)**

Returns the variable for this RateRule.

**int RateRule\_isSetVariable (const RateRule\_t \*rr)**

Returns 1 if the variable of this RateRule has been set, 0 otherwise.

**void RateRule\_setVariable (RateRule\_t \*rr, const char \*sid)**

Sets the variable of this RateRule to a copy of sid.

## 2.15 Reaction.h

**Reaction\_t \* Reaction\_create (void)**

Creates a new Reaction and returns a pointer to it.

**Reaction\_t \* Reaction\_createWith ( const char \*sid, KineticLaw\_t \*kl, int reversible, int fast )**

Creates a new Reaction with the given id, KineticLaw, reversible and fast and returns a pointer to it. This convenience function is functionally equivalent to:

```
Reaction_t r = Reaction_create();           Reaction_setId(r, sid);
Reaction_setKineticLaw(r, kl); ...;
```

**void Reaction\_free (Reaction\_t \*r)**

Frees the given Reaction.

**void Reaction\_initDefaults (Reaction\_t \*r)**

Initializes the fields of this Reaction to their defaults:

- reversible = 1 (true) - fast = 0 (false) (L1 only)

**const char \* Reaction\_getId (const Reaction\_t \*r)**

Returns the id of this Reaction.

**const char \* Reaction\_getName (const Reaction\_t \*r)**

Returns the name of this Reaction.

**KineticLaw\_t \* Reaction\_getKineticLaw (const Reaction\_t \*r)**

Returns the KineticLaw of this Reaction.

**int Reaction\_getReversible (const Reaction\_t \*r)**

Returns the reversible status of this Reaction.

**int Reaction\_getFast (const Reaction\_t \*r)**

Returns the fast status of this Reaction.

**int Reaction\_isSetId (const Reaction\_t \*r)**

Returns 1 if the id of this Reaction has been set, 0 otherwise.

**int Reaction\_isSetName (const Reaction\_t \*r)**

Returns 1 if the name of this Reaction has been set, 0 otherwise.

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

**int Reaction\_isSetKineticLaw (const Reaction\_t \*r)**

Returns 1 if the KineticLaw of this Reaction has been set, 0 otherwise.

**int Reaction.isSetFast (const Reaction.t \*r)**

Returns 1 if the fast status of this Reaction has been set, 0 otherwise.  
In L1, fast is optional with a default of false, which means it is effectively always set.  
In L2, however, fast is optional with no default value, so it may or may not be set to a specific value.

**void Reaction.moveIdToName (Reaction.t \*r)**

Moves the id field of this Reaction to its name field (iff name is not already set). This method is used for converting from L2 to L1.

**void Reaction.moveNameToId (Reaction.t \*r)**

Moves the name field of this Reaction to its id field (iff id is not already set). This method is used for converting from L1 to L2.

**void Reaction.setId (Reaction.t \*r, const char \*sid)**

Sets the id of this Reaction to a copy of sid.

**void Reaction.setName (Reaction.t \*r, const char \*string)**

Sets the name of this Reaction to a copy of string (SName in L1).

**void Reaction.setKineticLaw (Reaction.t \*r, KineticLaw.t \*kl)**

Sets the KineticLaw of this Reaction to the given KineticLaw.

**void Reaction.setReversible (Reaction.t \*r, int value)**

Sets the reversible status of this Reaction to value (boolean).

**void Reaction.setFast (Reaction.t \*r, int value)**

Sets the fast status of this Reaction to value (boolean).

**ListOf.t \* Reaction.getListOfReactants (Reaction.t \*r)**

Returns the list of Reactants for this Reaction.

**ListOf.t \* Reaction.getListOfProducts (Reaction.t \*r)**

Returns the list of Products for this Reaction.

**ListOf.t \* Reaction.getListOfModifiers (Reaction.t \*r)**

Returns the list of Modifiers for this Reaction.

**void Reaction.addReactant (Reaction.t \*r, SpeciesReference.t \*sr)**

Adds the given reactant (SpeciesReference) to this Reaction.

**void Reaction\_addProduct (Reaction\_t \*r, SpeciesReference\_t \*sr)**

Adds the given product (SpeciesReference) to this Reaction.

**void Reaction\_addModifier (Reaction\_t \*r, ModifierSpeciesReference\_t \*msr)**

Adds the given modifier (ModifierSpeciesReference) to this Reaction.

**SpeciesReference\_t \* Reaction\_getReactant (const Reaction\_t \*r, unsigned int n)**

Returns the nth reactant (SpeciesReference) of this Reaction.

**SpeciesReference\_t \* Reaction\_getReactantById (const Reaction\_t \*r, const char \*sid)**

Returns the reactant (SpeciesReference) in this Reaction with the given id or NULL if no such reactant exists.

**SpeciesReference\_t \* Reaction\_getProduct (const Reaction\_t \*r, unsigned int n)**

Returns the nth product (SpeciesReference) of this Reaction.

**SpeciesReference\_t \* Reaction\_getProductById (const Reaction\_t \*r, const char \*sid)**

Returns the product (SpeciesReference) in this Reaction with the given id or NULL if no such product exists.

**ModifierSpeciesReference\_t \* Reaction\_getModifier (const Reaction\_t \*r, unsigned int n)**

Returns the nth modifier (ModifierSpeciesReference) of this Reaction.

**ModifierSpeciesReference\_t \* Reaction\_getModifierById (const Reaction\_t \*r, const char \*sid)**

Returns the modifier (ModifierSpeciesReference) in this Reaction with the given id or NULL if no such modifier exists.

**unsigned int Reaction\_getNumReactants (const Reaction\_t \*r)**

Returns the number of reactants (SpeciesReferences) in this Reaction.

**unsigned int Reaction\_getNumProducts (const Reaction\_t \*r)**

Returns the number of products (SpeciesReferences) in this Reaction.

**unsigned int Reaction\_getNumModifiers (const Reaction\_t \*r)**

Returns the number of modifiers (ModifierSpeciesReferences) in this Reaction.

**void Reaction\_unsetName (Reaction\_t \*r)**

Unsets the name of this Reaction. This is equivalent to: `safe_free(r->name); r->name = NULL;`

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

---

**void Reaction\_unsetKineticLaw (Reaction.t \*r)**

Unsets the KineticLaw of this Reaction. This is equivalent to: `r->kineticLaw = NULL;`

---

**void Reaction\_unsetFast (Reaction.t \*r)**

Unsets the fast status of this Reaction.

In L1, fast is optional with a default of false, which means it is effectively always set.

In L2, however, fast is optional with no default value, so it may or may not be set to a specific value.

---

**int ReactionIdCmp (const char \*sid, const Reaction.t \*r)**

The ReactionIdCmp function compares the string sid to r->id.

Returns an integer less than, equal to, or greater than zero if sid is found to be, respectively, less than, to match or be greater than r->id. Returns -1 if either sid or r->id is NULL.

## 2.16 Rule.h

```
const char * Rule_getFormula (const Rule_t *r)
```

Returns the formula for this Rule.

```
const ASTNode_t * Rule_getMath (const Rule_t *r)
```

Returns the math for this Rule.

```
int Rule_isSetFormula (const Rule_t *r)
```

Returns true (non-zero) if the formula (or equivalently the math) for this Rule has been set, false (0) otherwise.

```
int Rule_isSetMath (const Rule_t *r)
```

Returns true (non-zero) if the formula (or equivalently the math) for this Rule has been set, false (0) otherwise.

```
void Rule_setFormula (Rule_t *r, const char *string)
```

Sets the formula of this Rule to a copy of string.

```
void Rule_setMath (Rule_t *r, ASTNode_t *math)
```

Sets the math of this Rule to the given ASTNode.

The node **is not copied** and this Rule **takes ownership** of it; i.e. subsequent calls to this function or a call to Rule\_free() will free the ASTNode (and any child nodes).

```
void Rule_setFormulaFromMath (const Rule_t *r)
```

This function is no longer necessary. LibSBML now keeps formula strings and math ASTs synchronized automatically. The function is kept around for backward compatibility (and is used internally).

```
void Rule_setMathFromFormula (const Rule_t *r)
```

This function is no longer necessary. LibSBML now keeps formula strings and math ASTs synchronized automatically. The function is kept around for backward compatibility (and is used internally).

## 2.17 RuleType.h

**RuleType\_t RuleType\_forName (const char \*name)**

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

**const char \* RuleType\_toString (RuleType\_t rt)**

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

## 2.18 SBase.h

**void SBase\_init (SBase.t \*sb, SBMLTypeCode.t tc)**

SBase "objects" are abstract, i.e., they are not created. Rather, specific "subclasses" are created (e.g., Model) and their SBASE\_FIELDS are initialized with this function. The type of the specific "subclass" is indicated by the given SBMLTypeCode.

**void SBase\_clear (SBase.t \*sb)**

Clears (frees) only the SBASE\_FIELDS of sb.

**SBMLTypeCode.t SBase\_getTypeCode (const SBase.t \*sb)**

Returns the type of 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.

**const char \* SBase\_getMetaId (const SBase.t \*sb)**

Returns the metaid for this SBML object.

**const char \* SBase\_getNotes (const SBase.t \*sb)**

Returns the notes for this SBML object.

**const char \* SBase\_getAnnotation (const SBase.t \*sb)**

Returns the annotation 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 SBBase\_setNotes (SBBase.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 SBBase\_setAnnotation (SBBase.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 SBBase\_unsetMetaid (SBBase.t \*sb)**

Unsets the metaid for this SBML object. This is equivalent to: `safe_free(sb->metaid); s->metaid = NULL;`

---

**void SBBase\_unsetNotes (SBBase.t \*sb)**

Unsets the notes for this SBML object. This is equivalent to: `safe_free(sb->notes); s->notes = NULL;`

---

**void SBBase\_unsetAnnotation (SBBase.t \*sb)**

Unsets the annotation for this SBML object. This is equivalent to: `safe_free(sb->annotation); s->annotation = NULL;`

## 2.19 SBMLDocument.h

### **SBMLDocument\_t \* SBMLDocument\_create (void)**

Creates a new SBMLDocument and returns a pointer to it.  
The SBML level defaults to 2 and version defaults to 1.

### **SBMLDocument\_t \* SBMLDocument\_createWith (unsigned int level, unsigned int version)**

Creates a new SBMLDocument with the given level and version.

### **Model\_t \* SBMLDocument\_createModel (SBMLDocument\_t \*d)**

Creates a new Model inside this SBMLDocument and returns a pointer to it. This convenience function is functionally equivalent to:

```
d->model = Model_create();
```

### **Model\_t \* SBMLDocument\_createModelWith (SBMLDocument\_t \*d, const char \*sid)**

Creates a new Model inside this SBMLDocument and returns a pointer to it. The name field of this Model is set to a copy of sid.

### **void SBMLDocument\_free (SBMLDocument\_t \*d)**

Frees the given SBMLDocument.

### **unsigned int SBMLDocument\_getLevel (const SBMLDocument\_t \*d)**

Returns the level of this SBMLDocument.

### **unsigned int SBMLDocument\_getVersion (const SBMLDocument\_t \*d)**

Returns the version of this SBMLDocument.

### **ParseMessage\_t \* SBMLDocument\_getWarning (SBMLDocument\_t \*d, unsigned int n)**

Returns the nth warning encountered during the parse of this SBMLDocument or NULL if  $n \geq \text{getNumWarnings}() - 1$ .

### **ParseMessage\_t \* SBMLDocument\_getError (SBMLDocument\_t \*d, unsigned int n)**

Returns the nth error encountered during the parse of this SBMLDocument or NULL if  $n \geq \text{getNumErrors}() - 1$ .

### **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  $n \geq \text{getNumFatals}() - 1$ .

### **Model\_t \* SBMLDocument\_getModel (SBMLDocument\_t \*d)**

Returns the Model associated with this SBMLDocument.

**unsigned int SBMLDocument.getNumWarnings (const SBMLDocument.t \*d)**

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

**unsigned int SBMLDocument.getNumErrors (const SBMLDocument.t \*d)**

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

**unsigned int SBMLDocument.getNumFatals (const 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:

N Warning(s): line: (id) message

**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. `SBMLDocument.getNumErrors(d) == 0`, no output will be sent to stream. The format of the output is:

N Error(s): line: (id) message

**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. `SBMLDocument.getNumFatals(d) == 0`, no output will be sent to stream. The format of the output is:

N Fatal(s): line: (id) message

**void SBMLDocument.setLevel (SBMLDocument.t \*d, unsigned int level)**

Sets the level of this SBMLDocument to the given level number. Valid levels are currently 1 and 2.

**void SBMLDocument.setVersion (SBMLDocument.t \*d, unsigned int version)**

Sets the version of this SBMLDocument to the given version number. Valid versions are currently 1 and 2 for SBML L1 and 1 for SBML L2.

**void SBMLDocument.setModel (SBMLDocument.t \*d, Model.t \*m)**

Sets the Model of this SBMLDocument to the given Model. Any previously defined model is unset and freed.

**unsigned int SBMLDocument.checkConsistency (SBMLDocument.t \*d)**

Performs a set of semantic consistency checks on the document. Query the results by calling `getWarning()`, `getNumError()`, and `getNumFatal()`.

Returns the number of failed checks (errors) encountered.

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

```
@deprecated use SBMLDocument_checkConsistency() instead.
```

## 2.20 SBMLReader.h

### **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.

### **const char \* SBMLReader\_getSchemaFilenameL1v1 (const SBMLReader\_t \*sr)**

Returns the schema filename used by this SBMLReader to validate SBML Level 1 version 1 documents.

### **const char \* SBMLReader\_getSchemaFilenameL1v2 (const SBMLReader\_t \*sr)**

Returns the schema filename used by this SBMLReader to validate SBML Level 1 version 2 documents.

### **const char \* SBMLReader\_getSchemaFilenameL2v1 (const SBMLReader\_t \*sr)**

Returns the schema filename used by this SBMLReader to validate SBML Level 2 version 1 documents.

### **XMLSchemaValidation\_t SBMLReader\_getSchemaValidationLevel(const SBMLReader\_t \*sr)**

Returns the schema validation level used by this SBMLReader.

### **SBMLDocument\_t \* SBMLReader\_readSBML (SBMLReader\_t \*sr, const char \*filename)**

Reads an SBML document from the given file. If filename does not exist or is not an SBML file, a fatal error will be logged. Errors can be identified by their unique ids, e.g.:

```
SBMLReader sr; SBMLDocument_t d;
sr = SBMLReader_create(); SBMLReader_setSchemaValidationLevel(sr,
XML_SCHEMA_VALIDATION_BASIC); SBMLReader_setSchemaFilenameL1v1("sbml-l1v1.xsd");
SBMLReader_setSchemaFilenameL1v2("sbml-l1v2.xsd");
SBMLReader_setSchemaFilenameL2v1("sbml-l2v1.xsd");
d = SBMLReader_readSBML(reader, filename);
if (SBMLDocument_getNumFatals(d) > 0) ParseMessage_t pm
= SBMLDocument_getFatal(d, 0); if (ParseMessage_getId(pm) ==
SBML_READ_ERROR_FILE_NOT_FOUND) if (ParseMessage_getId(pm) ==
SBML_READ_ERROR_NOT_SBML) ;/code;
```

Returns a pointer to the SBMLDocument read.

### **SBMLDocument\_t \* SBMLReader\_readSBMLFromString (SBMLReader\_t \*sr, const char \*xml)**

Reads an SBML document from the given XML string.  
The XML string must be complete and legal XML document. Among other things, it must start with an XML processing instruction. For e.g.,:

```
!DOCTYPE sbml SYSTEM "http://www.sbml.org/SBML.dtd" [
?xml version='1.0' encoding='UTF-8' ?;
```

This method will log a fatal error if the XML string is not SBML. See the function documentation for SBMLReader\_readSBML(filename) for example error checking code.  
Returns a pointer to the SBMLDocument read.

---

**void SBMLReader\_setSchemaFilenameL1v1 (SBMLReader\_t \*sr, const char \*filename)**

Sets the schema filename used by this SBMLReader to validate SBML Level 1 version 1 documents.

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\_setSchemaFilenameL1v2 (SBMLReader\_t \*sr, const char \*filename)**

Sets the schema filename used by this SBMLReader to validate SBML Level 1 version 2 documents.

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\_setSchemaFilenameL2v1 (SBMLReader\_t \*sr, const char \*filename)**

Sets the schema filename used by this SBMLReader to validate SBML Level 2 version 1 documents.

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.

---

**SBMLDocument\_t \* readSBML (const char \*filename)**

Reads an SBML document from the given file. If filename does not exist or is not an SBML file, a fatal error will be logged. Errors can be identified by their unique ids, e.g.:

```
}; SBMLDocument_t d = SBMLReader_readSBML(reader, filename);
if (SBMLDocument_getNumFatals(d) > 0) ParseMessage_t pm
= SBMLDocument_getFatal(d, 0); if (ParseMessage_getId(pm) ==
SBML_READ_ERROR_FILE_NOT_FOUND) if (ParseMessage_getId(pm) ==
SBML_READ_ERROR_NOT_SBML) ;/code;
```

Returns a pointer to the SBMLDocument read.

---

**SBMLDocument\_t \* readSBMLFromString (const char \*xml)**

Reads an SBML document from the given XML string.

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'?>
```

This method will log a fatal error if the XML string is not SBML. See the function documentation for readSBML(filename) for example error checking code.

Returns a pointer to the SBMLDocument read.

---

## 2.21 SBMLWriter.h

### **SBMLWriter\_t \* SBMLWriter\_create (void)**

Creates a new SBMLWriter and returns a pointer to it.  
By default the character encoding is UTF-8 (CHARACTER\_ENCODING\_UTF\_8).

### **void SBMLWriter\_free (SBMLWriter\_t \*sw)**

Frees the given SBMLWriter.

### **void SBMLWriter\_setProgramName (SBMLWriter\_t \*sw, const char \*name)**

Sets the name of this program, i.e. the one about to write out the SBMLDocument. If the program name and version are set (setProgramVersion()), the following XML comment, intended for human consumption, will be written at the beginning of the document:  
|!- Created by |program name| version |program version| on yyyy-MM-dd HH:mm with libsbml version |libsbml version|. -|

### **void SBMLWriter\_setProgramVersion (SBMLWriter\_t \*sw, const char \*version)**

Sets the version of this program, i.e. the one about to write out the SBMLDocument. If the program version and name are set (setProgramName()), the following XML comment, intended for human consumption, will be written at the beginning of the document:  
|!- Created by |program name| version |program version| on yyyy-MM-dd HH:mm with libsbml version |libsbml version|. -|

### **int SBMLWriter\_writeSBML ( SBMLWriter\_t \*sw, const SBMLDocument\_t \*d, const char \*filename )**

Writes the given SBML document to filename.  
Returns non-zero on success and zero if the filename could not be opened for writing.)

### **char \* SBMLWriter\_writeSBMLToString (SBMLWriter\_t \*sw, const 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 the string on success and NULL if one of the underlying Xerces or Expat components fail (rare).

### **int writeSBML (const SBMLDocument\_t \*d, const char \*filename)**

Writes the given SBML document to filename. This convenience function is functionally equivalent to:

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

Returns non-zero on success and zero if the filename could not be opened for writing.)

### **char \* writeSBMLToString (const 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. This convenience function is functionally equivalent to:

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

Returns the string on success and NULL if one of the underlying Xerces or Expat components fail (rare).

## 2.22 SimpleSpeciesReference.h

```
const char * SimpleSpeciesReference_getSpecies (const SimpleSpeciesReference_t *ssr)
```

Returns the species for this SimpleSpeciesReference.

```
int SimpleSpeciesReference_isSetSpecies (const SimpleSpeciesReference_t *ssr)
```

Returns 1 if the species for this SimpleSpeciesReference has been set, 0 otherwise.

```
void SimpleSpeciesReference_setSpecies ( SimpleSpeciesReference_t *ssr, const char *sid )
```

Sets the species of this SimpleSpeciesReference to a copy of sid.

```
const char * SimpleSpeciesReference_getId (const SimpleSpeciesReference_t *ssr)
```

Returns the id for the species reference.

```
void SimpleSpeciesReference_setId (SimpleSpeciesReference_t *ssr, const char *sid)
```

Sets the id for the given SpeciesReference.

```
int SimpleSpeciesReference_isSetId (const SimpleSpeciesReference_t *ssr)
```

Returns 0 if the id is not set.

```
void SimpleSpeciesReference_unsetId (SimpleSpeciesReference_t *ssr)
```

Sets the id to the empty string.

```
int SimpleSpeciesReferenceCmp ( const char *sid, const SimpleSpeciesReference_t *ssr )
```

The SimpleSpeciesReferenceCmp function compares the string sid to ssr->species. Returns an integer less than, equal to, or greater than zero if sid is found to be, respectively, less than, to match or be greater than ssr->species. Returns -1 if either sid or ssr->species is NULL.

## 2.23 SpeciesConcentrationRule.h

**SpeciesConcentrationRule\_t \* SpeciesConcentrationRule\_create (void)**

Creates a new SpeciesConcentrationRule and returns a pointer to it.

**SpeciesConcentrationRule\_t \* SpeciesConcentrationRule\_createWith ( const char \*formula, RuleType\_t type, const char \*species )**

Creates a new SpeciesConcentrationRule with the given formula, type and species and returns a pointer to it. This convenience function is functionally equivalent to:

```
SpeciesConcentrationRule_t scr = SpeciesConcentrationRule_create();
Rule_setFormula((Rule_t ) scr, formula); AssignmentRule_setType((AssignmentRule_t
) scr, type); ...;
```

**void SpeciesConcentrationRule\_free (SpeciesConcentrationRule\_t \*scr)**

Frees the given SpeciesConcentrationRule.

**const char \* SpeciesConcentrationRule\_getSpecies (const SpeciesConcentrationRule\_t \*scr)**

Returns the species of this SpeciesConcentrationRule.

**int SpeciesConcentrationRule\_isSetSpecies (const SpeciesConcentrationRule\_t \*scr)**

Returns 1 if the species of this SpeciesConcentrationRule has been set, 0 otherwise.

**void SpeciesConcentrationRule\_setSpecies ( SpeciesConcentrationRule\_t \*scr, const char \*sname )**

Sets the species of this SpeciesConcentrationRule to a copy of sname.

## 2.24 SpeciesReference.h

**SpeciesReference\_t \* SpeciesReference\_create (void)**

Creates a new SpeciesReference and returns a pointer to it.

**SpeciesReference\_t \* SpeciesReference\_createWith ( const char \*species, double stoichiometry, int denominator )**

Creates a new SpeciesReference with the given species, stoichiometry and denominator and returns a pointer to it. This convenience function is functionally equivalent to:

```
SpeciesReference_t r = SpeciesReference_create();
SpeciesReference_setSpecies(r, species);           r->stoichiometry =
stoichiometry; ...;
```

**void SpeciesReference\_free (SpeciesReference\_t \*sr)**

Frees the given SpeciesReference.

**void SpeciesReference\_initDefaults (SpeciesReference\_t \*sr)**

Initializes the fields of this SpeciesReference to their defaults:  
- stoichiometry = 1 - denominator = 1

**const char \* SpeciesReference\_getSpecies (const SpeciesReference\_t \*sr)**

Returns the species of this SpeciesReference.

**double SpeciesReference\_getStoichiometry (const SpeciesReference\_t \*sr)**

Returns the stoichiometry of this SpeciesReference.

**const ASTNode\_t \* SpeciesReference\_getStoichiometryMath (const SpeciesReference\_t \*sr)**

Returns the stoichiometryMath of this SpeciesReference.

**int SpeciesReference\_getDenominator (const SpeciesReference\_t \*sr)**

Returns the denominator of this SpeciesReference.

**int SpeciesReference\_isSetSpecies (const SpeciesReference\_t \*sr)**

Returns 1 if the species of this SpeciesReference has been set, 0 otherwise.

**int SpeciesReference\_isSetStoichiometryMath (const SpeciesReference\_t \*sr)**

Returns 1 if the stoichiometryMath of this SpeciesReference has been set, 0 otherwise.

**void SpeciesReference\_setSpecies (SpeciesReference\_t \*sr, const char \*sname)**

Sets the species of this SpeciesReference to a copy of sname.

**void SpeciesReference\_setStoichiometry (SpeciesReference\_t \*sr, double value)**

Sets the stoichiometry of this SpeciesReference to value.

**void SpeciesReference\_setStoichiometryMath (SpeciesReference\_t \*sr, ASTNode\_t \*math)**

Sets the stoichiometryMath of this SpeciesReference to the given ASTNode.

The node **is not copied** and this SpeciesReference **takes ownership** of it; i.e. subsequent calls to this function or a call to SpeciesReference\_free() will free the ASTNode (and any child nodes).

**void SpeciesReference\_setDenominator (SpeciesReference\_t \*sr, int value)**

Sets the denominator of this SpeciesReference to value.

## 2.25 Species.h

### **Species\_t \* Species\_create (void)**

Creates a new Species and returns a pointer to it.

### **Species\_t \* Species\_createWith( const char \*sid, const char \*compartment, double initialAmount, const char \*substanceUnits, int boundaryCondition, int charge )**

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

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

### **void Species\_free (Species\_t \*s)**

Frees the given Species.

### **void Species\_initDefaults (Species\_t \*s)**

Initializes the fields of this Species to their defaults:

- boundaryCondition = 0 (false) - constant = 0 (false) (L2 only)

### **const char \* Species\_getId (const Species\_t \*s)**

Returns the id of this Species

### **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.

### **double Species\_getInitialConcentration (const Species\_t \*s)**

Returns the initialConcentration of this Species.

### **const char \* Species\_getSubstanceUnits (const Species\_t \*s)**

Returns the substanceUnits of this Species.

### **const char \* Species\_getSpatialSizeUnits (const Species\_t \*s)**

Returns the spatialSizeUnits of this Species.

**const char \* Species\_getUnits (const Species\_t \*s)**

Returns the units of this Species (L1 only).

**int Species\_getHasOnlySubstanceUnits (const Species\_t \*s)**

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

**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.

**int Species\_getConstant (const Species\_t \*s)**

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

**int Species\_isSetId (const Species\_t \*s)**

Returns 1 if the id 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 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\_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 (L1 only).

**int Species.isSetCharge (const Species.t \*s)**

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

**void Species.moveIdToName (Species.t \*s)**

Moves the id field of this Species to its name field (iff name is not already set). This method is used for converting from L2 to L1.

**void Species.moveNameToId (Species.t \*s)**

Moves the name field of this Species to its id field (iff id is not already set). This method is used for converting from L1 to L2.

**void Species.setId (Species.t \*s, const char \*sid)**

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

**void Species.setName (Species.t \*s, const char \*string)**

Sets the name of this Species to a copy of string (SName in L1).

**void Species.setCompartment (Species.t \*s, const char \*sid)**

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

**void Species.setInitialAmount (Species.t \*s, double value)**

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

**void Species.setInitialConcentration (Species.t \*s, double value)**

Sets the initialConcentration 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 of this Species to a copy of sid.

**void Species.setSpatialSizeUnits (Species.t \*s, const char \*sid)**

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

**void Species.setUnits (Species.t \*s, const char \*sname)**

Sets the units 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 of this Species to value (boolean).

**void Species\_setCharge (Species.t \*s, int value)**

Sets the charge 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).

**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\_unsetInitialConcentration (Species.t \*s)**

Unsets the initialConcentration of this Species.

**void Species\_unsetSubstanceUnits (Species.t \*s)**

Unsets the substanceUnits of this Species. This is equivalent to: safe\_free(s->substanceUnits); s->substanceUnits = NULL;

**void Species\_unsetSpatialSizeUnits (Species.t \*s)**

Unsets the spatialSizeUnits of this Species. This is equivalent to: safe\_free(s->spatialSizeUnits); s->spatialSizeUnits = NULL;

**void Species\_unsetUnits (Species.t \*s)**

Unsets the units of this Species (L1 only).

**void Species\_unsetCharge (Species.t \*s)**

Unsets the charge of this Species.

**int SpeciesIdCmp (const char \*sid, const Species.t \*s)**

The SpeciesIdCmp function compares the string sid to species->id.

Returns an integer less than, equal to, or greater than zero if sid is found to be, respectively, less than, to match, or be greater than species->id. Returns -1 if either sid or species->id is NULL.

## 2.26 UnitDefinition.h

### **UnitDefinition\_t \* UnitDefinition\_create (void)**

Creates a new UnitDefinition and returns a pointer to it.

### **UnitDefinition\_t \* UnitDefinition\_createWith (const char \*sid)**

Creates a new UnitDefinition with the given id and returns a pointer to it. This convenience function is functionally equivalent to:

```
UnitDefinition_setId(UnitDefinition_create(), sid);
```

### **UnitDefinition\_t \* UnitDefinition\_createWithName (const char \*string)**

Creates a new UnitDefinition with the given name and returns a pointer to it. This convenience function is functionally equivalent to:

```
UnitDefinition_setName(UnitDefinition_create(), string);
```

### **void UnitDefinition\_free (UnitDefinition\_t \*ud)**

Frees the given UnitDefinition.

### **const char \* UnitDefinition\_getId (const UnitDefinition\_t \*ud)**

Returns the id of this UnitDefinition.

### **const char \* UnitDefinition\_getName (const UnitDefinition\_t \*ud)**

Returns the name of this UnitDefinition.

### **int UnitDefinition\_isSetId (const UnitDefinition\_t \*ud)**

Returns non-zero if the id of this UnitDefinition has been set, zero otherwise.

### **int UnitDefinition\_isSetName (const UnitDefinition\_t \*ud)**

Returns non-zero if the name of this UnitDefinition has been set, zero otherwise.

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

### **int UnitDefinition\_isVariantOfArea (const UnitDefinition\_t \*ud)**

Returns non-zero if this UnitDefinition is a variant of the builtin type area, i.e. square metres with only arbitrary variations in scale, multiplier, or offset values, zero otherwise.

### **int UnitDefinition\_isVariantOfLength (const UnitDefinition\_t \*ud)**

Returns non-zero if this UnitDefinition is a variant of the builtin type length, i.e. metres with only arbitrary variations in scale, multiplier, or offset values, zero otherwise.

### **int UnitDefinition\_isVariantOfSubstance (const UnitDefinition\_t \*ud)**

Returns non-zero if this UnitDefinition is a variant of the builtin type substance, i.e. moles or items with only arbitrary variations in scale, multiplier, or offset values, zero otherwise.

**int UnitDefinition\_isVariantOfTime (const UnitDefinition\_t \*ud)**

Returns non-zero if this UnitDefinition is a variant of the builtin type time, i.e. seconds with only arbitrary variations in scale, multiplier, or offset values, zero otherwise.

**int UnitDefinition\_isVariantOfVolume (const UnitDefinition\_t \*ud)**

Returns non-zero if this UnitDefinition is a variant of the builtin type volume, i.e. litre or cubic metre with only arbitrary variations in scale, multiplier, or offset values, zero otherwise.

**void UnitDefinition\_moveIdToName (UnitDefinition\_t \*ud)**

Moves the id field of this UnitDefinition to its name field (iff name is not already set). This method is used for converting from L2 to L1.

**void UnitDefinition\_moveNameToId (UnitDefinition\_t \*ud)**

Moves the name field of this UnitDefinition to its id field (iff id is not already set). This method is used for converting from L1 to L2.

**void UnitDefinition\_setId (UnitDefinition\_t \*ud, const char \*sid)**

Sets the id of this UnitDefinition to a copy of sid.

**void UnitDefinition\_setName (UnitDefinition\_t \*ud, const char \*string)**

Sets the name of this UnitDefinition to a copy of string (SName in L1).

**void UnitDefinition\_unsetName (UnitDefinition\_t \*ud)**

Unsets the name of this UnitDefinition. This is equivalent to: `safe_free(ud->name); ud->name = NULL;`  
In SBML L1, a UnitDefinition name is required and therefore **should always be set**. In L2, name is optional and as such may or may not be set.

**void UnitDefinition\_addUnit (UnitDefinition\_t \*ud, Unit\_t \*u)**

Adds the given Unit to this UnitDefinition.

**ListOf\_t \* UnitDefinition\_getListOfUnits (UnitDefinition\_t \*ud)**

Returns the list of Units for 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)**

Returns the number of Units in this UnitDefinition.

**int UnitDefinitionIdCmp (const char \*sid, const UnitDefinition\_t \*ud)**

The UnitDefinitionIdCmp function compares the string sid to ud->id.

Returns an integer less than, equal to, or greater than zero if sid is found to be, respectively, less than, to match or be greater than ud->id. Returns -1 if either sid or ud->id is NULL.

## 2.27 UnitKind.h

### **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.

### **int UnitKind\_isValidUnitKindString (const char \*string)**

Returns nonzero if string is the name of a valid unitKind.

## 2.28 Unit.h

### **Unit\_t \* Unit\_create (void)**

Creates a new Unit and returns a pointer to it.

### **Unit\_t \* Unit\_createWith (UnitKind.t kind, int exponent, int scale)**

Creates a new Unit with the given kind, exponent and scale and returns a pointer to it. This convenience function is functionally equivalent to:

```
Unit_t u = Unit_create(); Unit_setKind(kind); Unit_setExponent(exponent);  
...;
```

### **void Unit\_free (Unit\_t \*u)**

Frees the given Unit.

### **void Unit\_initDefaults (Unit\_t \*u)**

Initializes the fields of this Unit to their defaults:

- exponent = 1 - scale = 0 - multiplier = 1.0 - offset = 0.0

### **UnitKind.t Unit\_getKind (const Unit\_t \*u)**

Returns the kind of this Unit.

### **int Unit\_getExponent (const Unit\_t \*u)**

Returns the exponent of this Unit.

### **int Unit\_getScale (const Unit\_t \*u)**

Returns the scale of this Unit.

### **double Unit\_getMultiplier (const Unit\_t \*u)**

Returns the multiplier of this Unit.

### **double Unit\_getOffset (const Unit\_t \*u)**

Returns the offset of this Unit.

### **int Unit\_isAmpere (const Unit\_t \*u)**

Returns non-zero if the kind of this Unit is 'ampere', zero otherwise.

### **int Unit\_isBecquerel (const Unit\_t \*u)**

Returns non-zero if the kind of this Unit is 'becquerel', zero otherwise.

### **int Unit\_isCandela (const Unit\_t \*u)**

Returns non-zero if the kind of this Unit is 'candela', zero otherwise.

**int Unit.isCelsius (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'Celsius', zero otherwise.

**int Unit.isCoulomb (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'coulomb', zero otherwise.

**int Unit.isDimensionless (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'dimensionless', zero otherwise.

**int Unit.isFarad (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'farad', zero otherwise.

**int Unit.isGram (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'gram', zero otherwise.

**int Unit.isGray (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'gray', zero otherwise.

**int Unit.isHenry (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'henry', zero otherwise.

**int Unit.isHertz (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'hertz', zero otherwise.

**int Unit.isItem (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'item', zero otherwise.

**int Unit.isJoule (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'joule', zero otherwise.

**int Unit.isKatal (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'katal', zero otherwise.

**int Unit.isKelvin (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'kelvin', zero otherwise.

**int Unit.isKilogram (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'kilogram', zero otherwise.

**int Unit.isLitre (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'litre' or 'liter', zero otherwise.

**int Unit.isLumen (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'lumen', zero otherwise.

**int Unit.isLux (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'lux', zero otherwise.

**int Unit.isMetre (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'metre' or 'meter', zero otherwise.

**int Unit.isMole (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'mole', zero otherwise.

**int Unit.isNewton (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'newton', zero otherwise.

**int Unit.isOhm (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'ohm', zero otherwise.

**int Unit.isPascal (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'pascal', zero otherwise.

**int Unit.isRadian (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'radian', zero otherwise.

**int Unit.isSecond (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'second', zero otherwise.

**int Unit.isSiemens (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'siemens', zero otherwise.

**int Unit.isSievert (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'sievert', zero otherwise.

**int Unit.isSteradian (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'steradian', zero otherwise.

**int Unit.isTesla (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'tesla', zero otherwise.

**int Unit.isVolt (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'volt', zero otherwise.

**int Unit.isWatt (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'watt', zero otherwise.

**int Unit.isWeber (const Unit.t \*u)**

Returns non-zero if the kind of this Unit is 'weber', zero otherwise.

**int Unit.isSetKind (const Unit.t \*u)**

Returns non-zero if the kind of this Unit has been set, zero otherwise.

**void Unit.setKind (Unit.t \*u, UnitKind.t kind)**

Sets the kind of this Unit to the given UnitKind.

**void Unit.setExponent (Unit.t \*u, int value)**

Sets the exponent of this Unit to the given value.

**void Unit.setScale (Unit.t \*u, int value)**

Sets the scale of this Unit to the given value.

**void Unit.setMultiplier (Unit.t \*u, double value)**

Sets the multiplier of this Unit to the given value.

**void Unit.setOffset (Unit.t \*u, double value)**

Sets the offset of this Unit to the given value.

**int Unit.isBuiltin (const char \*name)**

Returns non-zero if name is one of the five SBML builtin Unit names ('substance', 'volume', 'area', 'length' or 'time'), zero otherwise.

---

## References

- Bornstein, B. J. (2004). LibSBML 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.