

Design
Explorer and
Viewer for
Iterative
SBML
Enhancement of
Representations

VERSION 1.0 (November 2015)

**Manual for creating SBML
Level 3 packages**

SBML Team

Contents

1	Getting started	5
1.1	Functionality.....	5
1.2	Prerequisites	6
1.2.1	Source code for libSBML	6
1.2.2	Integration and testing	6
1.2.3	Basic TeX files.....	7
1.2.4	Basic documentation	7
1.2.5	UML diagrams	7
1.2.6	Available bundles	8
1.2.7	Useful links.....	8
1.3	Setting up the Deviser Edit tool	8
2	Defining an SBML Level 3 Package.....	10
2.1	Define the general package information.....	10
2.2	Add the version number	13
2.3	Add class information.....	14
2.3.1	An SBML ‘element’ or ‘class’	14
2.3.2	General class description	15
2.3.3	Adding attributes and child elements	17
2.3.4	A note on repeated information	23
2.3.5	Example 1 - Adding a class with no containing ListOf.....	24
2.3.6	Example 2 - Adding a class with a containing ListOf.....	25
2.3.7	Example 3 – Adding a base class and derived classes.....	28
2.4	Add plugin information	33
2.4.1	What is a plugin ?	33
2.4.2	General plugin information.....	34
2.4.3	Example 4 – Extending a core element	35
2.4.4	Example 5 – Extending a core element with attributes only	36
2.4.5	Example 6 – Extending a non-core element	37
2.5	Add enum information	37
2.5.1	Example 7 – Adding an enumeration	37
2.6	Mappings	40
2.7	Results.....	41
2.7.1	Validating the description.....	42
3	Using Deviser	44

3.1	View UML diagrams.....	44
3.2	Generate libSBML package code.....	46
3.3	Generate basic specification documentation.....	47
3.4	Integrate and test the package with libSBML.	47
3.5	Command line.....	48
4	Appendix A	49
5	Appendix B	51
6	References	51

Figure 1	The 'Preferences' sheet - adding information about other required software.	8
Figure 2	The 'Package' sheet - Illustrating the first step in defining the 'foo' package.	10
Figure 3:	The Numbers and Offsets information box	11
Figure 4	The 'requires additional code' check box.	12
Figure 5	The 'Version' sheet	13
Figure 6	Snapshot of part of libSBML class hierarchy	15
Figure 7	The 'Class' sheet	16
Figure 8:	Attributes of type 'array' and 'enum'	20
Figure 9:	UML diagram of CSGTransformation from SBML L3 'spatial' package specification	22
Figure 10:	DeviserEdit description of CSGTransformation	23
Figure 11	Defining the FooKineticLaw class.....	24
Figure 12	Defining the FooParameter class.	25
Figure 13:	The UML diagram produced by Deviser Edit following the definition of package 'Foo' in	27
Figure 14	Defining the base class 'FooRule'.	28
Figure 15	Defining the Algebraic class	29
Figure 16:	Defining the Assignment class.	30
Figure 17:	The UML diagram from DeviserEdit of the ListOfFooRules class	32
Figure 18	libSBML class hierarchy showing 'plugins' to the Model class	33
Figure 19	The 'Plugin' sheet.	34
Figure 20	Defining the extension of SBML Level 3 Core Reaction by package foo.	35
Figure 21	Defining the extension of SBML Level 3 Core Model by package foo.....	36
Figure 22	Defining the extension of SBML Level 3 Qual Transition by package foo.....	37
Figure 23	Defining the Extra class which has an attribute of type enum.....	38
Figure 24	Defining the Sign enumeration.	39
Figure 25	Identifying the origin of classes from other packages	40
Figure 26	The complete description of the foo package	41
Figure 27	Validating the package description	42
Figure 28:	The UML window	44
Figure 29:	The yUML text representation of the diagram	45
Figure 30	The Generate window	46

Figure 31 Integration and Testing tab selected on the Generate window.	47
SBML snippet 1: The <sbml> element declaring use of the package 'foo'	11
SBML snippet 2: SBML representation of a Reaction object	14
SBML snippet 3: SBML Spatial using a text element that is an array of numbers....	19
SBML snippet 4: The kineticLaw element as reflecting the definition of package Foo in Examples 1 and 2	26
SBML snippet 5: The XML output for the ListOfFooRules defined in Example 3	31
Code example 1: Functions for manipulating attributes with simple types	18
Code example 2: Functions produced for child element of 'array' Type	19
Code example 3: Function produced for an attribute of type 'enum'	20
Table 1: The 'element', 'lo_element' and 'inline_lo_element' types	21
Table 2: Expected values for the Element field based on attribute Type	21

1 Getting started

Deviser allows a user to define an SBML L3 package and produce libSBML code which can be overlaid onto the existing libSBML source tree to add support for the package.

Deviser can also use the package description to create a basic specification files for the package based on the SBML Level 3 package class specification.

Both creating code or tex files can be done via a command line interface.

The Deviser Edit tool provides a GUI to facilitate defining the necessary information; as well as a convenient way of invoking the functionality. It also provides some additional functionality (view UML, create PDF, integrate and test code) to aid the process of definition and testing.

1.1 Functionality

The Deviser Edit tool outputs an XML [1] definition of the package. This definition is then used by Deviser code to generate the requested files. Samples are included with the source code and the full text created by the examples used in this manual is given in [Appendix A](#).

There is command line version of the function that can be used to invoke the deviser functionality on the XML file. This is the `deviser.py` file found in the generator directory.

```
deviser.py [--generate][--latex] input-file.xml
```

This program takes as input a Deviser XML file and either

- `--generate (-g)` generates the libSBML code for the package or
- `--latex (-l)` generates a LaTeX scaffold for its specification.

The basic Deviser Edit tool provides a number of forms that allow a user to define an SBML L3 package. Once defined the following functions are available.

1. Generate the necessary libSBML [2] code for the package.
2. Integrate and test the package with libSBML.
3. Generate TeX files [3] for basic specification of the package.
4. Generate a pdf of a basic specification document for the package.
5. Create and view a UML [4] diagram.

Deviser is not a standalone package and other software may be necessary in order to access certain functionality. The main functionality, that is generating code files, is invoked using a Python Interpreter which will be necessary when using Deviser via the command line.

1.2 Prerequisites

Note that since this is a beta release of Deviser we are not bundling all of the dependencies. Future releases will hope to reduce the prerequisites.

TO DO: What about this release ?

1.2.1 Source code for libSBML

In order to generate code for libSBML a Python Interpreter is necessary. Our code has been tested on Python 2.7, 3.3 and 3.4.

- Python Interpreter

1.2.2 Integration and testing

Deviser Edit does allow you to automatically integrate and test your newly created code with libSBML. In order to do this you will need to have CMake (<http://www.cmake.org/>), SWIG (<http://www.swig.org/>) and a C++ compiler available. In addition you will require the libSBML source code and the source code for the libSBML dependencies.

- Python Interpreter
- CMake
- SWIG
- C++ compiler
- libSBML source code
- libSBML dependencies source code

1.2.3 Basic TeX files

Deviser will generate the necessary TeX files with only a Python Interpreter present.

- Python Interpreter

1.2.4 Basic documentation

In order to create a PDF it will be necessary to have access to a version of pdflatex. On Windows having MiKTeX (<http://miktex.org/>) installed provides this functionality. The PDF generation also requires the SBML documentation class files sbmlpkgspec (<https://sourceforge.net/projects/sbml/files/specifications/tex/>).

- Python Interpreter
- pdflatex (MiKTeX on Windows OS)
- sbmlpkgspec

Note on a standard Linux OS we found it necessary to install the following packages:

- xzdec
- texlive-latex-base
- texlive-latex-extra
- texlive-fonts-extra

and run the following from the command line

- tlmgr init-usertree
- tlmgr install bbding
- tlmgr install fourier

TO DO: Add anything Mac specific

1.2.5 UML diagrams

Deviser allows you to create and view very basic UML diagrams based on the classes specified. It uses the free yUML (<http://yuml.me/>) web service. Thus it will be necessary to be connected to the internet to create UML diagrams.

- Internet connection

1.2.6 Available bundles

TO DO: What are we releasing this time ?

1.2.7 Useful links

1. libSBML source code (latest release):
<https://sourceforge.net/projects/sbml/files/libsbml/5.11.4/stable/libSBML-5.11.4-core-src.tar.gz/download>
2. libSBML source code (latest code):
<https://sourceforge.net/p/sbml/code/HEAD/tree/trunk/libsbml/>
3. libSBML dependencies (for Windows users):
<https://github.com/sbmlteam/libSBML-dependencies>
4. SBML package specification template files:
<https://sourceforge.net/projects/sbml/files/specifications/tex/sbmlpkgspec-1.6.0.tar.gz/download>

1.3 Setting up the Deviser Edit tool

In order to access the functionality for generation it is necessary to tell the Deviser Edit tool where it will find things on your system.

Select Edit->Edit Preferences (Preferences on Mac OSX)

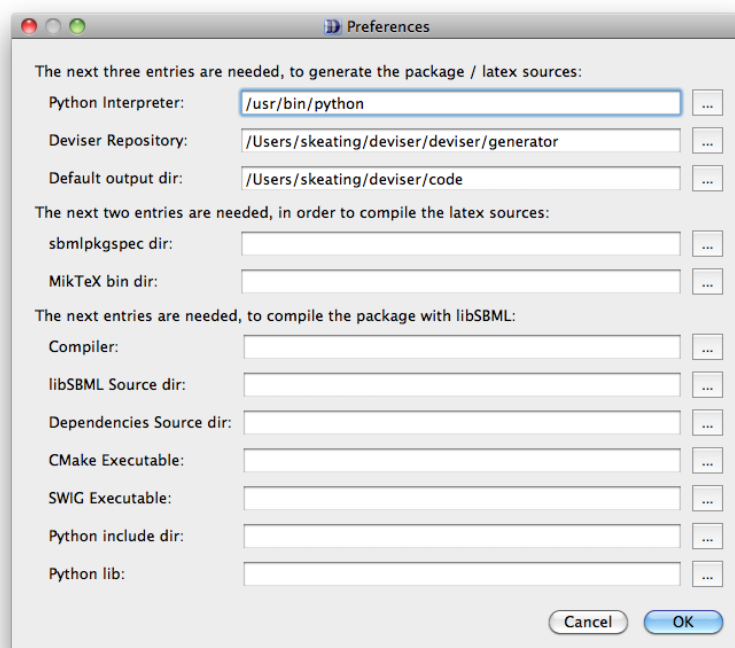


Figure 1 The 'Preferences' sheet - adding information about other required software.

Fill in or browse to the location of the files requested by each field. Note it is not necessary to fill in all the fields if you are not intending to use all the functionality. [Figure 1](#) illustrates a case where you could generate code and TeX files but not generate a PDF or integrate the code automatically.

Without any of this information the Deviser Edit tool will save the XML description and create UML diagrams.

The fields requested are:

Python Interpreter. This is location of the python executable. This is necessary to invoke any functionality of Deviser.

Deviser repository. This is the location of the deviser/generator directory. This will have been bundled with an installer but may also be obtained directly from our GitHub repository [<https://github.com/sbmlteam/deviser>].

Default output dir. The location where generated files should be written.

sbmlpkgspec dir. The location of the SBML documentation class files sbmlpkgspec.

MikTeX Bin dir. The location of the MikTeX executables.

Compiler. The location of the C++ compiler to be used when building libSBML with integrated package code.

libSBML Source dir. The location of the libSBML source tree. This should be the top-level libSBML directory.

Dependencies Source dir. The location of the libSBML dependencies source code. Note that particularly on Windows it is necessary for the libSBML dependencies to be built with the same Compiler as that to be used to build libSBML. Deviser Edit allows a user to specify the location of the source code for the dependencies and offers an option to build them if this should be necessary.

CMake executable. The location of the CMake executable. Deviser creates the necessary CMake files to allow package code to be integrated. This will be necessary if you intend to use Deviser Edit to integrate and build your code. Note Deviser does not support any other build system.

SWIG executable. The location of the SWIG executable. Since Deviser uses a python interpreter to create code, the integration and testing will create the Python binding of libSBML. SWIG is necessary for this.

Python include dir: The location of the python include files. This is necessary to build and test the Python binding of libSBML.

Python lib. The location of the Python library files. This is necessary to build and test the Python binding of libSBML.

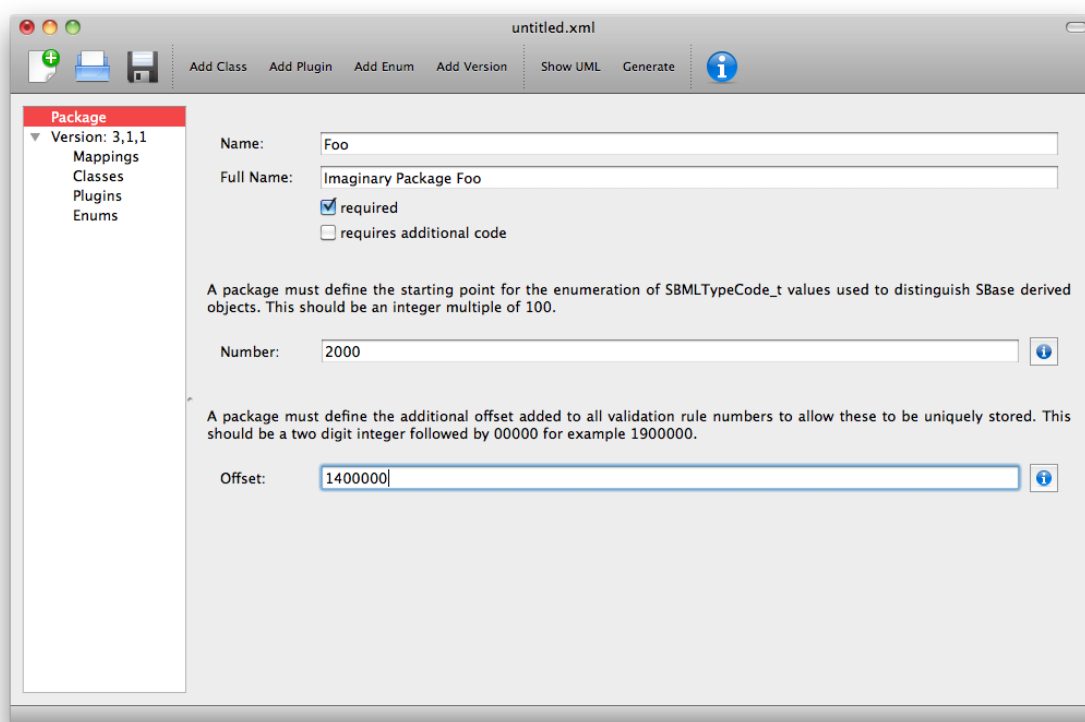
2 Defining an SBML Level 3 Package

SBML Level 3 is the most recent specification of SBML. It is a modular language, with a core comprising a complete format that stands alone [5]. Level 3 packages may be added to this core to provide additional, optional features. Deviser provides a way of defining the package that can then be used by generation code to create code for libSBML and text for specifications.

It is necessary to define the structure of the SBML Level 3 package before invoking other functionality available within Deviser. A series of sheets guide you through the process in what we hope is an intuitive manner. For the purpose of this manual we will work step by step through creating an imaginary package 'Foo'. Note the completed file is available as part of the Deviser code in the 'samples' folder and can be opened using the Deviser Edit tool.

2.1 Define the general package information

Start the Deviser Edit tool and select 'Package' from the tree on the left hand side.



The screenshot shows the 'Package' sheet in the Deviser Edit tool. The left sidebar contains a tree view with 'Package' selected, and sub-items: 'Version: 3,1,1', 'Mappings', 'Classes', 'Plugins', and 'Enums'. The main area has the following fields and options:

- Name:** Foo
- Full Name:** Imaginary Package Foo
- ☒ required
- ☐ requires additional code
- Number:** 2000
- Offset:** 1400000

Below the 'Number' field, there is a text box with the instruction: "A package must define the starting point for the enumeration of SBMLTypeCode_t values used to distinguish SBase derived objects. This should be an integer multiple of 100."

Below the 'Offset' field, there is a text box with the instruction: "A package must define the additional offset added to all validation rule numbers to allow these to be uniquely stored. This should be a two digit integer followed by 00000 for example 1900000."

Figure 2 The 'Package' sheet - Illustrating the first step in defining the 'foo' package.

The **Name** field is the short name that will be used as a prefix for the package e.g. 'foo'.

The **Full Name** field is the name that will be used to refer to the package in documentation e.g. 'Imaginary Package Foo'.

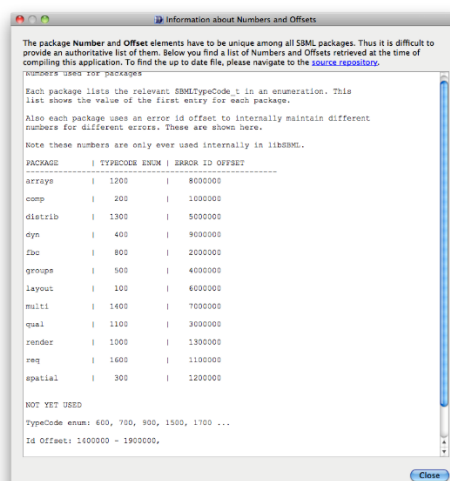
The **required** checkbox is used to indicate whether the package may change the mathematical interpretation of the core model and corresponds to the required attribute on the <sbml> element declaring this package (see [SBML snippet 1](#)).

```
<sbml xmlns=http://www.sbml.org/sbml/level3/version1/core  
      xmlns:foo=http://www.sbml.org/sbml/level3/version1/foo/version1  
      level="3" version="1" foo:required="true">
```

SBML snippet 1: The <sbml> element declaring use of the package 'foo'

The **Number** field is the starting point for the enumeration of the typecodes for this package. Pressing the information button will pop-up with information regarding the Number and Offset values used by existing L3 packages ([Figure 3](#)).

The **Offset** field is the number added to the validation rules given in the specification to allow this to be identified uniquely in code. Pressing the information button will pop-up with information regarding the Number and Offset values used by existing L3 packages.



The package Number and Offset elements have to be unique among all SBML packages. Thus it is difficult to provide an authoritative list of them. Below you find a list of Numbers and Offsets retrieved at the time of compiling this application. To find the up to date file, please navigate to the [source repository](#).

Numbers used for packages

Each package lists the relevant SBMLTypeCode_t in an enumeration. This list shows the value of the first entry for each package.

Also each package uses an error id offset to internally maintain different numbers for different errors. These are shown here.

Note these numbers are only ever used internally in libSBML.

PACKAGE	TYPECODE ENUM	ERROR ID OFFSET
arrays	1200	8000000
comp	200	1000000
distrib	1300	9000000
dyn	400	9000000
fnv	800	2000000
groups	500	4000000
layout	100	6000000
multi	1400	7000000
qual	1100	3000000
render	1000	1300000
req	1600	1100000
spatial	300	1200000

NOT YET USED

TypeCode enum: 600, 700, 900, 1500, 1700 ...

Id Offset: 1400000 - 1900000

Close

Figure 3: The Numbers and Offsets information box

The **requires additional code** checkbox can be used to indicate that there is additional code that will be required by the package that will not be generated by Deviser. Checking the 'requires additional code' box reveals further boxes that can be used to specify the location of the additional code files. Deviser will incorporate this code 'as-is'.

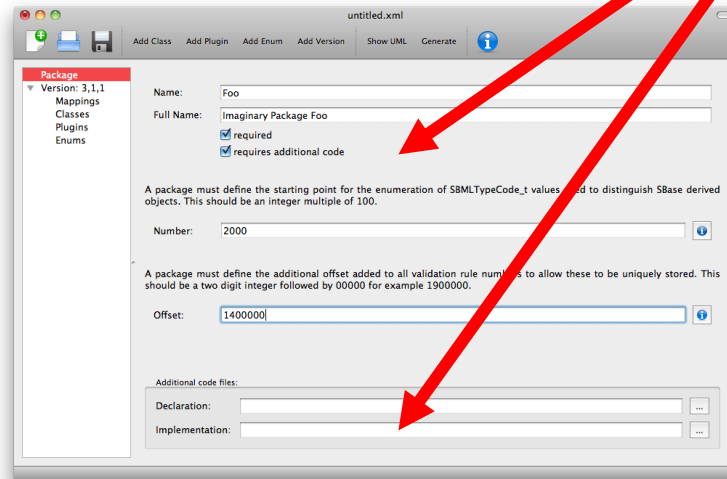


Figure 4 The 'requires additional code' check box.

Note this facility allows a user to include code for classes that are not defined in Deviser and perhaps do not follow the usual libSBML conventions for classes. When generating the code Deviser will merely copy files listed here into the sbml directory for the package.

2.2 Add the version number

Highlight 'Version' in the tree on the left hand side.

Fill in the core level and version and the package version numbers.

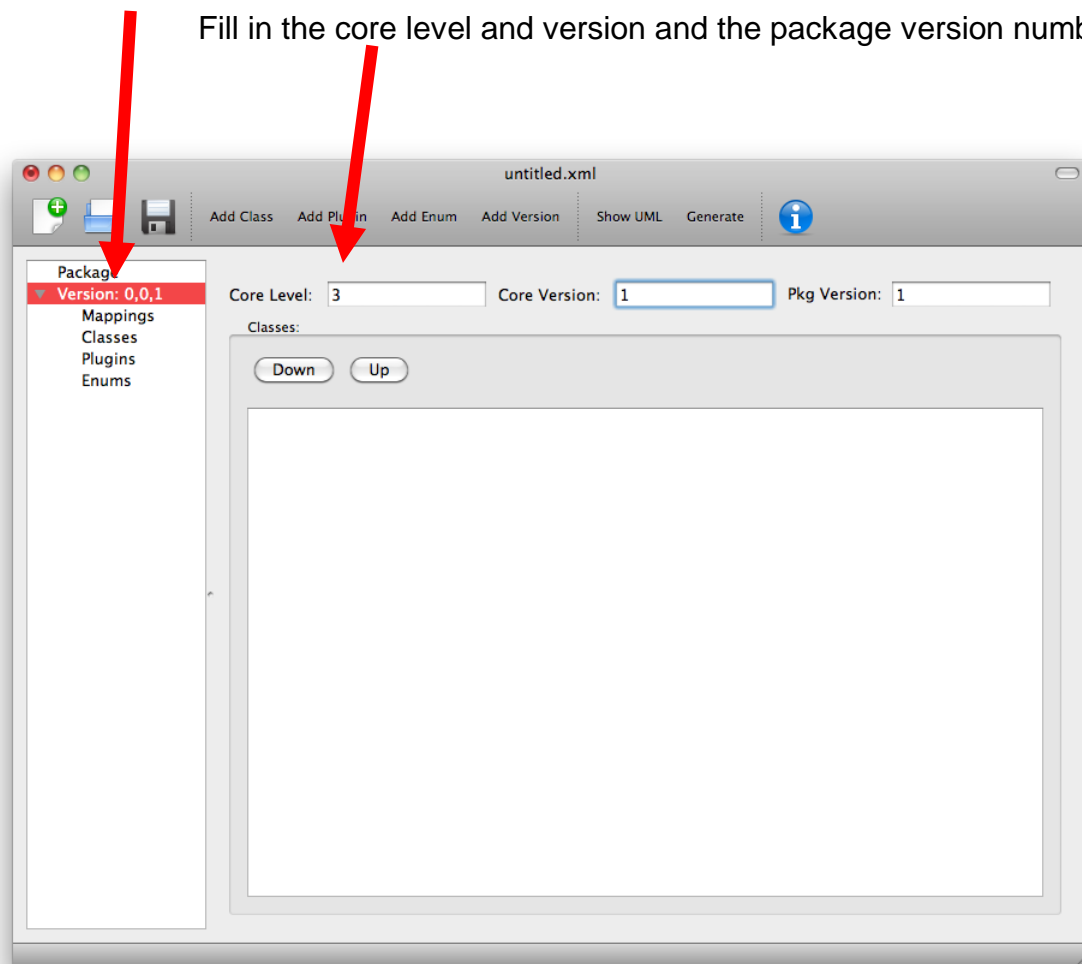


Figure 5 The 'Version' sheet

Note once classes have been specified they will be listed on this sheet (see [Figure 26](#)) and the order in which they are listed can be changed. This order dictates the order in which the generation code processes the classes. This can be useful in ensuring documentation is written out in a specific order.

The tree in the left hand panel shows the aspects of the package that can now be added i.e. Classes, Plugins and Enums. The Mappings sheet will be automatically populated when the description is complete. We shall work through the different aspects.

2.3 Add class information

This section describes how to specify a class. The first subsection gives a brief overview of what is meant by a ‘class’. The next two subsections give an overview of the information that needs to be provided and then we work through several examples.

2.3.1 An SBML ‘element’ or ‘class’

In SBML XML elements are used to capture the information relating to particular objects by means of attributes to specify characteristics of the element and where necessary child elements to provide further information. SBML generally uses an enclosing `listOf` element to group elements of the same type together. The names of attributes and elements are chosen to be intuitive and libSBML mimics these names and structure in its class definitions and API. This is illustrated in the figures below. We use class to mean the description of an XML element. In object-oriented programming languages (such as C++ or Java), this is represented as a class.

```
<listOfReactions>
  <reaction id="reaction_1" reversible="false" fast="false">
    <listOfReactants>
      <speciesReference species="X0" constant="true"/>
    </listOfReactants>
    <listOfProducts>
      <speciesReference species="S1" constant="true"/>
    </listOfProducts>
    <kineticLaw>
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <apply>
          <times/>
          <ci> K0 </ci>
          <ci> X0 </ci>
        </apply>
      </math>
    </kineticLaw>
  </reaction>
</listOfReactions>
```

SBML snippet 2: SBML representation of a Reaction object

Figure 6 shows a snapshot of libSBML class hierarchy corresponding to [SBML snippet 2](#). Note the correspondence of names and the getXYZ functions etc.

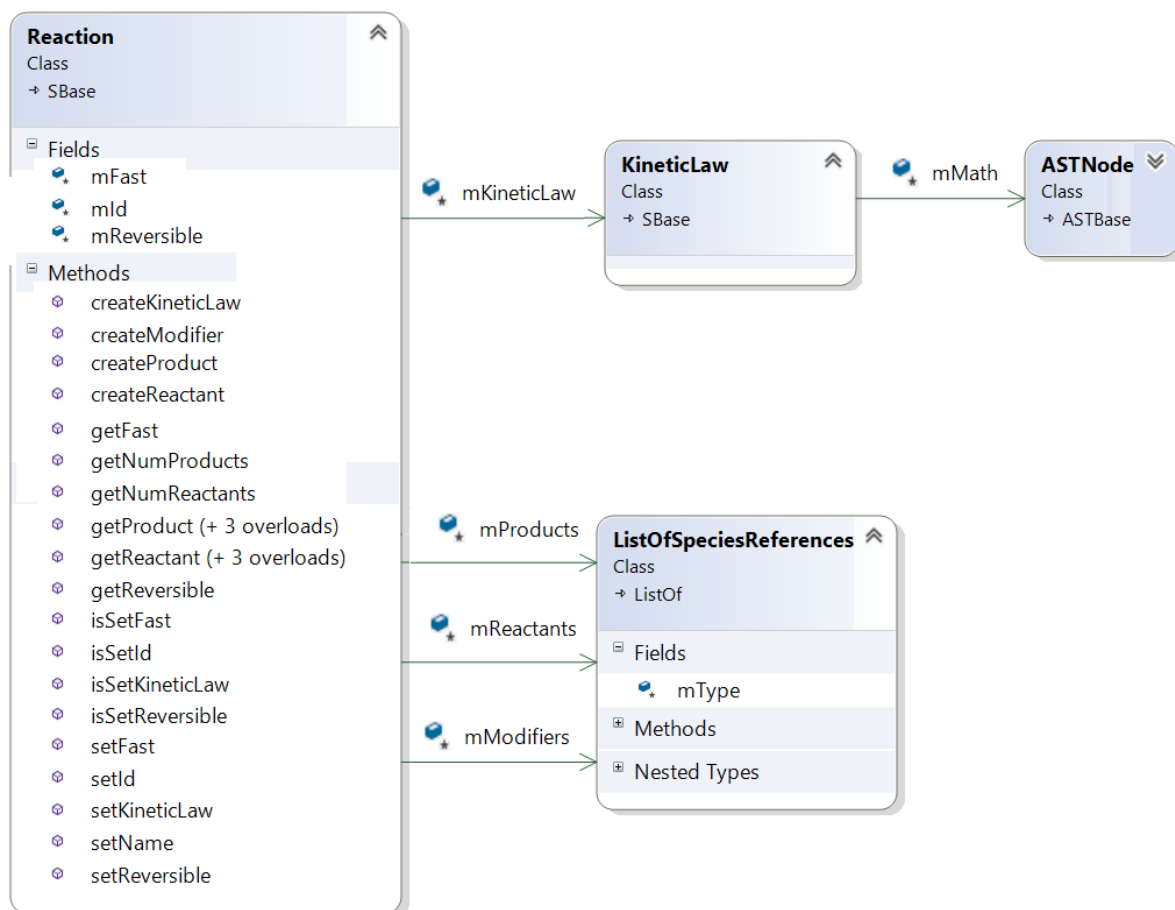


Figure 6 Snapshot of part of libSBML class hierarchy

2.3.2 General class description

We use class to mean the description of an XML element. You will need to specify the details for every new XML element that the package defines including classes that are abstract and/or used as base classes for other classes. You may find yourself repeating information but this is necessary to facilitate creating a valid definition that the auto-generation code can work with (see section [2.3.4](#)).

Select 'Add Class' from the toolbar or the 'Edit' menu.

untitled.xml

Add Class Add Plugin Add Enum Add Version Show UML Generate

Package
▼ Version: 3.1.1
Mappings
► Classes
Plugins
Enums

Name: class_0 BaseClass: XML ElementName: TypeCode: hasListOf isBaseClass requires additional code

Class attributes and child elements:

Name	Type	Element	Required	isBaseClass	XML name
------	------	---------	----------	-------------	----------

Figure 7 The 'Class' sheet

NOTE: Using the 'wand' button will populate the adjacent field with the value of the field that conforms most closely to SBML and libSBML conventions.

The **Name** field is the name of the class that will be used by the code generation (see XML ElementName below). This field is required and must be populated.

The **BaseClass** field gives a base class if this class derives from a base. Clicking the wand will populate the field with 'SBase', as this is the most common base class for libSBML classes. Note this field is a required field and leaving it blank implies that there is no base class for the class being specified.

The **TypeCode** field is a value that will be used in an enumeration of the types for this package. Clicking on the wand populates it with SBML_PACKAGE_CLASS where PACKAGE is the short package name given and CLASS is the name field for this class. This field is also required and cannot be left blank. You must populate it.

The **XML ElementName** is an optional field that can be used to specify the name of the element as it will appear in the XML output. This defaults to the class name with a lowercase first letter. An example of where this might be different from the default is if two packages use the same class name and it is necessary to distinguish between these in code. The example in [Figure 11](#) shows a case where we have reused the class 'KineticLaw' within our package foo and indicate that code should generate a class named FooKineticLaw but that text and the XML output should use 'kineticLaw' as the name of the element.

The **hasListOf** checkbox is used to indicate whether the element has a parent ListOf class. In SBML it is common for elements 'bar' to occur within a list of element 'listOfBars'. However some elements may occur without a containing ListOf. If this checkbox is selected code will also be generated for a ListOfXYZ class corresponding to the class being described.

The **isBaseClass** checkbox is used to indicate that the class being defined is in fact a base class for other classes within the specification.

The **requires additional code** checkbox can be used to indicate that there is additional code that will be required by the package that will not be generated by Deviser. Checking the 'requires additional code' box reveals further boxes that can be used to specify the location of the additional code files. Deviser will incorporate this code 'as-is'. Given here the additional code would expect to be part included within the class being generated. A case where this is useful is where the class may take data that might be compressed and the additional code files can be used to provide the functions to compress and decompress the data.

The **Class attributes and child elements** table is used to specify each attribute and child element for the class. These are added and deleted using the '+' and '-' buttons to the left of this table.

2.3.3 Adding attributes and child elements

Here we expand on the fields in the **Class attributes and child elements** table for a class as shown in [Figure 7](#).

The **Name** field gives the name of the attribute or child element. In the rare cases where this Name is not an exact match with the name that will appear in the XML the 'XML name' field can be used to override.

The **Type** field gives the type of the attribute or child.

Note the type used here allows the underlying Deviser auto-generation code to determine which data type should be used in code to represent the particular attribute. Deviser Edit does not restrict what can be used here – as users may be using types that Deviser has not anticipated. If the auto-generation code encounters a type with which it is unfamiliar it will add code assuming the object to be an attribute (not element) but use ‘FIX ME’ as the type declaration

The recognized types for an attribute are the datatypes allowed by SBML. These are (with accepted variations):

string, bool(ean), double, int(eger), unsigned int(eger), positive int(eger), non-negative int(eger), ID, SId, SIdRef, UnitSId, UnitSIdRef

and additionally Deviser Edit will accept and process:

array, enum, element, lo_element, inline_lo_element.

Appendix B gives the lists the types with their corresponding C++ data type.



Support for the SBOTerm type is not yet available but is planned.

It should be noted that the ‘Type’ used for each attribute/child element determines the functions that will be produced (see [Code example 1](#)).

```
[Type]  get[Name]    ()
bool    isSet[Name] ()
int     set[Name]    ([Type] value)
int     unset[Name] ()
```

Code example 1: Functions for manipulating attributes with simple types

2.3.3.1 Attribute/child element type 'array'

```
<spatial:sampledField spatial:id="SegmentedImage">
  0 0 1
</spatial:sampledField>
```

SBML snippet 3: SBML Spatial using a text element that is an array of numbers

The 'array' type refers to an XML element that may contain text that represents a list of numerical values of a particular type. For example the L3 Spatial Package uses a SampledField element that contains an 'array' of integers (see [SBML snippet 3](#)).

This information would be defined in the 'Class attributes and child elements' section of the Class description as an entry with

Name: the name to be used by code to store and manipulate this information

Type: array

Element: integer (the numeric type of the data)

and the corresponding functions are produced.

```
void    get[Name]      ([Type]* outArray)
bool    isSet[Name]    ()
int     set[Name]      ([Type]* inArray, int arrayLength)
int     unset[Name]    ()
```

Code example 2: Functions produced for child element of 'array' Type

2.3.3.2 Attribute/child element type 'enum'

If the attribute is of an enumeration type defined within the package it should have type 'enum' and the Element field should give the name of the enumeration. The enumeration is declared fully by adding an enumeration to the description (see Add enum information).

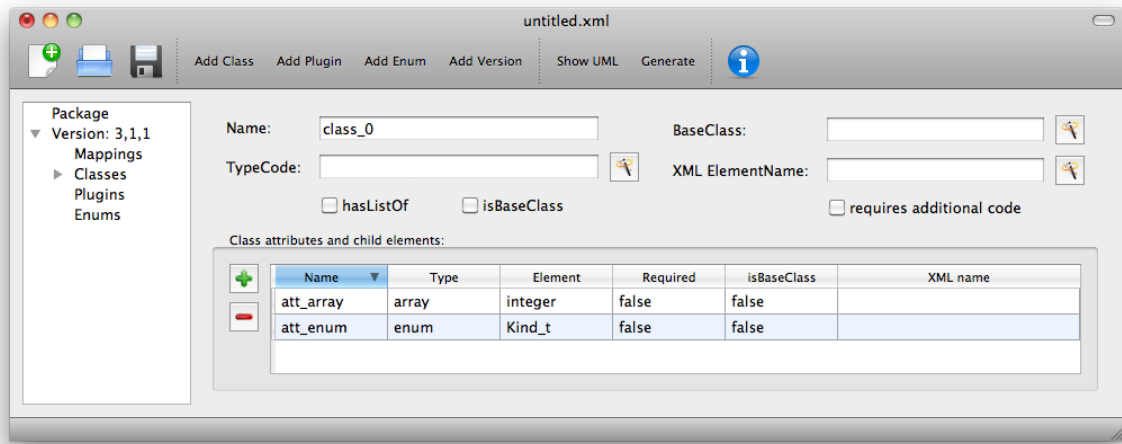


Figure 8: Attributes of type 'array' and 'enum'

[Element]	get[Name]	()
const std::string&	get[Name]AsString	()
bool	isSet[Name]	()
int	set[Name]	([Element] value)
int	set[Name]	(const std::string& value)
int	unset[Name]	()

Code example 3: Function produced for an attribute of type 'enum'

2.3.3.3 Attribute/child element types 'element' and 'lo_element'

Here the child refers to a single instance of another class. If that class is a ListOfClass 'lo_element' should be used. The name of the child element is given in the Element field; where the container is a listOf the Element field should be the child of the listOf. Table 1 gives examples of the expected XML and the functions produced for each type.

2.3.3.4 The type 'inline_lo_element'

On occasion an element may contain multiple children of the same type which are not specified as being within a listOf element. From a code point of view it is easier to consider these children as being within a listOf element as this provides functionality to access and manipulate potentially variable numbers of child elements. The 'inline_lo_element' type allows the user to specify that there are

multiple instances of the same child element but that these do not occur within a specified ListOf element. . [Table 1](#) gives examples of the expected XML and the functions produced.

Type element	XML output	Functions
	<pre><container> <parameter attributes= .../> </container></pre>	<pre>getParameter() isSetParameter() setParameter(Parameter*) unsetParameter() createParameter()</pre>
lo_element	<pre><container> <listOfParameters> <parameter attributes= .../> <parameter attributes= .../> ... </listOfParameters> </container></pre>	<pre>getListOfParameters() getParameter(index) getParameter(id) addParameter(Parameter*) getNumParameters() createParameter() removeParameter(index) removeParameter(id)</pre>
inline_lo_element	<pre><container> <parameter attributes= .../> <parameter attributes= .../> ... </container></pre>	<pre>getListOfParameters() getParameter(index) getParameter(id) addParameter(Parameter*) getNumParameters() createParameter() removeParameter(index) removeParameter(id)</pre>

Table 1: The 'element', 'lo_element' and 'inline_lo_element' types

The **Element** field provides additional information depending on the type of the object being described. [Table 2](#) describes how this field should be populated. Note the 'name' of an element or object refers to the ClassName of the appropriate object.

Type	Element field
array	type of data within the array
enum	The name of the enumeration
element	The name of the element
lo_element	The name of the element within the ListOf
inline_lo_element	The name of the element
SIdRef	The name of the object being referenced. (Limited to one element for now).
Any other	blank

Table 2: Expected values for the Element field based on attribute Type

NOTE TO LUCIAN: We are aware that an SIdRef may refer to several things, having only one entry here currently does not affect any code other than the fact that it will generate 'getObjectByElement' type functions for the Element referred to. We will expand this but it does still produce perfectly functioning code – it just does not have some additional frills.

The **Required** field indicates whether the attribute or child element is mandatory. On occasion SBML has conditional requirements e.g. you must set either StoichiometryMath or stoichiometry but you cannot have both. As yet Deviser does not deal with this situation. We recommend that if you need to facilitate this situation you mark both attributes as 'unrequired' and adjust the generated code accordingly.

The **isBaseClass** field indicates that the child element is a base class and not instantiated directly. This is a situation that will not commonly occur but happens when there is multiple nesting of classes. The current 'spatial' package defines a CSGTransformation that inherits from CSGNode but also contains an element of that type (see [Figure 9](#) and [Figure 10](#)).

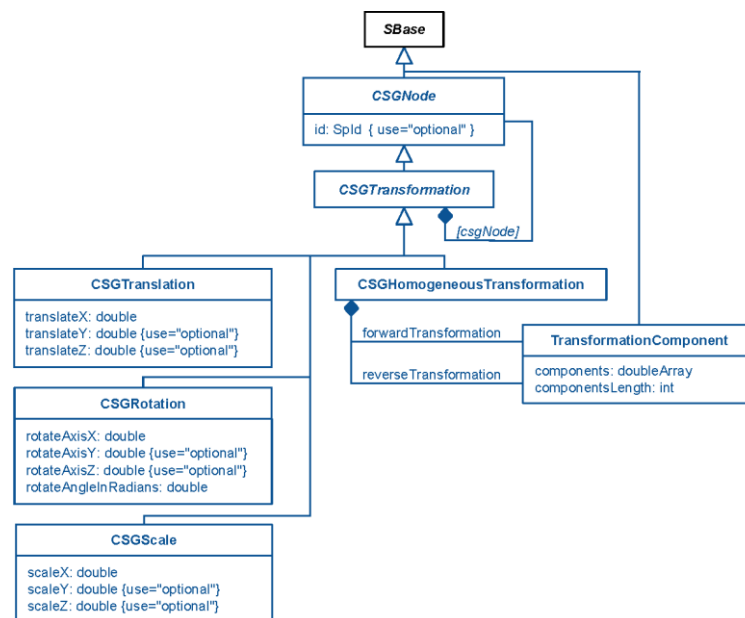


Figure 9: UML diagram of CSGTransformation from SBML L3 'spatial' package specification

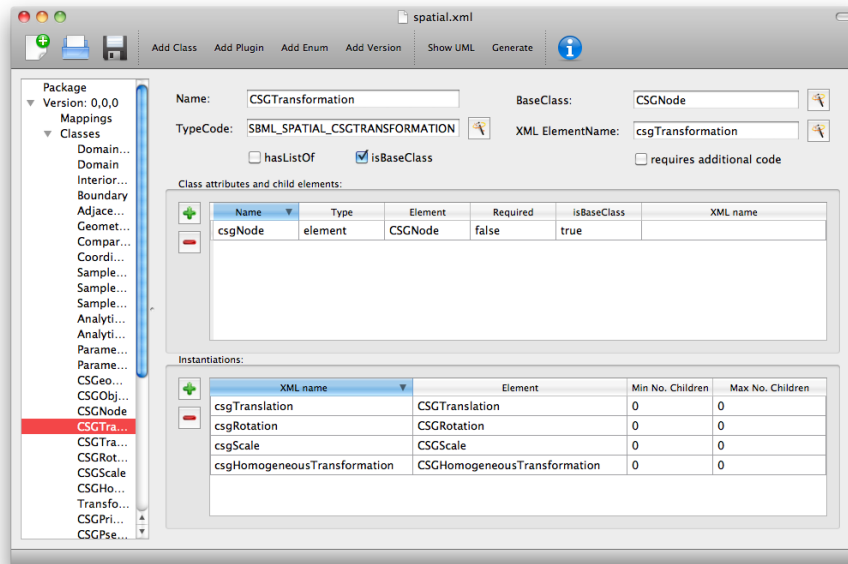


Figure 10: DeviserEdit description of CSGTransformation

Note that the child element 'csgNode' has been marked as a base class. This tells Deviser to generate code relevant to the instantiations of the csgNode class rather than for a concrete CSGNode child. For example you would get `createCSGTransformation()` rather than `createCSGNode()`.

The **XML name** field can be used to specify the name of the element as it will appear in the XML output where this may differ from the Name field. For attributes it is unlikely that the Name used will differ from the XML name; however if the object being listed is an element or listOf element there may be situations where they differ – as in Example 2 below.

2.3.4 A note on repeated information

Users may become aware of the fact that at times they are entering duplicate information. For example if a child element is used that does not have the default XML Name then this will be declared both when describing the Class for that element AND when listing the child element occurrence (see Example 2). Also, classes derived from a base class are listed as Instantiations of that class when it would be possible to work out this information from the BaseClass information given for each class.

Deviser Edit **does require this information to be duplicated** as this facilitates the storing of unfinished definitions and allows the definition to be validated to an extent. It also means that each sheet contains all the pertinent information for the Class being specified rather than this information being distributed across various sheets in the GUI.

2.3.5 Example 1 - Adding a class with no containing ListOf

Here we define the KineticLaw class for our imaginary package 'foo'.

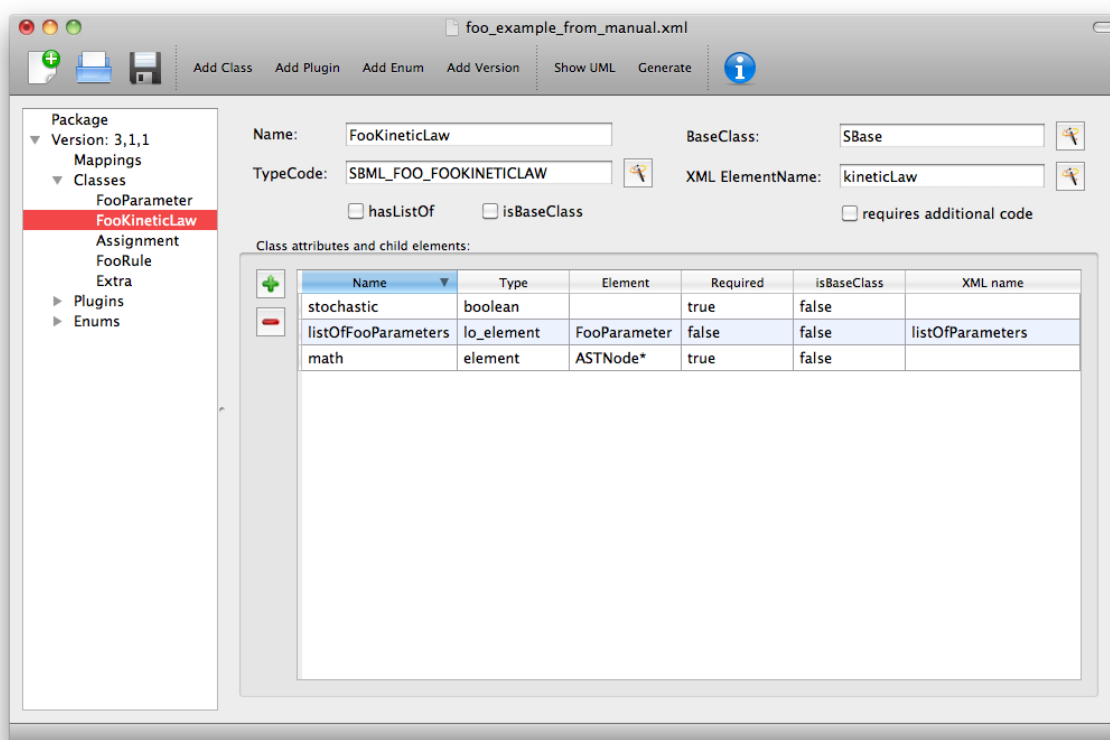


Figure 11 Defining the FooKineticLaw class.

We know that libSBML already contains a class KineticLaw and so we use a class name that reflects the package and class i.e. 'FooKineticLaw' and we specify that the XML ElementName will be 'kineticLaw'. Thus the generated code will use a class 'FooKineticLaw' that will not conflict with existing libSBML classes but would output this in XML as an element <foo:kineticLaw>. This causes no conflict as XML Namespaces keep elements completely separate.

Our class has three attributes/child elements.

The first is a boolean attribute called 'stochastic', which is mandatory. So we add the name 'stochastic', the type 'boolean' and change the required status to 'true'.

The second child is a ListOfParameters. Again we know that name will conflict with the class ListOfParameters so we add the name 'listOfFooParameters', the type 'lo_element', the element 'FooParameter' and state that the XML name is 'listOfParameters'. Note that we will need to specify the class FooParameter later on; which we do in Example 2.

The third child is a math element. So we add the name 'math', the type 'element' and the element 'ASTNode*'. Note that Deviser does specifically recognize the elements ASTNode and XMLNode and treats them appropriately.

2.3.6 Example 2 - Adding a class with a containing ListOf

Here we specify the FooParameter class used by the FooKineticLaw that we specified in Example 1.

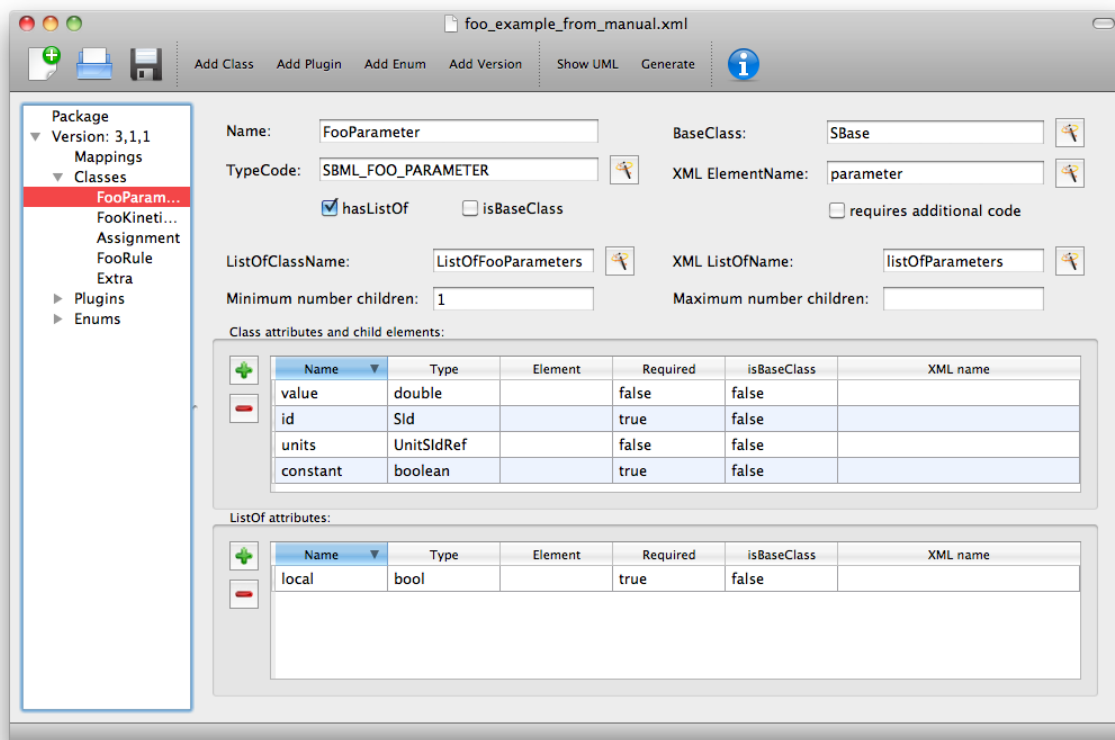


Figure 12 Defining the FooParameter class.

The **hasListOf** checkbox has been selected and a number of additional fields appear.

The **ListOfClassName** is the name used in code for the class representing the ListOf object. Again it need only be populated if the default of 'ListOfBars' is inappropriate.

The **XML ListOfName** field is the XML name for the list of objects. It only needs to be populated if there is a difference in name between XML and code. It will default to 'listOfBars' where 'Bar' is the class name.

In our example we have populated these fields as we have used a class name 'FooParameter' but will have XML names of 'parameter' and 'listOfParameters'.

The **Minimum number of children** field is used to indicate the minimum number of child objects of type Bar a ListOfBars expects. Currently in SBML ListOf elements cannot be empty and so must have a minimum of 1 child; which we have indicated in our example. Leaving this blank implies there is no stated minimum value for the number of children.

The **Maximum number of children** field is used to indicate the maximum number of child objects a ListOf expects. Leaving this blank implies there is no stated maximum value for the number of children.

The **ListOf attributes** table (which has the same fields as the table for entering class attributes and child elements) allows you to add attributes to the ListOf class.

Figure 13 shows the UML diagram produced by Deviser Edit of the package 'foo' as described so far in examples 1 and 2 while the corresponding SBML is shown in SBML snippet 4.

```
<foo:kineticLaw foo:stochastic="false">
  <foo:listOfParameters foo:local="true">
    <foo:parameter foo:id="p1" foo:constant="true"/>
  </foo:listOfParameters>
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    ...
  </math>
</foo:kineticLaw>
```

SBML snippet 4: The kineticLaw element as reflecting the definition of package Foo in Examples 1 and 2

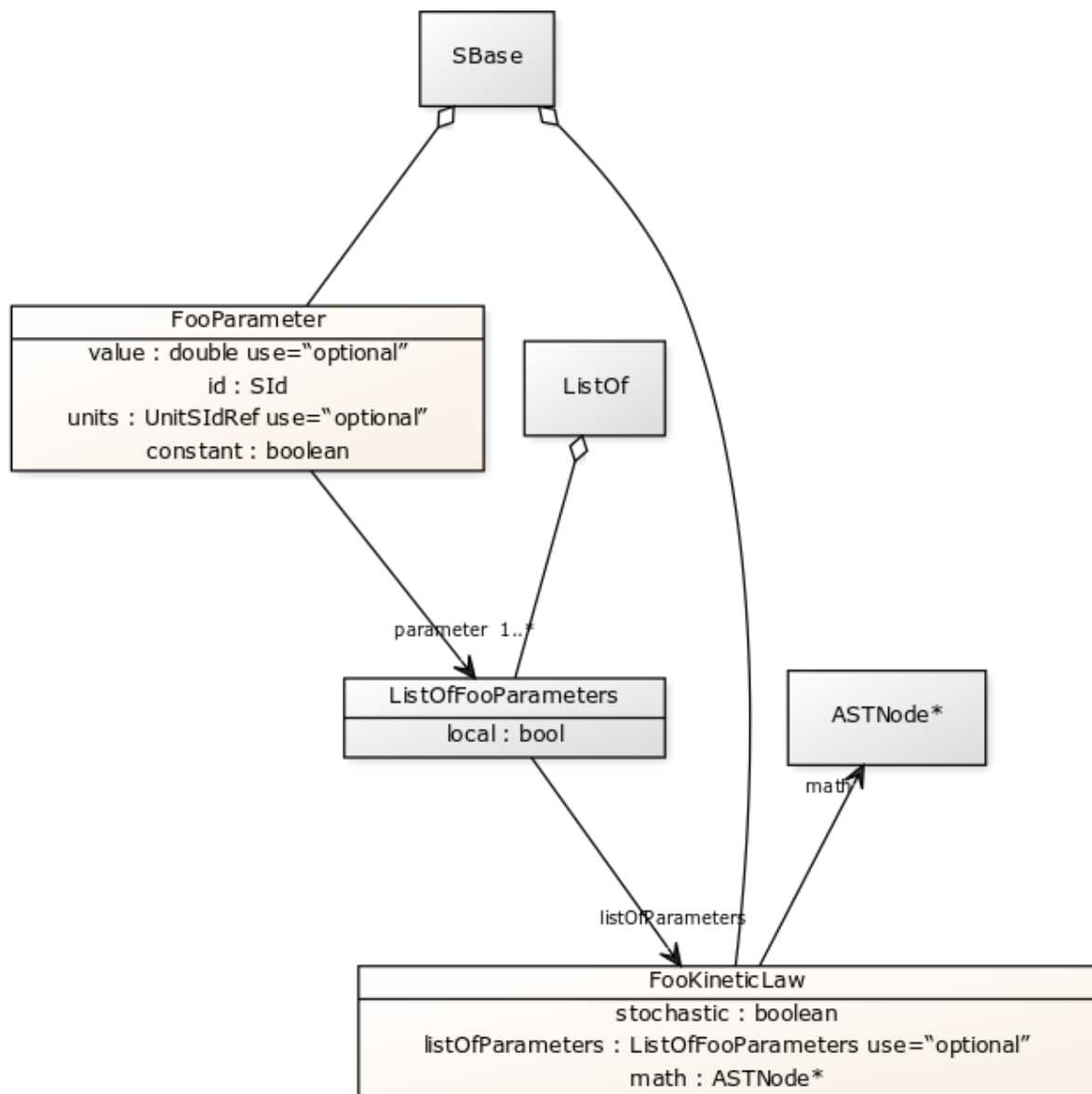


Figure 13: The UML diagram produced by Deviser Edit following the definition of package 'Foo' in Examples 1 and 2

2.3.7 Example 3 – Adding a base class and derived classes

Here we define a class that will be used as a base class for others (see [Figure 14](#)).

The screenshot shows a software interface for defining a class. The sidebar on the left contains a tree view with the following structure:

- Package
 - Version: 3,1,1
 - Mappings
 - Classes
 - FooParam...
 - FooKineti...
 - Assignment**
 - Extra
 - Plugins
 - Enums

The main form contains the following fields and checkboxes:

- Name: FooRule
- TypeCode: SBML_FOO_RULE
- BaseClass: SBase
- XML ElementName: (empty)
- ☒ hasListOf
- ☒ isBaseClass
- ☐ requires additional code
- ListOfClassName: (empty)
- XML ListOfName: (empty)
- Minimum number children: 1
- Maximum number children: (empty)

Below these fields are three tables:

Class attributes and child elements:

Name	Type	Element	Required	isBaseClass	XML name
math	element	ASTNode*	true	false	

ListOf attributes:

Name	Type	Element	Required	isBaseClass	XML name
------	------	---------	----------	-------------	----------

Instantiations:

XML name	Element	Min No. Children	Max No. Children
assignment	Assignment	0	0
algebraic	Algebraic	0	0

Figure 14 Defining the base class 'FooRule'.

This class is named FooRule and has a corresponding ListOf element. Note we have not filled in any alternative names so we will expect to get an element called listOfFooRules in the XML.

This class is a base class and we tick the isBaseClass checkbox. The **Instantiations** table then appears.

The **Instantiations** table allows you to specify the class(es) that will be derived from this base class. Note Deviser Edit expects these to be listed here – even if the information could be determined elsewhere. Entries in this table do not define a class, the definition of the class should be created as a separate class entry. Only classes that directly inherit from this class need be listed; it may be that the classes listed are themselves base classes for further classes. These should be listed as the Instantiations on the relevant base class description.

2.3.7.1 Instantiations fields

The **XML name** field specifies the XML name of the object.

The **Element** field specifies a class that will be derived from this FooRule base class.

The **Min No. Children** field is used to specify a minimum number of children that this element may have.

The **Max No. Children** field is used to specify the maximum number of children.

Note that sometimes a specific instantiation adds further requirements. For example, where one class may contain children of the same base class there may be a requirement that it contains a certain number of children as with Associations in the FBC package an FBCAnd instantiation **MUST** have two children. Where there are no such requirements these fields should be left as '0'.

Here we have specified that the ListOfFooRules may contain objects of type Assignment or Algebraic. We specify Algebraic as a new class as in [Figure 15](#) and Assignment in [Figure 16](#).

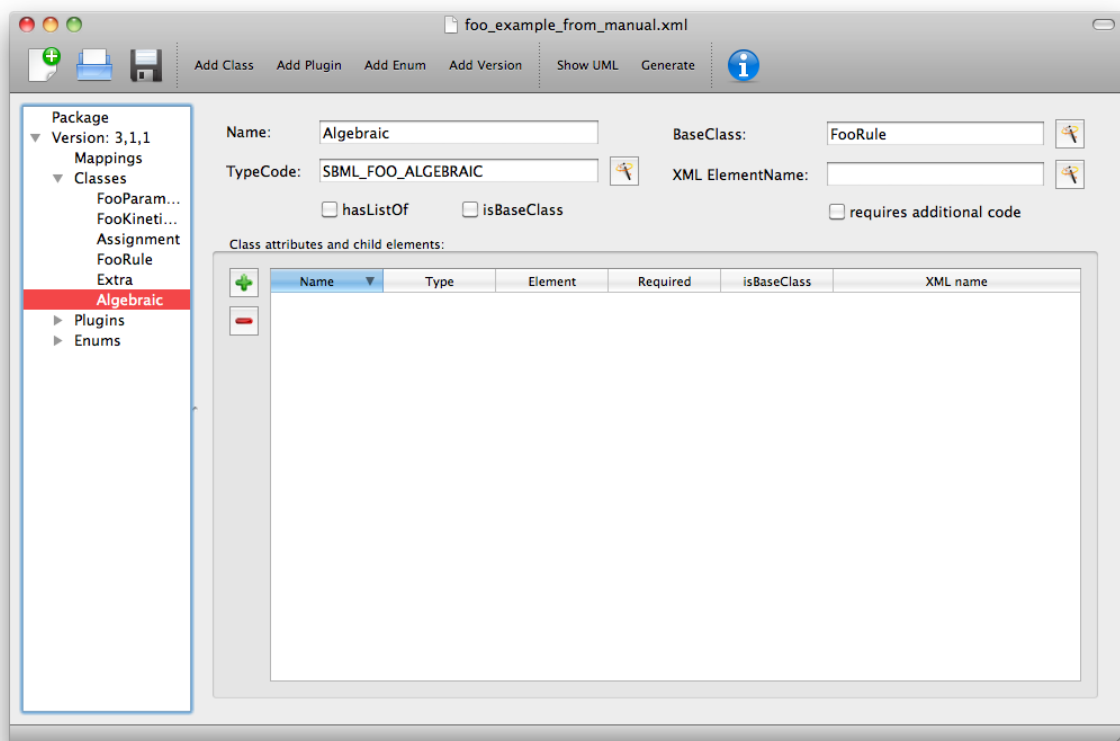


Figure 15 Defining the Algebraic class

Note that we have changed the BaseClass field to FooRule.

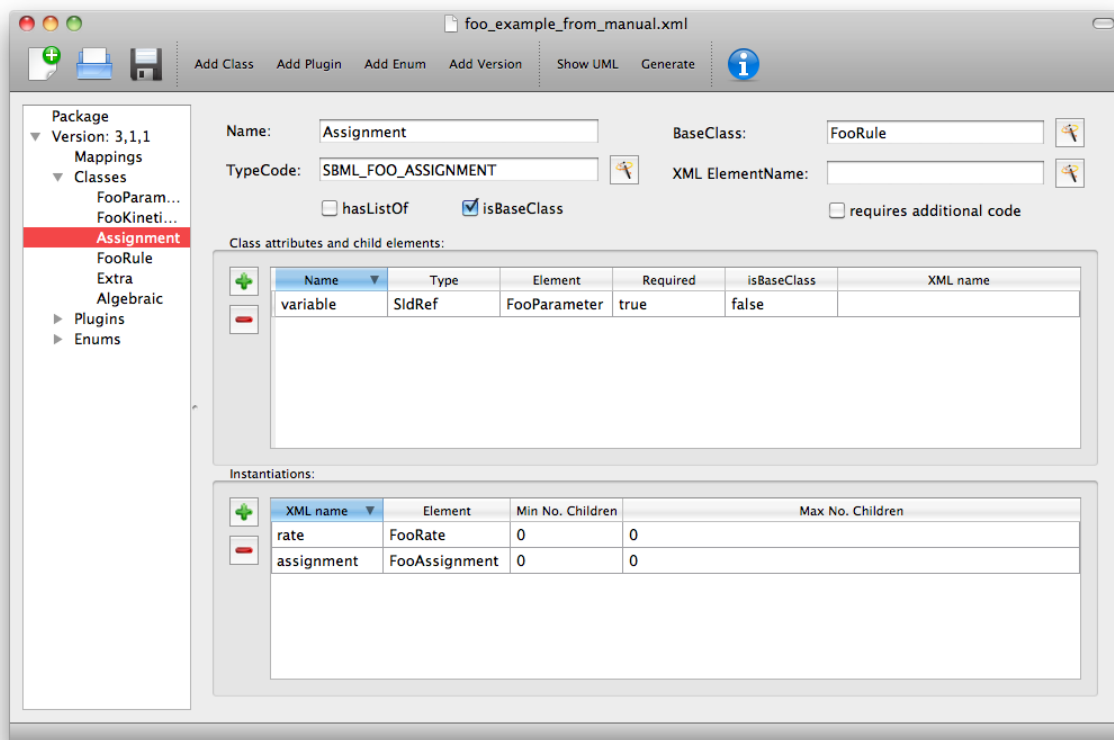


Figure 16: Defining the Assignment class.

The Assignment class illustrates a slightly more complex scenario. Here it derives from the baseClass FooRule and adds an attribute 'variable' that is a reference to a FooParameter. It also acts as a base class for two further classes FooRate and FooAssignment. [Figure 17](#) shows the hierarchy and [SBML snippet 5](#) the resulting XML.

```
<foo:listOfFooRules>
  <foo:assignment foo:variable="p">
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      ...
    </math>
  </foo:assignment>
  <foo:rate foo:variable="s">
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      ...
    </math>
  </foo:rate>
  <foo:algebraic>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      ...
    </math>
  </foo:algebraic>
</foo:listOfFooRules>
```

SBML snippet 5: The XML output for the ListOfFooRules defined in Example 3

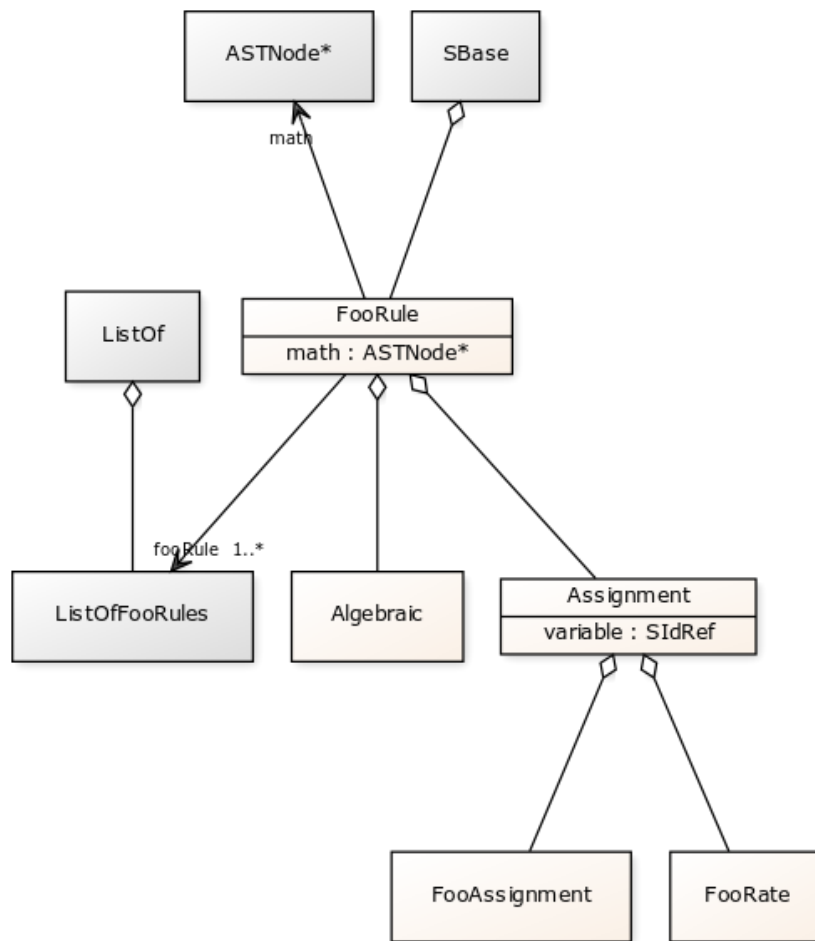


Figure 17: The UML diagram from DeviserEdit of the ListOfFooRules class

2.4 Add plugin information

2.4.1 What is a plugin ?

In order to extend SBML Level 3 Core with a package not only is it necessary to define new classes, it is also necessary to attach these elements to an existing point in an SBML model. The simplest case would be that a new element is added to the containing <sbml> element but the point of extension may be much further embedded within the SBML. Here (and indeed within libSBML) we use the term 'plugin' to specify the necessary information that links the new package classes with other classes. Code for any given class in any relevant function then checks whether it has a plugin attached and passes control to the plugin if necessary. [Figure 18](#) shows two plugins on the Model class, one by the 'qual' package and the other by the 'fbc' package. Note the names reflect the package and the object being extended.

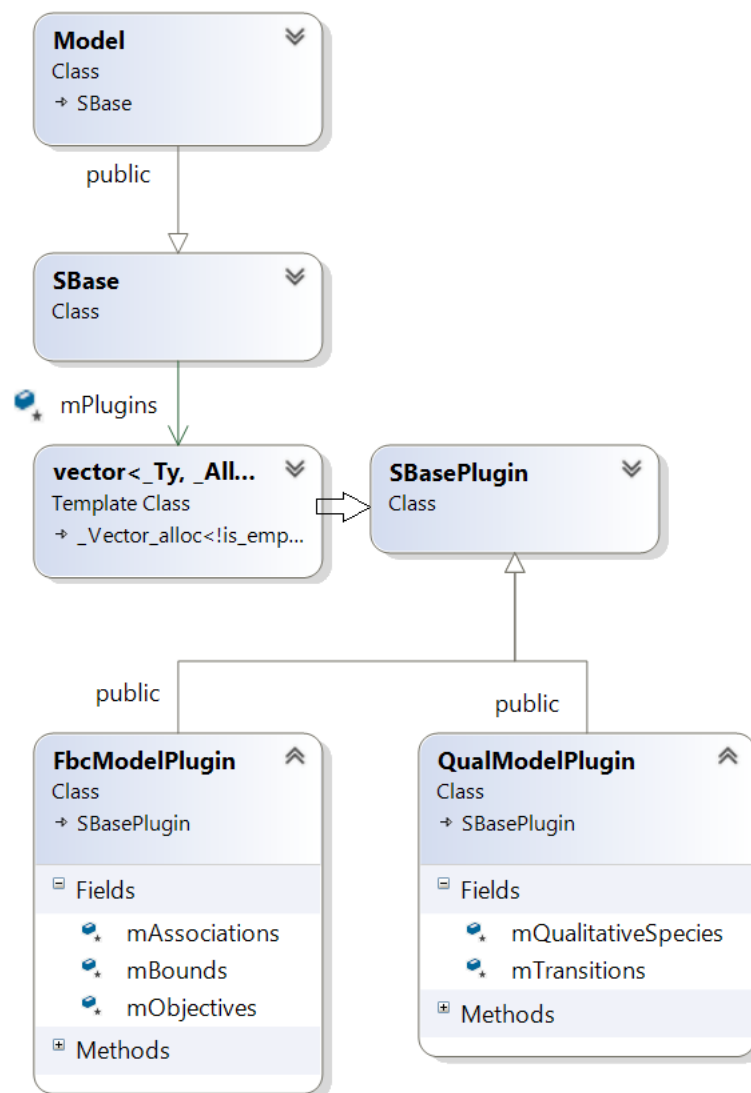


Figure 18 libSBML class hierarchy showing 'plugins' to the Model class

2.4.2 General plugin information

Plugin information describes the elements that are extended by the new classes defined within a package. The elements to be extended may come from SBML Level 3 Core or another SBML Level 3 package.

Select 'Add Plugin' from the toolbar or the 'Edit' menu.

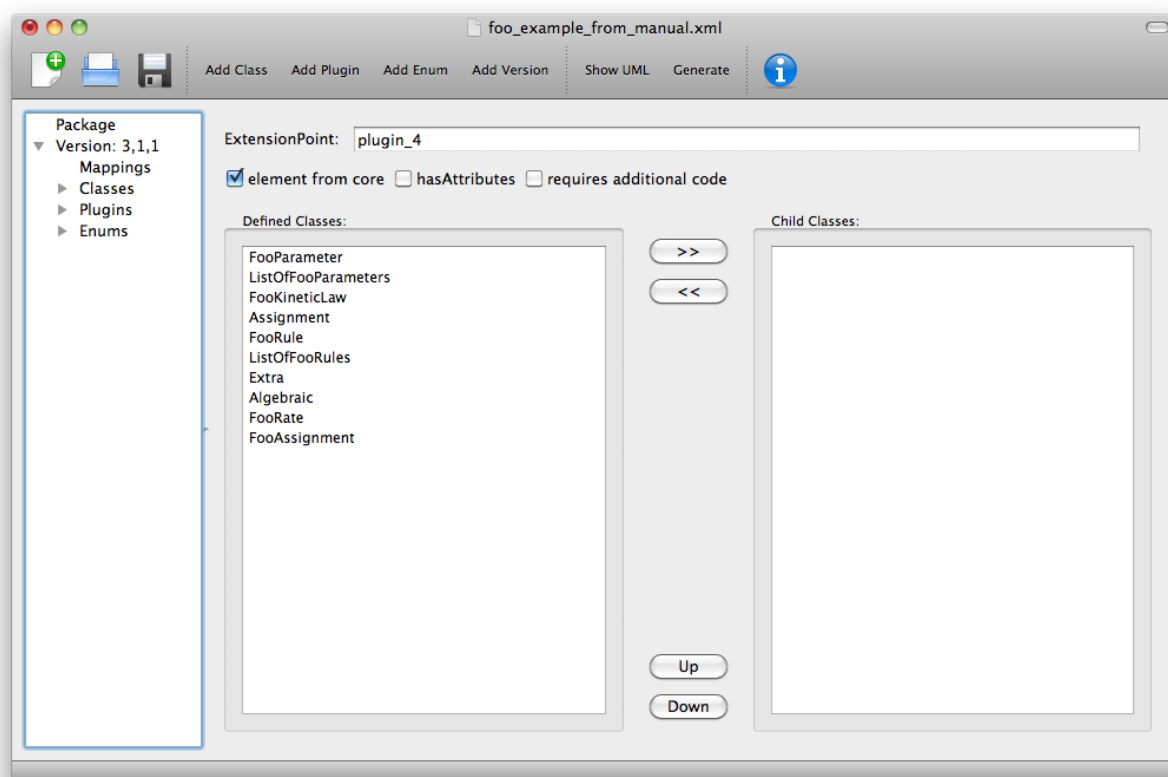


Figure 19 The 'Plugin' sheet.

The **ExtensionPoint** field is used to specify the name of the element that is being extended. This will be the name of the class as used by libSBML.

The **element from core** checkbox is used to specify whether the object being extended originates in SBML Core or another Level 3 package.

The **hasAttributes** checkbox should be ticked if the package is going to extend an object with attributes rather than (or as well as) elements.

As on other sheets the **requires additional code** checkbox can be used to indicate that there is additional code that will be required by the package that will not be generated by Deviser. Checking the 'requires additional code' box reveals further boxes that can be used to specify the location of the additional code files. Deviser will incorporate this code 'as-is'.

The sheet for adding a plugin lists the classes that have already been specified (**Defined Classes**) and are 'available' to extend an object. These can be selected and moved into the **Child Classes** column.

2.4.3 Example 4 – Extending a core element

Here we are going to specify that the 'foo' package extends the SBML Level 3 Core Reaction with the new FooKineticLaw class.

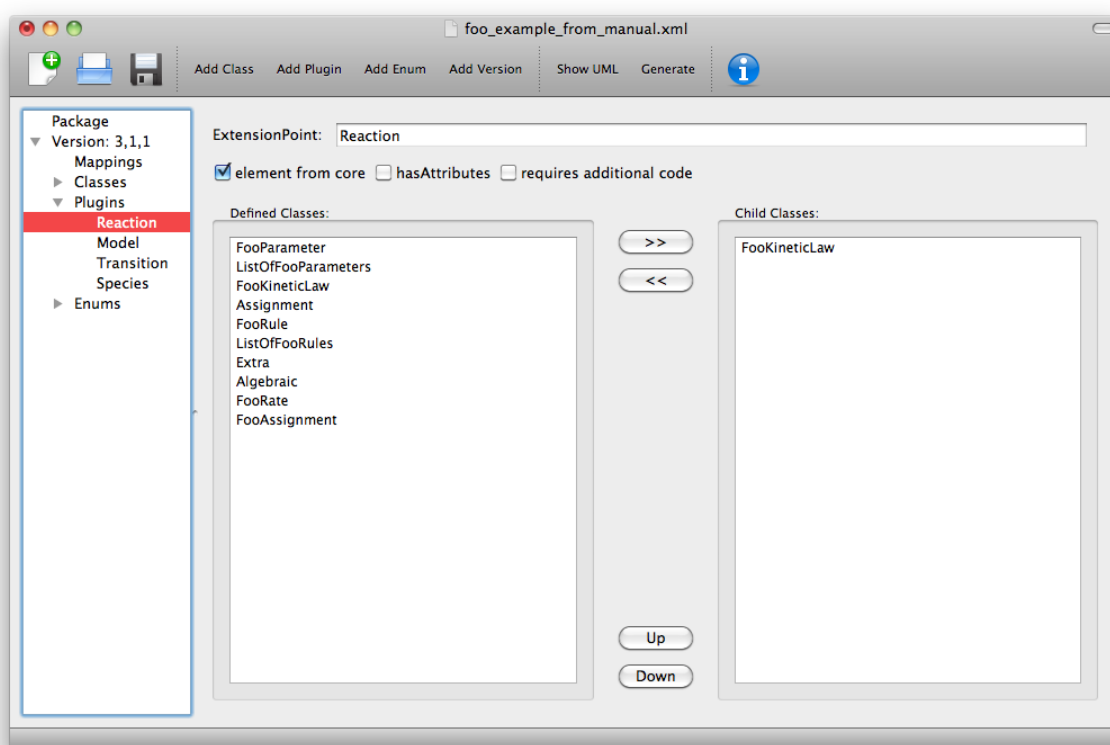


Figure 20 Defining the extension of SBML Level 3 Core Reaction by package foo.

We fill in the **ExtensionPoint** with 'Reaction', tick the checkbox to note that the element is from core. Highlight FooKineticLaw in the **Defined Classes** column and use the arrows to move it to the **Child Classes** column. Essentially this is telling Deviser to generate the class FooReactionPlugin which will expect to have a data member of type FooKineticLaw class.

2.4.4 Example 5 – Extending a core element with attributes only

Here we declare that the **ExtensionPoint** is Model from core and tick the **hasAttributes** checkbox.

The table **Child attributes and child elements** appears. This is used for adding attributes and child elements as previously described. Here we specify that the Model will have a required boolean attribute ‘useFoo’ from the foo package (Figure 21). Note that it is not necessary to specify child elements that originate in the package being defined i.e. those that have already been listed as **Child classes**.

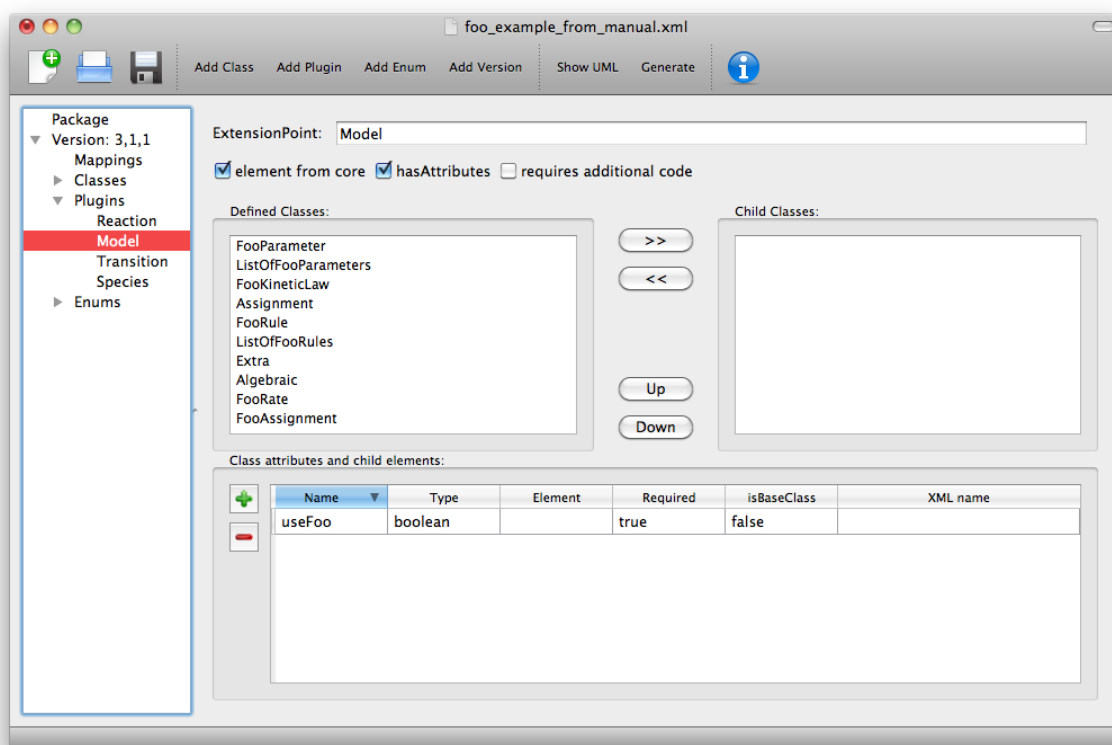


Figure 21 Defining the extension of SBML Level 3 Core Model by package foo.

2.4.5 Example 6 – Extending a non-core element

Here we declare that the **ExtensionPoint** is Transition from the Qualitative Models (qual) Package. The package foo adds the ListOfFooRules object to the Transition object.

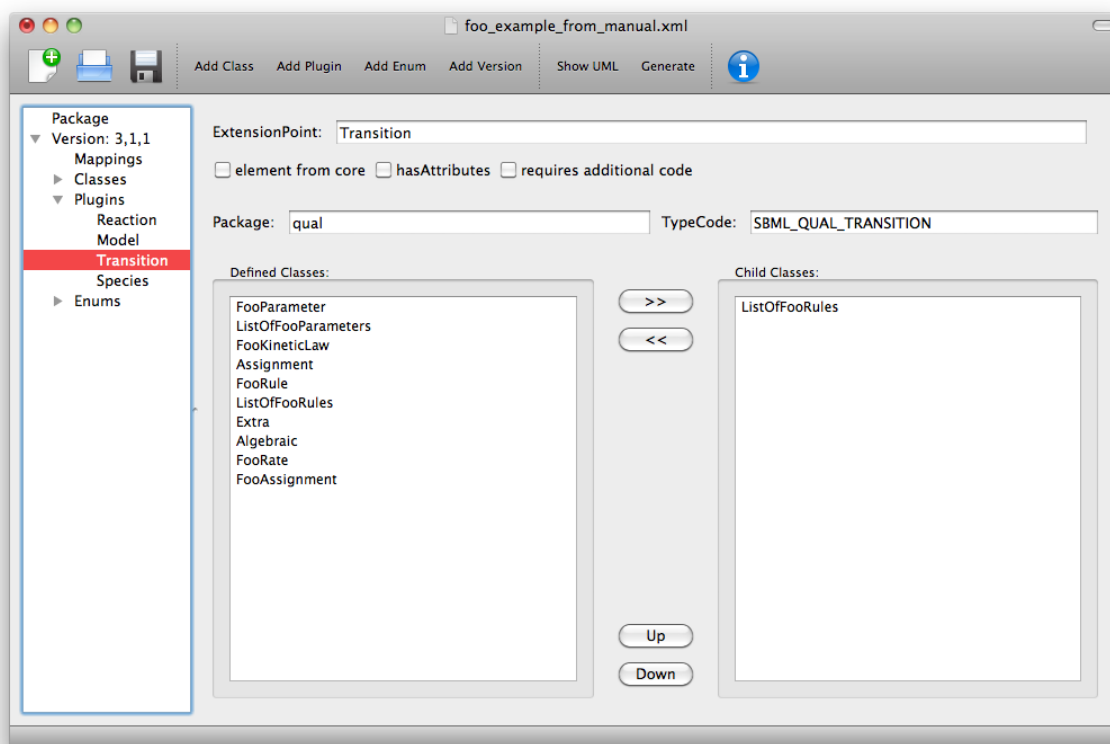


Figure 22 Defining the extension of SBML Level 3 Qual Transition by package foo.

2.5 Add enum information

SBML allows users to define data types as enumerations of allowed values. Here we describe how to specify these.

2.5.1 Example 7 – Adding an enumeration

Assume we have an object 'extra' that has an attribute called 'sign' which is of an enumeration type 'Sign'. Firstly we define the class 'Extra' and specify the attribute. In this case the **Type** of the attribute is 'enum' and the **Element** field gives the name of the enumeration type 'Sign' as shown in [Figure 23](#).

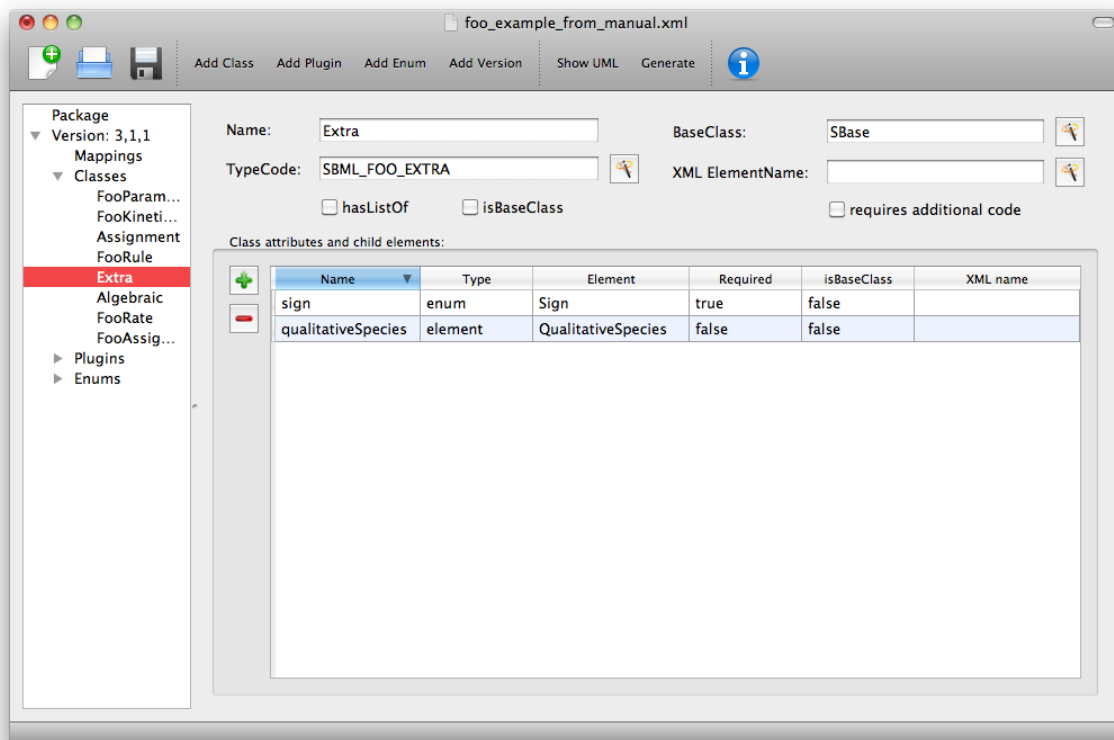


Figure 23 Defining the Extra class which has an attribute of type enum.

Then it is necessary to specify the enumeration itself. Use the **Add Enum** button from the toolbar or Edit menu.

The **Name** field is used to declare the name of the enumeration, in this case Sign. Note that when generating code Deviser will append an ‘_t’ to this name.

The table is used to specify the individual allowed values of the enumeration.

The **Name** field is the enumeration value that will appear in the enumeration itself.

The **Value** field gives the corresponding string value of that member of the enumeration.

Here (Figure 24) we have specified that the enumeration sign has three possible values: ‘positive’, ‘negative’ and ‘neutral’. Note the names used reflect the individual values and the package in which they originate. It is not necessary to add a default or “unknown” value – Deviser will do this when generating code.

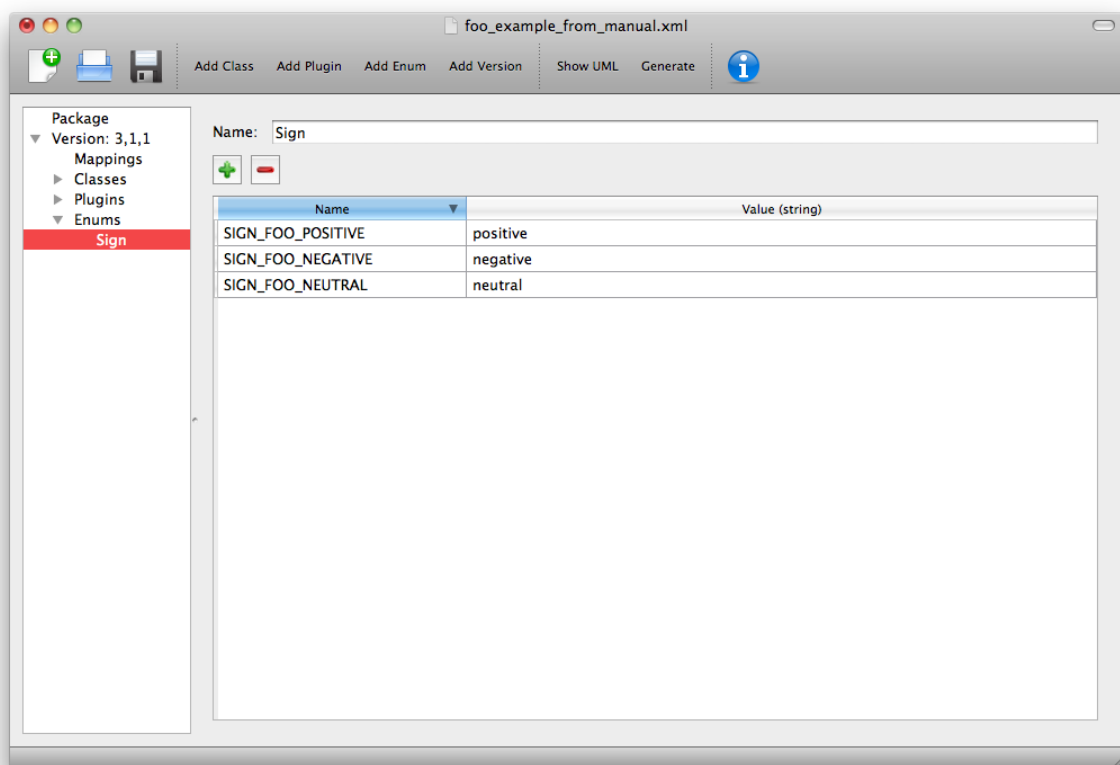


Figure 24 Defining the Sign enumeration.

2.6 Mappings

Once the class and plugin descriptions are complete the Deviser Edit tool will determine if there are any classes present that do not originate from core or the package being described. Select 'Mappings' from the tree in the panel on the left hand side. The tool will have prepopulated this with any relevant classes and all that remains is for the package information to be filled in.

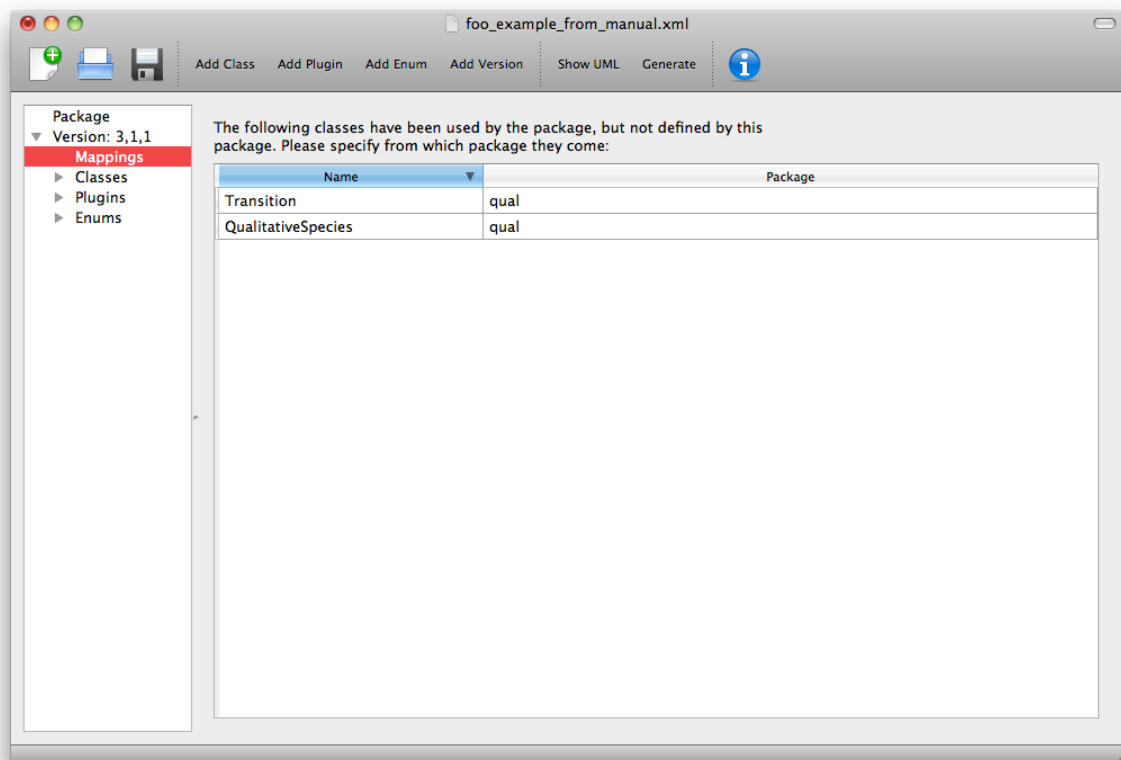


Figure 25 Identifying the origin of classes from other packages

The **Package** field is used to add the name of the package in which the class given in the **Name** field originates. In our example we have used the Transition and QualitativeSpecies classes both of which originate in the 'qual' package, so this information is added. Note on this sheet only the **Package** column can be edited. The **Name** column is populated by the tool.

2.7 Results

Select 'Version' from the tree in the panel on the left hand side. Now that all the classes have been defined these are listed here (see [Figure 26](#)) and the ordering can be adjusted. The order will dictate the order of the relevant section in the TeX documents.

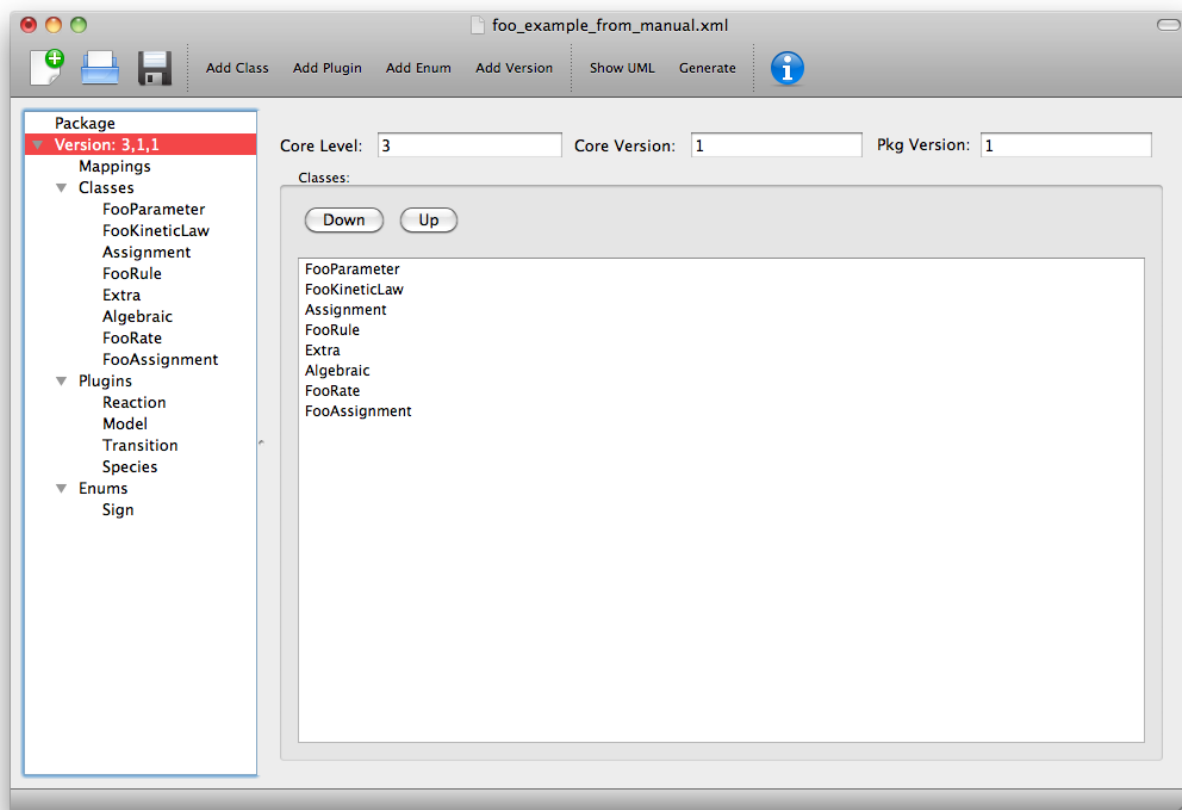


Figure 26 The complete description of the foo package

The Deviser Edit tool creates an XML description from the definition that is used by other code to generate UML, TeX and libSBML code.

Note this underlying XML file can be saved at any point and reopened using Deviser Edit or with any XML Editor. The full description of the Imaginary Foo Package used in the Examples can be seen in Appendix A or is available in the `deviser/samples` directory.

2.7.1 Validating the description

NOTE TO LUCIAN: We have not yet finished implementing all the validation. The plan is to provide a list here of all the possible errors and warnings that might be reported.

There are two further options on the Edit menu that have not yet been discussed.

Validate Description runs a series of internal checks on the information provided and prompts the user to fill in any required fields.

A pop-up window (Figure 27) will appear with either an error message or a confirmation that everything is consistent. The Copy button can be used to copy the contents of the report to the clipboard and thus makes them available for pasting elsewhere.

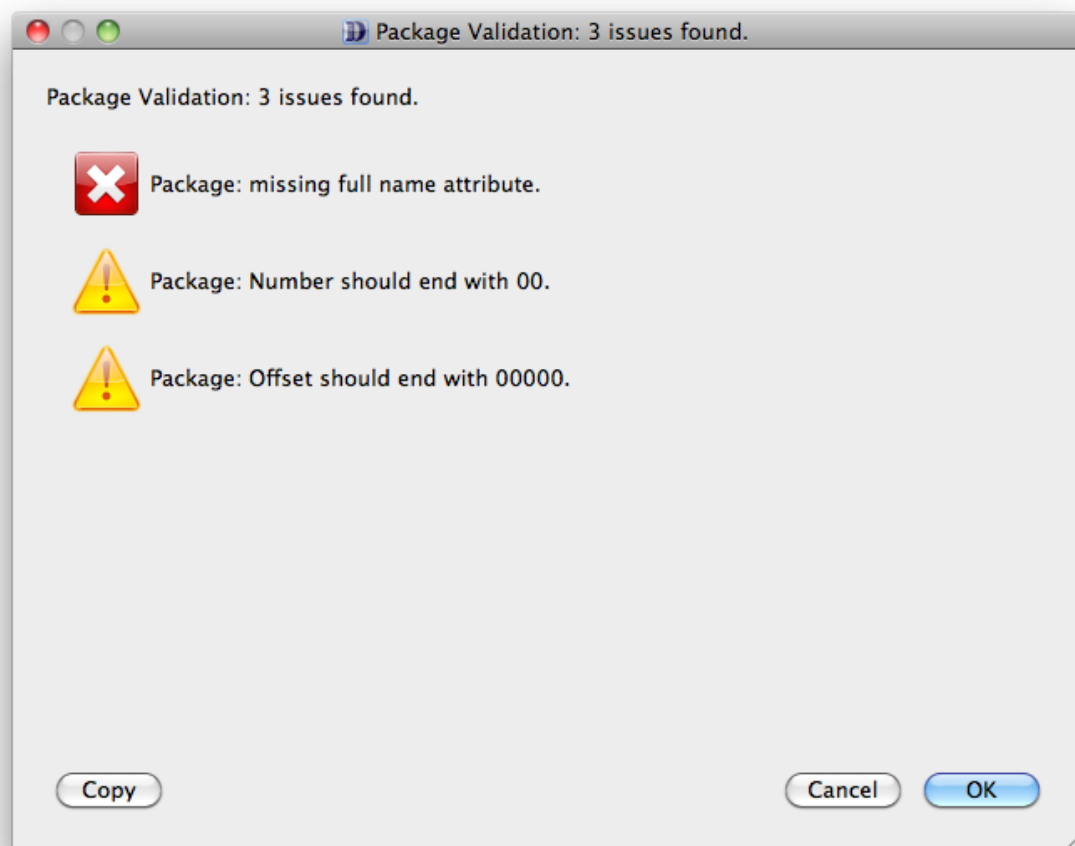


Figure 27 Validating the package description

Fix Errors provides a direct way of validating and then correcting any inconsistencies. It is advisable to use **Validate Description** following **Fix Errors** as some errors cannot be automatically fixed.

3 Using Deviser

The Deviser Edit tool allows you to specify the details of an SBML L3 Package. In addition it can be used to perform a number of tasks directly. Some of these require additional software on your system and Deviser Edit needs to be given the information about where to find these. This is described in Section 1.3.

3.1 View UML diagrams.

Once you have specified at least one class you can click on the **Show UML** button on the toolbar or select **Show UML** from the Tool menu. Remember that this uses the yUML webservice and so will only work if you have an internet connection.

The UML output will appear (Figure 28). Note that each time this option is selected a new diagram is generated from the definition; the more complex the definition the longer it will take.

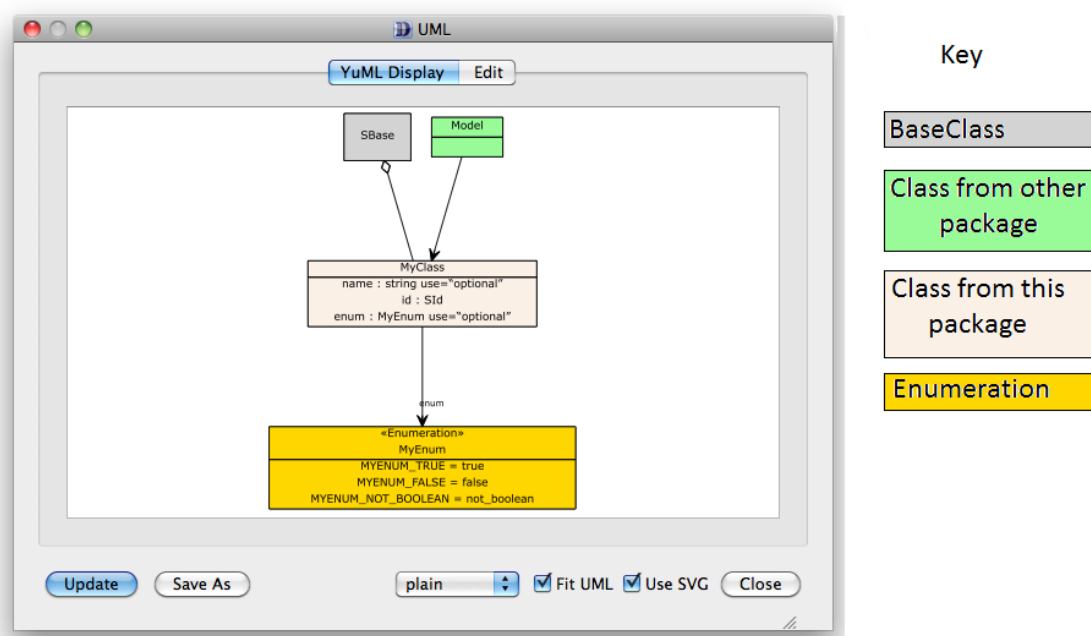


Figure 28: The UML window

You will note that not all inheritance is determined. Deviser does not try to infer inheritance of classes from outside this package definition. However these can be added using the **Edit** window (Figure 29).

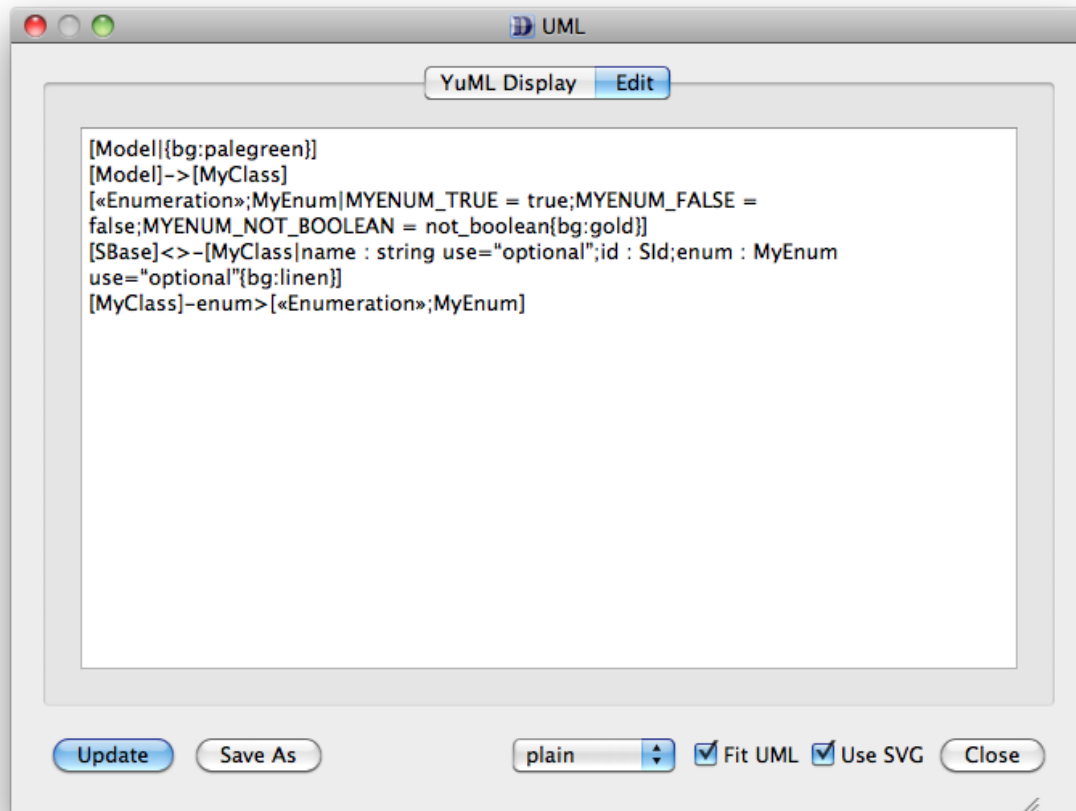


Figure 29: The yUML text representation of the diagram

The **Edit** tab allows access to the text format of the diagram which can be edited and updated. Note the update applies to the UML diagram only, not the defined package.

The **Update** button will update the diagram in line with any changes that have been made to the text description on the Edit panel.

The **SaveAs** button allows the user to navigate to a location of their choice and save the diagram in a graphic format. PNG, JPG, PDF, SVG and yUML are all supported.

There is a selection box and two check boxes that allow the user to change the format of the diagram. These are left for the user to experiment with.

3.2 Generate libSBML package code.

Click on the ‘Generate’ button on the toolbar and the Generate window (Figure 30) will appear.

The **Generate Package Code** button will then create the package code and put it in a directory name ‘foo’ (i.e. the name of the package) in the specified output directory. This code can then be archived and expanded over the libSBML source tree.

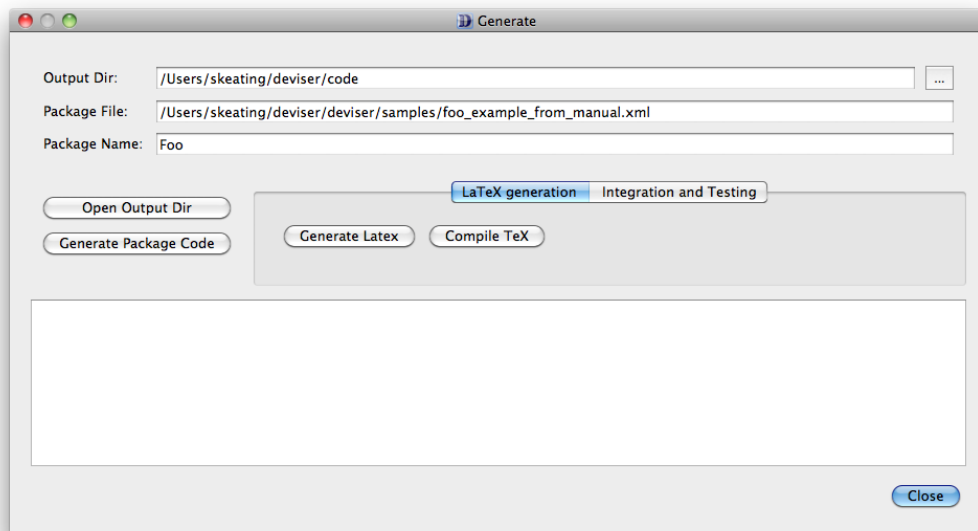


Figure 30 The Generate window

3.3 Generate basic specification documentation.

In the Generate window there is a **LaTeX Generation** tab with two further buttons.

The **Generate Latex** button will generate three TeX files.

- body.tex describing each class and its attributes with their types and cardinalities.
- apdx-validation.tex listing the validation rules.
- macros.tex listing the classes and creating commands for cross-referencing

The **Compile TeX** button takes the generated TeX files, creates another main.tex based on the sbmlpkgspec requirements and generates a basic specification document as a pdf.

3.4 Integrate and test the package with libSBML.

Click the **Integration and Testing** tab in the Generate window and a further set of buttons are revealed ([Figure 31](#)).

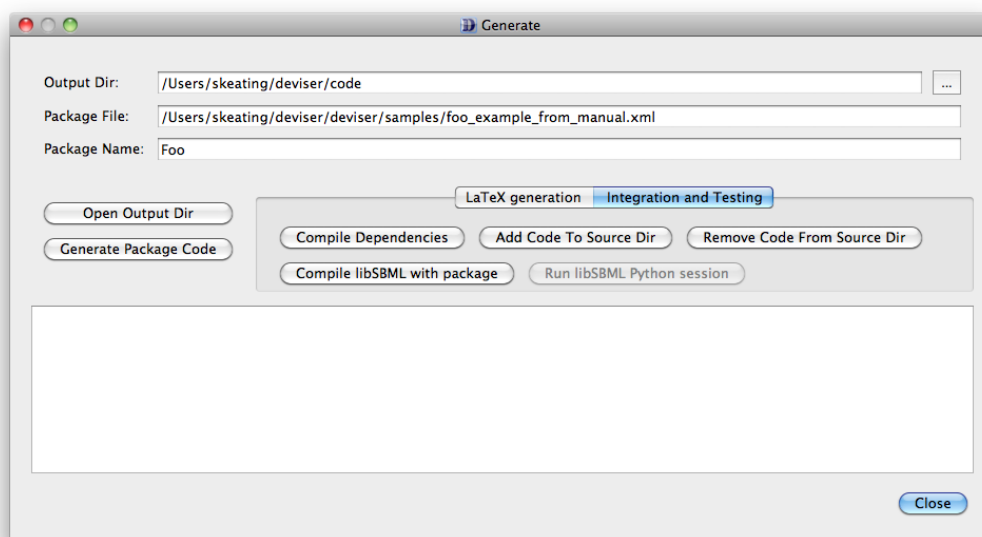


Figure 31 Integration and Testing tab selected on the Generate window.

Compile Dependencies compiles the dependencies with the specified C++ compiler to ensure that these are compatible with the libSBML build.

Add Code to Source Dir places the code generated for package foo within the libSBML source tree.

Remove Code from Source Dir removes the code generated for package foo from the libSBML source tree.

Compile libSBML with package enables package foo within the libSBML build and runs the build.

Run libSBML Python session (Windows only) starts an interactive python session with the package loaded to facilitate manual testing of the code.

3.5 Command line

The command line function `deviser.py` can be invoked directly.

```
deviser.py [--generate][--latex] input-file.xml
```

This program takes as input a Deviser XML file and either

- `--generate (-g)` generates the libSBML code for the package
This generates the complete src package that can be zipped and then unzipped over an existing libSBML src tree.
- `--latex (-l)` generates a LaTeX scaffold for its specification.
This generates the `adx-validation.tex`, `macros.tex` and `body.tex` files which can then be integrated into a package pdf using the `sbmlpkgspec` files.

Generation of a pdf or integration with libSBML can then be done manually.

4 Appendix A

```
<?xml version="1.0" encoding="UTF-8"?>
<package name="Foo" fullname="Imaginary Package Foo" number="2000"
  offset="1400000" version="1" required="true">
  <versions>
    <pkgVersion level="3" version="1" pkg_version="1">
      <elements>
        <element name="FooParameter" typeCode="SBML_FOO_PARAMETER"
          hasListOf="true" minNumListOfChildren="1" baseClass="SBase"
          abstract="false" elementName="parameter" listOfName="listOfParameters"
          listOfClassName="ListOfFooParameters">
          <attributes>
            <attribute name="value" required="false" type="double" abstract="false"/>
            <attribute name="id" required="true" type="SId" abstract="false"/>
            <attribute name="units" required="false" type="UnitSIdRef" abstract="false"/>
            <attribute name="constant" required="true" type="boolean" abstract="false"/>
          </attributes>
          <listOfAttributes>
            <listOfAttribute name="local" required="true" type="bool" abstract="false"/>
          </listOfAttributes>
        </element>
        <element name="FooKineticLaw" typeCode="SBML_FOO_KINETIC_LAW"
          hasListOf="false" baseClass="SBase" abstract="false"
          elementName="kineticLaw">
          <attributes>
            <attribute name="stochastic" required="true" type="boolean" abstract="false"/>
            <attribute name="listOfFooParameters" required="false" type="lo_element"
              element="FooParameter" xmlName="listOfParameters" abstract="false"/>
            <attribute name="math" required="true" type="element" element="ASTNode*"
              abstract="false"/>
          </attributes>
        </element>
        <element name="Assignment" typeCode="SBML_FOO_ASSIGNMENT"
          hasListOf="false" baseClass="FooRule" abstract="true">
          <attributes>
            <attribute name="variable" required="true" type="SIdRef"
              element="FooParameter" abstract="false"/>
          </attributes>
          <concretes>
            <concrete name="rate" element="FooRate"/>
            <concrete name="assignment" element="FooAssignment"/>
          </concretes>
        </element>
        <element name="FooRule" typeCode="SBML_FOO_RULE" hasListOf="true"
          minNumListOfChildren="1" baseClass="SBase" abstract="true">
          <attributes>
            <attribute name="math" required="true" type="element" element="ASTNode*"
              abstract="false"/>
          </attributes>
        </element>
      </elements>
    </pkgVersion>
  </versions>
</package>
```

```

</attributes>
<concretes>
  <concrete name="assignment" element="Assignment"/>
  <concrete name="algebraic" element="Algebraic"/>
</concretes>
</element>
<element name="Extra" typeCode="SBML_FOO_EXTRA" hasListOf="false"
  baseClass="SBase" abstract="false">
  <attributes>
    <attribute name="sign" required="true" type="enum" element="Sign"
      abstract="false"/>
    <attribute name="qualitativeSpecies" required="false" type="element"
      element="QualitativeSpecies" abstract="false"/>
  </attributes>
</element>
<element name="Algebraic" typeCode="SBML_FOO_ALGEBRAIC"
  hasListOf="false" baseClass="FooRule" abstract="false"/>
<element name="FooRate" typeCode="SBML_FOO_FOORATE" hasListOf="false"
  baseClass="Assignment" abstract="false" elementName="rate"/>
<element name="FooAssignment" typeCode="SBML_FOO_FOOASSIGNMENT"
  hasListOf="false" baseClass="Assignment" abstract="false"
  elementName="assignment"/>
</elements>
<plugins>
  <plugin extensionPoint="Reaction">
    <references>
      <reference name="FooKineticLaw"/>
    </references>
  </plugin>
  <plugin extensionPoint="Model">
    <attributes>
      <attribute name="useFoo" required="true" type="boolean" abstract="false"/>
    </attributes>
  </plugin>
  <plugin typecode="SBML_QUAL_TRANSITION" package="qual"
    extensionPoint="Transition">
    <references>
      <reference name="ListOfFooRules"/>
    </references>
  </plugin>
  <plugin extensionPoint="Species">
    <references>
      <reference name="Extra"/>
    </references>
  </plugin>
</plugins>
<enums>
  <enum name="Sign">
    <enumValues>
      <enumValue name="SIGN_FOO_POSITIVE" value="positive"/>
    </enumValues>
  </enum>

```

```

    <enumValue name="SIGN_FOO_NEGATIVE" value="negative"/>
    <enumValue name="SIGN_FOO_NEUTRAL" value="neutral"/>
  </enumValues>
</enum>
</enums>
<mappings>
  <mapping name="Transition" package="qual"/>
  <mapping name="QualitativeSpecies" package="qual"/>
</mappings>
</pkgVersion>
</versions>
</package>

```

5 Appendix B

Data types recognized by Deviser Edit and the corresponding C++ data type.

Deviser Edit attribute Type	Corresponding C++ data type
string	std::string
boolean	bool
double	double
integer	int
unsigned integer	unsigned int
positive integer	unsigned int
non-negative integer	unsigned int
ID	std::string
SId	std::string
SIdRef	std::string
UnitSId	std::string
UnitSIdRef	std::string

6 References

- [1] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Mahler and F. Yergeau, "Extensible markup language (XML) 1.0 3rd edition," 4 February 2004. [Online]. Available: <http://www.w3.org/TR/2004/REC-xml-20040204/>.
- [2] B. J. Bornstein, S. M. Keating, A. Jouraku and M. Hucka, "LibSBML: An API Library for SBML," *Bioinformatics*, pp. 880-881, 2008.

- [3] “TeX,” [Online]. Available: <http://en.wikipedia.org/wiki/TeX>.
- [4] “Unified Modeling Language™ (UML®) Resource Page,” [Online]. Available: <http://www.uml.org/>.
- [5] M. Hucka, F. Bergmann, S. Hoops, S. Keating, S. Sahle, J. Schaff, L. Smith and D. Wilkinson, “The Systems Biology Markup Language (SBML): Language Specification for Level 3 Version 1 Core,” 2010. [Online]. Available: <http://sbml.org/Documents/Specifications>.
- [6] “CMake,” [Online]. Available: <https://cmake.org/>.

TO DO: Change the L3 ref to the new published version