Deviser Documentation

Release 1.0

SBML Team

Jan 30, 2018

CONTENTS

CHAPTER

ONE

BACKGROUND

The Systems Biology Markup Language (SBML) provides a *lingua franca* for respresenting computational models in a machine readable format. This faciliates both the exchange of models between different software platforms and the re-use of these models for subsequent research.

1.1 SBML Level 3 and Packages

SBML Level 3 (L3) is being developed using a modular approach. A core specification (SBML L3 Core) captures the basics of a model, with optional additional packages providing extensions relevant to particular areas of modeling. There are several L3 packages at various stages of development encompassing such things as flux balance modeling and models involving spatial components. An up to date list of these packages can be found at the sbml.org website.

1.2 LibSBML

LibSBML is a free open-source API library for working with SBML. Support for L3 packages in libSBML is also provided in a modular fashion using CMake (a system for managing the build process) to allow users to configure and build with options to enable/disable the various L3 packages. A wide range of programming languages are supported by libSBML which uses SWIG (an interface compiler) to create the different interfaces necessary. Options to configure and build these interfaces are also available using the CMake system.

As a general note it is worth pointing out that a release of libSBML does include targeted installers for several languages and operating systems; although for users of Deviser these will not be relevant. Further information is available from the libSBML page.

A complete L3 package for libSBML includes the C++ and C code, as well as all the necessary SWIG and CMake files to allow the package to be integrated into an existing libSBML build and have all the language interfaces available.

1.3 Development of L3 Packages

Each SBML L3 Package that has been proposed goes through a period of development by a focussed group of people (known as the package working group or PWG) who are either interested and/or actively trying to use the additional functionality provided by the package. This leads to a number of iterations of the package design as the PWG work towards a specification that is agreed upon and meets the requirements of the parties involved.

Deviser has been developed to provide a fast, efficient means of changing both API code and specification documentation to facilitate the update of documentation and implementation necessary as the package itself evolves.

CHAPTER

TWO

GETTING STARTED

Deviser allows a user to define an SBML Level 3 (L3) package and produce libSBML code that can be integrated with the existing libSBML source tree to add libSBML support for the package.

Deviser can also use the package description to create basic specification files for the package based on the LaTeX style sbmlpkgspec for SBML Level 3 package documentation.

The creation of both code and TeX files can be controlled via a command-line interface.

The full Deviser package includes Deviser Edit; a graphical user interface (GUI) for defining the necessary details of the package being defined. The GUI also provides a convenient way of invoking the Deviser functionality to create code or TeX files. Other capabilities of Deviser Edit (view UML, create PDF, integrate and test code) further aid the process of defining an SBML L3 package and testing the code generated by Deviser.

2.1 Functionality

The Deviser Edit tool outputs an XML definition of the SBML L3 package you have defined. This definition is then used by Deviser to generate either code or TeX files. Samples of definitions output by Deviser Edit are included in the *devisersamples* directory within the Deviser distribution, and the full text of the XML definition created by the examples used in this manual is given in *Appendix A: Example package description*. Note it is not necessary to use Deviser Edit to produce this XML definition; the definition can be produced by hand but should match the format used in the samples provided.

2.1.1 Using command line

A command line interface is available to produce code files and TeX files. This invokes the deviser.py function that is available in the *devisergenerator* directory of the Deviser distribution.

This is invoked as:

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

This program takes as input a Deviser XML file and either

- generates libSBML code for the package, when the comman-line argument is –generate or -g
- generates the LaTeX files for the package specification, when the command-line argument is –latex or -l

2.1.2 Using Deviser Edit

The basic Deviser Edit tool allows a user to define an SBML L3 package by filling in requested information. Once defined, the following functions are available:

1. Generate libSBML code for the package.
2. Integrate and test the package code within libSBML.
3. Generate TeX files for basic specification of the package.
4. Generate a PDF of a basic specification document for the package.
5. Create and view a UML diagram of the package specification.

2.2 Prerequisites

The main functionality (generating code or TeX files) is invoked using a Python Interpreter (see Section 4). This release of Deviser has been tested with Python 2.6, 2.7, 3.3 and 3.4. Other functions provided by Deviser Edit may require further software. The prerequisites for each function are listed below.

2.2.1 Integration and testing

Deviser Edit allows you to automatically integrate and test your newly created code with libSBML. In order to do this you will need to have the following additional software installed/available:

- CMake
- SWIG
- a C++ compiler
- libSBML source code
- libSBML dependencies source code

2.2.2 Basic documentation

In order to create a PDF document of the package specification it will be necessary to have the following installed/available:

- pdflatex
- sbmlpkgspec (the LaTeX style for SBML L3 package documentation)

Note that using pdflatex may involve installing other packages depending on the operating system.

On Windows we successfully used MiKTEX (MiKTeX (pronounced mick-tech) is an up-to-date implementation of TeX/LaTeX and related programs for Windows) see Section 2.2.5.

2.2.3 UML diagrams

Deviser allows you to create and view very basic UML diagrams based on the classes specified. Since it uses the free yUML (http://yuml.me/) web service, you will be necessary to be connected to the internet to create UML diagrams. The Deviser Edit tool requires the OpenSSL library (https://www.openssl.org/) to access yUML.

2.2.4 Available bundles

Installers and source code bundles are available from https://github.com/sbmlteam/deviser/releases .

2.2.5 Useful links

1. libSBML source code (latest release): https://sourceforge.net/projects/sbml/files/libsbml/5.16.0/stable/libSBML-5.16.0-core-src.tar.gz/download **Note that there may be a later release available.**
2. libSBML source code (latest code): https://sourceforge.net/p/sbml/code/HEAD/tree/trunk/libsbml/
3. libSBML dependencies: https://github.com/sbmlteam/libSBML-dependencies
4. SBML package specification LaTeX template files: https://sourceforge.net/projects/sbml/files/specifications/tex/sbmlpkgspec-1.6.0.tar.gz/download
5. CMake: http://www.cmake.org/
6. SWIG: http://www.swig.org/
7. MiKTEX: http://miktex.org/

2.3 Setting up the Deviser Edit tool

Before you can generate code/files/diagrams it is necessary to tell the Deviser Edit tool where it will find things on your system.

Data types not fully supported by Deviser can still be used as types for attributes by adding them via the Deviser Edit tool.

Deviser Edit highlights required fields. The settings provide the option to customize the color used for this highlighting.

Setting these values is done by selecting

Edit->Edit Preferences (Preferences on Mac OSX)

and choosing the **Deviser Settings**, the **User Defined Types** or the **Misc** tab.

Settings used here will be saved and persist between sessions.

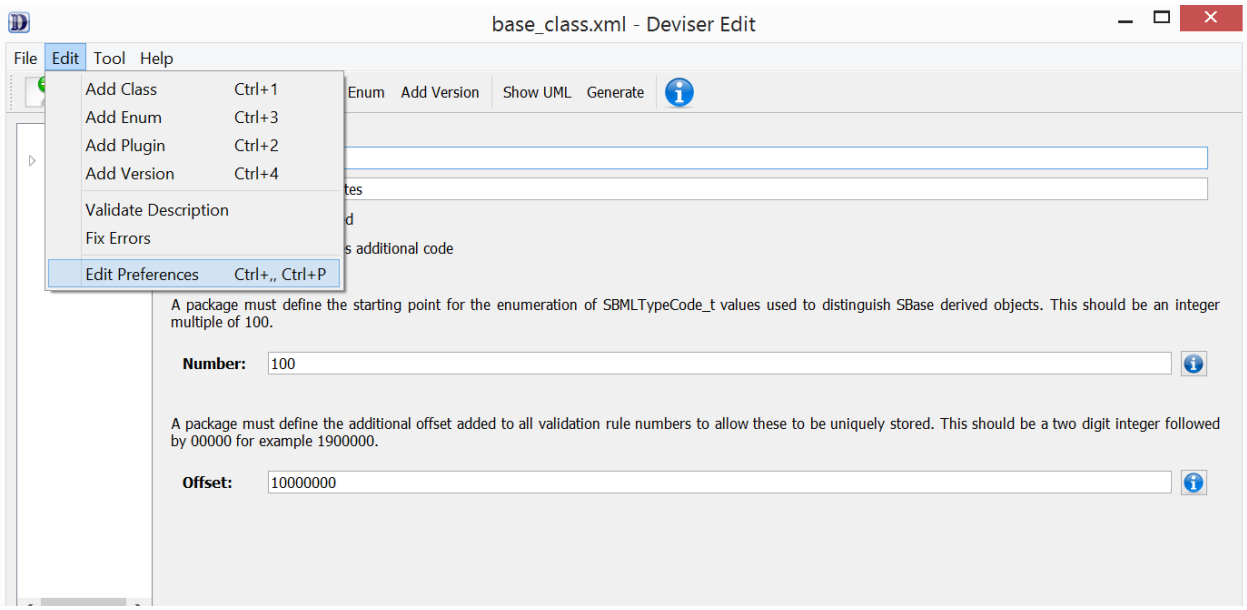

Figure 2.1: The *Edit* menu.

2.3.1 Deviser Settings

It is not necessary to fill in all the fields if you do not intend to use all the functionality. With nothing entered by the user, Deviser Edit will save the XML description file and it would also be possible to generate UML diagrams.

Enter or browse to the location of the files requested by each field.

To generate code or LaTeX source the fields requested are:

- **Python Interpreter**. This is the location of the Python executable, which is necessary to invoke any functionality of Deviser.
- **Deviser repository**. This is the location of the directory containing the scripts used to generate code/LaTeX. If you installed Deviser/Deviser Edit using one of the installers provided this will be the **deviser** directory located in your installation path. This directory may also be obtained directly from our GitHub repository [https://github.com/sbmlteam/deviser].
- **Default output dir**. The location where you want generated files to be written.

To compile the LaTeX source the fields requested are:

- **sbmlpkgspec dir**. The location of the SBML documentation style files sbmlpkgspec.
- **TeX Bin dir**. The location of the TeX executables.

To integrate and compile the code with libSBML the fields requested are:

- **Compiler**. The location of the C++ compiler to be used when building libSBML to include the integrated package code.
- **libSBML Source dir**. The location of the libSBML source tree (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 used to build libSBML. Deviser Edit offers an option to build them if this should be necessary.

- **CMake executable**. The location of the CMake executable, necessary if you want Deviser Edit to integrate your package code into an existing libSBML build. Note Deviser/Deviser Edit do not support any build system other than CMake.
- **SWIG executable**. The location of the SWIG executable. When Deviser Edit invokes the compilation of libSBML with the newly integrated package it configures and builds the Python binding, in addition to the C++ library. SWIG is necessary for this. It is hoped in future to add a facility to directly test the build using a Python interface.

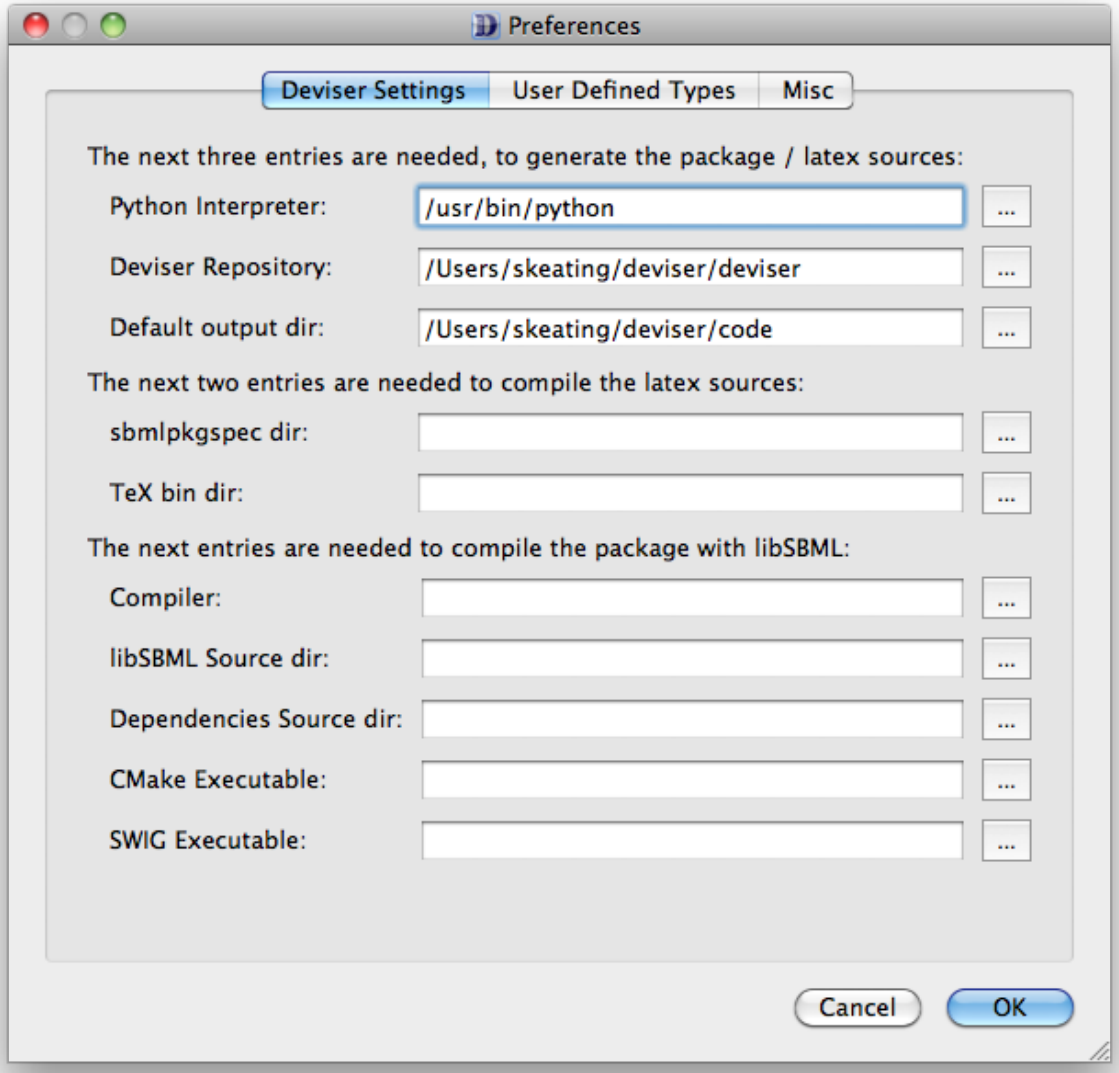

Figure 2.2: The Deviser Settings tab from the Preferences window; adding information about other required software.

Figure 2.2 illustrates a case where you want to be able to generate code and TeX files but do not want to generate a PDF or integrate the code automatically.

2.3.2 User Defined Types

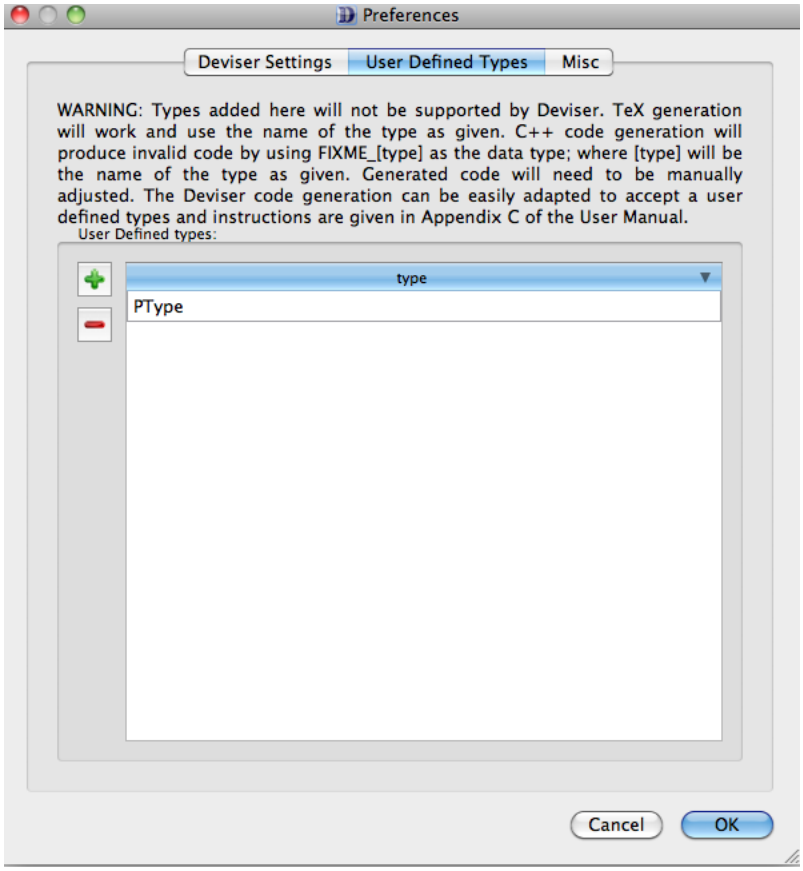

Figure 2.3: The User Defined Types tab from the Preferences window; adding information about other data types.

To generate functionable code, all data types used in the package must be known to Deviser. Deviser has built-in support for the predefined types declared by SBML L3 Core (see Section 3.3.3.3) but various SBML L3 packages may introduce unsupported types. To enable Deviser to handle these new types, the new types should be added using the **User Defined Types** tab on the **Preferences** window. Use the '+' button to add a new type and then adjust the name as required. To remove a type, highlight it and press the '-' button. The new type will then appear in the dropdown list of Types available when you enter attribute information. (For example, Figure 2.3 demonstrates adding a type called *PType*.)

Note any C++ code produced will use the type 'FIXME_[type]' where [type] is the type name entered by the user (see Listing 2.1). The resulting code will not compile and will need to be manually adjusted.

Listing 2.1: Code generated for an attribute named 'P' of user defined type 'PType'.

```
FIXME_PType getP() const;

int setP(FIXME_PType p);
```

2.3.3 Misc

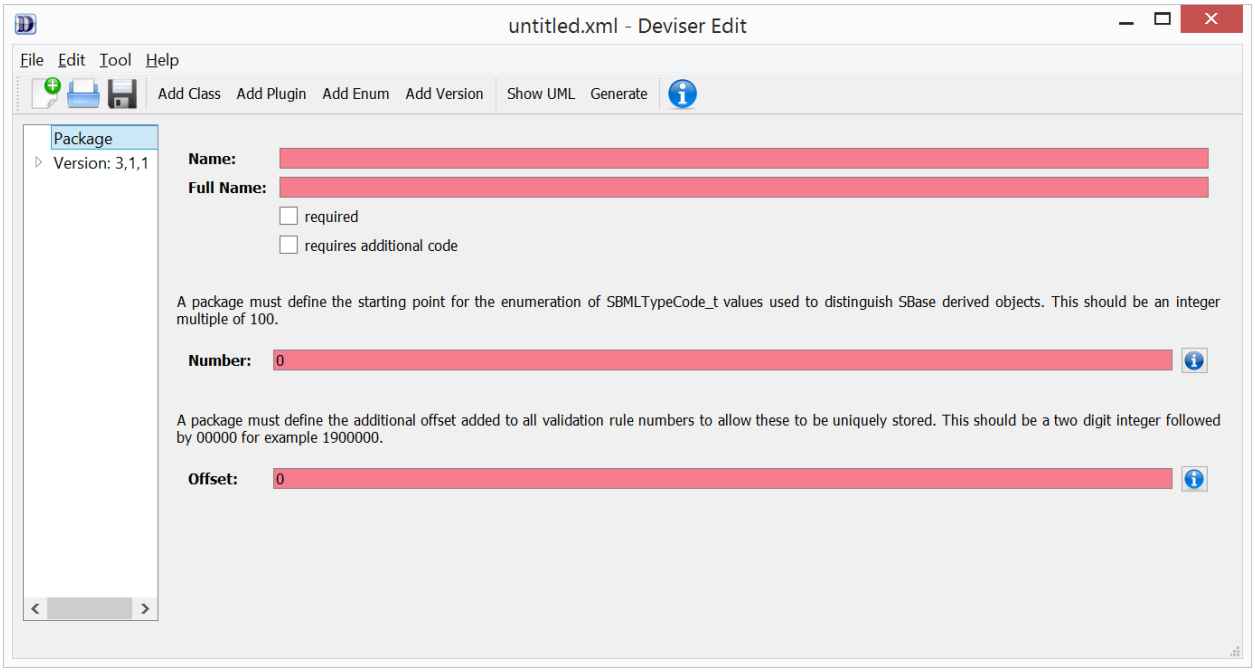

Figure 2.4: A window from Deviser Edit showing missing required information boxes highlighted.

When the information in a text box is required, Deviser Edit highlights the field that has missing information (see Figure 2.4). Users can customize the color used to highlight missing field using the **Misc** tab on the **Preferences** window. We leave experimenting with that to the user.

CHAPTER

THREE

DEFINING AN SBML LEVEL 3 PACKAGE

SBML Level 3 is a modular language, with a core comprising a complete format that stands alone. Level 3 packages may be added to this core to provide additional, optional features. Deviser provides a unique way of defining the package that facilitates the creation 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 panels 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.

3.1 Define the general package information

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

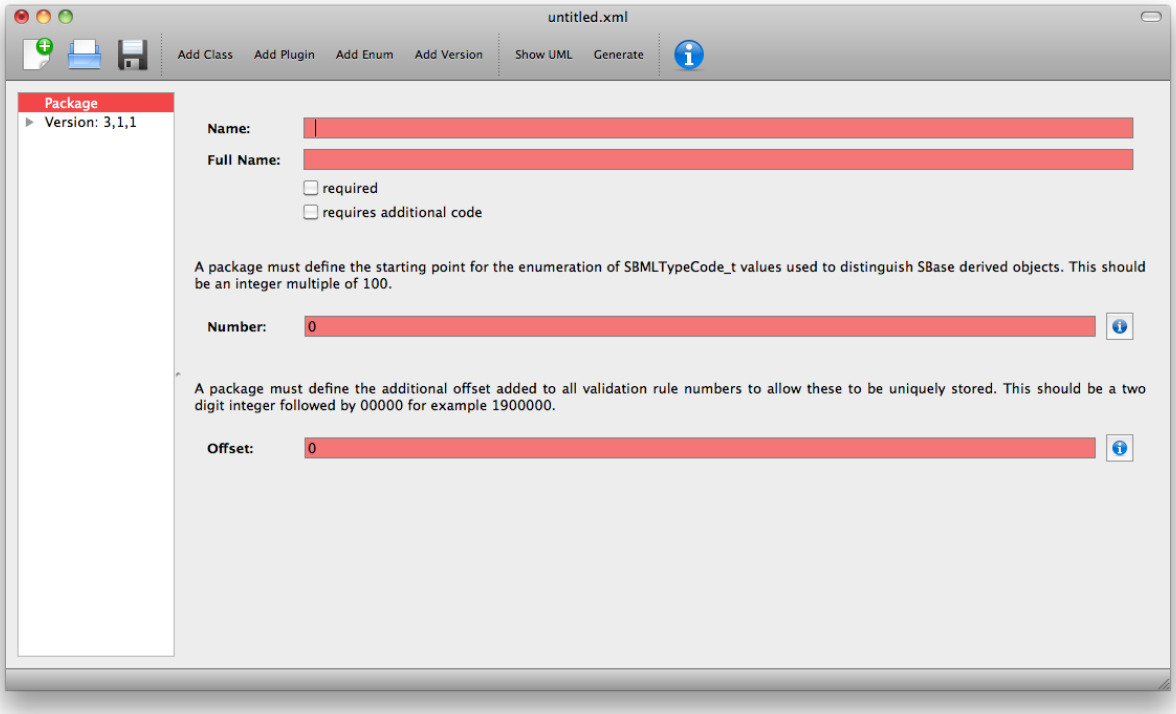

Figure 3.1: The 'Package' panel with no information.

Fields colored red are mandatory and will remain red until a valid value has been entered.

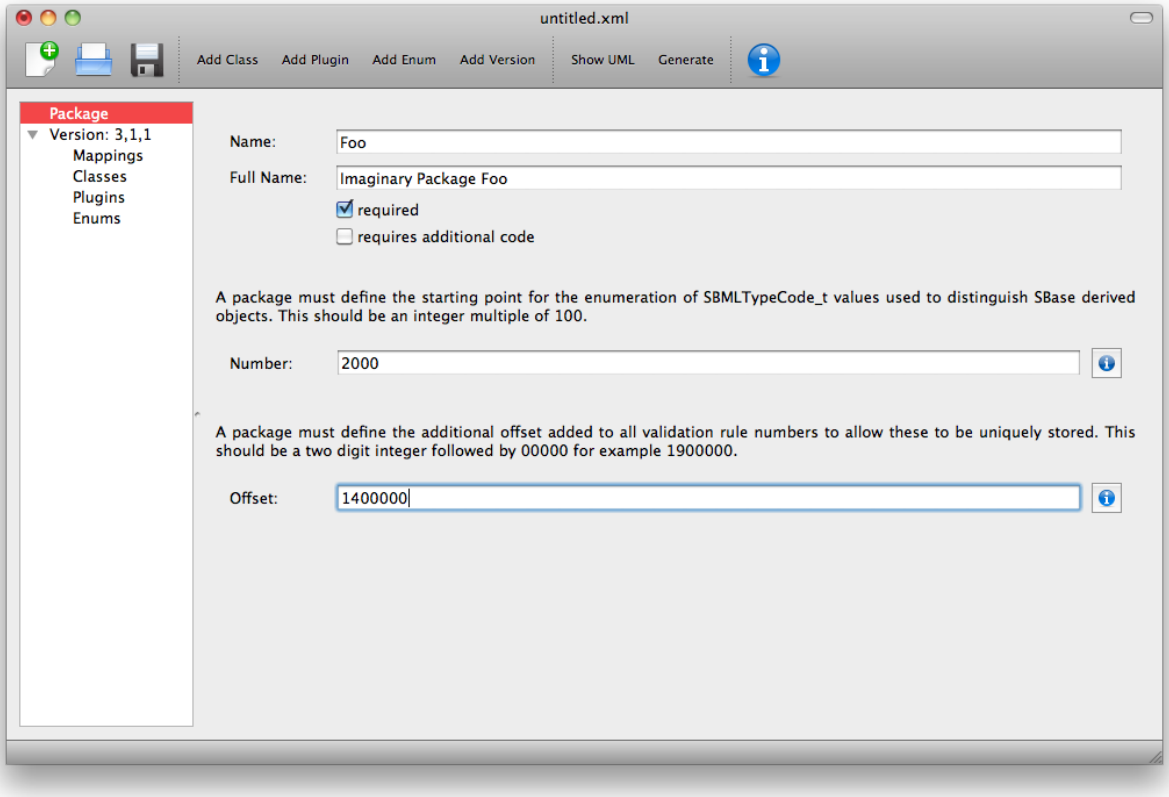

Figure 3.2: The 'Package' panel - Illustrating the first step in defining the 'foo' package.

3.1.1 The Package Name field

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

3.1.2 The Package Full Name field

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

3.1.3 The Package required checkbox

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 Listing 3.1).

Listing 3.1: The <sbml> element showing the required attribute.'.

```
<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">
```

3.1.4 The Package requires additional code checkbox

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.

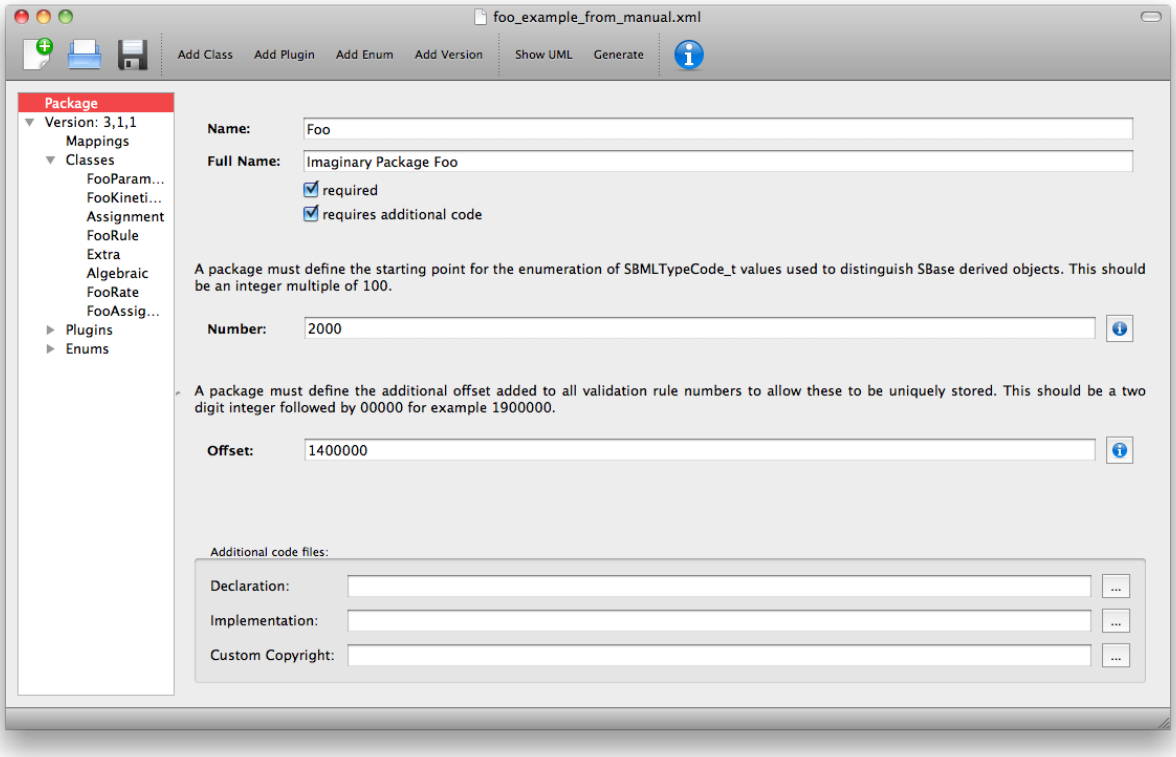

Figure 3.3: The 'requires additional code' check box.

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.

3.1.4.1 Custom copyright

The 'Custom copyright' field allows user to specify a text file containing a custom copyright that will be added to all files in addition to the SBML Team copyright. It is possible to completely replace the copyright with appropriate acknowledgement. See Section 8 .

3.1.5 The Package Number field

The **Number** field is the starting point for the enumeration of the typecodes for this package. This is a required field.

Clicking the information button will generate a pop-up window with information regarding the Number and Offset values used by existing L3 packages (Figure 3.4).

3.1.6 The Package Offset field

The **Offset** field is the number added to the validation rules given in the specification to allow this to be identified uniquely in code. This is a required field.

Clicking the information button will generate a pop-up window with information regarding the Number and Offset values used by existing L3 packages (Figure 3.4).

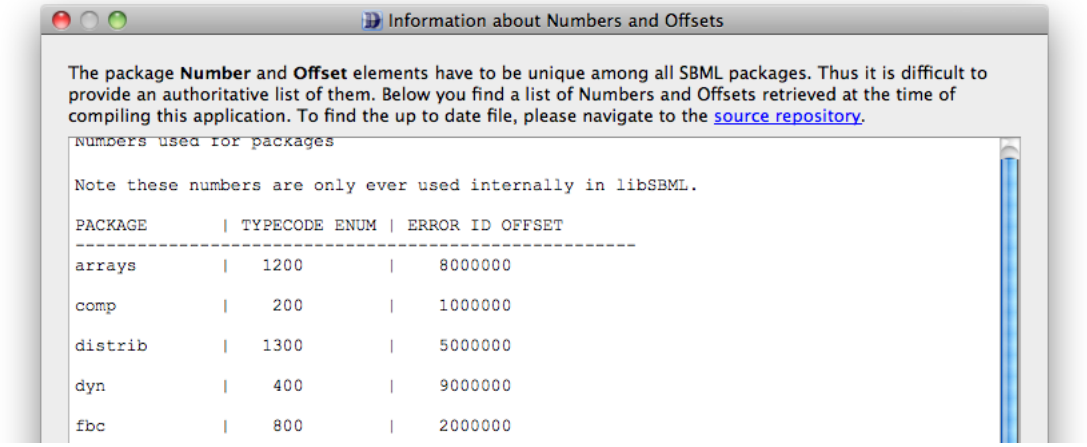

Figure 3.4: The Numbers and Offsets information box.

3.2 Add the version number

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

Fill in the core level and version and package version numbers. These default to **Core level** '3' and **Core version** '1' as SBML L3V1 is the only official SBML Level 3 version at the time of writing. The package version (**Pkg Version**) defaults to '1'. Using Deviser to specify more than one version of a package is discussed in Section 3.8.

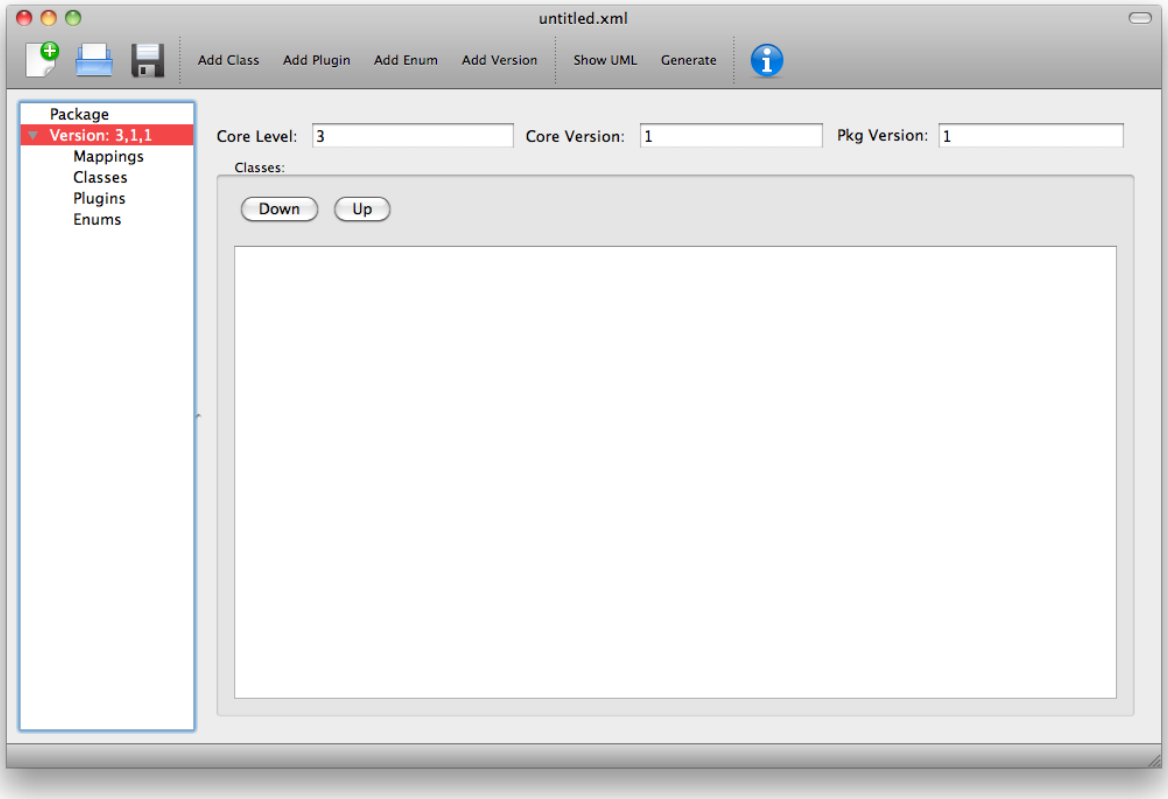

Figure 3.5: The 'Version' panel.

Once classes have been specified they will appear on this panel (see Figure 3.28). The order in which they're listed dictates the order in which the generation code processes the classes. This list can be rearranged, which is helpful in ensuring documentation is written in a specific sequence.

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 panel will be automatically populated when the description is complete.

3.3 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.

3.3.1 What is a 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. Deviser Edit uses '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 object.

SBML snippet 1: A SBML Level 3 Core ListOfReactions element.

```
<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>
```

Figure 3.6 shows a snapshot of libSBML class hierarchy corresponding to *SBML snippet 1 above*. Note the correspondence of names and the getXYZ functions etc.

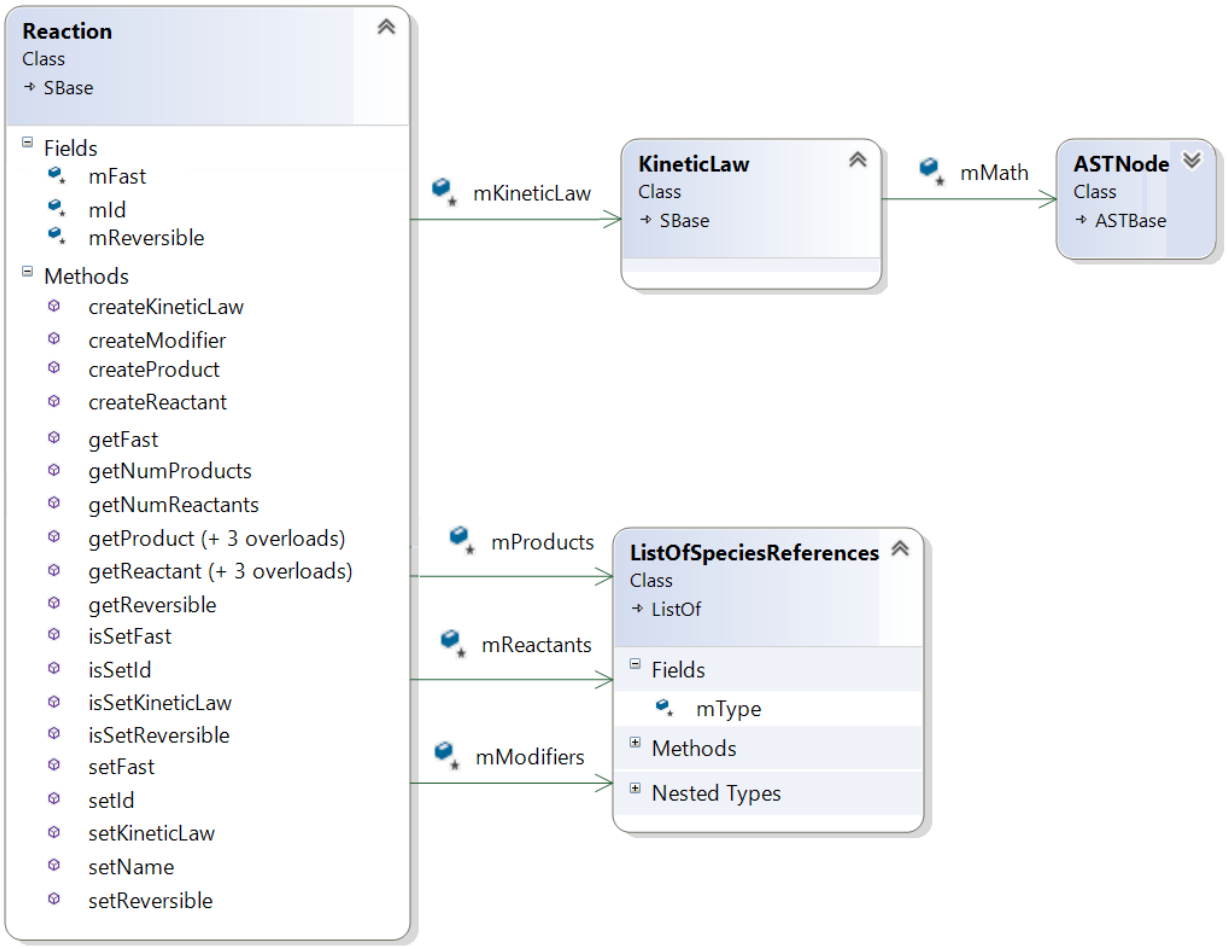

Figure 3.6: Snapshot of part of libSBML class hierarchy. The 'Reaction' class has fields mFast, mId and mReversible which correspond to the attributes of the <reaction> element within the SBML. It has a child member of type 'KineticLaw' which itself has a child member of type 'ASTNode'. These capture the information contained within the <kineticLaw> element and it's subelement <math>. The list of Methods for the 'Reaction' class show the correspondance between the element/attribute names used in the SBML and the function names used by libSBML.

3.3.2 General class information

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 3.3.6).

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

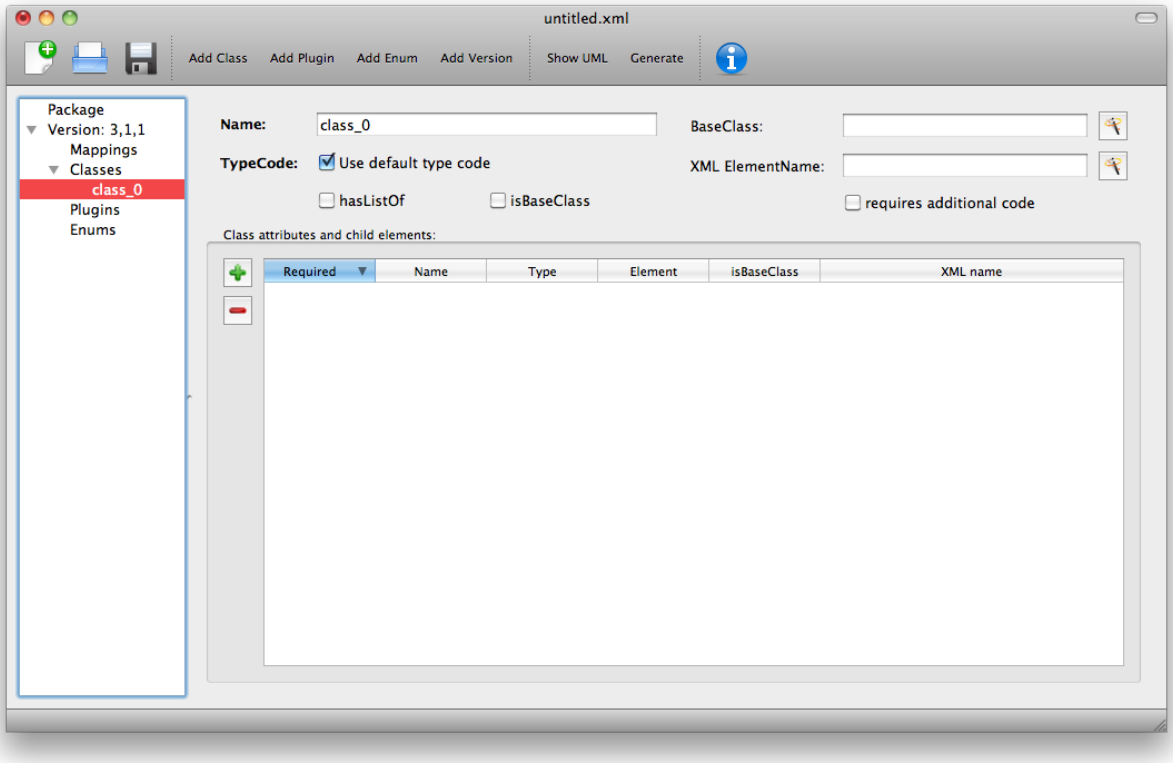

Figure 3.7: The 'Class' panel

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.

Fields that are madatory will remain red until populated.

3.3.2.1 The Class Name field

The **Name** field is the name of the class that will be used by the code generation (see Section 3.3.2.4). This field is required and must be populated.

3.3.2.2 The Class BaseClass field

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. Code generated using Deviser assumes that all classes ultimately derive from SBase as it uses the generic SBase code for reading and writing to and from an XML document. Therefore, leaving the field empty implies that the base class is SBase.

3.3.2.3 The Class TypeCode checkbox and field

The **TypeCode** is a value that will be used in an enumeration of the types for this package. This is used in code to distinguish between SBase derived classes. Initially Deviser Edit displays a check box with the caption **Use default type code**. Leaving this checked implies that it is acceptable for Deviser to use a default typecode of SBML_PACKAGE_CLASS where PACKAGE is the short package name given and CLASS is the name field for this class.

Unchecking the box will reveal a text box with the 'wand' icon. This field is mandatory. The user must enter a value. Note that using the 'wand' will populate the field with the default typecode SBML_PACKAGE_CLASS as above.

3.3.2.4 The Class XML ElementName field

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 3.13 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.

3.3.2.5 The Class hasListOf checkbox

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.

When the **hasListOf** checkbox is selected further information is required and several additional fields appear. These are detailed in *a later section*.

Note leaving this box unchecked means that the corresponding ListOf class has not been defined and any further references to such a ListOf class are invalid.

3.3.2.6 The Class isBaseClass checkbox

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

When the **isBaseClass** checkbox is selected further information is required and an additional table appears. The information required is detailed in the *instantiations section*.

3.3.2.7 The Class requires additional code checkbox

The **requires additional code** checkbox can be used to indicate that there is additional code that will be required by this class, 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 contents of the supplied header file supplied will be included at the end of the class declaration, with the contents of the supplied implementation file being included with the implementation file for this class. 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.

3.3.2.8 The Class attributes and child elements table

This 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. The possible entries are explained in detail in the Section *Adding attributes and child elements*.

3.3.3 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 3.7. These fields are identical to those in the **ListOf attributes** table that appears when the **hasListOf** checkbox is checked.

3.3.3.1 The attribute/element Required checkbox

The **Required** field indicates whether the attribute or child element is mandatory in terms of the SBML definition.

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.

3.3.3.2 The attribute/element Name field

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 Name supplied.

Note to avoid compilation issues with libSBML SBase objects the following attribute/element names should not be used:

> annotation, attribute, column, cvterms, level, line, metaid, model, modelhistory, notes, packageversion, prefix, sboterm, uri, version

3.3.3.3 The attribute/element Type field

The **Type** field gives the type of the attribute or child. This is a drop-down list giving the types that are supported by Deviser.

The recognized types for an attribute are the datatypes allowed by SBML. These are:

> string, boolean, double, integer, unsigned integer, positive integer,
>
> non-negative integer, ID, IDREF, SId, SIdRef, UnitSId, UnitSIdRef

and additionally Deviser Edit uses:

> array, enum, element, lo_element, inline_lo_element, vector.

Appendix B: Data Types lists the SBML types with their corresponding C++ data type. The additional types accepted by Deviser are explained in detail in the sections below.

It should be noted that the 'Type' used for each attribute/child element determines the code generated functions that will be produced. For attributes with one of the accepted SBML types the functions produced are shown below:

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

       where
           [Type]
                 is a placeholder for the appropriate C++ type
           [Name]
                 is a placenolder for the name of the attribute
```

Deviser Edit provides the ability to add types to the drop-down list which will facilitate using types that are not supported. Section *User Defined Types* provides information on how to use this facility.

3.3.3.3.1 Attribute/child element type 'array'

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 below).

SBML snippet 2: An SBML Level 3 Spatial SampledField element.

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

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

Required true/false as appropriate

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

Type array

Element integer (the numeric type of the data)

Figure 3.8 shows the Deviser Edit entry for the SampledField class. Note it also includes an attribute to record the length of the array. This proved useful when using this sort of construct.

The code generator produces the following code for an attribute of type 'array':

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

      where
            [Type]
                  is a placeholder for the appropriate C++ type
            [Name]
                  is a placeholder for the attribute name
                  given to the array
```

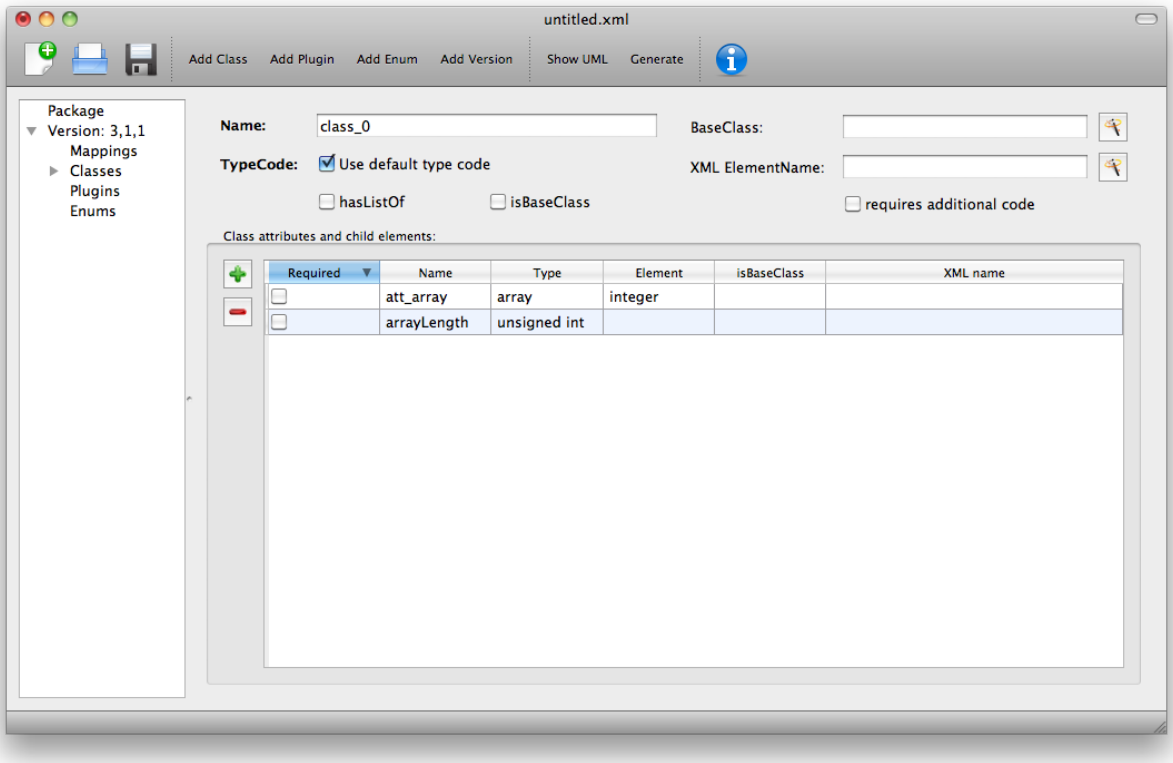

Figure 3.8: Attributes of the SampledField class.

3.3.3.3.2 Attribute/child element type 'enum'

An attribute can have a type corresponding to an enumeration type defined within the package. In this case the attribute has type 'enum' and would be defined as an entry with the following field values:

Required true/false as appropriate

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

Type enum

Element the name of the enumeration

The enumeration is declared fully by adding an enumeration to the package description (see Section 3.5).

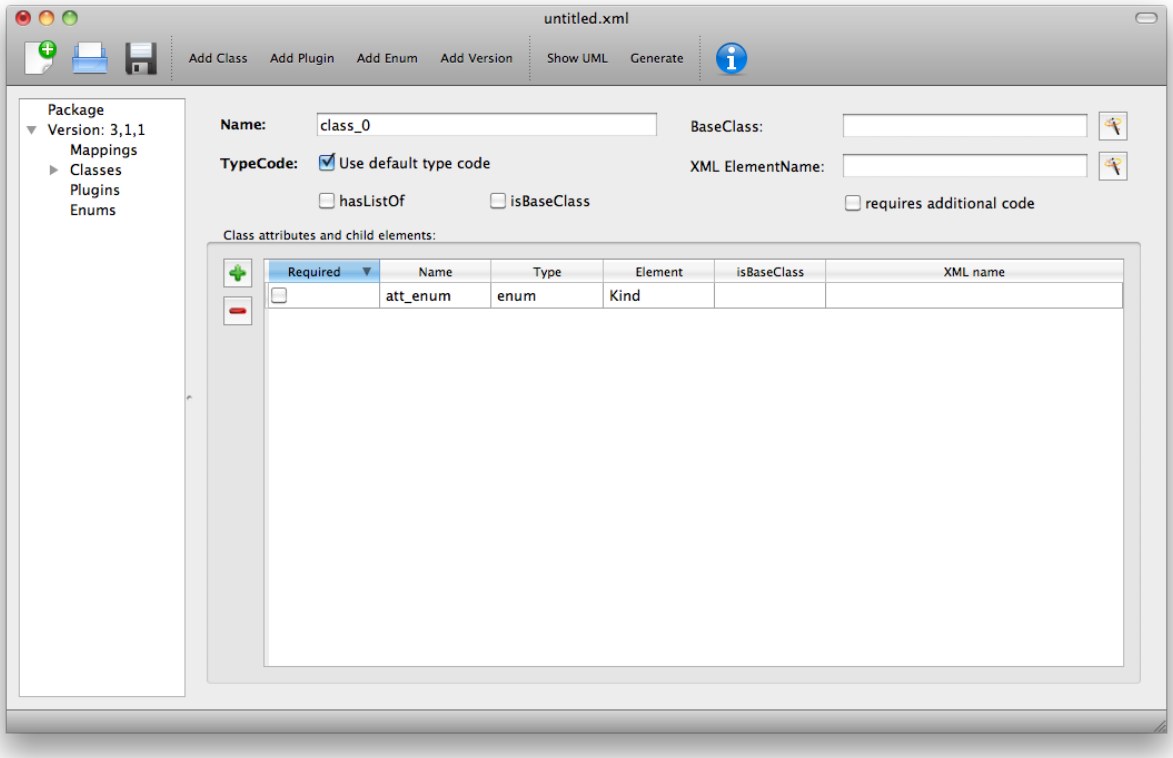

Figure 3.9: Attribute of type 'enum'

The code generator produces the following code for an attribute of type 'enum':

```
[EnumType_t]     get[Name]           ()
std::string&     get{Name}AsString   ()
bool             isSet[Name]         ()
int              set[Name]           ([EnumType_t] value)
int              set[Name]           (std::string& value)
int              unset[Name]         ()

      where
            [EnumType_t]
                  is a placeholder for the C++ type of the enumeration
            [Name]
                  is a placeholder for the attribute name
```

It should be noted that libSBML convention uses an '_t' for all enumeration types. This will be added by Deviser if necessary.

3.3.3.3.3 Attribute/child element type 'element'

This type can be used to define a child element of the defining class. The type 'element' refers to a single instance of another class that is a child of the defining class.

SBML snippet 3: An SBML Level 3 Core Event element.

```
<event id="event1" name="event1" useValuesFromTriggerTime="true">
    <trigger initialValue="true" persistent="true">
        <math xmlns="http://www.w3.org/1998/Math/MathML">
            <apply>
                <lt/>
                <ci> S1 </ci>
                <cn> 0.1 </cn>
            </apply>
        </math>
    </trigger>
    <listOfEventAssignments>
        <eventAssignment variable="S1">
            <math xmlns="http://www.w3.org/1998/Math/MathML">
               <cn type="integer"> 1 </cn>
            </math>
        </eventAssignment>
    </listOfEventAssignments>
</event>
```

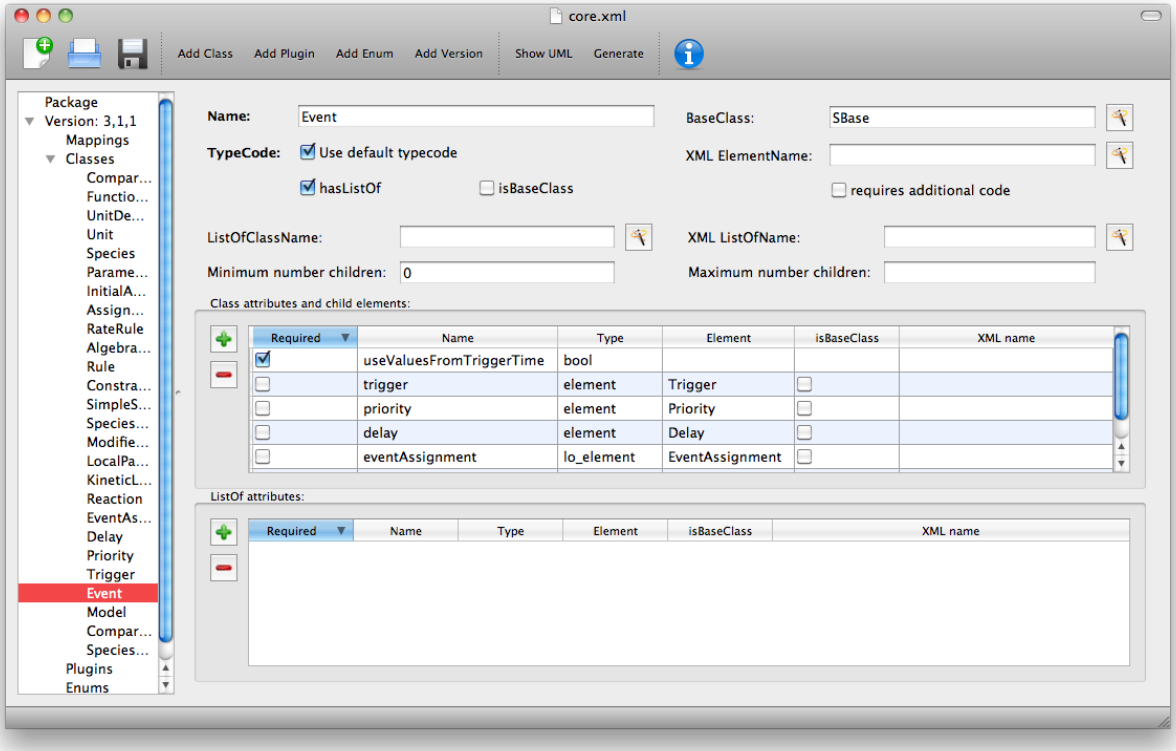

Figure 3.10: Class with child elements of type 'element' and 'lo_element'

SBML snippet 3 above shows an Event from SBML Level 3 Core which has a Trigger child element. The Deviser Edit entries in the Attributes and child element table for the Event class are shown in Figure 3.10.

Table 1 gives examples of the expected XML and the functions produced for type 'element'.

3.3.3.3.4 Attribute/child element type 'lo_element'

This type is used to define a child that is an instance of a 'ListOf' class. In the *SBML Event* shown the listOfEventAssignments is defined as a child of type 'lo_element' (see Figure 3.10). *Table 1* details the corresponding XML output and functions generated.

Table 1: The XML output and generated functions for each of the Deviser child element types.

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

3.3.3.3.5 Attribute/child element 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.

3.3.3.3.6 Attribute/child element type 'vector'

The 'vector' type refers to an XML element that may contain text that represents a list of values of a particular type. This is similar to the 'array' type but will use the C++ std::vector class as a type.

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

Required true/false as appropriate

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

Type vector

Element integer (the type of the data)

The code generator produces the following code for an attribute of type 'vector':

```
std::vector<[Type]>&  get[Name]      ()
bool                  has[Name]s     ()
unsigned int          getNum[Name]s  ()
int                   set[Name]      (std::vector<[Type]>& value)
int                   add[Name]      ([Type] value)
int                   clear[Name]s   ()

      where
            [Type]
                  is a placeholder for the appropriate C++ type
            [Name]
                  is a placeholder for the attribute name
                  given to the vector
```

3.3.3.4 The attribute/element Element field

The **Element** field provides additional information depending on the type of the attribute/child element being described. *Table 2* describes how and when this field should be populated. Note the 'name' of an element or object refers to the ClassName of the appropriate object.

Table 2: The expected entries in the 'Element' field depending on the 'Type'.

Type	Element field
array	type of data within the array
enum	The name of the enumeration
element	The class name of the element
lo_element	The class name of the element within the ListOf
inline_lo_element	The class name of the element
SIdRef	Comma separated list of the class name of multiple objects that can be referenced
Any other	blank

Note that Deviser does specifically recognize the elements ASTNode and XMLNode and treats them appropriately as elements that will contain either MathML or XML content respectively. Other class names that are listed are assumed to be parsed as classes belonging to libSBML; either those being defined by this package or ones defined in L3 core or other available L3 package code. Deviser Edit will prompt users for this information in the section on *Mappings*.

3.3.3.5 The attribute/element isBaseClass field

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 3.11 and Figure 3.12).

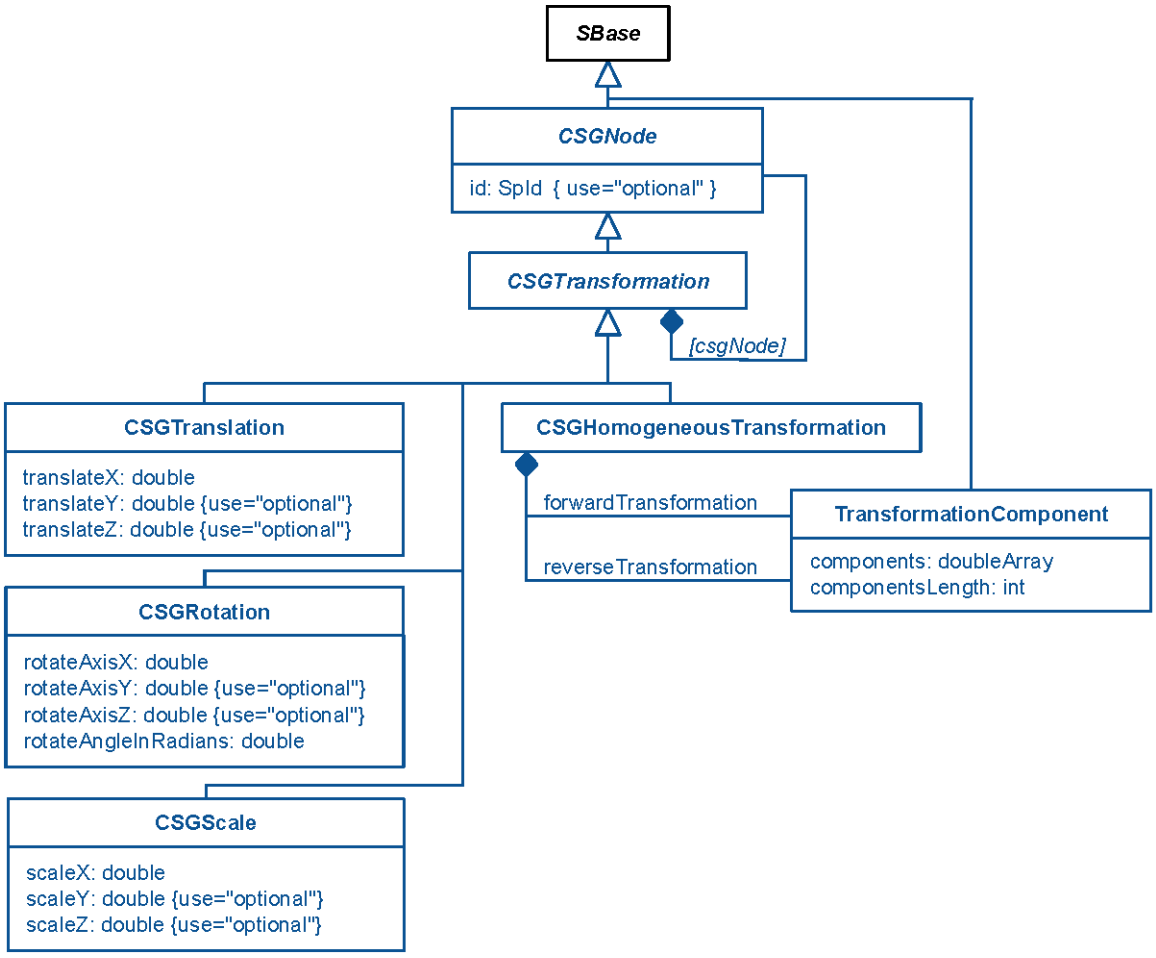

Figure 3.11: UML diagram of CSGTransformation from SBML L3 'spatial' package specification

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, instead of getting a 'createCSGNode()' function, you would get create functions for all the instantiations of the base class: createCSG-Transformation(); createCSGPrimitive() etc.

3.3.3.6 The attribute/element XML Name field

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.

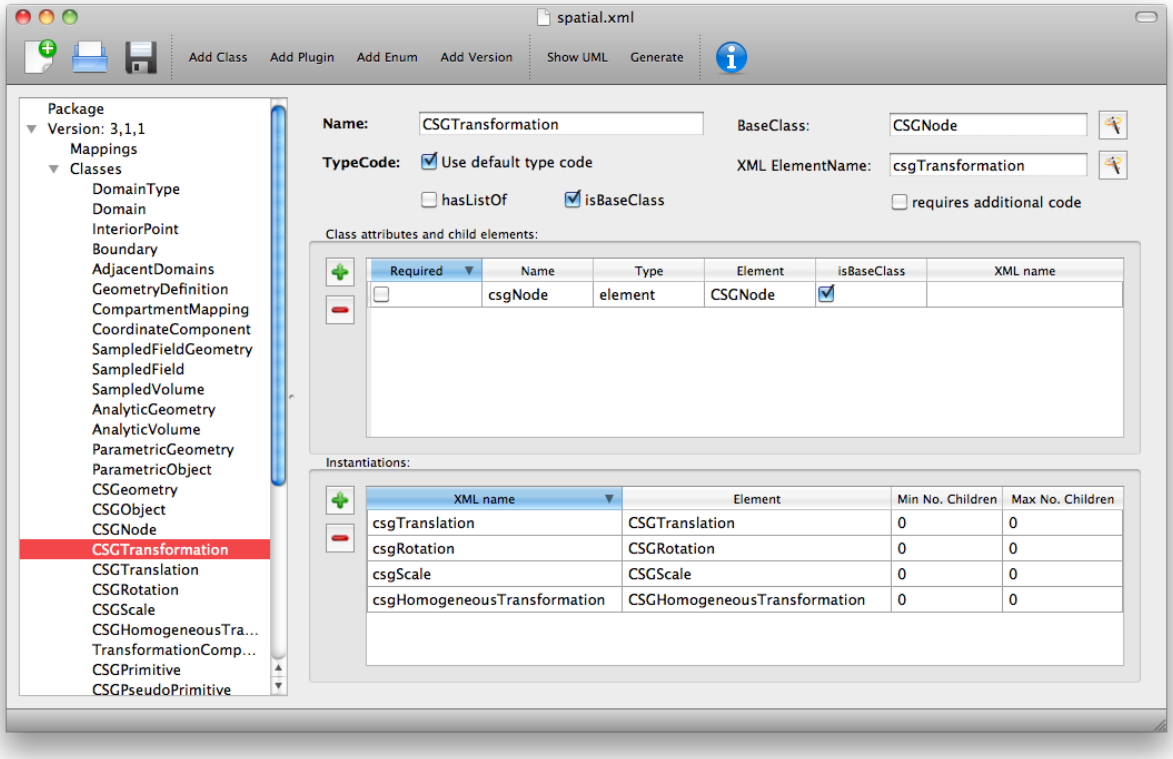

Figure 3.12: Deviser Edit description of CSGTransformation

3.3.4 Adding further ListOf information

When the **hasListOf** checkbox is selected four other fields appear.

3.3.4.1 The listof ListOfClassName field

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

3.3.4.2 The listof XML ListOfName field

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.

3.3.4.3 The listof Minimum number of children field

The **Minimum number of children** field is used to indicate the minimum number of child objects of type Bar a ListOfBars expects. In SBML L3V1 ListOf elements cannot be empty so code generation will treat the value as '1'. In SBML L3V2 a value of '0' indicates that a ListOfBars object may have no Bar children and a value of '1' indicates there must be at least one Bar child object present in the list.

3.3.4.4 The listof Maximum number of children field

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.

At present these values do not impact on the code generated for the classes. In future versions they will be applied as validation rules.

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. This table could also be used in the very rare case where a listOf element contains a child that is not the same type as the expected children. For example the Qualitative Models Package defines a listOfFunctionTerms that must contain one instance of a defaultTerm in addition to the functionTerm children.

3.3.5 Adding instantiations information

When the isBaseClass checkbox is selected 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.

3.3.5.1 The instantiations XML Name field

The **XML name** field specifies the XML name of the object. This is a required field and must have a value.

3.3.5.2 The instantiations Element field

The **Element** field specifies a class that will be derived from this base class. This is a required field and should be the name of a Class defined within the package.

3.3.5.3 The instantiations Min No. Children field

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

3.3.5.4 The instantiations Max No. Children field

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'.

3.3.6 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 some extent. It also means that each panel contains all the pertinent information for the Class being specified rather than this information being distributed across various panels in the GUI.

3.3.7 Example 1 - Adding a class with no containing ListOf

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

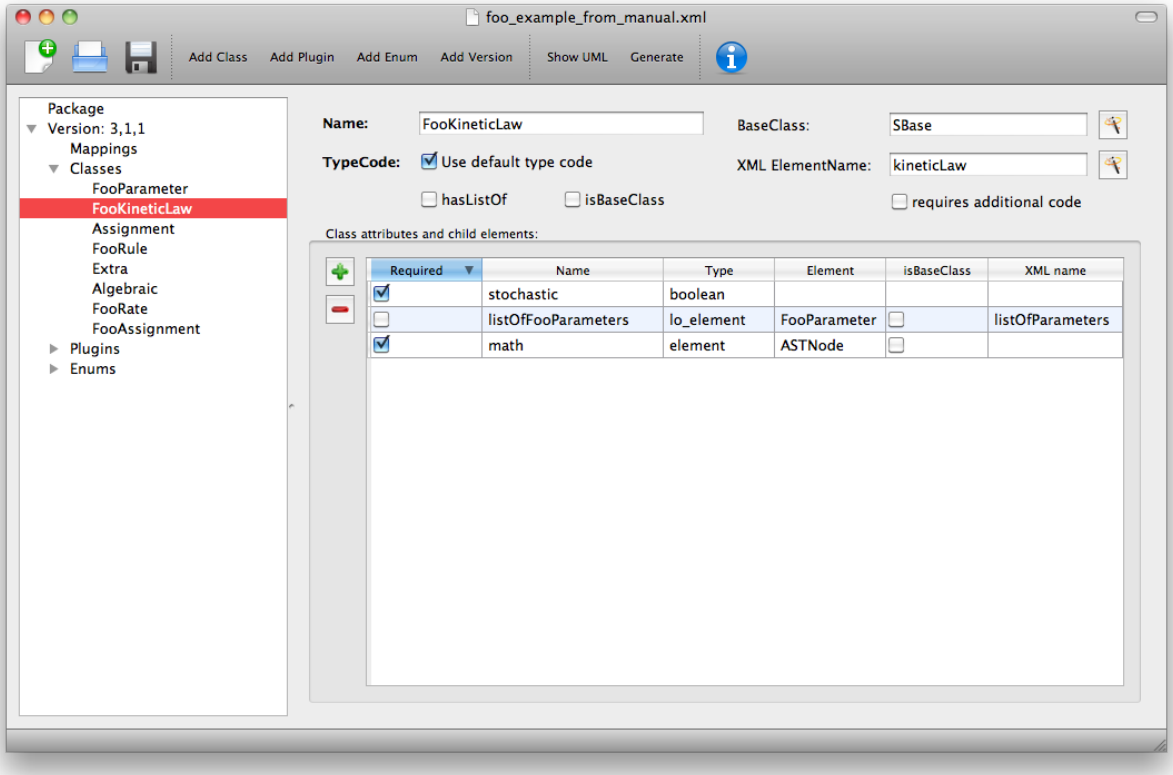

Figure 3.13: Defining the FooKineticLaw class.

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*'. *As mentioned above* Deviser does specifically recognize the elements ASTNode and XMLNode.

3.3.8 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*.

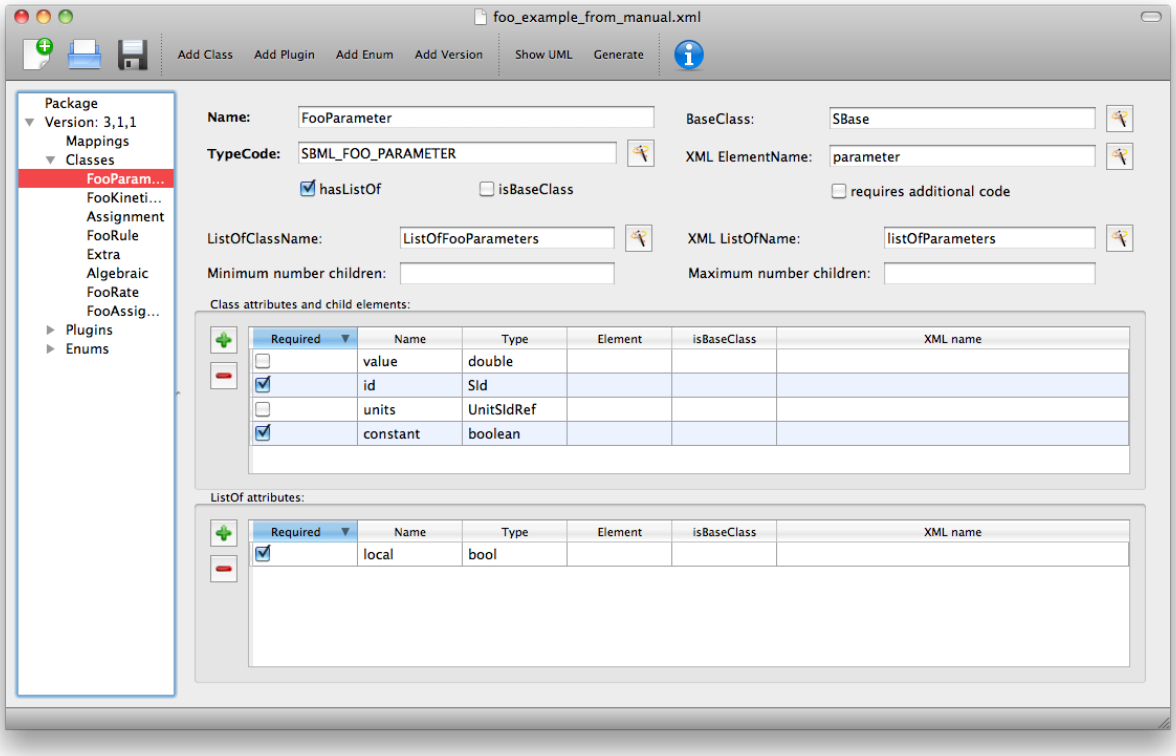

Figure 3.14: Defining the FooParameter class.

The **hasListOf** checkbox has been selected the additional fields appear.

In our example we have populated the **ListOfClassName** and **XML ListOfName** fields as we have used a class name 'FooParameter' but want to have XML names of 'parameter' and 'listOfParameters'.

Figure 3.15 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*.

SBML Snippet 4: The 'foo' kineticLaw element as defined in Examples 1 and 2.

```
<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>
```

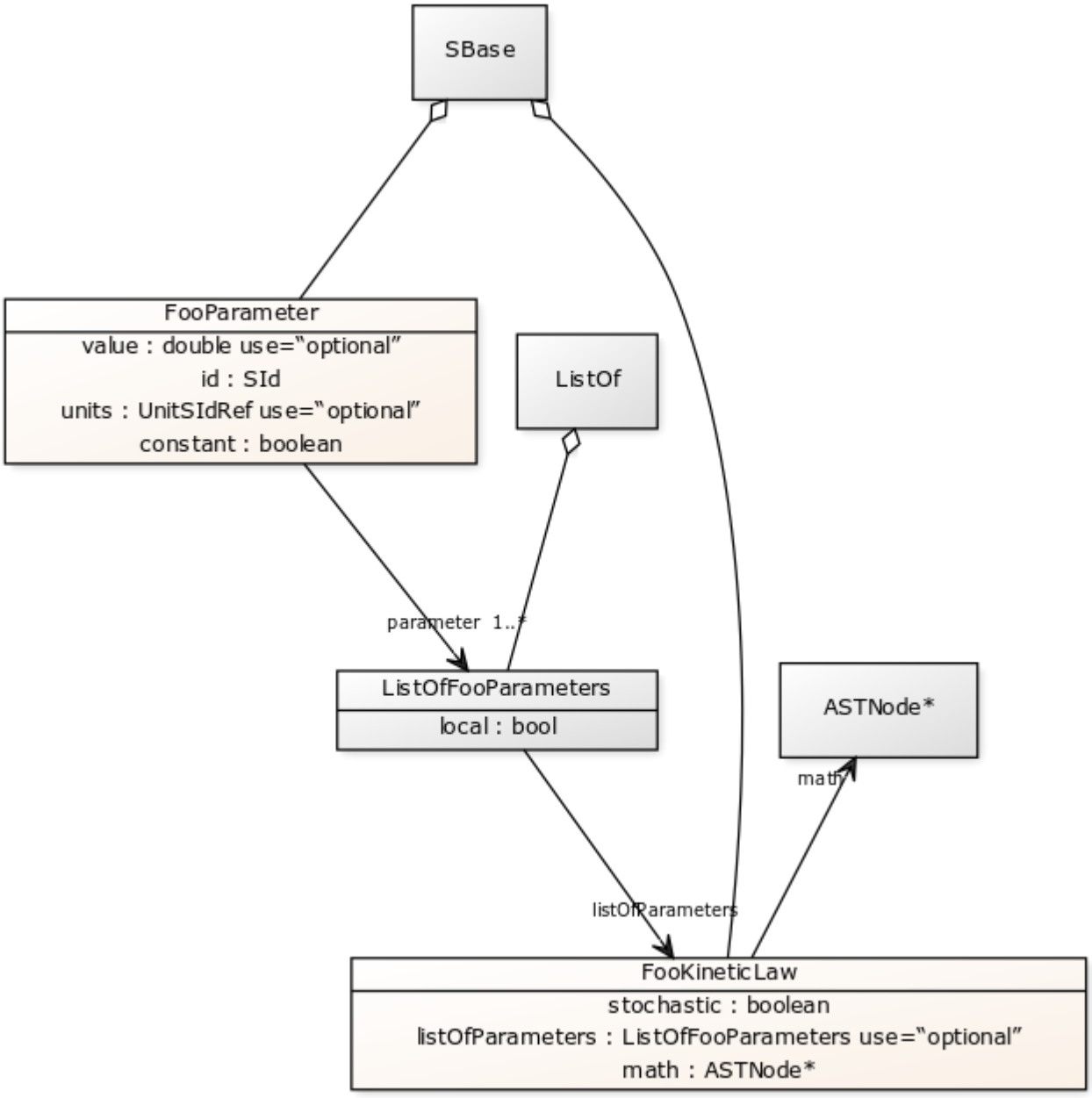

Figure 3.15: The UML diagram produced by Deviser Edit following the definition of package 'Foo' in Examples 1 and 2

3.3.9 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 3.16).

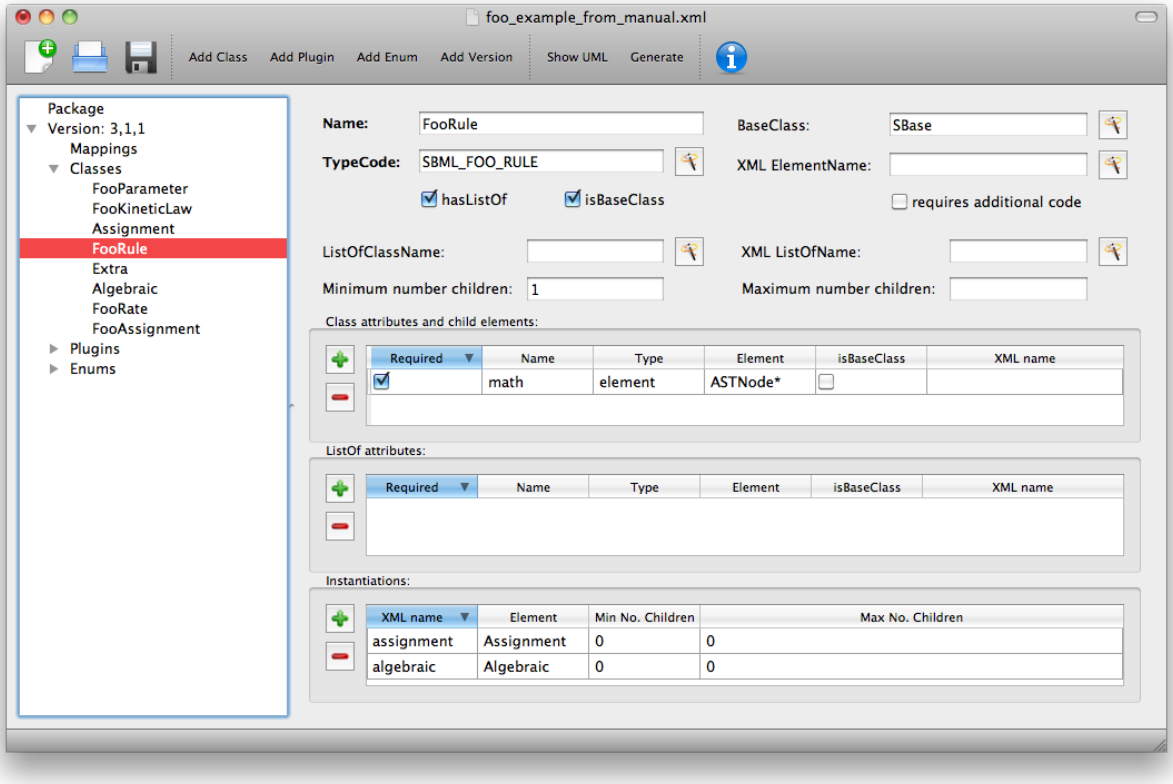

Figure 3.16: Defining the base class 'FooRule'.

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.

Here we have specified that the ListOfFooRules may contain objects of type Assignment or Algebraic. We specify Algebraic as a new class in Figure 3.17 and Assignment in Figure 3.18.

Note that we have changed the BaseClass field to FooRule.

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 have also acts as a base class for two further classes FooRate and FooAssignment. Figure 3.19 shows the hierarchy and *SBML snippet 5* the resulting XML.

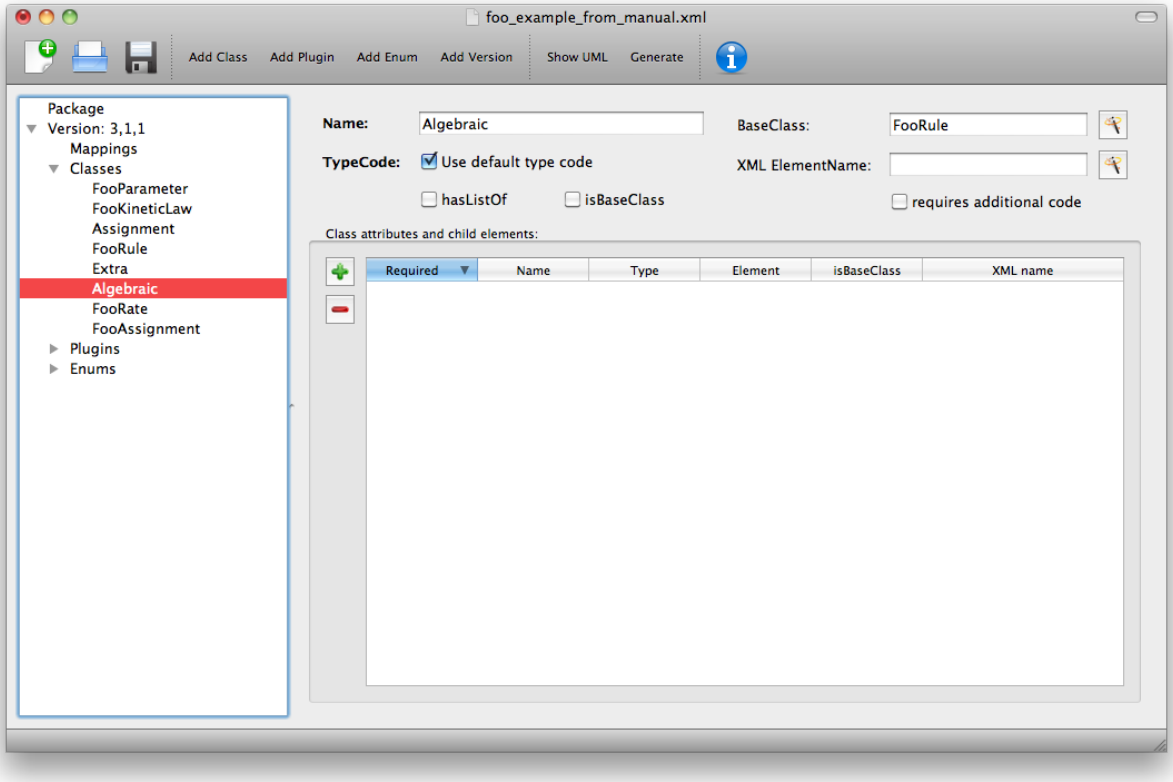

Figure 3.17: Defining the Algebraic class

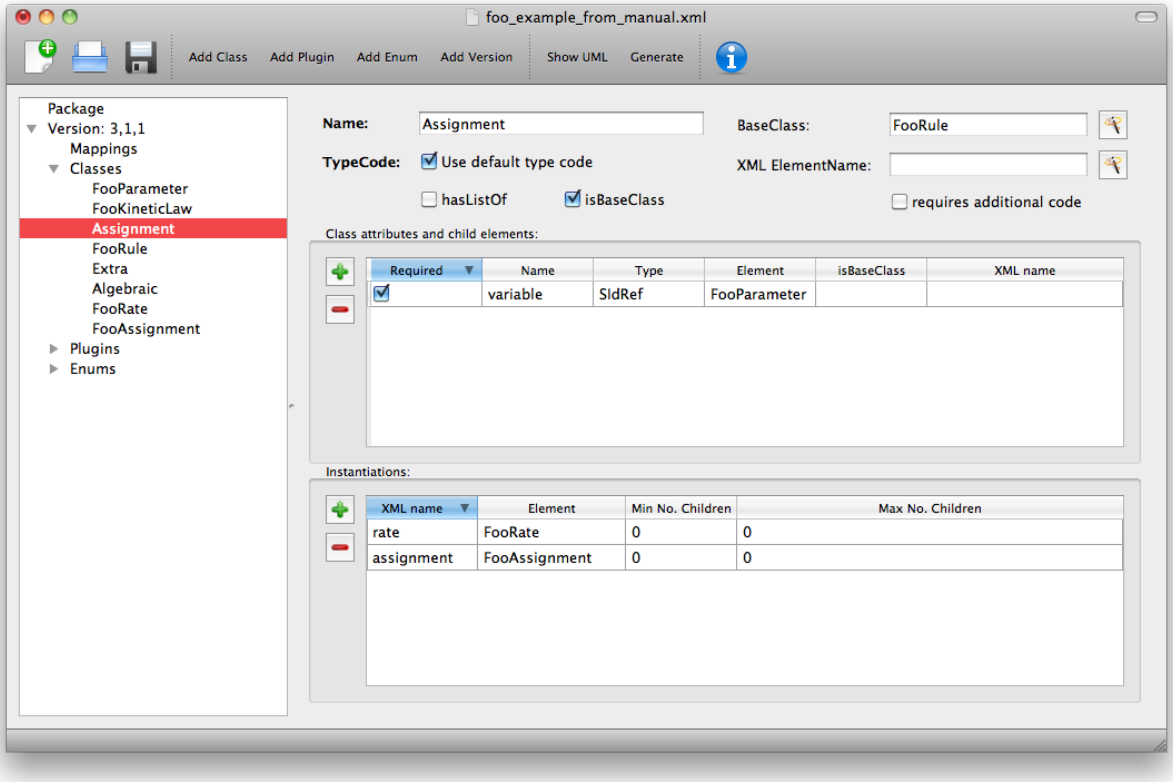

Figure 3.18: Defining the Assignment class.

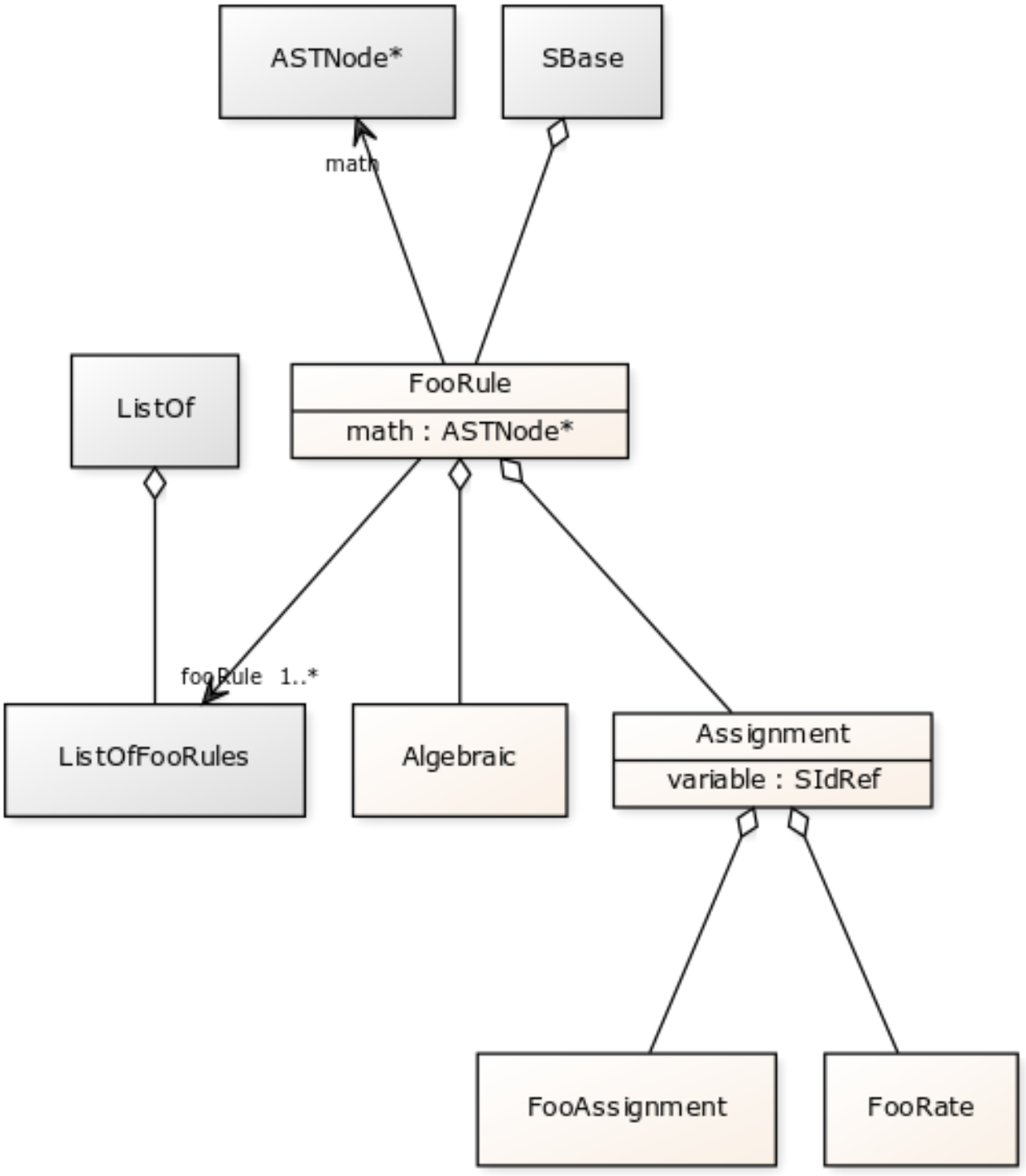

Figure 3.19: The UML diagram from Deviser Edit of the ListOfFooRules class

SBML Snippet 5: The listOfFooRules element as defined in Example 3.

```
<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>
```

3.4 Add plugin information

3.4.1 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 3.20 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.

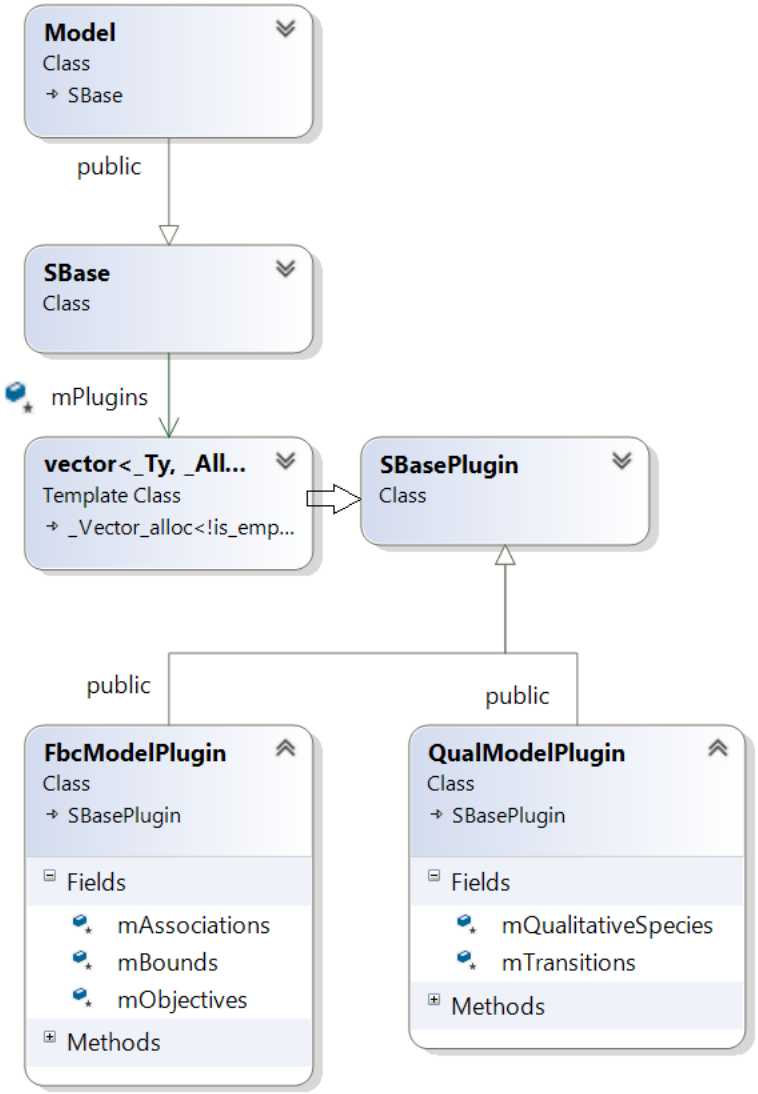

Figure 3.20: libSBML class hierarchy showing 'plugins' to the Model class

3.4.2 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.

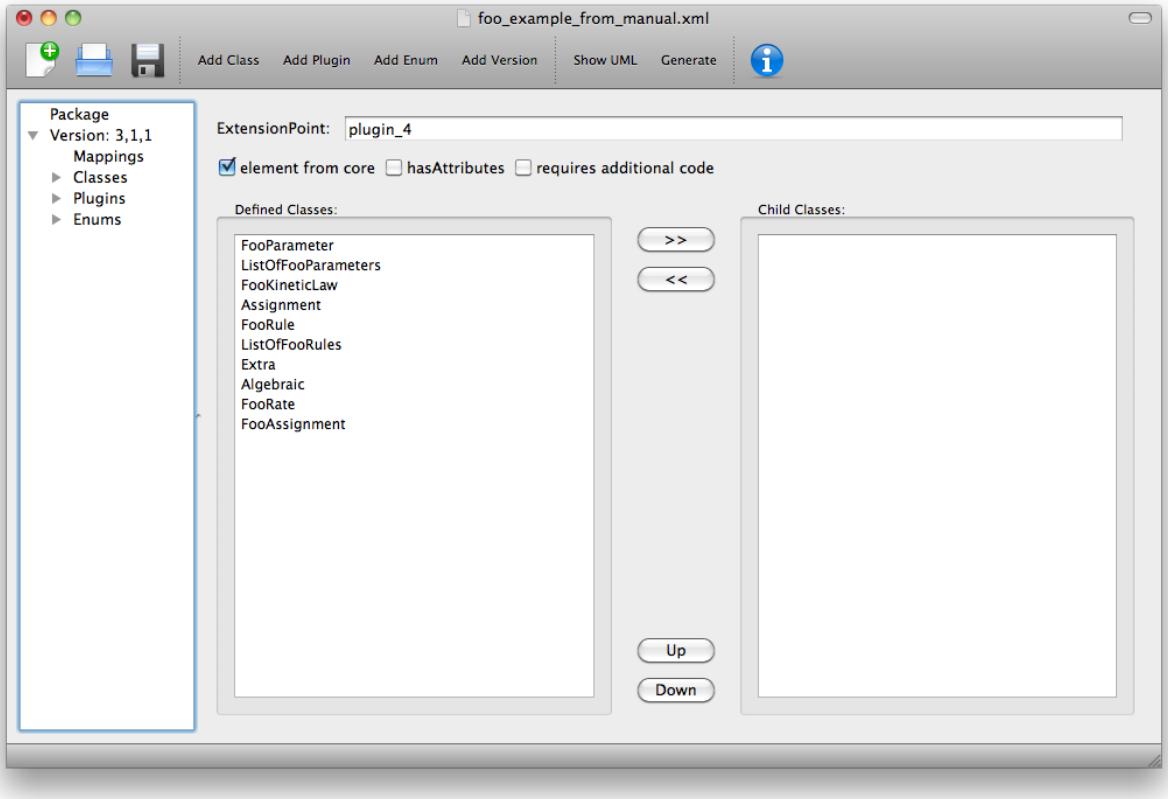

Figure 3.21: The 'Plugin' panel.

3.4.2.1 The Plugin ExtensionPoint field

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.

3.4.2.2 The Plugin element from core checkbox

The **element from core** checkbox is used to specify whether the object being extended originates in SBML Core or another Level 3 package. It is checked by default as to date the majority of SBML L3 packages have only extended elements from SBML core. Unchecking the box reveals the **Package** and **TypeCode** fields discussed below.

3.4.2.3 The Plugin hasAttributes checkbox

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

3.4.2.4 The Plugin requires additional code checkbox

As with the class panel *The Class requires additional code checkbox* the **requires additional code** checkbox can be used to indicate that there is additional code that will be required by this plugin 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'.

3.4.2.5 The Plugin Defined Classes and Child Classes boxes

The panel 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.

The **Up** and **Down** buttons can be used to reorder the classes that have been added as extensions for the given extension point. This will impact the order in which Deviser deals with plugins and thus will affect typecode enumerations and the order in which plugin objects are documented.

3.4.2.6 Adding other package information

3.4.2.6.1 The plugin Package field

In cases where the **ExtensionPoint** does not originate in SBML L3 Core Deviser needs to know in which L3 Package the class does originate.

3.4.2.6.2 The plugin TypeCode field

In cases where the **ExtensionPoint** does not originate in SBML L3 Core Deviser also needs to know the TypeCode that libSBML has used for the object being extended. It will be necessary for the user to consult libSBML documentation (or code) to determine this value.

3.4.3 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.

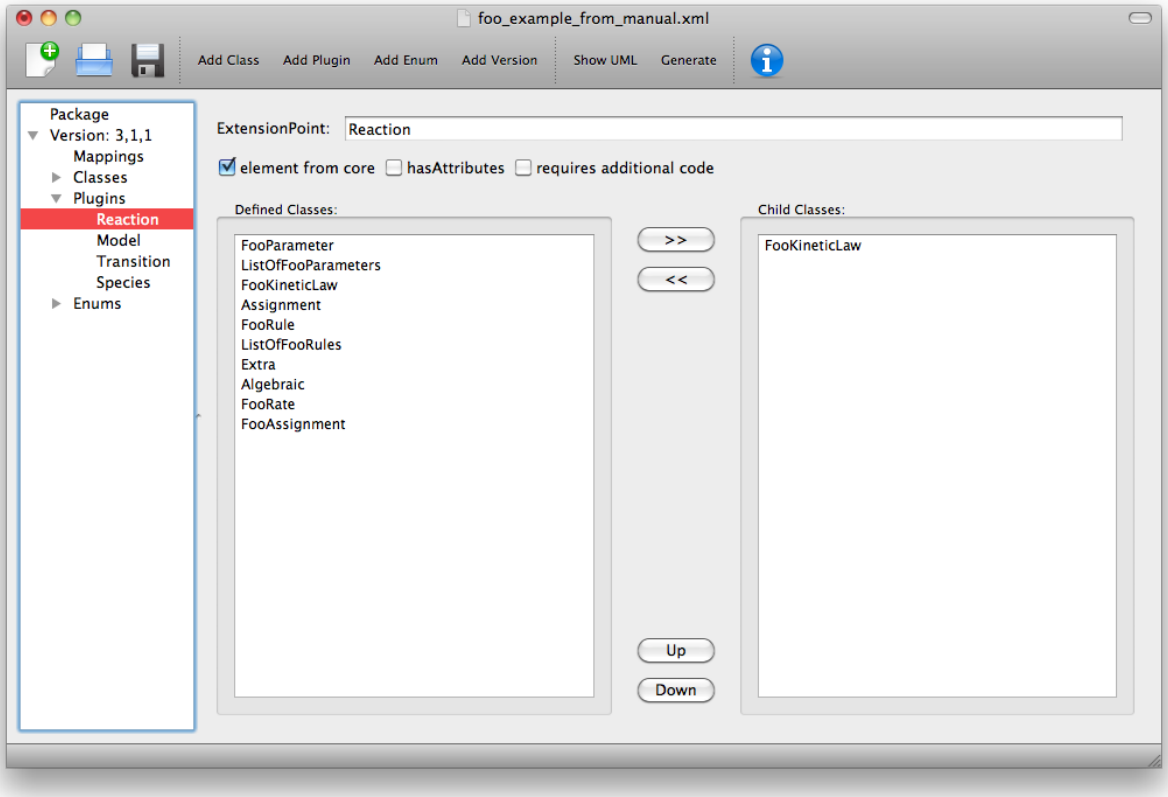

Figure 3.22: Defining the extension of SBML Level 3 Core Reaction by package foo.

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 and the functions necessary to create and manipulate it (as shown below).

```
class LIBSBML_EXTERN FooReactionPlugin : public SBasePlugin
{
protected:
  FooKineticLaw* mFooKineticLaw;
public:
  const FooKineticLaw* getFooKineticLaw() const;
  bool isSetFooKineticLaw() const;
  int setFooKineticLaw(const FooKineticLaw* fooKineticLaw);
  FooKineticLaw* createFooKineticLaw();
  int unsetFooKineticLaw();
  ...
}
```

3.4.4 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 3.23). Note that it is not necessary to specify child elements that originate in the package being defined as these that have already been listed as **Child classes**.

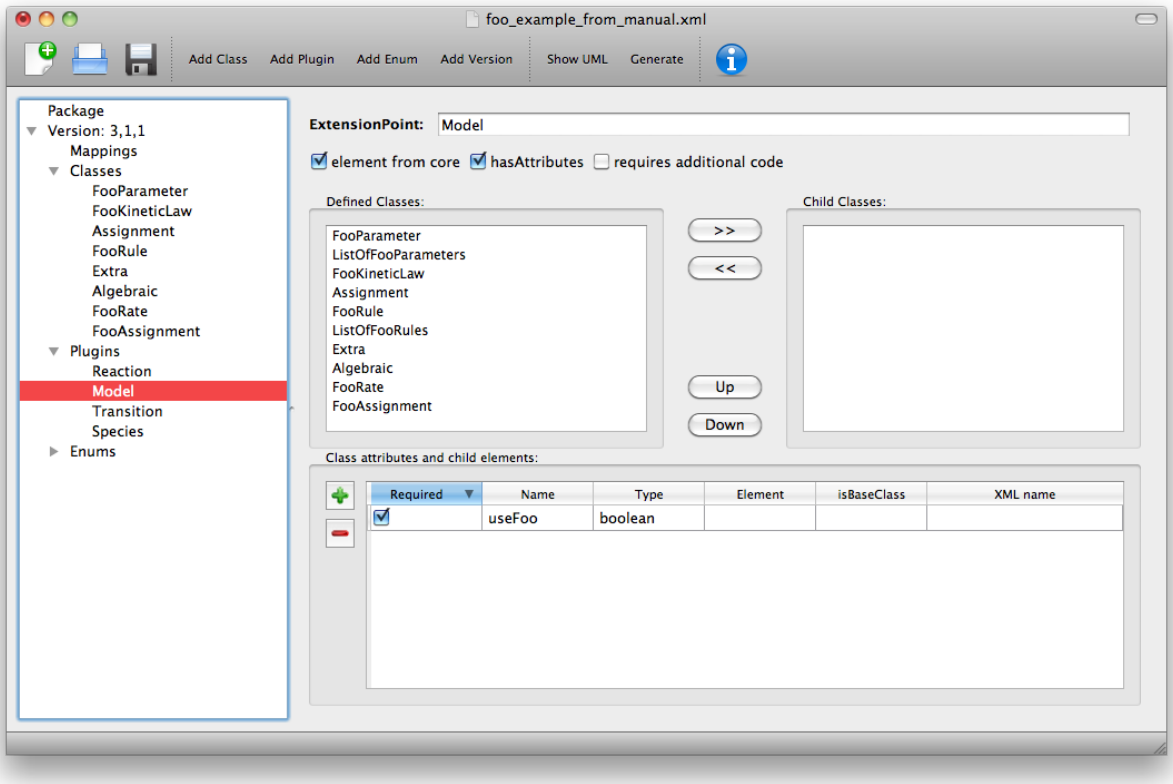

Figure 3.23: Defining the extension of SBML Level 3 Core Model by package foo.

3.4.5 Example 6 – Extending a non-core element

Here we declare that the **ExtensionPoint** is Transition from the Qualitative Models (qual) Package. Unchecking the **element from core** checkbox reveals the **Package** and **TypeCode** fields which have been filled in as appropriate. The package 'foo' adds the ListOfFooRules object to the Transition object.

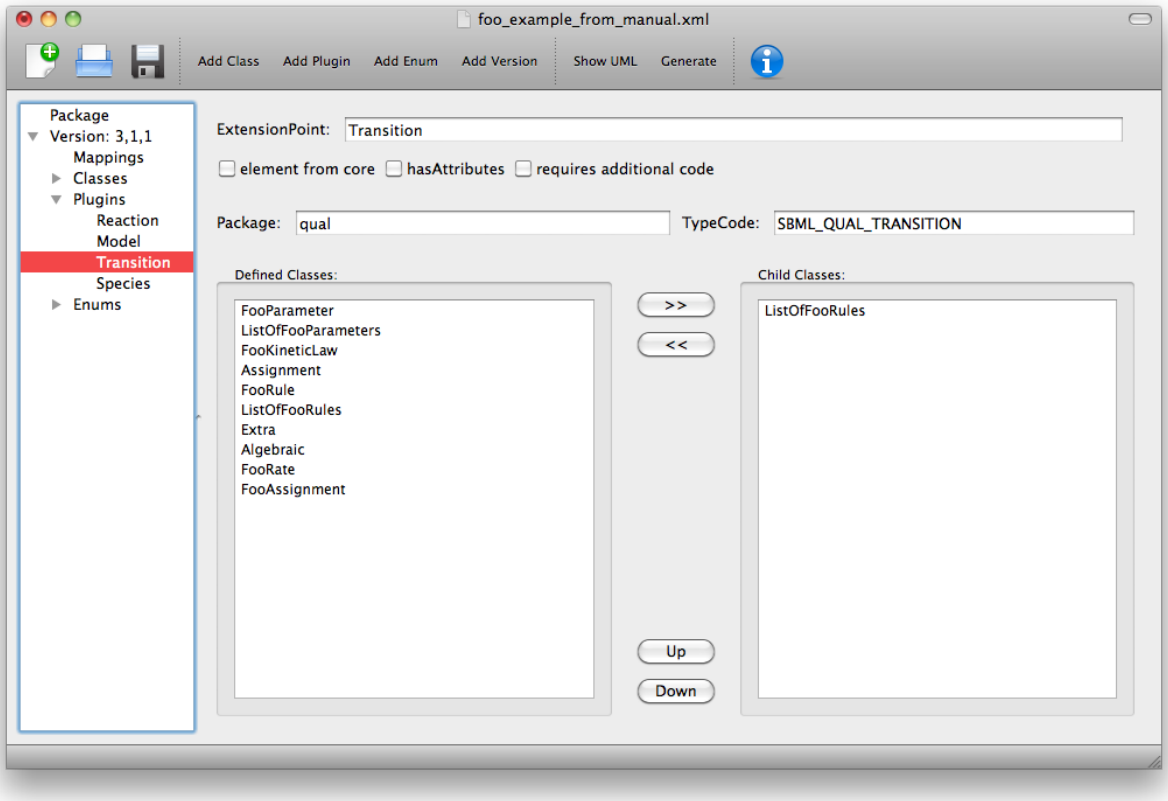

Figure 3.24: Defining the extension of SBML Level 3 Qual Transition by package foo.

3.5 Add enum information

SBML allows users to define data types as enumerations of allowed values. Section *Attribute/child element type 'enum'* describes using 'enum' as an attribute **Type**. Here we describe how to fully specify the enumeration.

3.5.1 General enum information

3.5.1.1 The Enum Name field

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 if it does not already have a name of the format Name_t.

3.5.1.2 The Enum Name/Value table

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

3.5.1.2.1 The enum table Name field

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

3.5.1.2.2 The enum table Value field

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

Each entry must have both a **Name** and a **Value**.

3.5.1.3 The Enum Quick Add field

This field can be used to facilitate creating the enumeration Name-Value pairs. Enter the string value in this field and press the 'wand'. The string will be added as an enumeration **Value** with a **Name** of PACKAGE_ENUM_VALUE.

3.5.2 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 3.25.

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

The **Name** filed is Sign (which corresponds to the **Element** field in the attribute table). Figure 3.26 shows that we have specified that the enumeration sign has three possible values: 'positive', 'negative' and 'neutral. Note we used the **Quick Add** field to enter 'neutral' which resulted in the enumeration FOO_SIGN_NEUTRAL.

It is not necessary to add a default or "unknown" value – Deviser will do this when generating code.

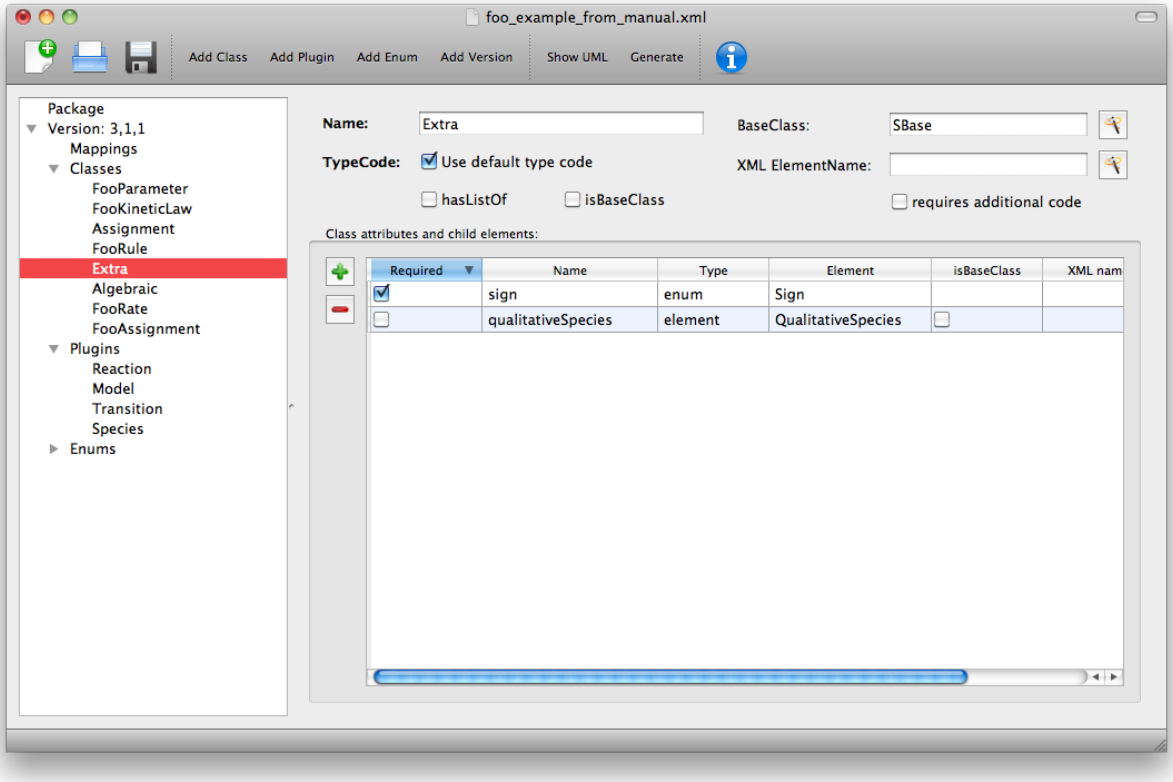

Figure 3.25: Defining the Extra class which has an attribute of type enum.

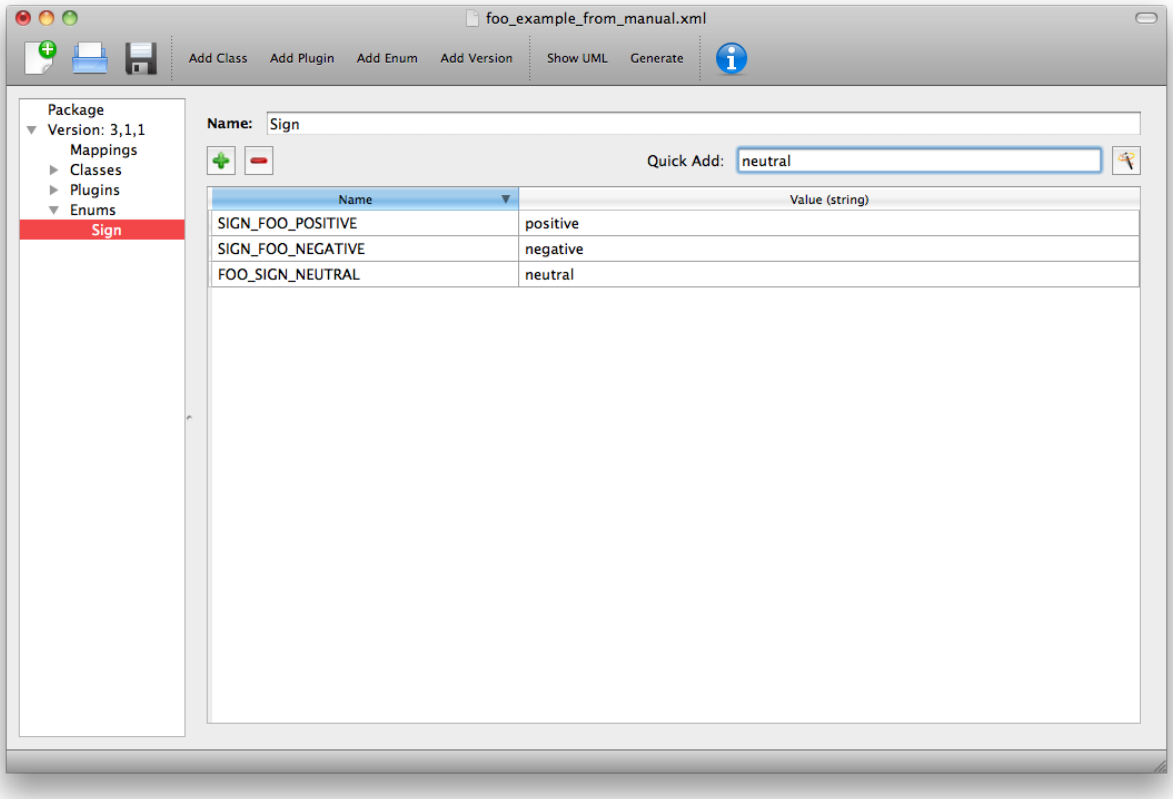

Figure 3.26: Defining the Sign enumeration.

3.6 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.

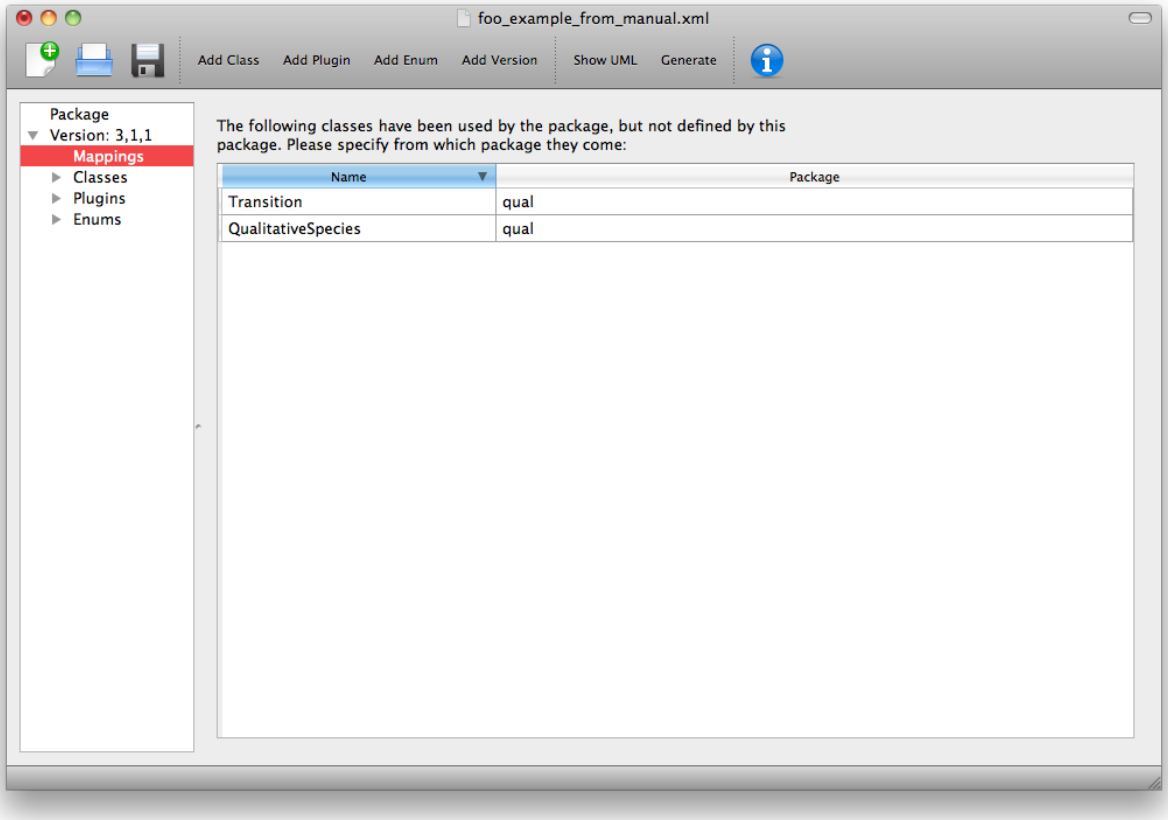

Figure 3.27: Identifying the origin of classes from other packages

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 panel only the **Package** column can be edited. The **Name** column is populated by the tool.

3.7 Overview of a defined package

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 3.28) and the ordering can be adjusted. The order will dictate the order of the relevant sections in the TeX documents.

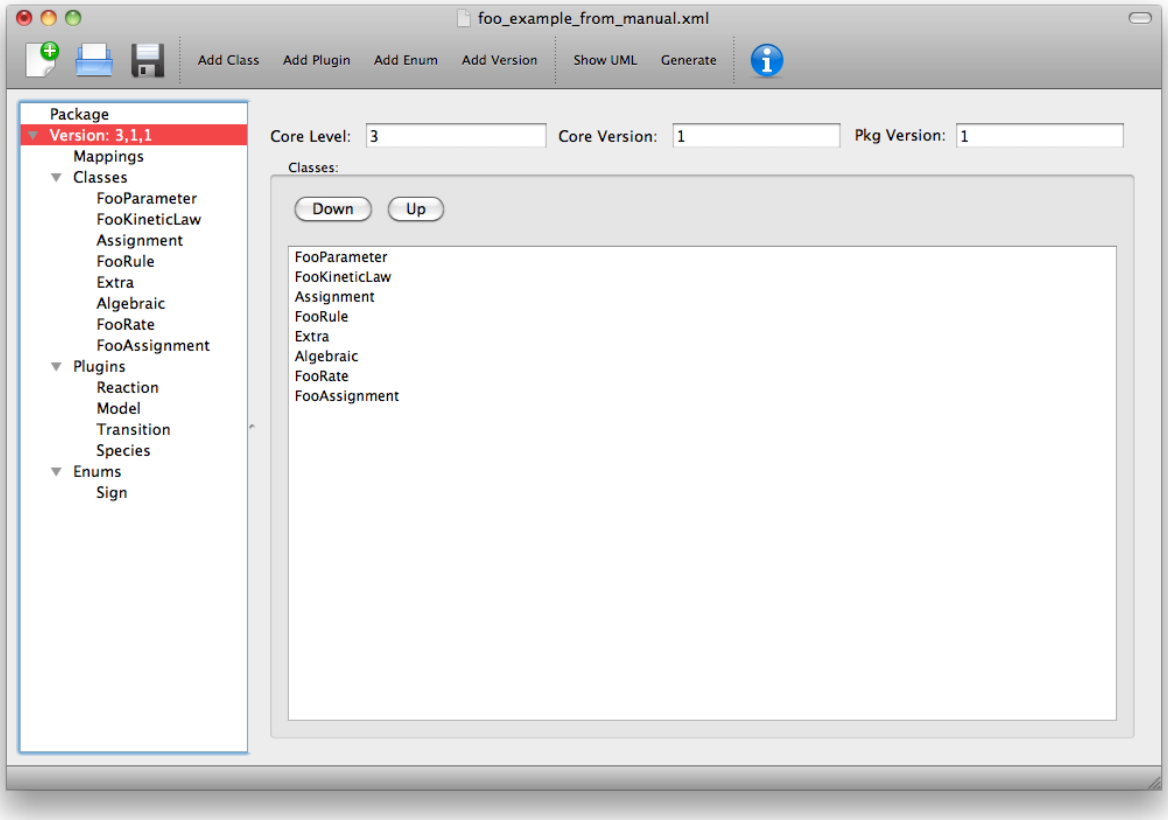

Figure 3.28: The complete description of the foo package

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: Example package description* or is available in the deviser/samples directory.

3.7.1 Validating the description

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

3.7.1.1 The Validate Description option

Validate Description runs a series of internal checks on the information provided and produces a list of Errors and Warnings. When invoked a pop-up window (Figure 3.29) will appear with either a list of errors and/or warnings 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.

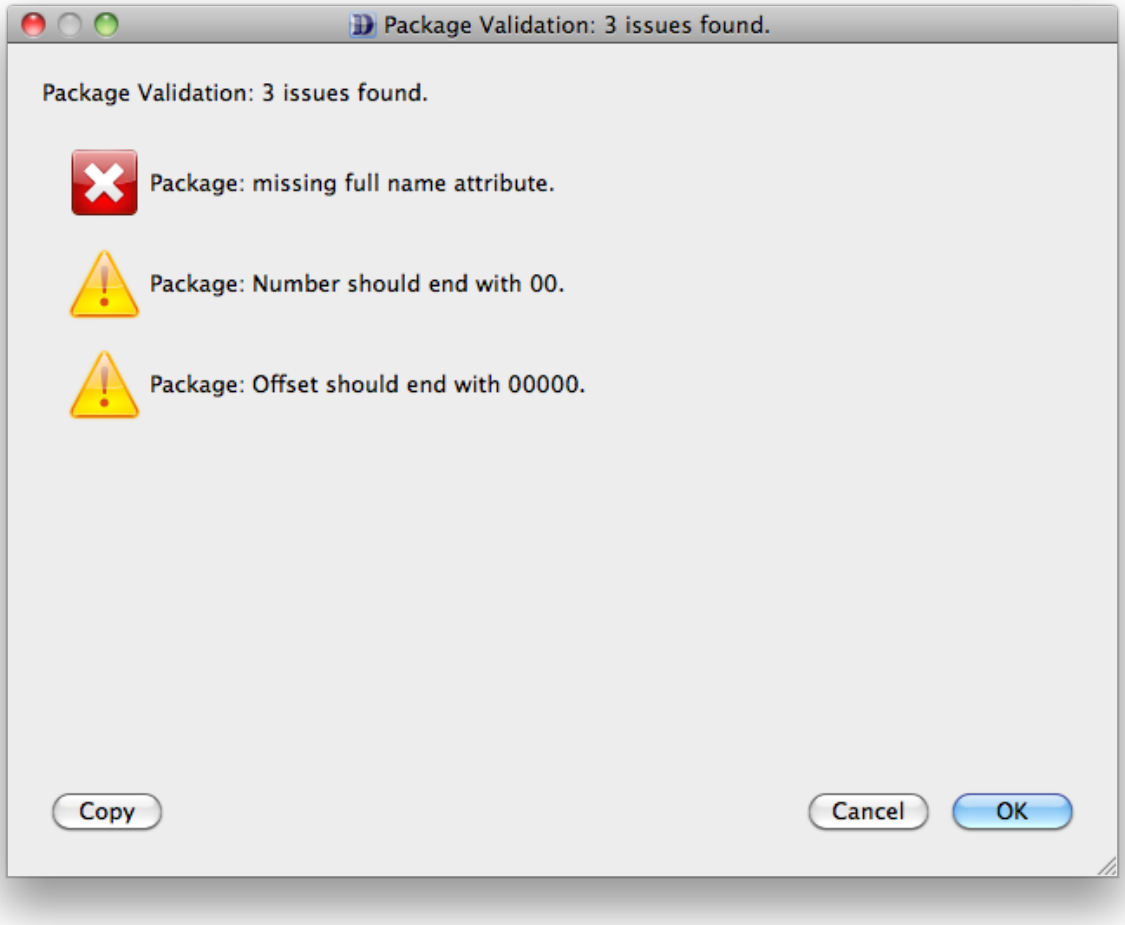

Figure 3.29: Validating the package description

Appendix C: Validation gives a list of the errors and warnings that may be issued by Deviser Edit with references to the relevant sections in the main text.

3.7.1.2 The Fix Errors option

Fix Errors provides a direct way of validating and then correcting any inconsistencies. Deviser Edit will run the validation checks and then automatically correct any issues, where this is possible. It is advisable to use **Validate Description** following **Fix Errors** as some errors cannot be automatically fixed.

3.8 Defining multiple versions of a package

> **Warning:** Latex generation does not, as yet, deal with multiple versions.

Deviser code generation neatly handles multiple versions of a package.

Each version must be specified completely. Using the **Add Version** button from the menu or toolbar will create a second version that can be populated in the same way as described in this manual.

Since new versions of a package are likely to resemble existing versions the entire version can be duplicated by right-clicking on the Version header in the tree view on the left panel and selecting **Duplicate** (see Figure 3.30).

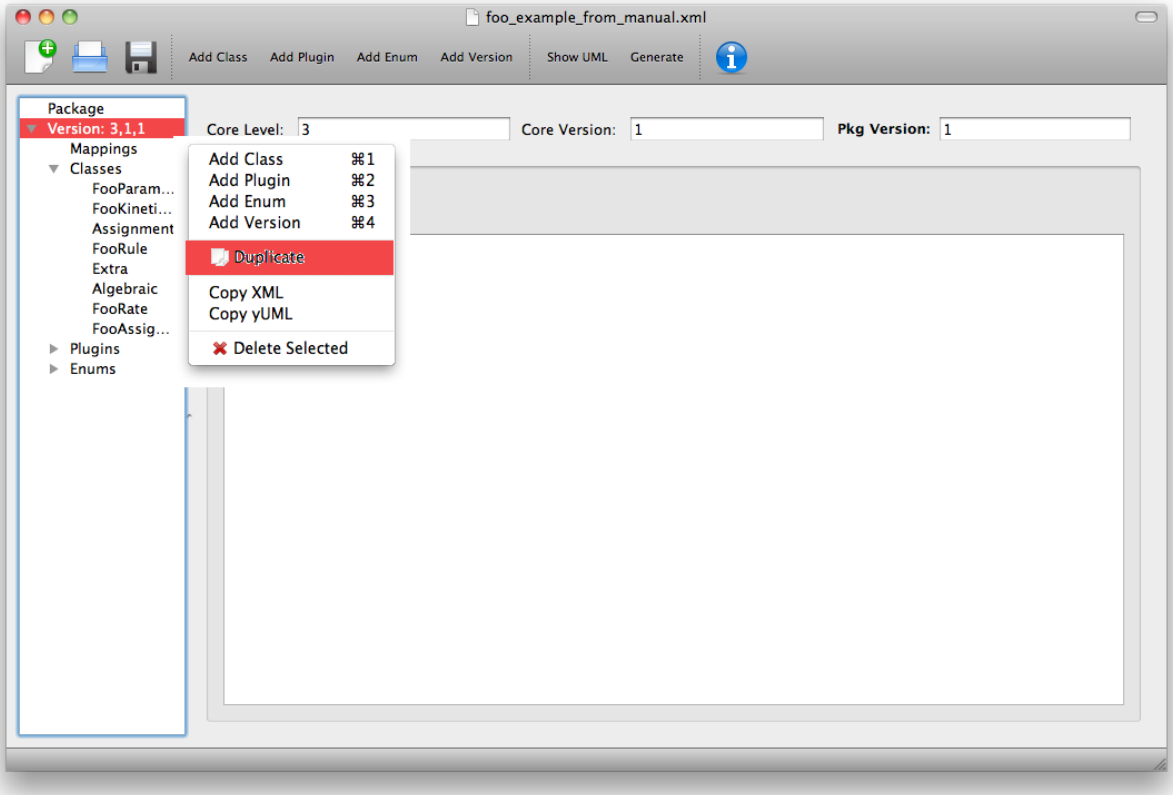

Figure 3.30: Duplicating a version of a package.

This creates a complete copy of the previous package version and gives it the next package version number (see Figure 3.31). The individual objects can then be edited or removed as necessary and any additional objects added.

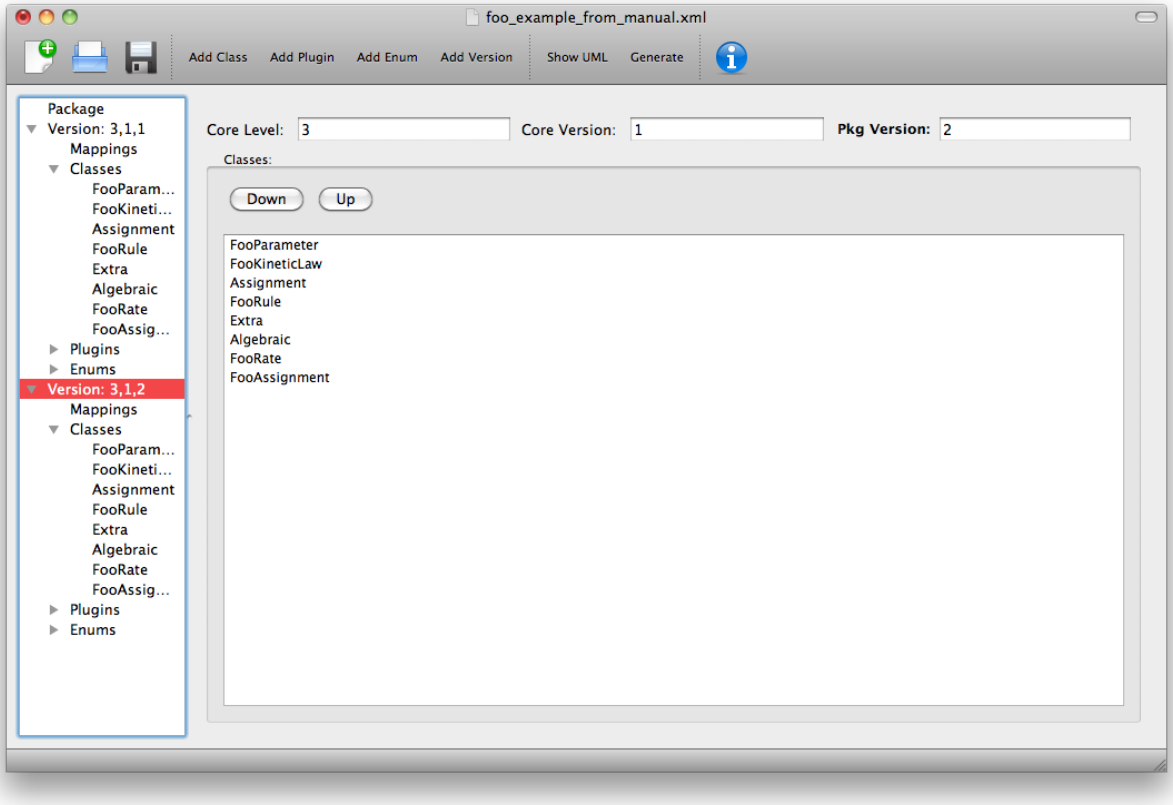

Figure 3.31: A second version of the 'foo' package duplicated from Version 1. Note how the classes listed reflect Version 1.

CHAPTER

FOUR

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 *Setting up the Deviser Edit tool*.

4.1 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 4.1). 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.

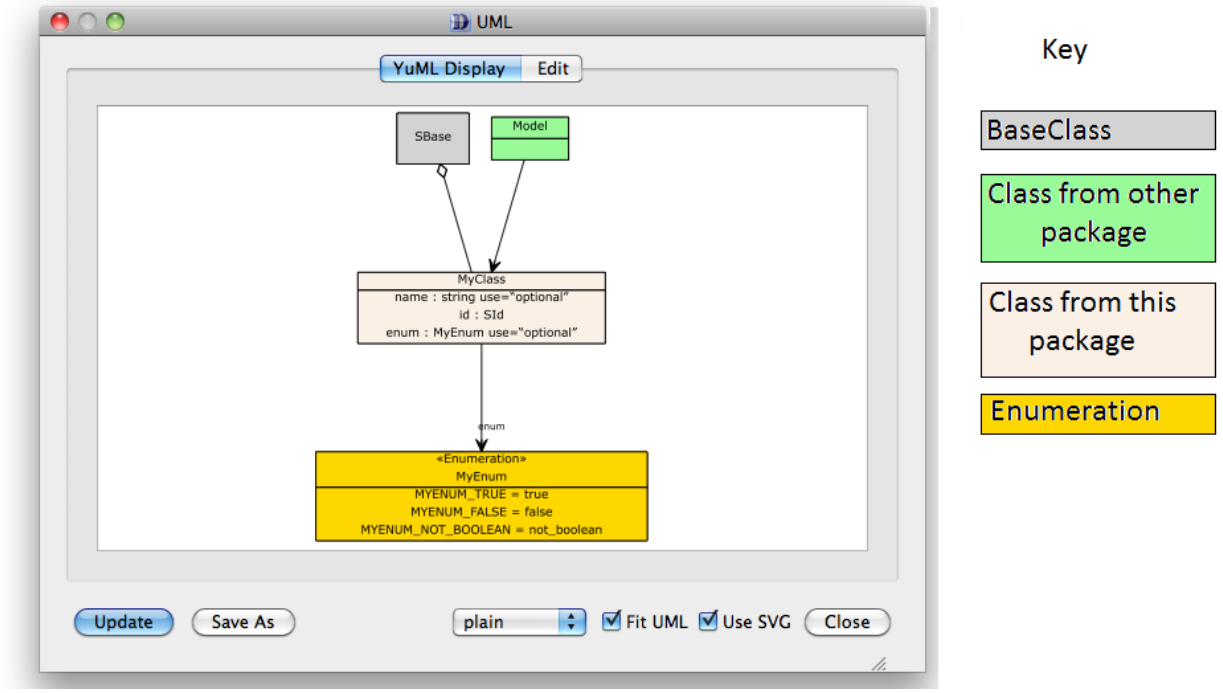

Figure 4.1: The UML window

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 4.2).

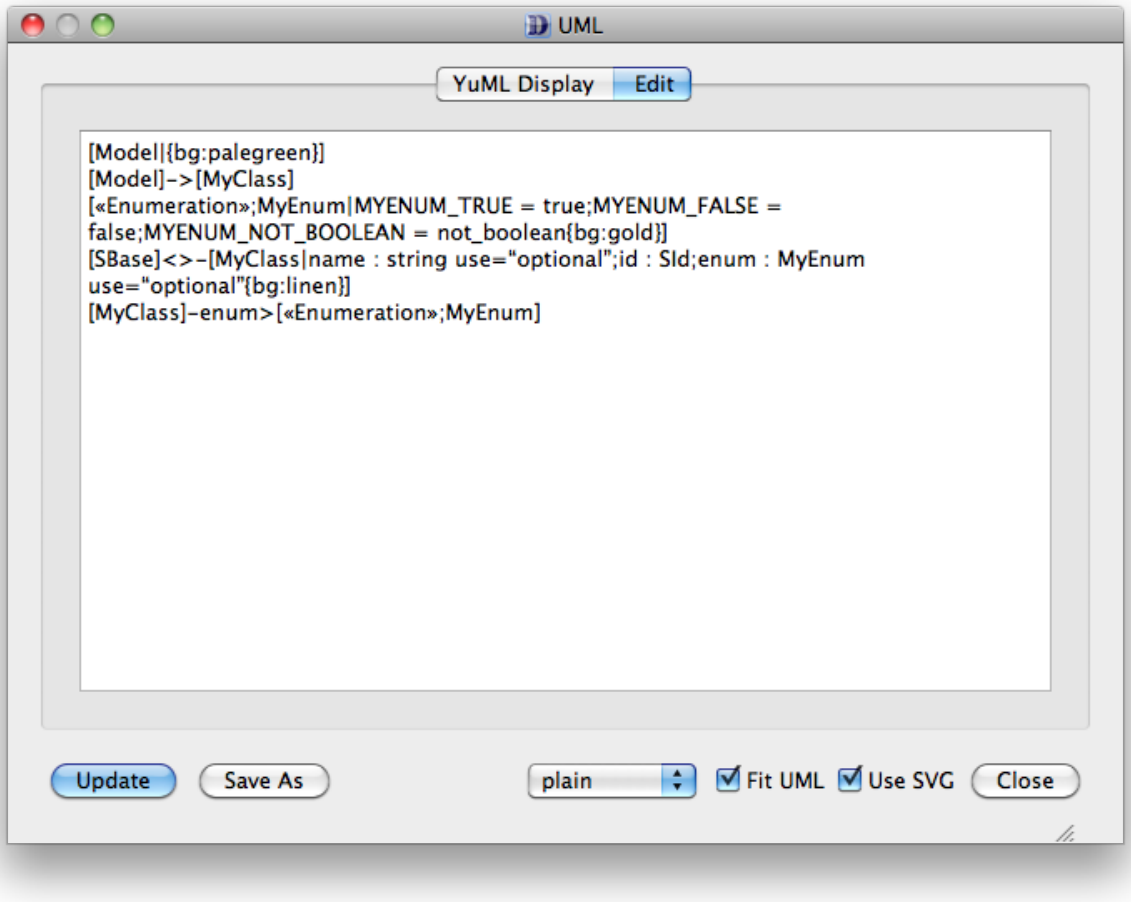

Figure 4.2: The yUML text representation of the diagram

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.

4.2 Generate package code/documentation

Click on the **Generate** button on the toolbar and the Generate window (Figure 4.3) will appear. Prior to displaying the window the validation checks (see Section 3.7.1.1) are run and any problems reported to the user.

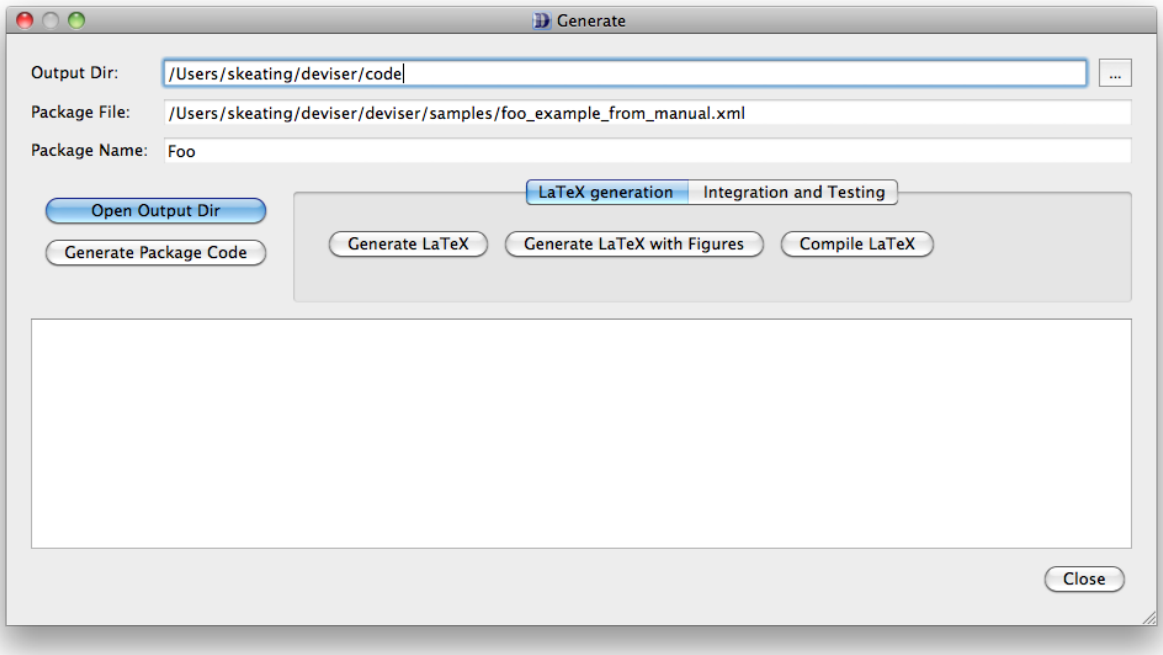

Figure 4.3: The Generate window

The **Output Dir**, **Package File** and **Package Name** fields are all automatically populated using the information provided. It is recommended that you have saved to a file before attempting generation.

The **Generate Package Code** button will then create the package code and put it in a directory named '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.

The **Open Output Dir** button provides a shortcut to the output directory specified.

Further options are available via the **LaTeX Generation** and **Integration and Testing** tabs. Functionality provided here will require access to further software as detailed in the *Prerequisites* section.

4.2.1 Generate basic specification documentation.

In the Generate window there is a **LaTeX Generation** tab with three further buttons (Figure 4.3).

4.2.1.1 The Generate LaTeX button

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

These will be located in a directory named 'foo-spec' within the specified output directory.

4.2.1.2 The Generate LaTeX with Figures button

The **Generate LaTeX with Figures** button will call the yUML services to generate figures for each of the classes and plugins specified within the package description. It also creates an overall diagram of the whole package. These are placed in a subdirectory figures within the 'foo-spec' directory.

The three TeX files are then generated

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

These will be located in a directory named 'foo-spec' within the specified output directory.

4.2.1.3 The Compile TeX button

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. This will be located within the 'foo-spec' directory.

4.2.2 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 4.4).

4.2.2.1 The Compile Dependencies button

Compile Dependencies compiles the dependencies with the specified C++ compiler to ensure that these are compatible with the libSBML build. Note that particularly on Windows OS it is necessary for the libSBML dependencies to be built with the same compiler as that to be used to build libSBML. If the user has specified the location of the source code for the dependencies and the C++ compiler (see Section 2.3.1) then this button will invoke a build of the dependencies.

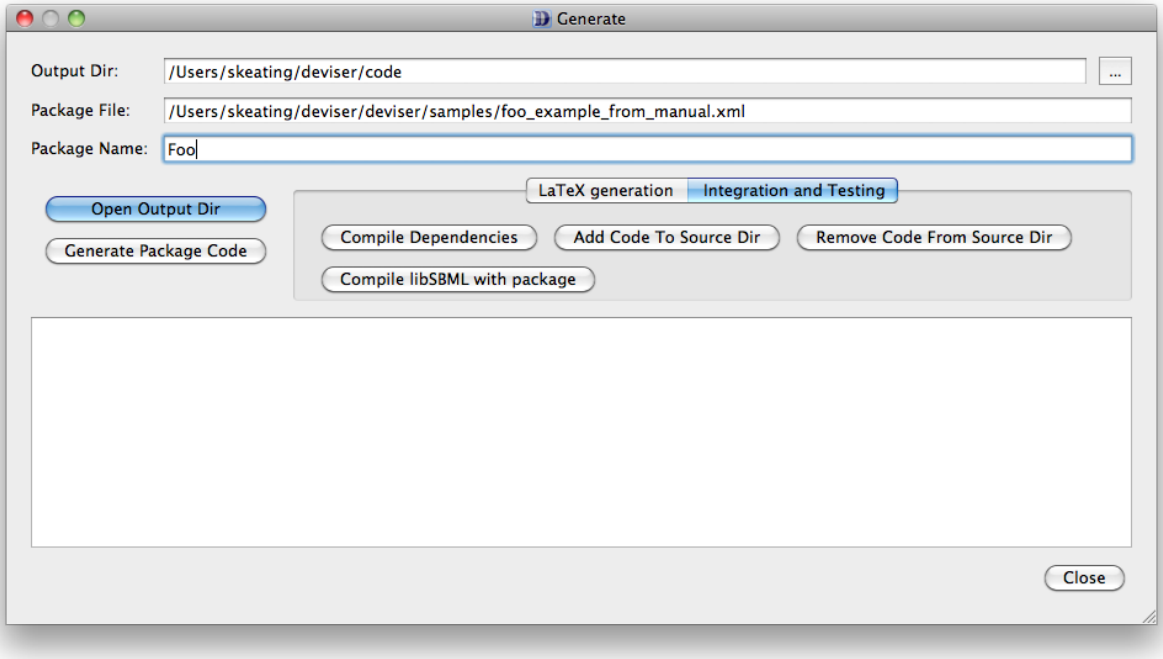

Figure 4.4: Integration and Testing tab selected on the Generate window.

4.2.2.2 The Add Code to Source Dir button

Add Code to Source Dir places the code generated for package foo within the libSBML source tree. Note that this merely takes the output from the code generation step and places a copy of it within the source tree. Thus two copies of the code now exist. One in the Deviser **Output Dir** and one embedded within the specified libSBML source tree. If the code is changed, either in the tree or by regenerating using Deviser, then the other 'copy' is NOT affected.

This does mean that using this button more than once will overwrite files. Care need to be taken if code needs to be manually adjusted.

4.2.2.3 The Remove Code to Source Dir button

Remove Code from Source Dir removes the code generated for package foo from the libSBML source tree. The removed code is merely deleted and not saved elsewhere. Thus any changes made to the code within the tree will be lost.

4.2.2.4 The Compile libSBML with package button

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

4.3 Utility functions in Deviser Edit

There are a number of functions to facilitate use of Deviser Edit and reporting of issues. Using the tree view on the left right clicking on an object reveals a menu as shown in Figure 4.5.

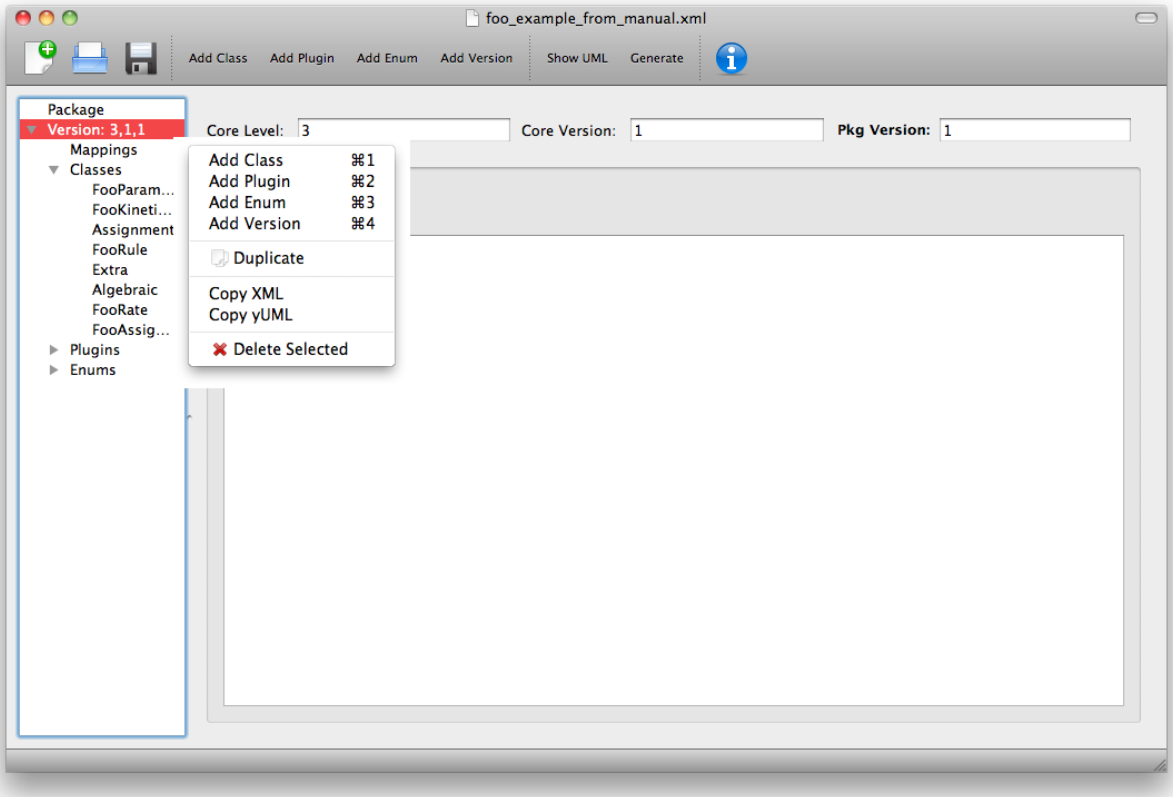

Figure 4.5: Utility menu.

4.3.1 The Add XYZ items

The various **Add** items will add the appropriate object to the Version selected. This will work from any object within the tree. So in Figure 4.5 highlighting *FooRate* and selecting *Add Enum* would add a new Enum to the list for Version 3,1,1.

4.3.2 The Duplicate item

The **Duplicate** item creates a copy of the selected item (where this makes sense). So it is possible to duplicate individual classes, plugins or enums but the headers *Mappings*, *Classes*, *Plugins* and *Enums* cannot be duplicated. It is also possible to duplicate the entire version, which creates a new version with identical elements as the one being copied. Section *Defining multiple versions of a package* gives more details.

In Figure 4.5 highlighting *FooRate* and selecting *Duplicate* would add a new class to the list for Version 3,1,1. This class would be identical to *FooRate* except it would have the name *FooRate_copy*.

4.3.3 The Copy XML item

The **Copy XML** item copies the Deviser XML description of the selected object to the clipboard; thus making it available for pasting elsewhere. Similarly to **Duplicate** this works for individual classes, plugins, enums and versions but not for the other headers.

Example of text copied when highlighting *Model* from the *Plugins* and selecting *Copy XML*

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin extensionPoint="Model">
  <attributes>
    <attribute name="useFoo" required="true" type="boolean" abstract="false"/>
  </attributes>
</plugin>
```

4.3.4 The Copy yUML item

The **Copy yUML** item copies the yUML text description of the selected object to the clipboard; thus making it available for pasting elsewhere. Similarly to **Duplicate** this works for individual classes, plugins, enums and versions but not for the other headers.

Example of text copied when highlighting *Sign* from the *Enums* and selecting *Copy yUML*

```
[«Enumeration»;Sign|SIGN_FOO_POSITIVE = positive;SIGN_FOO_NEGATIVE = negative;SIGN_FOO_NEUTRAL =␣
↪neutral{bg:gold}]
```

4.3.5 The Delete Selected item

The **Delete Selected** item removes the item from the description. Similarly to **Duplicate** this works for individual classes, plugins, enums and versions but not for the other headers.

4.4 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.

CHAPTER

FIVE

APPENDIX A: EXAMPLE PACKAGE DESCRIPTION

The following is the XML description produced by the Deviser Edit tool and used by further generation facilities within Deviser. The example given corresponds to the examples used within the documentation to specify the "Imaginary Package Foo".

```
<?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_FOOKINETIC_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>
          <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"/>
            <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>
```

CHAPTER

SIX

APPENDIX B: DATA TYPES

The SBML attribute 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
IDREF	std::string
SId	std::string
SIdRef	std::string
UnitSId	std::string
UnitSIdRef	std::string

CHAPTER

SEVEN

APPENDIX C: VALIDATION

Severity	Reference	Sample Message
Error	*The Package Name field*	Package: missing name attribute.
Error	*The Package Full Name field*	Package: missing full name attribute.
Warning	*The Package requires additional code checkbox*	Package: Number should end with 00.
Warning	*The Package Offset field*	Package: Offset should end with 00000.
Error	*The attribute/element Name field*	The attribute 'foo_name' of 'foo_classname uses a restricted name which will cause compilation issues.
Warning	*The attribute/element Element field*	The attribute 'foo_name' of 'foo_classname' is of type SIdRef, but has no element set. This is needed for validation purposes.
Error	*The attribute/element Element field*	The attribute 'foo_name' is of type 'foo_type' but has no element defined, this is a required attribute for that type.
Warning	*The attribute/element Element field*	The attribute 'foo_name' of type 'foo_type' but has an element 'foo_element' defined, this attribute will not be used.
Error	*The attribute/element Type field*	The attribute 'foo_name' has no type defined, this is a required attribute.
Warning	*The attribute/element Type field*	The attribute 'foo_name' is of type 'foo_type', this type is not known by DEVISER and will have to be changed manually in the generated code.
Error	*The Enum Name/Value table*	EnumValue 'count' of enum 'foo_enum_name' has no value, this is a required attribute
Error	*The Enum Name/Value table*	EnumValue 'count' of enum 'foo_enum_name' has no name, this is a required attribute
Warning	*The Class hasListOf checkbox*	Class 'foo_classname' uses a list of 'foo_other', which is not marked having a list.
Warning	*The Class hasListOf checkbox*	Plugin for 'foo_extension_point' uses a list of 'foo_other', which is not marked having a list.
Error	*The instantiations Element field*	The instantiation 'foo_xml_name' has no element defined, this is a required attribute.
Error	*The instantiations XML Name field*	In class 'FooRule' an instantiation has no XML name, this is a required attribute.
Warning	*The Class isBaseClass checkbox*	Class 'foo_classname' is marked 'isBaseClass', but not used.
Error	*The Class TypeCode checkbox and field*	The element 'foo_classname' has no typecode defined, and it can not be generated automatically.

The table lists the errors/warnings that may be produced by validating a package being defined within Deviser Edit.

Note: The error messages will be populated with the names of attributes/classes/types etc. as illustrated by the 'foo_xxx' placeholders above.

CHAPTER

EIGHT

APPENDIX D: LICENSING THE CODE PRODUCED

The Deviser and DeviserEdit code is released under the GNU LESSER GENERAL PUBLIC LICENSE (LGPL) Version 2.1. A copy of this is included with the deviser code and is available here.

Note: Code produced by Deviser must be released under the LGPL v2.1 license.

Currently the code produced will include mention of the LGPL license as well as the Copyright information for the current SBML Team. Deviser users may add their own Copyright information or indeed replace the existing Copyright providing a reference is made to Deviser.

At present the way to change what is produced at the top of each file involves adjusting the Deviser source code directly. In the file BaseFile.py located at deviser/generator/base_files in the deviser source tree the function **write_libsbml_licence(self)** produces the license and copyright information.

```
def write_libsbml_licence(self):
    self.write_blank_comment_line()
    self.write_comment_line('<!---------------------------------------'
                            '---------------------------------')
    self.write_comment_line('This file is part of libSBML.  Please visit '
                            'http://sbml.org for more information about '
                            'SBML, and the latest version of libSBML.')
    self.write_blank_comment_line()
    self.write_comment_line('Copyright (C) 2013-2015 jointly by the '
                            'following organizations:')
    self.write_comment_line('    1. California Institute of Technology, '
                            'Pasadena, CA, USA')
    self.write_comment_line('    2. EMBL European Bioinformatics '
                            'Institute (EMBL-EBI), Hinxton, UK')
    self.write_comment_line('    3. University of Heidelberg, Heidelberg, '
                            'Germany')
    self.write_blank_comment_line()
    self.write_comment_line('Copyright (C) 2009-2013 jointly by the '
                            'following organizations:')
    self.write_comment_line('    1. California Institute of Technology, '
                            'Pasadena, CA, USA')
    self.write_comment_line('    2. EMBL European Bioinformatics '
                            'Institute (EMBL-EBI), Hinxton, UK')
    self.write_blank_comment_line()
    self.write_comment_line('Copyright (C) 2006-2008 by the California '
                            'Institute of Technology,')
    self.write_comment_line('    Pasadena, CA, USA ')
    self.write_blank_comment_line()
    self.write_comment_line('Copyright (C) 2002-2005 jointly by the '
                            'following organizations:')
    self.write_comment_line('    1. California Institute of Technology, '
                            'Pasadena, CA, USA')
    self.write_comment_line('    2. Japan Science and Technology Agency, '
                            'Japan')
    self.write_blank_comment_line()
```

```
    self.write_comment_line('This library is free software; you can '
                            'redistribute it and/or modify it under the '
                            'terms of the GNU Lesser General Public '
                            'License as published by the Free Software '
                            'Foundation.  A copy of the license agreement'
                            ' is provided in the file named "LICENSE.txt"'
                            ' included with this software distribution '
                            'and also available online as http://sbml.org'
                            '/software/libsbml/license.html')
    self.write_comment_line('----------------------------------------------'
                            '---------------------------- -->')
```

Users should remove or replace the copyright text as appropriate.