

# MathSBML.m

## The Mathematica SBML Reader

Version 2.1.5( $\beta$  5) (September 17, 2003)

*Author:* Bruce E. Shapiro

The Systems Biology Workbench Development Group  
ERATO Kitano Symbiotic Systems Project  
Control and Dynamical Systems, MC 107-81  
California Institute of Technology  
<http://www.sbml.org>

## Table of Contents

1. Introduction
  2. Using MathSBML
    - 2.1 Using SBMLRead
    - 2.2 Example
    - 2.3 Plotting with SBMLPlot
    - 2.4 Using MathSBML as an SBML file interpreter
  3. Function Reference
  4. Predefined Rate Law Function Reference
  5. Model Builder Function Reference
- References

# 1. Introduction

The [MathSBML](#) package provides facilities for reading models expressed in SBML and converting the models to systems of ordinary differential equations for simulation and plotting in *Mathematica*.

The current release of [MathSBML](#) is version 2.1.5 and can read SBML Level 1 (Version 1 and Version 2) and SBML Level 2. Future releases will be able to read higher levels and versions.

For details on SBML and *Mathematica* the user should consult the references listed at the end of this document.

## Installation Instructions

[MathSBML](#) requires *Mathematica* 4.2 or above. It will not work properly with earlier versions of *Mathematica*. [MathSBML](#) is also fully compatible with *Mathematica* version 5.0, and additional functionality, notably the solution of differential-algebraic equations (DAEs) becomes available with with version 5.0.

The only file required is [MathSBML.m](#). Do not edit or modify this file, as it is machine-generated. The file can be copied anywhere on your hard drive. To use the file you need to include the line

```
<< path/MathSBML.m
```

where path is the full path name, relative to your home directory, of the directory in which you have placed [MathSBML](#). An alternative syntax is

```
Get["MathSBML.m", Path → path]
```

*Mathematica* knows about a set of default directories that it will search when looking for a file. If you put your files in one of these directories, you do not need to specify the path; e.g., you can omit the "path" from the above [Get](#) (<<) command:

```
<< MathSBML.m
```

A message such as the following will be displayed on the next line to indicate that [MathSBML](#) has completed loading correctly.

```
MathSBML 2.1.3 (28 Aug 2003) loaded 28-August-2003 10:41:01.254403
```

If you get any other message, then an error has occurred during loading.

You can determine the current path by entering `$Path` at the *Mathematica* prompt. Additional information on the *Mathematica* path is given in the *Mathematica* help files.

The file `MathSBML.nb` contains the source code for `MathSBML.m`. This file is not required unless you want to change the functionality of `MathSBML`. If you open this file in *Mathematica* you may get a warning message to the effect that the file has been edited outside of *Mathematica*. This is because of the manner in which version information is added to the file by the software repository and does not affect the coding contents of the file. This version information is stored in text boxes within the file.

## Functions Provided by This Package

`MathSBML` contains three functions that can be invoked directly: `SBMLRead`, `SBMLNDSolve`, and `SBMLPlot`.

`SBMLRead` is the primary function provided by this package - it reads a model encoded in SBML into *Mathematica*, converting the model into a system of differential and algebraic equations, if such information is contained in the model. It also contains options to generate a formatted listing of the model, as well as call `SBMLNDSolve` and `SBMLPlot` automatically upon reading the model. It is discussed in greater detail in the following section.

`SBMLNDSolve` solves the system of differential-algebraic equations produced by `SBMLRead` using `NDSolve`. The user can optionally pass the output of `SBMLRead` directly to `NDSolve` without using `SBMLNDSolve`. (The solution of DAE's requires *Mathematica* Version  $\geq 5.0$ ; the solution of ODEs can be performed with earlier *Mathematica* versions) Examples of how to perform both of these operations are given below.

`SBMLPlot` can be used to generate plots of the resulting solutions. Plots can also be generated directly with `Plot`.

`SBMLWrite` will read an SBML file and translate it into another format. At the present time, the only output formats supported are "Fortran", "HTML", and "XPP". It is anticipated that additional formats will be added in future releases.

`SBMLCopy` will read one SBML file and copy it to a second file, processing the file to "pretty-print" the XML with indentations as determined by *Mathematica*'s `ExportString` function. The purpose of this function is to make a machine-generated XML file more readable.

The `MathSBML Model Builder` consists of a suite of functions that can be used to build SBML Models manually. This suite is not complete, and additional functionality will be added in future releases.

These functions are illustrated below in detail.

## 2. Using MathSBML

### 2.1. Using SBMLRead

The basic function used to read SBML files is `SBMLRead`. This section describes what `SBMLRead` returns and what its basic options are. The following sections provide examples on using `SBMLRead`.

#### 2.1.1. SBMLRead Return Values

`SBMLRead` reads an SBML file and translates it into a *Mathematica* data structure consisting of *Mathematica* differential equations, initial conditions, a list of variables, and replacement rules for constant parameters. `SBMLRead` can also be used to generate an interpretive listing of the SBML file. The return value of `SBMLRead` is a *Mathematica* rule list of the form

```
{SBMLODES → list of differential equations,
 SBMLParameters → list of parameter rules,
 SBMLIC → list of initial conditions,
 SBMLSpecies → list of variables,
 SBMLAlgebraicRules → list of algebraic rules,
 SBMLUnitDefinitions → list of unit definitions,
 SBMLUnitAssociations → list of unit associations,
 SBMLReactions → list of reactions,
 SBMLFunctions → list of pure function definitions,
 SBMLNameIDAssociations → list id / name associations,
 SBMLEvents → list of events,
 SBMLModelName → name of the model,
 SBMLNumericalSolution → numericalSolution
}
```

By default, all of this information is returned. However, the user is allowed to inhibit return of any portions of this information by using the `return` option to `SBMLRead`. The items highlighted in red (`SBMLFunctions`, `SBMLNameIDAssociations`, and `SBMLEvents`) are only provided for SBML Level 2.0 files and above.

Here *list of variables* has the form

```
{var1[t], var2[t], ... }
```

where each of `var1`, `var2`, etc., are variables that are governed by rate laws in the SBML models; i.e., any species, parameter, or compartment that is described by a rate law in a rule, and species that are either products or reactants in reactions, and are not boundary conditions. Each species in the SBML model is translated into a time-dependent function in the *Mathematica* model. In level 2, the "id" field is used to identify the variable. In level 1, the "name" field is used.

The global variable (i.e., in the *Mathematica* context `Global``) `t` is reserved for time. It is anticipated that in future releases of MathSBML the name and context of the time variable can be reassigned at the user's direction. In contrast to the SBML specification, `SBMLRead` does not make use of the `csymbol` for time; this will also be corrected in future versions.

The *list of differential equations* has the form

```
{var1'[t] == expression1, var2'[t] == expression2, ... }
```

where *expression1*, *expression2*, ... are *Mathematica* expressions formed by applying all of the rules and reactions that affect that corresponding species. In level 2, the "id" field is used to identify all variables and constants in the expression. In level 1, the "name" field is used.

The *list of parameter rules* has the form

```
{par1 → expression1, par2 → expression2, ... }
```

where *par1*, *par2*, etc., are constant parameters or variable parameters described by *scalar* type *ParameterRules*; compartments with volumes that are either fixed or described by *scalar* type *CompartmentVolumeRules*; species that are described by *scalar* type *SpeciesConcentrationRules*; or species that are boundary conditions. The expressions are either constants (for fixed values) or algebraic expressions that give the value of the parameter. It is possible for the same parameter to be listed more than once in this list if the same local parameter name is used in multiple reactions. The parameters are listed in the same order in which they are defined in the SBML model. In level 2, the "id" field is used to identify the parameter. In level 1, the "name" field is used.

The *list of initial conditions* has the form

```
{var1[0] == value1, var2[0] == value2, ... }
```

where each of *var1*, *var2*, etc., are the same as defined in *list of variables* above. Their lengths of the lists of initial conditions, variables, and differential equations are all the same. As usual, In level 2, the "id" field is used to identify the variable. In level 1, the "name" field is used.

The *list of algebraic rules* has the form

```
{expression1==0, expression2==0, ... }
```

where *expression1*, *expression2*, ... are *Mathematica* expressions formed from the corresponding *algebraicRule* in the SBML file. In level 2, the "id" field is used to identify all variables. In level 1, the "name" field is used.

The *list of unit definitions* has the form

```
{name1→def1, name2→def2, ...}
```

where *name1*, *name2*, ... are *Mathematica* variables corresponding to the *name* field of the corresponding *unitDefinition*; and *def1*, *def2*, ... are *Mathematica* expressions defining the units in terms of the basic pre-defined units.

The *list of unit associations* has the form

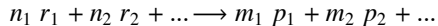
```
{name1→units1, name2→units2, ...}
```

where *name1*, *name2*, ... are *Mathematica* variables (species, parameters, or compartments) and *units1*, *units2*, ... are the units, either predefined or user-defined, corresponding to those variables.

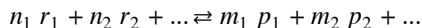
The *list of reactions* has the form

```
{reaction1, reaction2, ...}
```

where each of the reactions has the form



where  $r_1, r_2, \dots$  and  $p_1, p_2, \dots$  are the reactants and products in the reaction and  $n_1, \dots$  and  $m_1, \dots$ , are the corresponding composite stoichiometry for each reactant or product. In level 1, the composite stoichiometry is given by  $\sum (s_k / d_k)$  where  $s_k$  and  $d_k$  are the values of the `stoichiometry` and `denominator` field of the  $k$ th occurrence of the corresponding reactant or product in the reaction. In level 2, the stoichiometry is given by whatever mathematical formula is supplied. If the reaction is specified to be `reversible` in the model, then the double arrow form



is used instead. Note: it is anticipated that the particular arrow used in the reactions ( $\longrightarrow$ , `LongRightArrow`) will be replaced in subsequent versions of MathSBML with ( $\rightarrow$ , `ShortRightArrow`) because it provides more suitable infix-operator precedence. Both arrows are different from the `Rule` arrow ( $\rightarrow$ , keystrokes `->`).

The *list of pure function definitions* has the form

```
{id1→Function[...], id2→Function[...], ...}
```

where each `id1`, `id2`, ... are the function "id" fields specified in the SBML `Function[...]`, is a standard *Mathematica* technique for defining a function. A pure function definition can be applied to an argument in *Mathematica* as `Function[...][argument]`. To illustrate what this means, suppose that the SBML model has the following function definition to implement the function  $\text{foo}(x) = 1 / (1 + x^2)$ :

```
<functionDefinition id="foo">
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <lambda>
      <bvar><ci>x</ci></bvar>
      <apply>
        <times/>
        <cn>1</cn>
        <apply>
          <power/>
          <apply>
            <plus/>
            <apply>
              <power/>
              <ci>x</ci>
              <cn>2</cn>
            </apply>
            <cn>1</cn>
          </apply>
          <cn>-1</cn>
        </apply>
      </lambda>
    </math>
  </functionDefinition>
```

This function will be expressed by MathSBML as

```
Example`foo→Function[{x},1/(x^2 + 1)]
```

which is a direct *Mathematica* implementation of a lambda-form. To evaluate this function, say to find  $foo(y+z)$ , one would write

```
Function[{x},1/(x^2+1)][y+z]
```

to which *Mathematica* would return the expression

```
(1+(y+z)^2)^(-1).
```

Alternatively, one could define a function

```
foo[u_]:=Function[{x},1/(x^2 + 1)][u]
```

Typing `foo[y+z]` will evaluate the function at the point  $y+z$ , returning the same expression as before. More detail on pure functions is given *The Mathematica Book* [5th Edition, S. Wolfram, Cambridge University Press/Wolfram Media, 2003].

The *list of name/id associations* is a list of the form  $\{id1 \rightarrow name1, id2 \rightarrow name2, \dots\}$  describing the "id" and "name" fields for every object in the model for which a "name" is given.

The *list of events* describes the events in the SBML file in the following form,

```
{eventid1 → {"trigger" → trigger expression,
            "delay" → delay expression,
            "events" → {id1 → event expression, id2 → event expression, ...},
            eventid2 → {...}, ...}
```

where *eventid1*, *eventid2*, ... are the "id" fields of the named events; the *trigger expression* is the event trigger formatted as a *Mathematica* logical expression whose value will be True if and only if the trigger condition is met; the *delay expression* is a *Mathematica* algebraic expression giving the required time delay; and each *event expression* is a *Mathematica* algebraic expression that evaluates to the desired value as specified in the event.

The *numerical solution* is a list of interpolating functions as would typically be returned by NDSolve. Normally no numerical solution is generated by SBMLRead. To obtain a numerical solution the user would typically first call SBMLRead and then pass the result to SBMLNDSolve. For example,

```
m = SBMLRead["myfile.xml", ... ];
n = SBMLNDSolve[m, 200];
```

would read the file `myfile.xml` into the model `m` and then solve the model for 200 time units, returning a list of Interpolating Functions,

```
{{foo`var1 → InterpolatingFunction[{0, 200}], <>}[t],
  foo`var2 → InterpolatingFunction[{0, 200}], <>}[t], ...}
```

In this situation, the model `m` would contain  $\{\dots, SBMLNumericalSolution \rightarrow \{\}\}$ . Alternatively, the user can do this in a single step as

```
m = SBMLRead["myfile.xml",
            return → {..., SBMLNumericalSolution → 200, ...}, ... ];
```

in which case the value of `SBMLNumericalSolution` would be the same list of Interpolating Functions previously quoted. Using this option does not preclude the user for sending the model `m` to `SBMLNDSolve` again; however, the new solution will not overwrite the old solution in the model. Furthermore, the user can automatically send the output of `SBMLNDSolve`, if it is generated by `SBMLRead`, to `SBMLPlot`, but setting the option `PlotOptions` to anything besides a null list.

## 2.1.2.SBMLRead Options

The following table summarizes the options that are available to `SBMLRead`.

<u>Option</u>	<u>Value</u>	<u>Description</u>
<code>context</code>	<code>Automatic</code>	Context to use for model. <code>Context</code> → <code>Automatic</code> means generate context from the name of the model in the XML file. <code>Context</code> → <code>None</code> means place all variables in the Global` context. Note that if variables are placed in the Global` context then a clash with global or default <i>Mathematica</i> variables may occur, for example, the variables <i>E</i> and <i>Pi</i> , if they occur in the model, will be treated as the constants 2.718 ... and 3.14 ..., etc.
<code>defaultIC</code>	<code>Indeterminate</code>	Value to be used for initial conditions of variables when that are not assigned initial conditions in the model.
<code>defaultParameterValue</code>	<code>Indeterminate</code>	Value to be used for parameters that are not assigned values in the model.
<code>evaluateParameters</code>	<code>True</code>	Immediately evaluate all parameters before returning the model. Otherwise all parameters are returned in their symbolic form.
<code>return</code>	(see below)	Controls what gets returned by <code>SBMLRead</code>
<code>PlotOptions</code>	<code>{ }</code>	If <code>PlotOptions</code> → <code>True</code> or <code>PlotOptions</code> → <i>any non - null list</i> , then the numerical solution will automatically be plotted. If the numerical solution is not requested (See <code>return</code> , below) then this will generate an error. <code>PlotOptions</code> may simply be <code>True</code> or any list of options to be passed to <code>SBMLPlot</code> and/or <code>Plot</code> .
<code>verbose</code>	<code>False</code>	Produce an interpretive listing; described in greater detail below.

The option `return` controls what is returned by `SBMLRead`. It is a list of option/value pairs; the default value is:

```
return→{
  SBMLODES→True,
  SBMLIC→True,
  SBMLParameters→True,
  SBMLSpecies→True,
  SBMLAlgebraicRules→True,
  SBMLUnitDefinitions→True,
```

```

SBMLUnitAssociations→True,
SBMLReactions→True,
SBMLNameIDAssociations→True,
SBMLEvents→True,
SBMLFunctions→True,
SBMLModelName→True,
SBMLNumericalSolution→0}

```

If any of the boolean options are set to `False`, then the corresponding field of the same name is not returned by `.` If `SBMLNumericalSolution` is any positive number then the model will automatically be passed to `SBMLNDSolve`.

In addition, `SBMLRead` may be used to produce a detailed interpretive listing of an SBML file. The additional options required for this are discussed below.

### 2.1.3. Variable naming conventions

`SBMLRead` will attempt to match the variable names (objects of type `SName`) in the *Mathematica* version of the model as closely as possible to the name given in the SBML version of the model.

**SBML characters not allowed by *Mathematica*:** The character `"_"` is reserved for pattern matching in *Mathematica*, and will normally be replaced with the character `"_"`. A different underline character may be used by setting the value of the option `underscore` to `SBMLRead`. The default value is `underscore→"_"`. Hence an identifier `A_B` in the SBML becomes `A_B` in the *Mathematica* model. To force all underscores to be replaced by a happy face character (`[ESC]:[ESC]`), for example, use `SBMLRead[... , underscore→"☺"]`. The under-bracket character can be entered at the keyboard with either of the following keystroke sequences: the backslash character (`"\"`) immediately followed by the string `"[UnderBracket]"`, or the key sequence `[ESC] u [ESC]` (with no spaces between the characters). Note that when you hit the escape key, `[ESC]`, a vertical array of three short lines (`[ESC]`) will be displayed by *Mathematica*.

***Mathematica* representation of invalid characters in a model.** If a variable contains a character that is not in the standard `SName` set `{'a'..'z', 'A'..'Z', '0'..'9', ',', '.'}`, to ensure that the variable name is a legal *Mathematica* variable, the character will be replaced with the string `#n#` where `n` is the decimal unicode representation of the unusual character and `#` is the *Mathematica* (unicode 63268) character. (Aside note: The NumberSign character is not equivalent to the `"#"` on the keyboard, which has a unicode representation of 35. It may also be entered as `[ESC]#[ESC]` where `#` is the keyboard number sign, normally found on the number 3 key of US-standard keyboards, or as the backslash character (`"\"`) followed by the string `[NumberSign]`). Thus if there is an (invalid) SBML identifier `"hello[world]"` in the SBML model it will be represented as `"Hello#91#World#93#" in the Mathematica model. While this does not conform to the SBML standard, it provides consistent error recovery in the event of incorrectly formatted variables.`

**Scope of Variables.** SBML model variables are defined in a local context; the name of the context is determined by the name of the model. Thus if the SBML model `foo` contains species `A` and `B`, and global parameters `f` and `k`, they will be represented as `foo`A`, `foo`B`, `foo`f`, and `foo`k`, respectively. Local parameters `k` and `kf` defined in reactions `R1` and `R2` will become `foo`R1`k`, `foo`R1`kf`, `foo`R2`k`, and `foo`R2`kf`, respectively.

To override the use of the model name as the context, the `SBMLRead` option `context` may be used.

A short background on contexts is given in the following paragraphs; for additional information the user should refer to *The Mathematica Book* section 2.6.8.

**Background on Contexts.** *Mathematica* represents the scope of a symbol by its context. Different variables of the same name may exist independently of one another if they are defined in different contexts. The symbols `fred`foo` and

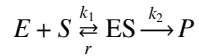
`barney`foo` represent to different variables, the first on context `fred`` and the second one in context `barney``. The "grave" or "backquote" character (unicode decimal representation 96, normally found just to the left of the number "1" key on most US keyboards) is used to separate the context name from the variable name.

Variables defined in the current context (which may be determined by entering `$Context`; the default value of `$Context` is `"Global`"`) do not have to be qualified by their context, although *Mathematica* is perfectly happy if you always use the full name `context`variable`. Normally all variables defined during an interactive *Mathematica* session are added to the `Global`` context. Thus if the variable `foo` is referenced during an interactive session, it is interpreted as `Global`foo`.

*Mathematica* also provides a global variable `$ContextPath` (its default value is `{Global`, System`}`) that gives a list of contexts to search, after `$Context`, in trying to find the definition of a symbol. After `MathSBML` is loaded (with the `<<MathSBML` command) then value of `$ContextPath` will be `{MathSBML`, Global`, System`}`. Thus if you refer to `foo`, *Mathematica* will first look for `MathSBML`foo`, then if it does not find it, will look for `Global`foo`, and then if it does not find it, will look for `System`foo`. If it still is not found, a new variable `$Context`foo` will be defined.

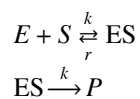
## 2.2. Example: Inport a model, run a simulation, and plot the results

Suppose we have a simple enzyme kinetic system



where E, S, and P represent enzyme, substrate, and product, respectively, and ES is an intermediate complex that is formed during the reaction, and the rate constants are  $k_1 = 3$ ,  $r = 6$ ,  $k_2 = 9$ . Suppose also that the amount of S and P are assumed to be constant, while the amount of E and ES are allowed to vary. Initial conditions are taken as  $E(0)=1$ ,  $ES(0)=0.01$ ,  $S(0)=1$ ,  $P(0)=0.01$ . These assumptions are taken to illustrate the properties of `MathSBML` and are not necessarily meant to represent a "real" biological system.

The SBML model shown in the box on the following page represents this model as a pair of reactions



where the k's in the two different reactions are assumed to be "local" parameters, i.e., the k in the first reaction is different from the k in the second reaction. Again, this assumption is made to illustrate `MathSBML` and would typically not be used by human-generated models (although it might occur in machine generated models).

Suppose that this model resides in the file "desktop/enzyme.xml". To read the file into a *Mathematica* model stored in the variable r, we would use

```
r = SBMLRead["desktop/enzyme.xml"]

{SBMLODES -> {EnzymeKinetics`E'[t] == -3. EnzymeKinetics`E[t] + 15. EnzymeKinetics`ES[t],
  EnzymeKinetics`ES'[t] == 3. EnzymeKinetics`E[t] - 15. EnzymeKinetics`ES[t]},
 SBMLIC -> {EnzymeKinetics`E[0] == 1., EnzymeKinetics`ES[0] == 0.01}, SBMLParameters ->
  {EnzymeKinetics`Cell -> 1., EnzymeKinetics`S -> 1., EnzymeKinetics`P -> 0.01},
 SBMLSpecies -> {EnzymeKinetics`E[t], EnzymeKinetics`ES[t]}, SBMLAlgebraicRules -> {},
 SBMLUnitDefinitions -> {substance -> mole, volume -> liter, time -> second},
 SBMLUnitAssociations -> {EnzymeKinetics`Cell -> volume,
  EnzymeKinetics`S -> substance, EnzymeKinetics`E -> substance,
  EnzymeKinetics`ES -> substance, EnzymeKinetics`P -> substance},
 SBMLReactions -> {EnzymeKinetics`E + EnzymeKinetics`S -> EnzymeKinetics`ES,
  EnzymeKinetics`ES -> EnzymeKinetics`E + EnzymeKinetics`P}}
```

Since there are two variable species (E, ES) and two fixed species (S, P), there are only two ODEs. The first three fields are directly compatible with `NDSolve`:

```
NDSolve[Join[SBMLODES /. r, SBMLIC /. r], SBMLSpecies /. r, {t, 0, 5}]
```

which returns:

```
{{EnzymeKinetics`E[t] -> InterpolatingFunction[{{0., 5.}}, <>][t],
  EnzymeKinetics`ES[t] -> InterpolatingFunction[{{0., 5.}}, <>][t]}}
```

```

<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level1" level="1" version="2">
  <model name="EnzymeKinetics">
    <listOfCompartments>
      <compartment name="Cell" volume="1"/>
    </listOfCompartments>
    <listOfSpecies>
      <species name="S" compartment="Cell" initialAmount="1"
        boundaryCondition="true"/>
      <species name="E" compartment="Cell" initialAmount="1"
        boundaryCondition="false"/>
      <species name="ES" compartment="Cell" initialAmount=".01"
        boundaryCondition="false"/>
      <species name="P" compartment="Cell" initialAmount="0.01"
        boundaryCondition="true"/>
    </listOfSpecies>
    <listOfReactions>
      <reaction name="Reaction1" reversible="true">
        <listOfReactants>
          <speciesReference species="S" stoichiometry="1"/>
          <speciesReference species="E" stoichiometry="1"/>
        </listOfReactants>
        <listOfProducts>
          <speciesReference species="ES" stoichiometry="1"/>
        </listOfProducts>
        <kineticLaw formula="k*S*E-r*ES">
          <listOfParameters>
            <parameter name="k" value="3"/>
            <parameter name="r" value="6"/>
          </listOfParameters>
        </kineticLaw>
      </reaction>
      <reaction name="Reaction2" reversible="false">
        <listOfReactants>
          <speciesReference species="ES" stoichiometry="1"/>
        </listOfReactants>
        <listOfProducts>
          <speciesReference species="E" stoichiometry="1"/>
          <speciesReference species="P" stoichiometry="1"/>
        </listOfProducts>
        <kineticLaw formula="k*ES">
          <listOfParameters>
            <parameter name="k" value="9"/>
          </listOfParameters>
        </kineticLaw>
      </reaction>
    </listOfReactions>
  </model>
</sbml>

```

The SBML model [enzyme.xml](#) used in section 2.2

The "[a/.b](#)" notation used above instructs *Mathematica* to evaluate all rules in the rule list [b](#) to expression [a](#). A rule in *Mathematica* is an expression such as  $x \rightarrow 7$ . Thus the expression  $(x+y)/. \{x \rightarrow 5, y \rightarrow 17+z*x\}$  (or the equivalent expression [ReplaceAll\[x+y, {x→5, y→17+z\\*x}](#)) would evaluate to  $22+z*x$ . If additional rules are included in the list that are not relevant to the expression "[a](#)", they are ignored. Note that rules are only evaluated once; the  $x$  in the second rule is not set equal to 5. To do this, one would use the expression  $(x+y)/. \{x \rightarrow 5, y \rightarrow 17+z*x\}$  (or its equivalent expression [ReplaceRepeated\[x+y, {x→5, y→17+z\\*x}](#)) which would evaluate to  $22+5$  ([ReplaceRepeated](#) can lead to infinite loops and should be used with care).

Since the above command is somewhat arduous to type in, the function [SBMLNDSolve](#) is available. The following syntax produces the same identical results as the preceding one:

```
n = SBMLNDSolve[r, 5]
```

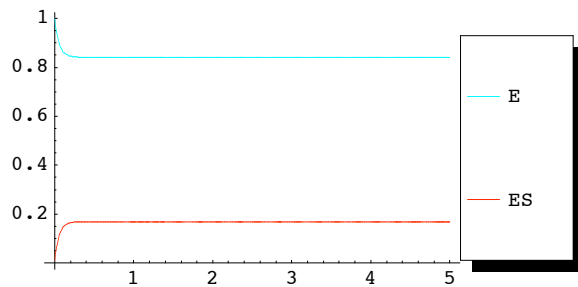
which returns:

```
{{EnzymeKinetics`E[t] → InterpolatingFunction[{{0., 5.}}, <>][t],  
  EnzymeKinetics`ES[t] → InterpolatingFunction[{{0., 5.}}, <>][t]}}
```

[SBMLNDSolve](#) accepts any options that are valid for [NDSolve](#). Additional details are provided in the Function Reference below.

The results may be plotted using [SBMLPlot](#), which accepts as input the output of [SBMLNDSolve](#).

```
SBMLPlot[n];
```



Observe that the context is not shown in the plot legend; in fact, it is `EnzymeKinetics`E` and `EnzymeKinetics`ES` that are plotted. This is done to improve readability of the plot.

## 2.3. Plotting with SBMLPlot

The output of `SBMLNDSolve` can be sent directly to `Plot`; however, because this can be tedious the function `SBMLPlot` is provided. It has the following formats:

<b>Format</b>	<b>Description</b>
<code>SBMLPlot[solution, options]</code>	Plot all variables in the solution for the entire duration of the simulation.
<code>SBMLPlot[solutions, {tbegin, tend}, options]</code>	Plot all variables in the solution from $t = tbegin$ to $t = tend$ .
<code>SBMLPlot[solution, {var1, var2, ...}, {tbegin, tend}, options]</code>	Plot <code>var1</code> , <code>var2</code> , ... from $t = tbegin$ to $t = tend$ .

Any valid option for `Plot` may be used. In addition, the option

`type→"Log"`

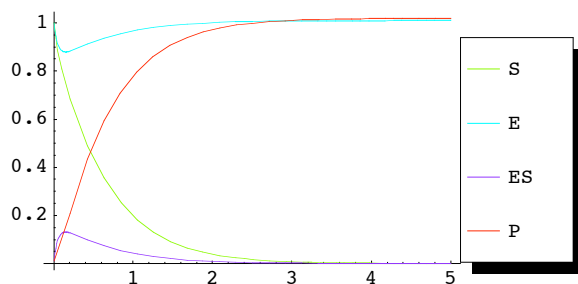
can be used to plot the y-axis on a log scale. For additional control over logarithmic plots the user should directly use `Graphics`Graphics`LogPlot` (to plot  $\log(y)$  as a function of  $x$ ), `Graphics`Graphics`LogLinearPlot` (to plot  $y$  as a function of  $\log(x)$ ), `Graphics`Graphics`LogLogPlot` (to plot  $\log(y)$  as a function of  $\log(x)$ ) (All of these functions are in the subcontext ``Graphics` of package `Graphics`, hence the word `Graphics` is repeated).

Suppose we modify the file `enzyme.xml` so that all the variables are allowed to change. We can do this by setting `boundaryCondition="false"` for all the variables in the SBML file, Suppose we name the modified file `"desktop/enzyme2.xml"`. Now we will have four variables in our model.

```
r = SBMLRead["desktop/enzyme2.xml"];
n = SBMLNDSolve[r, 5]

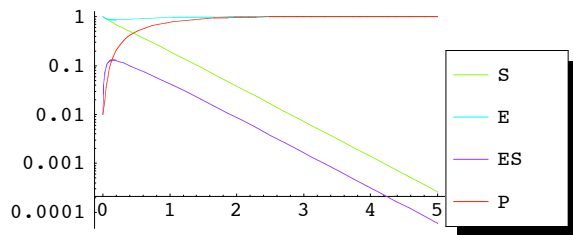
{{EnzymeKinetics`S[t] → InterpolatingFunction[{{0., 5.}}, <>][t],
  EnzymeKinetics`E[t] → InterpolatingFunction[{{0., 5.}}, <>][t],
  EnzymeKinetics`ES[t] → InterpolatingFunction[{{0., 5.}}, <>][t],
  EnzymeKinetics`P[t] → InterpolatingFunction[{{0., 5.}}, <>][t]}}
```

`SBMLPlot[n]`



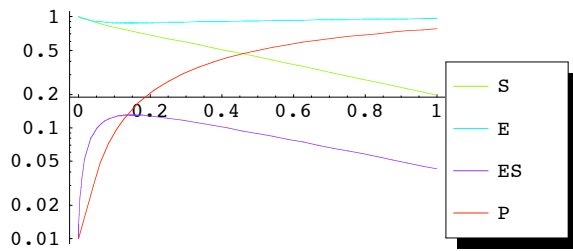
We can also plot the concentrations logarithmically,

`SBMLPlot[n, type → "Log"]`



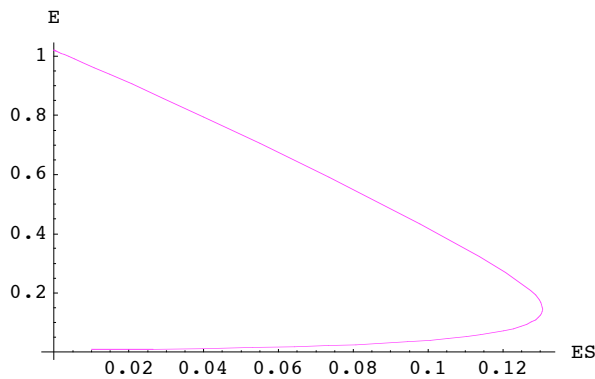
or zoom in on an interesting section of the plot:

```
SBMLPlot[n, {0, 1}, type -> "Log"]
```



The output of `SBMLNDSolve` can be sent directly to any standard *Mathematica* plotting function; for example, the reaction in the E-ES phase plane can be produced as follows:

```
ParametricPlot[Evaluate[{EnzymeKinetics`ES[t], EnzymeKinetics`P[t]} /. n],
  {t, 0, 5}, AxesLabel -> {"ES", "E"}, PlotStyle -> RGBColor[1, 0, 1];
```



## 2.4. Using MathSBML as an SBML file interpreter

The verbose option to `SBMLRead` allows one to produce readable listings of the files.

A number of options control the content of the verbose listing. These are summarized in the following table.

<b>Option</b>	<b>Value</b>	<b>Description</b>
<code>align</code>	<code>Left</code>	Alignment of verbose listing; any valid value for <code>TextAlignment</code> may be used.
<code>concise</code>	<code>False</code>	When <code>True</code> , overrides all verbose options. This option causes the most minimal output to be returned.
<code>shortenODES</code>	<code>True</code>	Display ODE's in short form, similar to the <code>Short[]</code> function.
<code>showKineticLaw</code>	<code>True</code>	Include <code>kineticLaw</code> in verbose reaction list.
<code>showReactionParameters</code>	<code>True</code>	Include list of local parameters for each reaction.
<code>stats</code>	<code>False</code>	Include a statistical summary of the file.
<code>verbose</code>	<code>False</code>	Print an interpretive listing of the SBML file.
<code>verboseContext</code>	<code>False</code>	Include the entire context of all variables in the verbose listing.

For example, to produce an interpretive listing of the file "`desktop/enzyme.xml`", we would use the following:

```
SBMLRead["desktop/enzyme.xml", verbose -> True];
```

The output produced by this command is illustrated on the following page; the fonts are reduced somewhat in size to fit the information on a single page.

**File Name:desktop/mathsbml–dbgfiles/enzyme.xml**  
**SBML Level 1 Version 2**

**Model: EnzymeKinetics**

**Unit Definitions**

----- None -----

**Compartments**

<u>Name</u>	<u>Volume</u>	<u>Units</u>	<u>Derived Units</u>	<u>Outside</u>
Cell	1.	volume	liter	...

**Species**

<u>Name</u>	<u>I.C.</u>	<u>Units</u>	<u>Derived Units</u>	<u>Charge</u>	<u>Compartment</u>	<u>Var/B.C.</u>
S	1.	substance	mole	0	Cell	Bound. Cond.
E	1.	substance	mole	0	Cell	Variable
ES	0.01	substance	mole	0	Cell	Variable
P	0.01	substance	mole	0	Cell	Bound. Cond.

**Global Parameters**

----- None -----

**Rules**

----- None -----

**Reactions**

<u>Name</u>	<u>Reaction</u>	<u>Parameters</u>	<u>Kinetic Law</u>
Reaction1	$E + S \rightleftharpoons ES$	$k \rightarrow 3.$ $r \rightarrow 6.$	$3.*E[t] - 6.*ES[t]$
Reaction2	$ES \rightarrow E + P$	$k \rightarrow 9.$	$9.*ES[t]$

**Differential Equations from Reactions**

<u>Species</u>	<u>Differential Equations</u>
E	$E'[t] == -3.*E[t] + 15.*ES[t]$
ES	$ES'[t] == 3.*E[t] - 15.*ES[t]$

## 3. Function Reference

This section contains usage information for `MathSBML` functions. This information may be retrieved during a *Mathematica* session any time after `MathSBML` has been imported by typing `?functionName`. Pre-defined rate laws for SBML level 1 are defined in section 4. The functions in this section are listed alphabetically.

Predefined SBML level-1 rate functions and functions in the Model Builder are not described in this section but are described in sections 4 and 5 respectively.

---

### ■ `MathSBML`SBMLCopy`

`SBMLCopy[input, output, options]` makes a copy of an SBML file to another SBML file, filtering the file through Mathematica's XML support to pretty-print the output in a more readable form. The content of the XML is unchanged, but the formatting/indentation is standardized. The names of the file must be specified as strings. If the output file already exists the output will be displayed on the screen. When checking for a pre-existing output file, the comparison is case-insensitive. `SBMLCopy[input]` will write the output to the screen instead of to a file.

**Options:**

`ExportOptions`→option list

`ImportOptions`→option list

**Example:**`SBMLCopy["myfile.xml", "yourfile.xml", ImportOptions→{CharacterEncoding→ "UTF8", ExportOptions→{CharacterEncoding→ "PrintableASCII"}}]`

For example, suppose the file "l1v1-minimal.xml" is formatted as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level1" level="1" version="1">
  <model>
    <listOfCompartments>
      <compartment name="x"/>
    </listOfCompartments>
    <listOfSpecies>
      <specie name="y" compartment="x" initialAmount="1"/>
    </listOfSpecies>
    <listOfReactions>
      <reaction name="x">
        <listOfReactants>
          <specieReference specie="y"/>
        </listOfReactants>
        <listOfProducts>
          <specieReference specie="y"/>
        </listOfProducts>
      </reaction>
    </listOfReactions>
  </model>
</sbml>
```

Then the expression

```
SBMLCopy["11v1-minimal.xml", "11v1-minimal-pretty-file.xml"]
```

will produce a second file, "11v1-minimal-pretty-file.xml", as follows. The name of the file produced will be returned as the value of the function.

```
<?xml version='1.0' encoding='UTF-8'?>
<sbml xmlns='http://www.sbml.org/sbml/level1'
  level='1'
  version='1'>
<model>
  <listOfCompartments>
    <compartment name='x' />
  </listOfCompartments>
  <listOfSpecies>
    <specie name='y'
      compartment='x'
      initialAmount='1' />
  </listOfSpecies>
  <listOfReactions>
    <reaction name='x'>
      <listOfReactants>
        <specieReference specie='y' />
      </listOfReactants>
      <listOfProducts>
        <specieReference specie='y' />
      </listOfProducts>
    </reaction>
  </listOfReactions>
</model>
</sbml>
```

---

## ■ MathSBML`SBMLNDSolve

`SBMLNDSolve[model, tmax, options]` evaluates `NDSolve` on an SBML model, where `model` is the output of `SBMLRead`, `tmax` is the duration of the `NDSolve` run, and `options` are any valid options for `NDSolve`.

Additional Notes and Limitations:

- (1) units are ignored by `SBMLNDSolve`
- (2) events are not currently processed by `SBMLNDSolve`. It is anticipated that this functionality will be added in a later version of `SBMLNDSolve`
- (3) The inclusion of algebraic constraints (rules) along with differential equations to produce a system of Differential-Algebraic Equations requires Mathematica Version  $\geq 5.0$ ; version checking is performed dynamically.

## ■ Example

```
r = SBMLRead["desktop/enzyme.xml", concise -> True];
n = SBMLNDSolve[r, 300, MaxSteps -> 5000]

{{EnzymeKinetics`E[t] -> InterpolatingFunction[{{0., 300.}}, <>][t],
  EnzymeKinetics`ES[t] -> InterpolatingFunction[{{0., 300.}}, <>][t]}}

{{X[t] -> InterpolatingFunction[{{0., 300.}}, <>][t],
  Y[t] -> InterpolatingFunction[{{0., 300.}}, <>][t],
  Z[t] -> InterpolatingFunction[{{0., 300.}}, <>][t]}}
```

## ■ MathSBML`SBMLPlot

`SBMLPlot[solution, {var1, var2,...}, {tbegin, tend}, options]` plots the results of a simulation. `solution` is either the output of `SBMLNDSolve`

or the output of an `SBMLRead` run with numerical solution enabled.

The variables named `var1, var2,..` are plotted from `t=tbegin` to `t=tend` on a single plot. The context of the variable name must be specified, i.e., if the model identifier is `foo`, to plot variables `A` and `C` only but not any other variable, one would invoke `SBMLPlot[solution, {foo`A,foo`C}, {tbegin,tend}]`.

`SBMLPlot[solutions,{tbegin,tend}, options]` plots all variables in the solution set for the specified time range.

`SBMLPlot[solution,options]` plots all variables for the entire duration of the run.

Any valid option for `Plot` may be used.

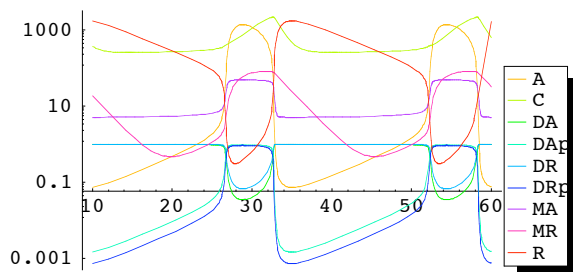
### Additional options:

`variables`→ `var` or `{var1, var2, ...}` or "All"; gives the names of the variables to be plotted. If this option is omitted all variables are plotted. The context of the variable name must be specified, i.e., if the model identifier is `foo`, to plot variables `A` and `C` only but not any other variable, one would specify `variables`→ `{foo`A,foo`C}`. If any of the variable names is `All` then all variables will be plotted. Note: This option is only available in `SBMLPlot[solution,options]` form.  
`type`→ "Log", if specified, the y-axis is logarithmic.

## ■ Example

The following example reads the file "`vilar.xml`" that is available on the <http://www.sbml.org> model repository, runs a simulation for 100 time units, and plots the results on a logarithmic scale from `t=10` through `t = 60`.

```
r = SBMLRead["desktop/vilar.xml", concise -> True];
n = SBMLNDSolve[r, 100, MaxSteps -> 25000];
SBMLPlot[n, {10, 60}, type -> "Log"]
```



## ■ MathSBML`SBMLRead

### ? SBMLRead

SBMLRead[filename, options] returns an option list of the form:

```
{
SBMLODES→{v1'[t]==expression, v2'[t]==expression,...},
SBMLParameters→{k1→value,k2→value,...},
SBMLIC→{v1[0]==value,v2[0]==value,...},
SBMLSpecies→{v1[t],v2[2],...},
SBMLAlgebraicRules→{expression1==0, expression2==0, ...},
SBMLUnitDefinitions→{unitName1→unitDefinition1, unitName2→unitDefinition2,...},
SBMLUnitAssociations→{var1→units1,var2→units2,...},
SBMLReactions→{reaction1, reaction2,...},
SBMLModelName→name,
SBMLNameIDAssociations→{id1→name1,id2→name2},
SBMLFunctions→{function1→def, function2→def,...},
SBMLEvents→{event1def1, eventdef2,...},
SBMLNumericalSolution→numerical solution
}
```

where v1,v2,... give all of the species in the SBML file; the expression gives the derived differential equation for that species; value (following SBMLParameters or SBMLIC) is the SBML value of the associated parameter or initial condition; unitName1,... are user-defined units; unitDefinition1,... are expressions that give the unit definitions in terms of pre-defined units; units1,... are the units that correspond to variable var1,..., which can be either species, parameter, or compartment; reaction1,reaction2,... are the reactions in standard biochemical form  $\sum_i R_i \rightarrow \sum_j P_j$ , where R and P are the reactants and products in each reaction. SBMLFunctions, SBMLEvents, and SBMLNameIDAssociations only apply for Level 2. Event definitions have the form id→{"trigger"→expression,"delay"→expression,"events"→{var1→exp1,var2→exp2,...}} where id is the event id (assigned to Event\_n if not provided); and expression, etc., are Mathematica expressions. SBMLFunctions have the form {id→Function[...], id→Function[...],...} where Function[...] gives a pure Mathematica Function definition and id is the corresponding SBML id for the function. Any portion of the returned option list may be turned off with the option return. SBMLNumericalSolution is the output of SBMLNDSolve, if a numerical solution is requested.

The intention is that this format contains all information necessary to pass the model to NDSolve in the following manner:

```
r=SBMLRead[filename,options]
Apply[NDSolve,{Join[SBMLODES/.r, SBMLIC/.r]/.
(SBMLParameters/.r), SBMLSpecies/.r,{t,0,tmax},NDSolveOptions]}
```

where NDSolveOptions are any valid options for NDSolve and tmax is the duration of the NDSolve Rule.

Options for SBMLRead are:

**align**→"Left", alignment of verbose output on screen. Any valid values of TextAlignment may be used; since the value is passed directly to TextAlignment without error checking invalid values will result in the default value of TextAlignment ("Left"). This option is ignored unless verbose→True.

**concise**→False. When True, overrides whatever values are set to verbose, warnings, and stats to set all of them to False. This option minimizes the written output.

**context**→"Automatic" (default), assign all global model variables to a context given by the model name; thus variables x,y,z in them SBML become modelName`x, modelName`y, modelName`z in the Mathematica representation. Local parameters in reactions will be assigned a context modelName`reactionname, i.e., if

reaction20 in model foo has a parameter k it will be called foo`reaction20`k.

`context`→str, where str is any string. All variables will be assigned to the context str` (reaction parameters to context str`reactionname`) instead of modelname`. The terminating "`" character is optional.

`context`→"None", all model variables are placed in the Global` context. Be aware that this could be dangerous, as symbols in the model could collide with other variables previously defined in the Mathematica environment and lead to unexpected results.

`defaultIC`→Indeterminate, if reassigned, then all unspecified initial conditions will be set to the value of defaultIC. Otherwise they will remain Indeterminate.

`defaultParameterValue`→Indeterminate, if reassigned, then all unspecified parameter values will be set to the value of defaultIC. Otherwise they will remain Indeterminate.

`evaluateParameters`→True, immediately evaluate parameters in reactions, otherwise return reactions with symbolic parameters.

`NDSolveOptions`→{}, options to be passed to NDSolve; ignored unless SBMLNumericalSolution→n, withing return, as in return→{SBMLNumericalSolution→ 25, ...}, where n>0 is a number.

`PlotOptions`→{} contains a list of options to be passed to SBMLPlot, including any options to be passed to Plot. If this option is omitted or a null list, no plot is generated. A warning message will be generated if the user does not also request a numerical solution by setting SBMLNumericalSolution to a positive value as part of the return options.

`return`→{SBMLODES→True, SBMLIC→True, SBMLParameters→True, SBMLSpecies→True, SBMLAlgebraicRules→True, SBMLUnitDefinitions→True, SBMLUnitAssociations→True, SBMLReactions→True, SBMLFunctions→True, SBMLEvents→True, SBMLNameIDAssociations→True, SBMLModelName→True, SBMLNumericalSolution→0}; this option allows the user to control the return value of SBMLRead. All boolean items are returned except for those set to False. A numerical solution is only returned if SBMLNumericalSolution evaluates to a positive number. Setting return→None is equivalent to setting everything to False.

`shortenODES`→False, ignored unless verbose→True and showKineticLaw→True. If shortenODES→False (default), then the entire differential equation will be displayed in the verbose listing; if shortenODES→True then the Mathematica Short[...] version will be used.

`showKineticLaw`→True, ignored unless verbose→True; if showKineticLaw→True (default), the SBML kinetic law is shown in the reaction-listing of the verbose display; otherwise the kinetic laws are not displayed in the reactions table

`showReactionParameters`→True, ignored unless verbose→True; if showReactionParameters→True (default), the local parameters in each reaction are shown in the reaction-listing of the verbose display; otherwise the local parameters are not displayed in the reactions table

`stats`→False, print a statistical summary of the file

`underscore`→"\_", character (or string) that is used to replace the underscore ("\_") character in SBML identifiers.

`verbose`→False, if True, print an interpretive table of the SBML

`verbosecontext`→False, if True, the context (scope) of all variables will be indicated in the verbose display. If False, only the pure model variable will be indicated. This option will be ignored unless verbose→True.

`warnings`→ True, if False, warning messages will be suppressed.

#### Additional Notes and Limitations of SBMLRead

- (1) `SBMLRead` does not perform XML or SBML validation. If invalid SBML or XML is supplied, unexpected results can occur. In general, incorrectly formatted XML will cause *Mathematica's* `Import[...]` function to print an error message indicating the line number for the first error and then *Mathematica* will terminate.
- (2) `SBMLRead` is currently only compatible with SBML Level 1 (versions 1 and 2) and Level 2 (version 1). Subsequent releases will support higher levels.
- (3) In SBML Level 1, all of the mathematical functions (e.g., `abs`, `acos`, etc.) are fully supported. Thus `cos(x)` becomes `Cos[x]`, etc. Predefined rate law functions are recognized as functions but are not implemented. Thus, if the function `umr(argument list)` is specified in the SBML, it will be recognized as a predefined function and will be expanded in the *Mathematica* model as `umr[argument list]`. However, unlike the mathematical functions, no implementation is provided. Thus if the model contains these functions, the user must supply a *Mathematica* implementation for `umr`, etc.
- (4) By default, all parameters are replaced with their numerical values as specified in the model. This can be switched off using the option `evaluateParameters`, in which case `SBMLRead` will return a list of *Mathematica* replacement rules of the form `name`→`value`.

- (5) The topological relationship specified by the `outside` attribute in a compartment definition is ignored, although `SBMLRead` will display the relationship in the verbose listing. If no outside component exists, the *Mathematica* variable `Indeterminate` is displayed.
- (6) The `reversible` parameter of the `reaction` type is ignored with the following exception: reversible reactions in the list of reactions returned (`SBMLReactions`) will use the double arrow ( $\rightleftharpoons$ ) instead of the single forward arrow ( $\rightarrow$ ).
- (7) In SBML Level 1, Unspecified initial conditions and parameter values will be labeled as `Indeterminate` if they are not specified, and a warning message will be printed. Models with `Indeterminate` parameters and initial conditions will cause an error in `NDSolve`. To prevent this from happening, the user can optionally specify the options `defaultParameterValue` and `defaultIC` to set all `Indeterminate` parameter values and initial conditions. For example, `defaultParameterValue -> 1` will set all `Indeterminate` parameters equal to 1. Unspecified units will be labeled as `Indeterminate` in the verbose listing, but no association will be returned for indeterminate units.
- (8) Annotations and notes are ignored.

## ■ Example

```
r = SBMLRead["desktop/enzyme.xml", concise -> True]
```

```
{SBMLODES -> {EnzymeKinetics`E'[t] == -3. EnzymeKinetics`E[t] + 15. EnzymeKinetics`ES[t],
  EnzymeKinetics`ES'[t] == 3. EnzymeKinetics`E[t] - 15. EnzymeKinetics`ES[t]},
 SBMLIC -> {EnzymeKinetics`E[0] == 1., EnzymeKinetics`ES[0] == 0.01}, SBMLParameters ->
  {EnzymeKinetics`Cell -> 1., EnzymeKinetics`S -> 1., EnzymeKinetics`P -> 0.01},
 SBMLSpecies -> {EnzymeKinetics`E[t], EnzymeKinetics`ES[t]}, SBMLAlgebraicRules -> {},
 SBMLUnitDefinitions -> {substance -> mole, volume -> liter, time -> second},
 SBMLUnitAssociations -> {EnzymeKinetics`Cell -> volume,
  EnzymeKinetics`S -> substance, EnzymeKinetics`E -> substance,
  EnzymeKinetics`ES -> substance, EnzymeKinetics`P -> substance},
 SBMLReactions -> {EnzymeKinetics`E + EnzymeKinetics`S ==> EnzymeKinetics`ES,
  EnzymeKinetics`ES -> EnzymeKinetics`E + EnzymeKinetics`P}}
```

---

## ■ MathSBML`SBMLWrite

`SBMLWrite[options]` will write a model in a specified format as determined by the options.

Options are:

`inputfile`→*string*, name of SBML file that is to be converted (read). An inputfile is required.  
`outputfile`→*string*, name of file that output is written to. If not specified, the output is written to the screen  
`format`→*string*, type of output to produce. If no format is specified or an invalid format is specified the original model will be returned. Valid formats are: "Fortran", "HTML", "XPP" (the value of format is case-insensitive). It is anticipated that other formats will be added in future releases.

### Notes for XPP format:

- (1) XPP implementation is limited to ODEs, parameters, and initial conditions. It is anticipated that more complex forms will be allowed in future release of MathSBML.
- (2) SBML functions are not implemented in XPP files. Instead, they are instantiated in place.
- (3) By default, assignment rules are not instantiated before evaluation. To force evaluation of assignment rules, use the option `evaluateParameters`→`True`. This will also force evaluation of all parameters.
- (4) SBML events are not implemented in XPP files.
- (5) Parameters or initial conditions that are not set in the model will be assigned a value of `Indeterminate` in the XPP file, which is not a valid XPP value.

### Notes for Fortran Format:

- (1) Fortran format is developmental
- (2) The output file will contain the three subroutines `res`, `addp`, and `jac` required by `lsodi`
- (3) Documentation of `lsodi` can be found at <http://netlib.org/alliant/ode/prog/lsodi.f>
- (4) The output file will also contain a subroutine `init` that sets the initial condition.
- (5) The output file also contains two modules for each event in the file, a logical function `trigger_<event>` that returns the boolean value of the event's trigger given the values of all the system's state variables, and a subroutine `activate_<event>` that modifies the system's state variables as per the event.
- (6) The user is expected to implement his/her own driver software that utilizes the event files and calls `lsodi` or some other solver as required. An example (without events) is provided in the `lsodi` documentation.
- (7) Because `lsodi` does not directly support events, if the model contains events the user will need to write wrappers for the subroutines provided that make them compatible with his/her chosen solver.
- (8) Parameters in the model such as `foo`k` and `foo`r1`k` will become `fooxk` and `fooxr1xk`

### ■ Example - XPP output to screen

```
SBMLWrite[inputfile → "enzyme.xml", format → "XPP"]
```

```
# Model Name:      EnzymeKinetics
#
# Creation Time:   30-June-2003 16:51:05.965258
# User:           jamestkirk
# Machine:        Slartibartifast
# System:         PowerPC PowerMac MacOSX
# Generated by MathSBML 2.0 beta 16 (1 July 2003)
#
# Differential Equations
#
EnzymeKinetics_E'=-3.*EnzymeKinetics_E+15.*EnzymeKinetics_ES
EnzymeKinetics_ES'=3.*EnzymeKinetics_E-15.*EnzymeKinetics_ES
#
# Parameters
#
par EnzymeKinetics_Cell=1.
par EnzymeKinetics_S=1.
par EnzymeKinetics_P=0.01
#
# Initial Conditions
#
init EnzymeKinetics_E=1.
init EnzymeKinetics_ES=0.01
#
done
```

### ■ Example - XPP output to file

```
SBMLWrite[inputfile → "enzyme.xml", format → "XPP", outputfile → "desktop/test.ode"]
```

```
desktop/test.ode
```

### ■ Example - Fortran output for pure differential system without algebraic rules

```
SBMLWrite[inputfile → "enzyme.xml", format → "Fortran"]
```

```
C SBML Model Name:  EnzymeKinetics
C
C Generated by MathSBML 2.0.3 (7 Aug 2003)
C Creation Time:    7-August-2003 18:31:25.176289
C User:            jamestkirk
C Machine:         Slartibartifast
C Processor:       PowerPC
C Machine type:    PowerMac
```

```

C Oper. System:  MacOSX
C -----
C
C lsodi compliant SBML model
C Reference: http://netlib.org/alliant/ode/prog/lsodi.f
C
C -----
C
C This file contains the following modules:
C
C Module Name      Description
C -----
C addp (subroutine) Add A to any matrix - required by lsodi
C init (subroutine) Set initial conditions
C jac (subroutine)  Compute Jacobian - required by lsodi
C res (subroutine)  Calculate Residuals - required by lsodi
C
C -----
C
C      subroutine res(neq,t,y,s,r,ires)
C      double precision r,s,t,y
C
C This is subroutine res for lsodi
C This function computes the residuals  $r(i)=g(t,y)-A(t,y)(dy/dt)$ 
C for the linear-implicit system system  $(A)*(dy/dt)=g(t,y)$ 
C where A is a constant, possibly singular, matrix.
C
C Here A is diagonal with (restriction imposed by SBML, not lsodi)
C      A(i,i)=1, i=1,...,m, where m=# of odes in the SBML Model
C      A(i,i)=0, i=m+1,...,m+nrules, where nrules =# of algebraic rules
C      A(i,i)=0, i=nrules+1,...,nvars, where nvars is the total number
C      of variables in the system and nvars-nrules-m>0 is the
C      number of variables controlled purely by events
C
C If the system is purely differential
C the right-hand side of the system  $dydt = g(y,t)$  will
C be returned if s is zero-filled.
C
C Representation of Model Variables by the array y
C -----
C y(1) = E
C y(2) = ES
C
C      dimension r(2),s(2),y(2)
C      r(1)=-s(1) - 3.*y(1) + 15.*y(2)
C      r(2)=-s(2) + 3.*y(1) - 15.*y(2)
C      return
C      end
C
C
C -----
C
C      subroutine addp(neq, t, y, ml, mu, p, nrowp)
C
C Subroutine addp required by lsodi
C
C      double precision p, t, y
C      dimension y(2), p(nrowp,2)

```

```
integer i
Do i = 1,2
  p(i,i) = p(i,i)+1
End Do
return
end

C
C -----
C
C   subroutine jac (neq,t,y,s,ml,mu,p,nrowp)
C
C Subroutine jac required by lsodi, computes jacobian
C
C   dimension y(2), s(2),p(nrowp,2)
C
C   p(1,1)=-3.
C   p(1,2)=15.
C   p(2,1)=3.
C   p(2,2)=-15.
C
C   return
C   end
C
C -----
C
C Subroutine init(neq,y)
C double precision y
C dimension y(2)
C y(1)=1.
C y(2)=0.01
C return
C end
```

## 4. Predefined Rate Laws

The rate laws are implemented as described in the SBML Level 1 Version 2 final spec.

---

### ■ massi

`massi[S1,S2,...,Sn,k]` is a predefined rate law for Irreversible Mass Action Kinetics that returns a velocity  $v=k*S_1*S_2*S_3\cdots S_n$ ;  
`massi[k]` returns  $v=k$ ;  
`massi[]` returns  $v=1$ .

---

### ■ massr

`massr` is not implemented in MathSBML because `massr[S1,...,Pj,...,k1,k2]` can be unambiguously written as `massi[S1,...,k1]-massi[,Pj,...,k2]`.

---

### ■ uui

`uui[S,Vm,Km]` is a predefined rate law for Irreversible Simple Michaelis-Menten Kinetics that returns a velocity  $v = \frac{V_m * S}{K_m + S}$ .

---

### ■ uur

`uur[S,P,Vf,Vr,Kms,Kmp]` is a predefined rate law for Uni-Uni Reversible Simple Michaelis-Menten Kinetics that returns a velocity  $v = \frac{V_f (S / K_{ms}) - V_r (P / K_{mp})}{1 + (P / K_{mp}) + (S / K_{ms})}$ .

---

### ■ uuhr

`uuhr[S,P,Vf,Km1,Km2,Keq]` is a predefined rate law for Uni-Uni Reversible Simple Michaelis-Menten Kinetics with Haldane Adjustment. The rate law returned is  $v = \frac{(V_f / m1) (S - P / K_{eq})}{1 + S / m1 + P / K_{m2}}$ .

## ■ isour

isour[S, P, V<sub>f</sub>, K<sub>ms</sub>, K<sub>mp</sub>, K<sub>ii</sub>, K<sub>eq</sub>] is a predefined rate law for Iso

Uni-Uni kinetics. The rate law returned is 
$$\frac{V_f (S - P / K_{eq})}{(S (1 + P / K_{ii}) + K_{ms} (1 + P / K_{mp}))}$$
.

## ■ hilli

hilli[S, V, K, h] is a predefined rate law for Hill Kinetics. The rate law returned is 
$$v = \frac{V * S^h}{K^h + S^h}$$
.

## ■ hillr

hillr[S, P, V<sub>f</sub>, S<sub>half</sub>, P<sub>half</sub>, h, K<sub>eq</sub>] is a predefined rate law for

reversible Hill kinetics. The rate law is 
$$\frac{S \left(1 - \frac{P}{S K_{eq}}\right) \left(\frac{P}{P_{half}} + \frac{S}{S_{half}}\right)^{-1+h} V_f}{\left(1 + \left(\frac{P}{P_{half}} + \frac{S}{S_{half}}\right)^h\right) S_{half}}$$
.

## ■ hillmr

hillmr[S, P, M, S<sub>0.5</sub>, P<sub>0.5</sub>, M<sub>0.5</sub>, V<sub>f</sub>, K<sub>eq</sub>, h, α] is a predefined rate law for reversible Hill kinetics with one modifier. The rate law is

$$\frac{S \left(1 - \frac{P}{S K_{eq}}\right) \left(\frac{P}{P_{0.5}} + \frac{S}{S_{0.5}}\right)^{-1+h} V_f}{(K_1 + K_2) S_{0.5}}$$
 where  $K_1 = \left(\frac{P}{P_{0.5}} + \frac{S}{S_{0.5}}\right)^h$  and  $K_2 = \frac{1 + \left(\frac{V_f}{M_{0.5}}\right)^h}{1 + \alpha \left(\frac{V_f}{M_{0.5}}\right)^h}$ .

## ■ hillmmr

hillmmr[S, P, M, S<sub>0.5</sub>, P<sub>0.5</sub>, M<sub>0.5</sub>, M<sub>a</sub>, M<sub>a0.5</sub>, M<sub>b</sub>, M<sub>b0.5</sub>, V<sub>f</sub>, K<sub>eq</sub>, h, a, b, α<sub>1</sub>, α<sub>2</sub>, α<sub>12</sub>] is a predefined rate law for reversible Hill Kinetics with Two

Modifiers. The Rate Law is 
$$\frac{S \left(1 - \frac{P}{S K_{eq}}\right) \left(\frac{P}{P_{0.5}} + \frac{S}{S_{0.5}}\right)^{-1+h} V_f}{S_{0.5} (K_1 + K_2)}$$
 where  $K_1 = (P/P_{0.5} + S/S_{0.5})^h$  and  $K_2 = \frac{1 + (M_a / M_{a0.5})^h + (M_b / M_{b0.5})^h}{1 + (M_a / M_{a0.5})^h \alpha_1 + (M_b / M_{b0.5})^h \alpha_2 + (M_a / M_{a0.5})^h (M_b / M_{b0.5})^h \alpha_1 \alpha_2}$

## ■ usii

usii[S,V,K<sub>m</sub>,K<sub>i</sub>] is a predefined rate law for substrate inhibition

kinetics (irreversible). The rate law returned is  $\frac{S V}{\left(1 + \frac{S^2}{K_i} + \frac{S}{K_m}\right) K_m}$ .

## ■ usir

usir[S,P,V<sub>f</sub>,V<sub>r</sub>,K<sub>ms</sub>,K<sub>mp</sub>,K<sub>i</sub>] is a predefined rate law for substrate

inhibition kinetics (reversible). The rate law returned is  $\frac{\frac{S V_f}{K_{ms}} + \frac{P V_r}{K_{mp}}}{1 + \frac{S^2}{K_i} + \frac{P}{K_{mp}} + \frac{S}{K_{ms}}}$ .

## ■ uai

uai[S,V,K<sub>sa</sub>,K<sub>sc</sub>] is a predefined rate law for

substrate activation. The rate law returned is  $\frac{S^2 V}{K_{sa}^2 \left(1 + \frac{S^2}{K_{sa}^2} + \frac{S}{K_{sa}} + \frac{S}{K_{sc}}\right)}$ .

## ■ ucii

ucii[S,V,Inh, K<sub>m</sub>,K<sub>i</sub>] is a predefined rate law for competitive

inhibition (irreversible). The rate law returned is  $\frac{S V}{\left(1 + \text{Inh} / K_i + S / K_m\right) K_m}$ .

## ■ ucir

ucir[S, P,Inh, V<sub>f</sub>, V<sub>r</sub>, K<sub>ms</sub>, K<sub>mp</sub>, K<sub>i</sub>] is a predefined rate law for

competitive inhibition (reversible). The rate law returned is  $\frac{\frac{S V_f}{K_{ms}} - \frac{P V_r}{K_{mp}}}{1 + \frac{\text{Inh}}{K_i} + \frac{P}{K_{mp}} + \frac{S}{K_{ms}}}$ .

## ■ unii

unii[S, Inh, V, K<sub>m</sub>,K<sub>i</sub>] is a predefined rate law for noncompetitive

inhibition (irreversible). The rate law returned is  $\frac{S V}{\left(1 + \frac{\text{Inh}}{K_i} + \frac{S \left(1 + \frac{\text{Inh}}{K_i}\right)}{K_m}\right) K_m}$ .

## ■ unir

`unir[S, P, Inh, Vf, Vr, Kms, Kmp, Ki]` is a predefined rate law for noncompetitive

inhibition (reversible). The rate law returned is 
$$\frac{\frac{S V_f}{K_{ms}} - \frac{P V_r}{K_{mp}}}{1 + \frac{Inh}{K_i} + \left(1 + \frac{Inh}{K_i}\right) \left(\frac{P}{K_{mp}} + \frac{S}{K_{ms}}\right)} .$$

## ■ uuci

`uuci[S, Inh, V, Km, Ki]` is a predefined rate law for uncompetitive

inhibition (irreversible). The rate law returned is 
$$\frac{S V}{\left(1 + \frac{S \left(1 + \frac{Inh}{K_i}\right)}{K_m}\right) K_m} .$$

## ■ uucr

`uucr[S, P, Inh, Vf, Vr, Kms, Kmp, Ki]` is a predefined rate law for uncompetitive

inhibition (reversible). The rate law returned is 
$$\frac{\frac{S V_f}{K_{ms}} - \frac{P V_r}{K_{mp}}}{1 + \left(1 + \frac{Inh}{K_i}\right) \left(\frac{P}{K_{mp}} + \frac{S}{K_{ms}}\right)} .$$

## ■ umi

`umi[S, Inh, V, Km, Kis, Kic]` is a predefined rate law for mixed inhibition

kinetics (irreversible). The rate law returned is 
$$\frac{S V}{\left(1 + \frac{Inh}{K_{is}} + \frac{S \left(1 + \frac{Inh}{K_{ic}}\right)}{K_m}\right) K_m} .$$

## ■ umr

`umr[S, P, Inh, Vf, Vr, Kms, Kmp, Kis, Kic]` is a predefined rate law for mixed inhibition

kinetics (reversible). The rate law returned is 
$$\frac{\frac{S V_f}{K_{ms}} - \frac{P V_r}{K_{mp}}}{1 + \frac{Inh}{K_{is}} + \left(1 + \frac{Inh}{K_{ic}}\right) \left(\frac{P}{K_{mp}} + \frac{S}{K_{ms}}\right)} .$$

## ■ uaii

`uaii[S, Ac, V, Km, Ka]` is a predefined rate law for specific

action kinetics (irreversible). The rate law returned is 
$$\frac{S V}{\left(1 + \frac{K_a}{A_c} + \frac{S}{K_m}\right) K_m} .$$

---

## ■ uar

`uar[S,P,Ac,Vf, Vr, Kms, Kmp, Ka]` is a predefined rate law for specific

action kinetics (reversible). The rate law returned is  $\frac{\frac{S V_f}{K_{ms}} - \frac{P V_r}{K_{mp}}}{1 + \frac{K_a}{A_c} + \frac{P}{K_{mp}} + \frac{S}{K_{ms}}}$ .

---

## ■ ucti

`ucti[S, Ac, V, Km, Ka]` is a predefined rate law for catalytic

activation (irreversible). The rate law returned is  $\frac{S V}{\left(1 + \frac{K_a}{A_c} + \frac{S \left(1 + \frac{K_a}{A_c}\right)}{K_m}\right) K_m}$ .

---

## ■ uctr

`uctr[S, P, Ac, Vf, Vr, Kms, Kmp, Ka]` is a predefined rate law for catalytic

activation (reversible). The rate law returned is  $\frac{\frac{S V_f}{K_{ms}} - \frac{P V_r}{K_{mp}}}{1 + \frac{K_a}{A_c} + \left(1 + \frac{K_a}{A_c}\right) \left(\frac{P}{K_{mp}} + \frac{S}{K_{ms}}\right)}$ .

---

## ■ umai

`umai[S, Ac, V, Km, Kas, Kac]` is a predefined rate law for mixed activation

kinetics (irreversible). The rate law returned is  $\frac{S V}{\left(1 + \frac{K_{as}}{A_c} + \frac{S \left(1 + \frac{K_{ac}}{A_c}\right)}{K_m}\right) K_m}$ .

---

## ■ umar

`umar[S, P, Ac, Vf, Vr, Kms, Kmp, Kas, Kac]` is a predefined rate law for mixed activation

kinetics (reversible). The rate law returned is  $\frac{\frac{S V_f}{K_{ms}} - \frac{P V_r}{K_{mp}}}{1 + \frac{K_{as}}{A_c} + \left(1 + \frac{K_{ac}}{A_c}\right) \left(\frac{P}{K_{mp}} + \frac{S}{K_{ms}}\right)}$ .

## ■ uhmi

uhmi[S, M, V, K<sub>m</sub>, K<sub>d</sub>, a, b] is a predefined rate law for general hyperbolic

modifier kinetics (irreversible). The rate law returned is 
$$\frac{S V \left(1 + \frac{b M}{a K_d}\right)}{\left(1 + \frac{M}{K_d} + \frac{S \left(1 + \frac{M}{a K_d}\right)}{K_m}\right) K_m}$$
.

## ■ uhmr

uhmr[S, P, M, V<sub>f</sub>, V<sub>r</sub>, K<sub>ms</sub>, K<sub>mp</sub>, K<sub>d</sub>, a, b] is a predefined rate law for general hyperbolic

modifier kinetics (reversible). The rate law returned is 
$$\frac{\left(1 + \frac{b M}{a K_d}\right) \left(\frac{S V_f}{K_{ms}} - \frac{P V_r}{K_{mp}}\right)}{1 + \frac{M}{K_d} + \left(1 + \frac{M}{a K_d}\right) \left(\frac{P}{K_{mp}} + \frac{S}{K_{ms}}\right)}$$
.

## ■ ualii

ualii[S, Inh, V, K<sub>s</sub>, K<sub>ii</sub>, n, L] is a predefined rate law for allosteric

inhibition (irreversible). The rate law returned is 
$$\frac{S V \left(1 + \frac{S}{K_s}\right)^{-1+n}}{\left(L \left(1 + \frac{\text{Inh}}{K_{ii}}\right)^n + \left(1 + \frac{S}{K_s}\right)^n\right) K_s}$$
.

## ■ ordubr

ordubr[A, P, Q, V<sub>f</sub>, V<sub>r</sub>, K<sub>ma</sub>, K<sub>mq</sub>, K<sub>mp</sub>, K<sub>ip</sub>, K<sub>eq</sub>] is a predefined rate for

Ordered Uni-Bi Kinetics. The rate law returned is 
$$\frac{\left(A - \frac{P Q}{K_{eq}}\right) V_f}{A \left(1 + \frac{P}{K_{ip}}\right) + K_{ma} + \frac{(P Q - Q K_{mp} + P K_{mq}) V_f}{K_{eq} V_r}}$$
.

## ■ ordbur

ordbur[A, B, P, V<sub>f</sub>, V<sub>r</sub>, K<sub>ma</sub>, K<sub>mb</sub>, K<sub>mp</sub>, K<sub>ia</sub>, K<sub>eq</sub>] is a predefined rate for Ordered

Bi-Uni Kinetics. The rate law returned is 
$$\frac{\left(A B - \frac{P}{K_{eq}}\right) V_f}{A B + B K_{ma} + A K_{mb} + \frac{(P \left(1 + \frac{A}{K_{ia}}\right) + K_{mp}) V_f}{K_{eq} V_r}}$$
.

## ■ ordbbr

ordbbr[A, B, P, Q, V<sub>f</sub>, V<sub>r</sub>, K<sub>mA</sub>, K<sub>mB</sub>, K<sub>mP</sub>, K<sub>mQ</sub>, K<sub>iA</sub>, K<sub>iB</sub>, K<sub>iP</sub>, K<sub>eq</sub>] is a predefined rate for Ordered Bi-Bi Kinetics. The rate law returned is 
$$\frac{(A B - P Q / K_{eq}) V_f}{A B (1 + P / K_{iP}) + B K_{mA} + (A + K_{iA}) K_{mB} + K_1}$$
 where  $K_1 = (V_f / (K_{eq} V_r)) (Q K_2 + P (1 + A / K_{iA}) K_{mQ})$  and  $K_2 = (1 + P (1 + B / K_{iB}) + B K_{mA} / (K_{iA} K_{mB})) K_{mP}$ .

## ■ ppbr

ppbr[A, B, P, Q, V<sub>f</sub>, V<sub>r</sub>, K<sub>ma</sub>, K<sub>mb</sub>, K<sub>mp</sub>, K<sub>mq</sub>, K<sub>ia</sub>, K<sub>iq</sub>, K<sub>eq</sub>] is a predefined rate for Ping-Pong Bi-Bi Kinetics. The rate law returned is 
$$\frac{(A B - \frac{P Q}{K_{eq}}) V_f}{A B + B (1 + \frac{Q}{K_{iq}}) K_{ma} + A K_{mb} + \frac{(Q (P + K_{mp}) + P (1 + \frac{A}{K_{ia}}) K_{mq}) V_f}{K_{eq} V_r}}$$

## 5. Model Builder Function Reference

The MathSBML Model Builder consists of a suite of functions that allow the user to build up an SBML model manually from from standard *Mathematica* data structures.

The Model Builder is developmental and incomplete. The following functions are available in MathSBML 2.1.4:

<b>Function</b>	<b>Description</b>
<code>compartmentToSBML</code>	Returns a <code>&lt; compartment &gt;</code> or <code>a &lt; listOfCompartments &gt;</code>
<code>eventToSBML</code>	Returns an <code>&lt; event &gt;</code> or <code>&lt; listOfEvents &gt;</code>
<code>functionToSBML</code>	Returns a <code>&lt; functionDefinition ... &gt;</code> or <code>&lt; listOfFunctionDefinitions ... &gt;</code>
<code>InfixToMathML</code>	Converts an infix expressions to Content MathSBML.
<code>MathMLToInfix</code>	Converts MathML to an infix expression.
<code>parameterToSBML</code>	Returns a <code>&lt; parameter &gt;</code> or <code>&lt; listOfParameters &gt;</code>
<code>reactionToSBML</code>	Returns a <code>&lt; reaction &gt;</code> or <code>&lt; listOfReactions &gt;</code>
<code>ruleToSBML</code>	Returns a rule or <code>a &lt; listOfRules &gt;</code>
<code>speciesToSBML</code>	Returns a <code>&lt; species ... / &gt;</code> or <code>&lt; listOfSpecies .. / &gt;</code> block of SBML.

Corresponding to each of `compartmentToSBML`, `eventToSBML`, `functionToSBML`, `parameterToSBML`, `reactionToSBML`, `ruleToSBML`, and `speciesToSBML` there is a symbolic SBML function with the same interface, `compartmentToSymbolicSBML`, `eventToSymbolicSBML`, `functionToSymbolicSBML`, `parameterToSymbolicSBML`, `reactionToSymbolicSBML`, `ruleToSymbolicSBML`, and `speciesToSymbolicSBML`, which return symbolic XML (`XMLElement` trees) instead of pure text-based XML.

It is anticipated that additional functions will be included in future releases.

```
In[550]:=
  mathSBMLHelpReference[modelBuilder -> True]
```

---

## ■ compartmentToSBML

`compartmentToSBML[options]` returns a `<compartment .../>`  
`compartmentToSBML[{{options},{options},...}]` returns a `<listOfCompartments .../>`  
where each option list corresponds to a single `<compartment .../>` definition.  
Options are the same as valid compartment fields: `id`, `name`, `spatialDimensions`,  
`size`, `untis`, `outside`, `constant`. Default values are set as per the level 2 spec.

Example:

```
compartmentToSBML[id→ "bedrock", size→ "40000"]
```

returns the string:

```
compartment id="bedrock" name="bedrock" size="40000"/>
```

The input:

```
compartmentToSBML[{{id→ "c1",size→ 5},{id→ "c2",  
  spatialDimension→ 2},{id→ "c3",name→ "unsized_compartment_3"}}]
```

returns the string:

```
<listOfCompartments>  
<compartment id="c1"  
  name="c1"  
  size="5"/>  
<compartment id="c2"  
  name="c2"  
  spatialDimension="2"/>  
<compartment id="c3"  
  name="unsized_compartment_3"/>  
</listOfCompartments>
```

---

## ■ compartmentToSymbolicSBML

`compartmentToSymbolicSBML` returns Symbolic SBML for a  
compartment or `listOfCompartments`. Usage is identical to `compartmentToSBML`.

## ■ eventToSBML

eventToSBML[options] returns a single <event.../> definition.  
 eventToSBML[{{options},{options},...}] returns a <listOfEvents.../>, where each option list corresponds to a single event.

### Options:

id→event id, if not provided, defaults to eventn where n is an integer.  
 name→event name, if not provided, defaults to value of event.  
 trigger→expression, required, any Mathematica expression that evaluates to True or False.  
 delay→expression, optional, any expression that evaluates to a number.  
 timeUnits→units, optional units to use for time.  
 eventAssignment→{var<sub>1</sub>→expr<sub>1</sub>, var<sub>2</sub>→expr<sub>2</sub>, ...}, what should happen when the event is triggered, each variable var<sub>i</sub> is assigned to the value of expression expr<sub>i</sub>.

### Example 1: single event:

```
eventToSBML[id→"foo", name→"A basic event",
  timeUnits→ "hours", trigger→ (x>5), eventAssignment→ {x→ y+x, y→ 0}]
returns:
<event id="foo" name="A basic event" timeUnits="hours">
  <trigger>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <gt/>
        <ci>x</ci>
        <cn>5</cn>
      </apply>
    </math>
  </trigger>
  <listOfEventAssignments>
    <eventAssignment variable="x">
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <apply>
          <plus/>
          <ci>x</ci>
          <ci>y</ci>
        </apply>
      </math>
    </eventAssignment>
    <eventAssignment variable="y">
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <cn>0</cn>
      </math>
    </eventAssignment>
  </listOfEventAssignments>
</event>
```

### Example 2: listOfEvents:

```
eventToSBML[{{id→"yon", name→"y turned on", trigger→ x>17, eventAssignment→{y→0, z→
  100}}, {id→"yoff", name→"y turned off", trigger→x>20, eventAssignment→{y→100, z→0}}}]
returns:
<listOfEvents>
  <event id="yon" name="y turned on">
    <trigger>
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <apply>
          <gt/>
          <ci>x</ci>
          <cn>17</cn>
```

```

    </apply>
  </math>
</trigger>
<listOfEventAssignments>
  <eventAssignment variable="y">
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <cn>0</cn>
    </math>
  </eventAssignment>
  <eventAssignment variable="z">
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <cn>100</cn>
    </math>
  </eventAssignment>
</listOfEventAssignments>
</event>
<event id="yoff" name="y turned off">
  <trigger>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <gt/>
        <ci>x</ci>
        <cn>20</cn>
      </apply>
    </math>
  </trigger>
  <listOfEventAssignments>
    <eventAssignment variable="y">
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <cn>100</cn>
      </math>
    </eventAssignment>
    <eventAssignment variable="z">
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <cn>0</cn>
      </math>
    </eventAssignment>
  </listOfEventAssignments>
</event>
</listOfEvents>

```

---

## ■ eventToSymbolicSBML

eventToSymbolicSBML returns Symbolic SBML for  
 an event or a listOfEvents. Usage is identical to eventToSBML.

## ■ functionToSBML

functionToSBML[options] returns an SBML string for a <functionDefinition ... /> definition.

functionToSBML[{{options}}, {options},...]

returns an SBML string for a <listOfFunctionDefinitions ... />

Options:

arguments→x or {x<sub>1</sub>,x<sub>2</sub>,...}, names of the function arguments. At least one argument must be supplied. If no arguments are given a single dummy argument will be generated.

id→string, value of SBML id field; if not specified, a name

functionk, where k is an incrementing integer, will be supplied.

math→expression, Mathematica expression for the function return value in

terms of the arguments. If not specified, the value of 1 will be returned.

name→string, value of SBML name field if not specified, the value of the id field will be used.

Example 1, <functionDefinition ...>:

```
functionToSBML[id→ alfafa, arguments→ {horse, grass}, math→ ( 1/(horse*grass))]
```

returns the string:

```
<functionDefinition id="alfafa"
  name="alfafa">
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <lambda>
    <bvar>
      <ci>horse</ci>
    </bvar>
    <bvar>
      <ci>grass</ci>
    </bvar>
    <apply>
      <times/>
      <cn type="integer">1</cn>
    </apply>
    <power/>
    <apply>
      <times/>
      <ci>grass</ci>
      <ci>horse</ci>
    </apply>
      <cn type="integer">-1</cn>
    </apply>
  </lambda>
</math>
</functionDefinition>
```

Example 2, <listOfFunctionDefinitions ...>

```
functionToSBML[{{id→ QRT,arguments→ {x},
  math→ x^(1/4)},{id→ sinc, arguments→ {x}, math→ Sin[x]/x}}]
```

returns the string:

```
<listOfFunctionDefinitions>
<functionDefinition id="QRT"
  name="QRT">
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <lambda>
    <bvar>
```

```

      <ci>x</ci>
    </bvar>
  <apply>
    <power/>
    <ci>x</ci>
    <cn type="rational">1<sep/>4</cn>
  </apply>
</lambda>
</math>
</functionDefinition>
<functionDefinition id="sinc"
  name="sinc">
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <lambda>
    <bvar>
      <ci>x</ci>
    </bvar>
    <apply>
      <times/>
      <apply>
        <sin/>
        <ci>x</ci>
      </apply>
    </apply>
    <power/>
    <ci>x</ci>
    <cn type="integer">-1</cn>
  </apply>
</lambda>
</math>
</functionDefinition>
</listOfFunctionDefinitions>

```

---

## ■ functionToSymbolicSBML

functionToSymbolicSBML returns Symbolic SBML for a function or listOfFunctions. Usage is identical to functionToSBML.

---

## ■ InfixToMathML

InfixToMathML[expression\_] returns a MathML string representing the given Mathematica expression.

---

## ■ MathMLToInfix

MathMLToInfix[string\_] converts a MathML content string into an infix expression.

---

## ■ newModel

`newModel[]` resets various counters and variables used by the MathSBML Model Builder.  
Options include:  
`id`->string, model id  
`name`->string, model name.

---

## ■ parameterToSBML

`parameterToSBML[options]` returns a `<parameter.../>` definition.  
`parameterToSBML[{{options},{options},...}]` returns a `<listOfParameters.../>`  
where each option list corresponds to a single `<parameter.../>` definition.  
Options are identical to those for a parameter: `id`, `name`, `value`, `units`, `constant`.

Example:

```
parameterToSBML[id-> "Pi", name-> "pi", value-> 3.14]
```

returns

```
<parameter id="Pi" name="pi" value="3.14"/>
```

```
parameterToSBML[{{id->"pi", value->3.14}, {id->"e",value->
  2.718}, {id->"E",name->"Enzyme_Concentration",constant->"false",value->1}}]
```

returns

```
<listOfParameters>
<parameter id="pi"
  name="pi"
  value="3.14"/>
<parameter id="e"
  name="e"
  value="2.718"/>
<parameter id="E"
  name="Enzyme_Concentration"
  value="1"
  constant="false"/>
</listOfParameters>
```

---

## ■ parameterToSymbolicSBML

`parameterToSymbolicSBML` returns Symbolic SBML for a  
parameter or `listOfParameters`. Usage is identical to `parameterToSBML`.

## ■ reactionToSBML

reactionToSBML[options] returns a fragment of SBML (not an entire SBML model) corresponding to a single reaction. Any combination of options is allowed.

### Available Options:

reaction→  $(\sum e_i r_i \rightarrow \sum s_i p_i)$ , where  $r_i$  and  $p_i$  are names of reactants and products; and  $e_i$  and  $s_i$  are the corresponding stoichiometry expressions. The stoichiometry expressions must be either a number or have the form Stoichiometry[expr] where expr is any expression. If the option reaction is used then the options reactants, products, reactantStoichiometry, and productStoichiometry are ignored.

id → string - required reaction identifier, no default

name → string - reaction name.

reactants → {} (none, default), r, or {r<sub>1</sub>, r<sub>2</sub>, ...}, names of reactants. This option is ignored if the option reaction is used.

products → {} (none, default), p, or {p<sub>1</sub>, p<sub>2</sub>, ...}, names of products. This option is ignored if the option reaction is used.

modifiers → {} (none, default), m, or {m<sub>1</sub>, m<sub>2</sub>, ...}, names of modifiers

reactantStoichiometry → integer, expression, {e<sub>1</sub>, e<sub>2</sub>, ...}, where e<sub>j</sub> is any expression. The stoichiometries are listed in the same order as the names of the reactants. If fewer stoichiometries then reactants are specified the remaining reactants are assigned a stoichiometry of 1. This option is ignored if the option reaction is used.

productStoichiometry → integer, expression, {e<sub>1</sub>, e<sub>2</sub>, ...}, where e<sub>j</sub> is any expression. The stoichiometries are listed in the same order as the names of the products. If fewer stoichiometries then products are specified the remaining products are assigned a stoichiometry of 1. This option is ignored if the option reaction is used.

kineticLaw → any expression (no default); if no kinetic law is specified, none is included in the SBML

parameters → {i<sub>1</sub>→o<sub>1</sub>, i<sub>2</sub>→o<sub>3</sub>, ...}, where i<sub>1</sub>, i<sub>2</sub>, ... are the ids of the parameters, and o<sub>1</sub>, o<sub>2</sub>, ... are option lists of the form {name→v<sub>1</sub>, value→v<sub>2</sub>, units→v<sub>3</sub>, constant→v<sub>4</sub>}, or any subset thereof, where v<sub>1</sub>, etc., are the values of the name, value, units and constant field.

reversible → True

fast → False

### Example:

```
reactionToSBML[reaction→ (A+2B→ C),kineticLaw→
  k*A*B,parameters→ {k→ {value→ 12,name→ "Rate Constant"}}]
```

```
<reaction id="reaction2"
  name="reaction2"
  reversible="true"
  fast="false">
<listOfReactants>
<speciesReference species="A"/>
<speciesReference species="B">
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <cn type="integer">2</cn>
  </math>
</speciesReference>
</listOfReactants>
<listOfProducts>
<speciesReference species="C"/>
</listOfProducts>
<listOfModifiers/>
<kineticLaw timeUnits="time"
  substanceUnits="substance">
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <apply>
      <times/>
      <ci>A</ci>
```

```
    <ci>B</ci>
    <ci>k</ci>
  </apply>
</math>
<listOfParameters>
  <parameter id="k"
    value="12"
    name="Rate Constant"/>
</listOfParameters>
</kineticLaw>
</reaction>
```

---

## ■ reactionToSymbolicSBML

reactionToSymbolicSBML returns Symbolic SBML for a reaction or listOfReactions. Usage is identical to reactionToSBML.

---

## ■ ruleToSBML

ruleToSBML[options] returns a rule definition.  
 ruleToSBML[{{options},{options},...}] returns a <listOfRules.../>, where each option list corresponds to a separate rule.

Options:  
 type->algebraicRule,assignmentRule,or rateRule (no default)  
 variable->name of variable (variable n) if not specified.  
 math->formula for the rule.

Example 1: ruleToSBML[type-> assignmentRule, variable-> x, math-> x+y]

returns:  
 <assignmentRule variable="x">  
 <math xmlns="http://www.w3.org/1998/Math/MathML">  
 <apply>  
 <plus/>  
 <ci>x</ci>  
 <ci>y</ci>  
 </apply>  
 </math>  
 </assignmentRule>

Example2:

ruleToSBML[{{type->"algebraicRule",math->x^2}, {type->"rateRule",variable->y,math->2\*x}}]  
 returns:

<listOfRules>  
 <algebraicRule>  
 <math xmlns="http://www.w3.org/1998/Math/MathML">  
 <apply>  
 <power/>  
 <ci>x</ci>  
 <cn type="integer">2</cn>  
 </apply>  
 </math>  
 </algebraicRule>  
 <rateRule variable="y">  
 <math xmlns="http://www.w3.org/1998/Math/MathML">  
 <apply>  
 <times/>  
 <cn type="integer">2</cn>  
 <ci>x</ci>  
 </apply>  
 </math>  
 </rateRule>  
 </listOfRules>

---

## ■ ruleToSymbolicSBML

ruleToSymbolicSBML returns Symbolic SBML  
 for a rule or listOfRules. Usage is identical to ruleToSBML.

---

## ■ speciesToSBML

speciesToSBML[options] returns an SBML <species ... /> block for a single species  
 speciesToSBML[{option-list-1, option-list-2,...}] returns an SBML <listOfSpecies .../> block for a set of species described by the option lists.  
 Options have the same names as the SBML fields for species: id, name, compartment, initialAmount, initialConcentration, units, boundaryCondition, charge, constant

Example 1 (single species):

```
speciesToSBML[id→ "fred", name→ "Fred
  Flintstone", compartment→ "bedrock" , boundaryCondition→ "false",
  constant→ "true", initialConcentration→5, units→ "rocks" , charge→17]
```

returns the string

```
<species id="fred" name="Fred
  Flintstone" compartment="bedrock" boundaryCondition="false" constant="true" initialConcentration
  ="5" units="rocks" charge="17"/>
```

Example 2 (multiple species):

```
speciesToSBML[{{id→ "fred", compartment→ "bedrock"}, {id→ "barney",compartment→ "bedrock"}}]
```

returns the string

```
<listOfSpecies>
<species id="fred"
  name="fred"
  compartment="bedrock"
  boundaryCondition="false"
  constant="false"/>
<species id="barney"
  name="barney"
  compartment="bedrock"
  boundaryCondition="false"
  constant="false"/>
</listOfSpecies>
```

---

## ■ speciesToSymbolicSBML

speciesToSymbolicSBML returns Symbolic SBML for  
 a species or listOfSpecies. Usage is identical to speciesToSBML.

---

## ■ unitToSBML

unitToSBML[options] returns an SBML <unitDefinition .../>.  
 unitToSBML[{{options},{options},...}] returns a <listOfUnitDefinitions ...>  
 where each sub-option list corresponds to the option list of the first form.

Options:

id->no default, value of id field of unitDefiniton,  
 if not specified, supplied by program as unitn, where n is an integer  
 name->no default, if not supplied, value of id is used  
 unit->{}, contains an option list of the form "string"->{scale->0,  
 exponent->1, multiplier->1, offset->1}, where "string" is a quote-delimited  
 string indicated the value of the kind field of the <unit ...> expression.

Example:

```
unitToSBML[id-> "mmlh",name-> "millimoles_per_liter_per_hour",unit-> {"mole"->
  {scale-> -3}, "litre"-> {exponent-> -1}, "second"-> {scale-> 3600, exponent-> -1}}]
```

returns the string

```
<unitDefinition id="mmlh"
  name="millimoles_per_liter_per_hour">
<listOfUnits>
  <unit kind="mole"
    scale="-3"/>
  <unit kind="litre"
    exponent="-1"/>
  <unit kind="second"
    exponent="-1"
    scale="3600"/>
</listOfUnits>
</unitDefinition>
```

```
unitToSBML[{{id->"mmlh",name->"millimoles_per_liter_per_hour",
  unit->{"mole"->{scale->-3},"litre"->{exponent->-1},"second"->{scale->3600,
  exponent->-1}}},{id->"fred",unit->{"ampere"->{exponent->-1},"ohm"->{scale->4}}}]
```

returns

```
<listOfUnitDefinitions>
<unitDefinition id="mmlh"
  name="millimoles_per_liter_per_hour">
<listOfUnits>
  <unit kind="mole"
    scale="-3"/>
  <unit kind="litre"
    exponent="-1"/>
  <unit kind="second"
    exponent="-1"
    scale="3600"/>
</listOfUnits>
</unitDefinition>
<unitDefinition id="fred"
  name="fred">
<listOfUnits>
  <unit kind="ampere"
    exponent="-1"/>
  <unit kind="ohm"
    scale="4"/>
</listOfUnits>
</unitDefinition>
</listOfUnitDefinitions>
```

---

## ■ unitToSymbolicSBML

`unitToSymbolicSBML` returns Symbolic SBML  
for a unit or a list of units. Usage is identical to `unitToSBML`.

## References

[SBML Level 2 Spec] Andrew Finney, Michael Hucka (2003) Systems Biology Markup Language (SBML) Level 2: Structures and Facilities for Model Definitions, SBML Level 2, Version 1 (Final Draft) June 4, 2003

[SBML Level 1 Version 2 Spec] Michael Hucka, Andrew Finney, Herbert Sauro, Hamid Bolouri, Systems Biology Markup Language (SBML) Level 1: Structures and Facilities for Basic Model Definitions, SBML Level 1, Version 2 (Final Draft) 8 May 2003

[SBML Level 1 Version 1 Spec] Michael Hucka, Andrew Finney, Herbert Sauro, Hamid Bolouri, Systems Biology Markup Language (SBML) Level 1: Structures and Facilities for Basic Model Definitions, SBML Level 1, Version 1 (Final), 2 March 2001

[Hucka 2003] Hucka M, Finney A, Sauro HM, Bolouri H, Doyle JC, Kitano H, Arkin AP, Bornstein BJ, Bray D, Cornish-Bowden A, Cuellar AA, Dronov S, Gilles ED, Ginkel M, Gor V, Goryanin II, Hedley WJ, Hodgman TC, Hofmeyr JH, Hunter PJ, Juty NS, Kasberger JL, Kremling A, Kummer U, Le Novere N, Loew LM, Lucio D, P. Mendes P, Mjolsness ED, Nakayama Y, Nelson MR, Nielsen PF, Sakurada T, Schaff JC, Shapiro BE, Shimizu S, Spence HD, Stelling J, Takahashi K, M. Tomita M, Wagner J, Wang J (2003) The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics* 19 (4):513-523

[Wolfram 2003] Stephen Wolfram, *The Mathematica Book*, 5th edition, Wolfram Media (2003).