Design
Explorer and
Viewer for
Iterative
SBML
Enhancement of
Representations

VERSION 0.9 (May 2015)

Manual for creating SBML Level 3 packages

SBML Team

Please note that this is a beta release of the deviser code and some of the functionality may only be available on certain platforms.

Contents

1 Getting started 4
1.1 Functionality 4
1.2 Prerequisites 4
1.2.1 The Deviser Edit tool 4
1.2.2 UML diagrams 5
1.2.3 Source code for libSBML 5
1.2.4 Basic documentation 5
1.2.5 Integration and testing 6
1.2.6 Available bundles 6
1.2.7 Useful links 6
1.3 Setting up the Deviser Edit tool 7
2 Defining an SBML Level 3 Package 8
2.1 Specify the general package information 8
2.2 Add the version number 10
2.3 Add class information 11
2.3.1 An SBML ‘element’ or ‘class’ 11
2.3.2 General class description 12
2.3.3 Adding attributes and child elements 15
2.3.4 Example 1 - Adding a class with no containing ListOf 16
2.3.5 Example 2 - Adding a class with a containing ListOf 17
2.3.6 Example 3 – Adding a base class and derived classes 20
2.4 Add plugin information 23
2.4.1 What is a plugin ? 23
2.4.2 General plugin information 24
2.4.3 Example 4 – Extending a core element 25
2.4.4 Example 5 – Extending a core element with attributes only 26
2.4.5 Example 6 – Extending a non-core element 27
2.5 Add enum information 27
2.5.1 Example 7 – Adding an enumeration 27
2.6 Mappings 30
2.7 Results 31
2.7.1 Validating the description 31
3 Using Deviser 33
3.1 View UML diagrams. 33

3.2 Generate libSBML package code. 34
3.3 Generate basic specification documentation. 35
3.4 Integrate and test the package with libSBML. 35
3.5 Command line 36
4 Appendix A 37
5 References 39

Figure 1 Adding information about other software. 7
Figure 2 Illustrating the first step in specifying the 'foo' package. 8
Figure 3 The 'requires additional code' check box. 9
Figure 4 The Version sheet 10
Figure 5 SBML snippet showing element with attributes and child elements 11
Figure 6 Snapshot of part of libSBML class hierarchy 12
Figure 7 The 'Add Class' sheet 13
Figure 8 Definition of the FooKineticLaw class. 16
Figure 9 Specifying the FooParameter class. 17
Figure 10 Specifying the base class 'FooRule'. 20
Figure 11 Specifying the class Assignment. 21
Figure 12 libSBML class hierarchy showing 'plugins' to the Model class 23
Figure 13 The 'addPlugin' sheet. 24
Figure 14 Specifying the extension of SBML Level 3 Core Reaction by package foo. 25
Figure 15 Specifying the extension of SBML Level 3 Core Model by package foo. 26
Figure 16 Specifying the extension of SBML Level 3 Qual Transition by package foo. 27
Figure 17 Specifying the Extra class which has an attribute of type enum. 28
Figure 18 Specifying the Sign_t enumeration. 29
Figure 19 Identifying the origin of classes from other packages 30
Figure 20 The complete description of the foo package 31
Figure 21 Validating the package description 32
Figure 22 Report following the Fix Errors command 32
Figure 23 The UML windows 33
Figure 24 The Generate window 34
Figure 25 Integration and Testing tab selected on the Generate window. 35

1 Getting started

1.1 Functionality

The basic Deviser Edit tool allows a user to define an SBML L3 package. Once defined the following functions are available.

1. Create and view a UML (Unified Modeling Language™ (UML®) Resource Page) diagram.
2. Generate the necessary libSBML (Bornstein, Keating, Jouraku, & Hucka, 2008) code for the package.
3. Generate TeX files (TeX) and generate a pdf of a basic specification document for the package.
4. Integrate and test the package with libSBML.

The Deviser Edit tool outputs an XML (Bray, Paoli, Sperberg-McQueen, Mahler, & Yergeau, 2004) description of the package. This description is then used by Deviser code to generate the requested files. Samples are included with the source code and the full text created by the examples used in this manual is given in Appendix A.

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

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

This program takes as input a Deviser XML file and either

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

In most cases additional software will be necessary.

1.2 Prerequisites

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

1.2.1 The Deviser Edit tool

The Deviser Edit tool will run on Windows and on Linux systems with Mono installed. It does not as yet run on a Mac OS X system.

- Mono

1.2.2 UML diagrams

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

- Internet connection

1.2.3 Source code for libSBML

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

- Python Interpreter

1.2.4 Basic documentation

Deviser will generate the necessary TeX files with only a Python Interpreter present. However in order to create a PDF it will be necessary to have access to a version of pdflatex. On Windows having MiKTEX (http://miktex.org/) installed provides this functionality. The PDF generation also requires the SBML documentation class files sbmlpkgspec (https://sourceforge.net/projects/sbml/files/specifications/tex/).

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

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

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

and run the following from the command line

- tlmgr init-usertree

- tlmgr install bbding
- tlmgr install fourier

1.2.5 Integration and testing

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

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

1.2.6 Available bundles

This release includes an installer for Windows. This includes a Python interpreter and the sbmlpkgspec files in addition to the Deviser Edit tool and the Deviser code.

An archive for Linux is available. This includes the sbmlpkgspec files in addition to the Deviser Edit tool and Deviser code.

1.2.7 Useful links

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

1.3 Setting up the Deviser Edit tool

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

Select Edit->Edit Preferences

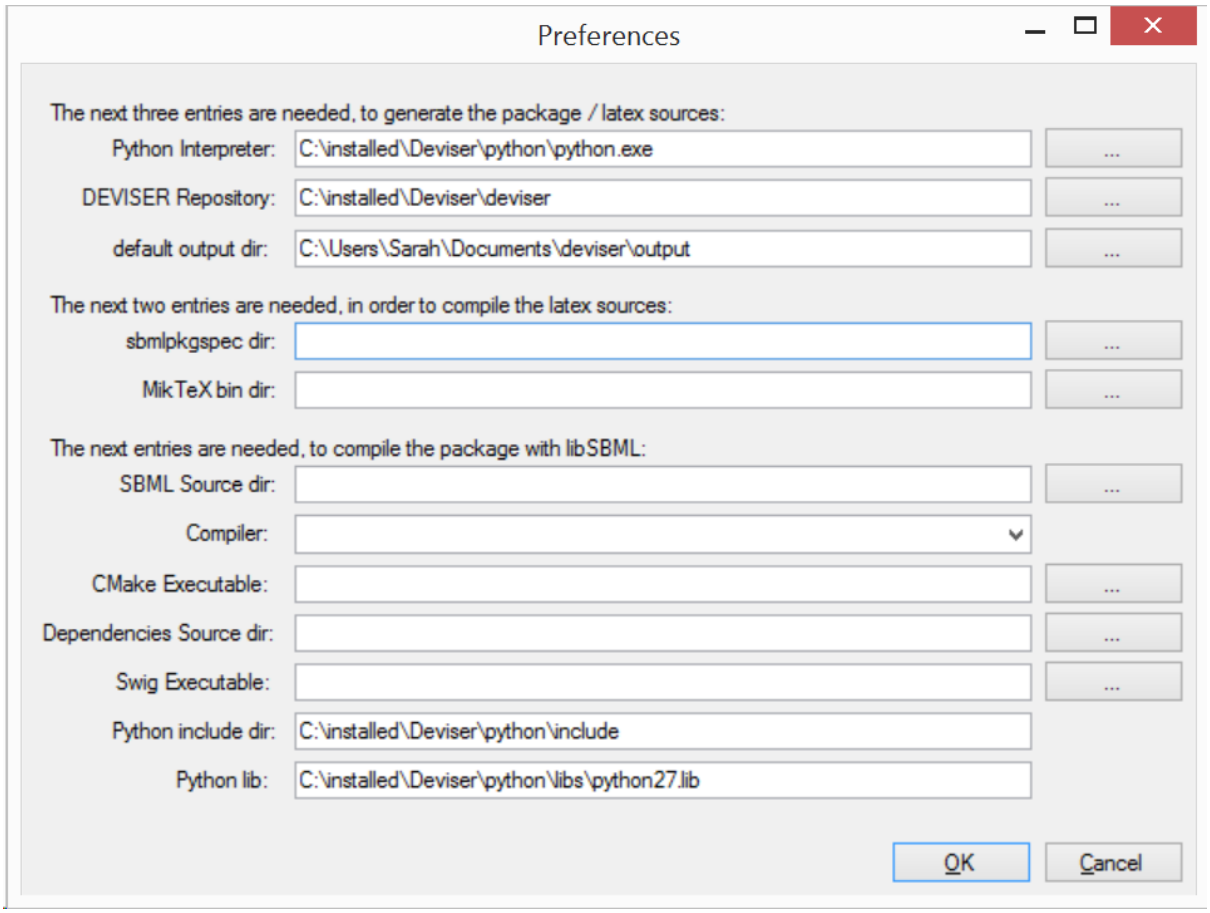


Figure 1 Adding information about other software.

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

2 Defining an SBML Level 3 Package

SBML Level 3 is the most recent specification of SBML. It is a modular language, with a core comprising a complete format that can be used alone (The Systems Biology Markup Language (SBML): Language Specification for Level 3 Version 1 Core, 2010). Additional Level 3 packages are layered on this core to provide additional, optional features.

It is necessary to specify the structure of the SBML Level 3 package before invoking other functionality available within Deviser. A series of sheets guide you through the process in an intuitive manner. For the purpose of this manual we will work step by step through creating an imaginary package 'Foo'.

2.1 Specify the general package information

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

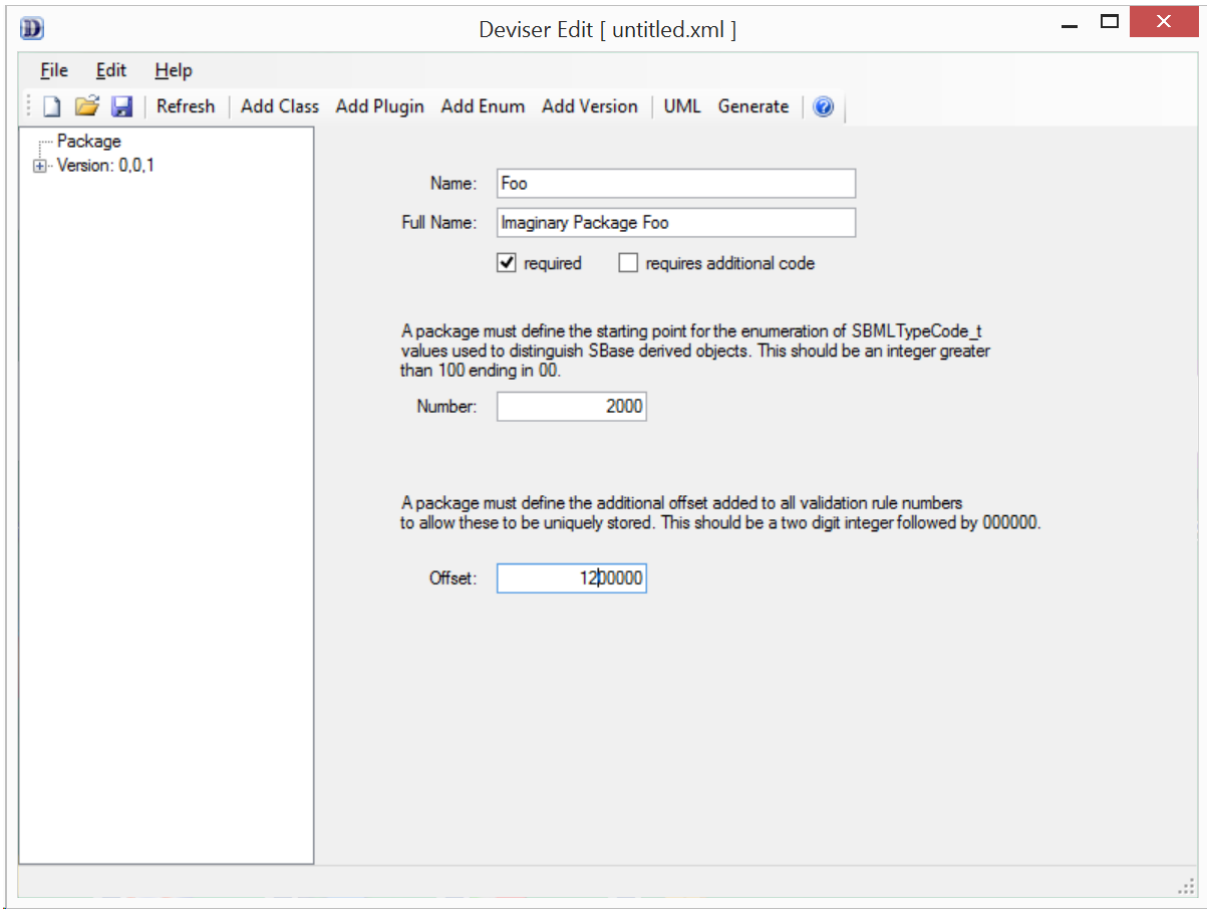


Figure 2 Illustrating the first step in specifying the 'foo' package.

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

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

The **required** checkbox is used to indicate whether the package may change the mathematical interpretation of the core model and corresponds to the required attribute on the <sbml> element declaring this package.

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

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

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

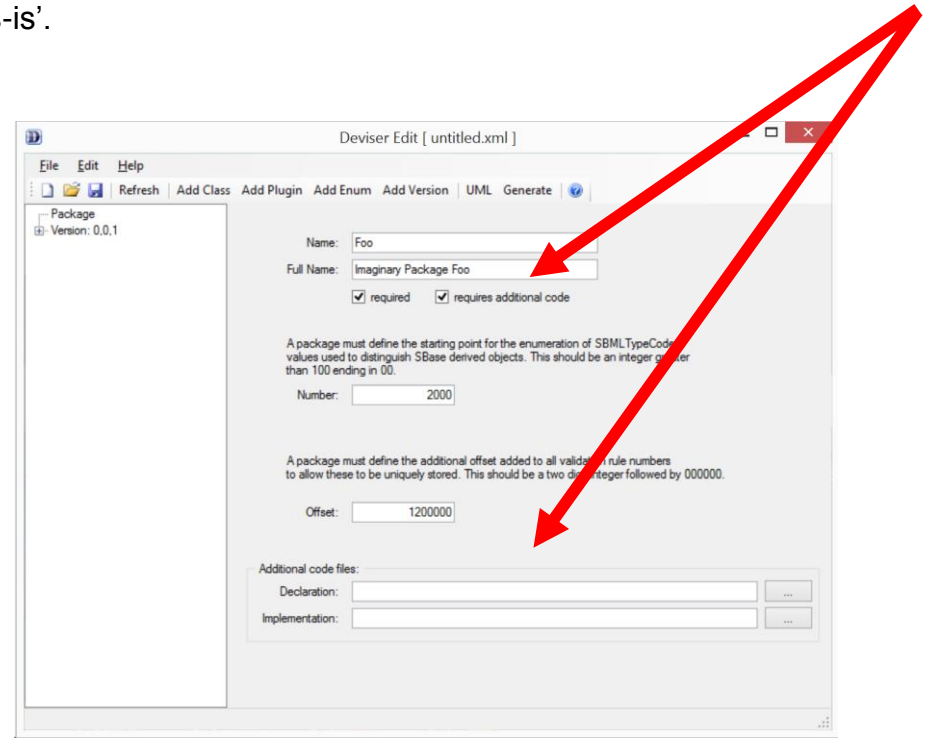


Figure 3 The ‘requires additional code’ check box.

2.2 Add the version number

Highlight ‘Version’ in the tree on the left hand side.

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

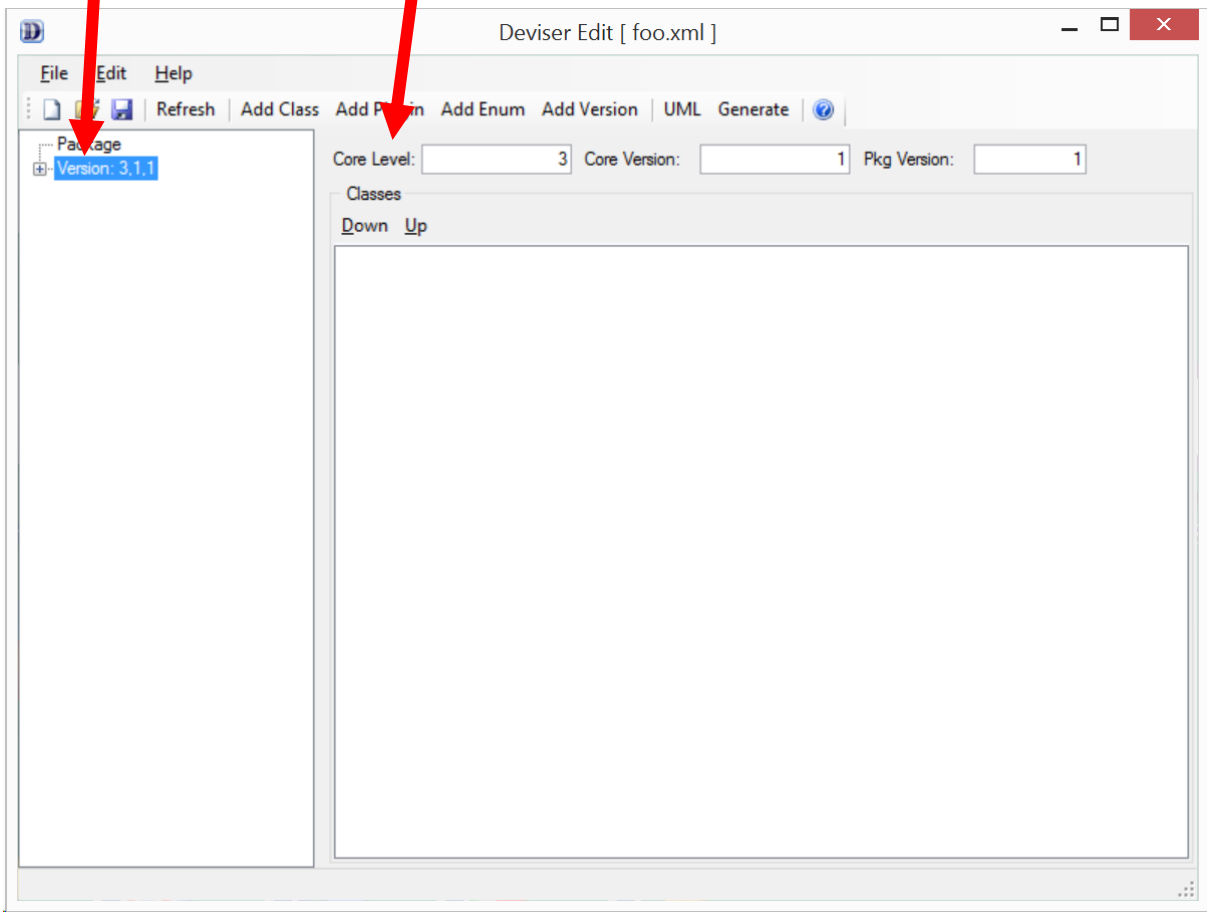


Figure 4 The Version sheet

Whilst Deviser Edit will allow you to specify more than one version for a given package the code does not yet deal with this appropriately.

Note once classes have been specified they will be listed on this sheet (see Figure 20) and the order in which they are listed can be changed. This order dictates the order in which the generation code processes the classes. This can be useful in ensuring documentation is written out in a specific order.

Expanding the tree in the left hand panel shows the aspects of the package that can now be added. We shall work through them here.

2.3 Add class information

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

2.3.1 An SBML ‘element’ or ‘class’

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

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

Figure 5 SBML snippet showing element with attributes and child elements

Figure 5 shows an SBML snippet whilst Figure 6 shows a snapshot of libSBML class hierarchy corresponding to this SBML. Note the correspondence of names and the getXYZ functions etc.

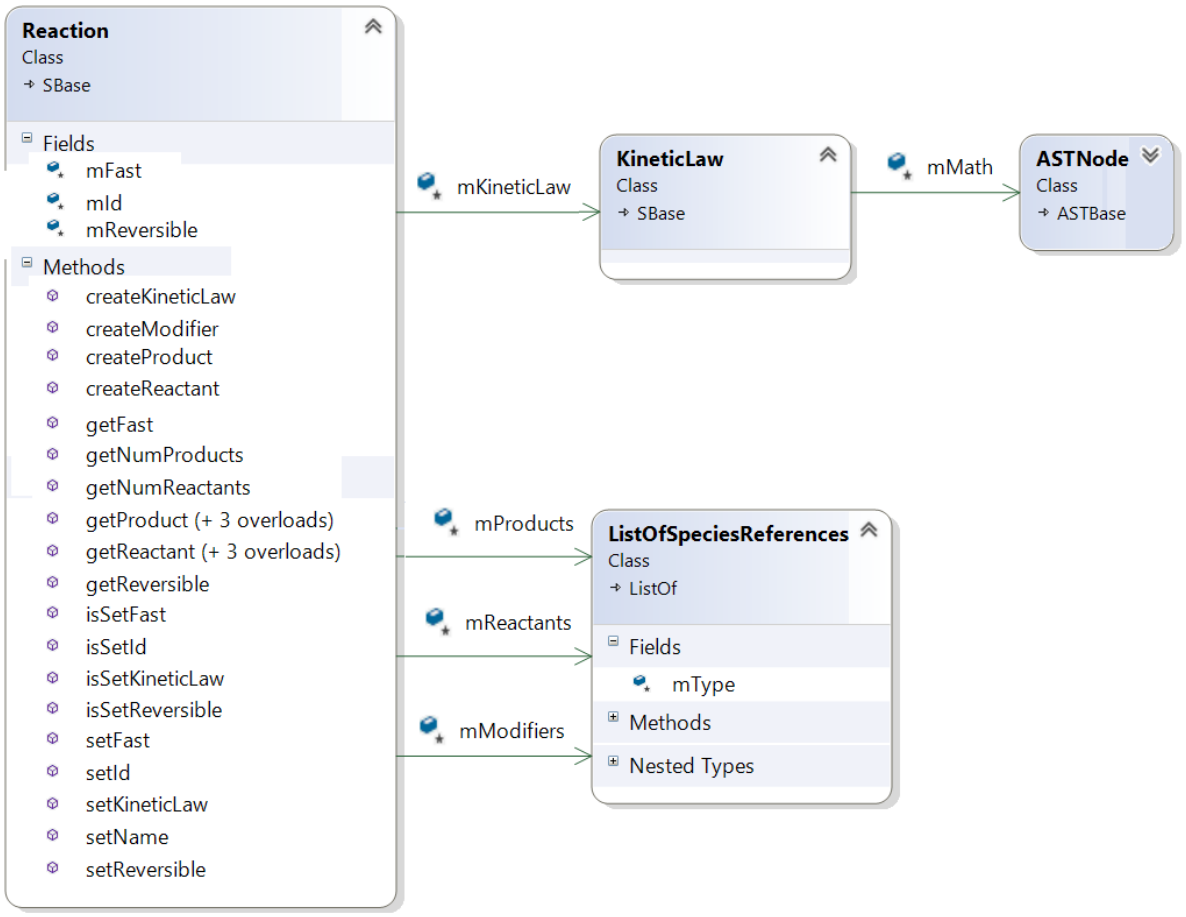


Figure 6 Snapshot of part of libSBML class hierarchy

2.3.2 General class description

We use class to mean the description of an XML element. You will need to specify the details for every new XML element that the package defines. This includes such things as attributes, child elements, whether the element occurs in a listOf etc.

Select ‘Add Class’ from the toolbar or the ‘Edit’ menu.

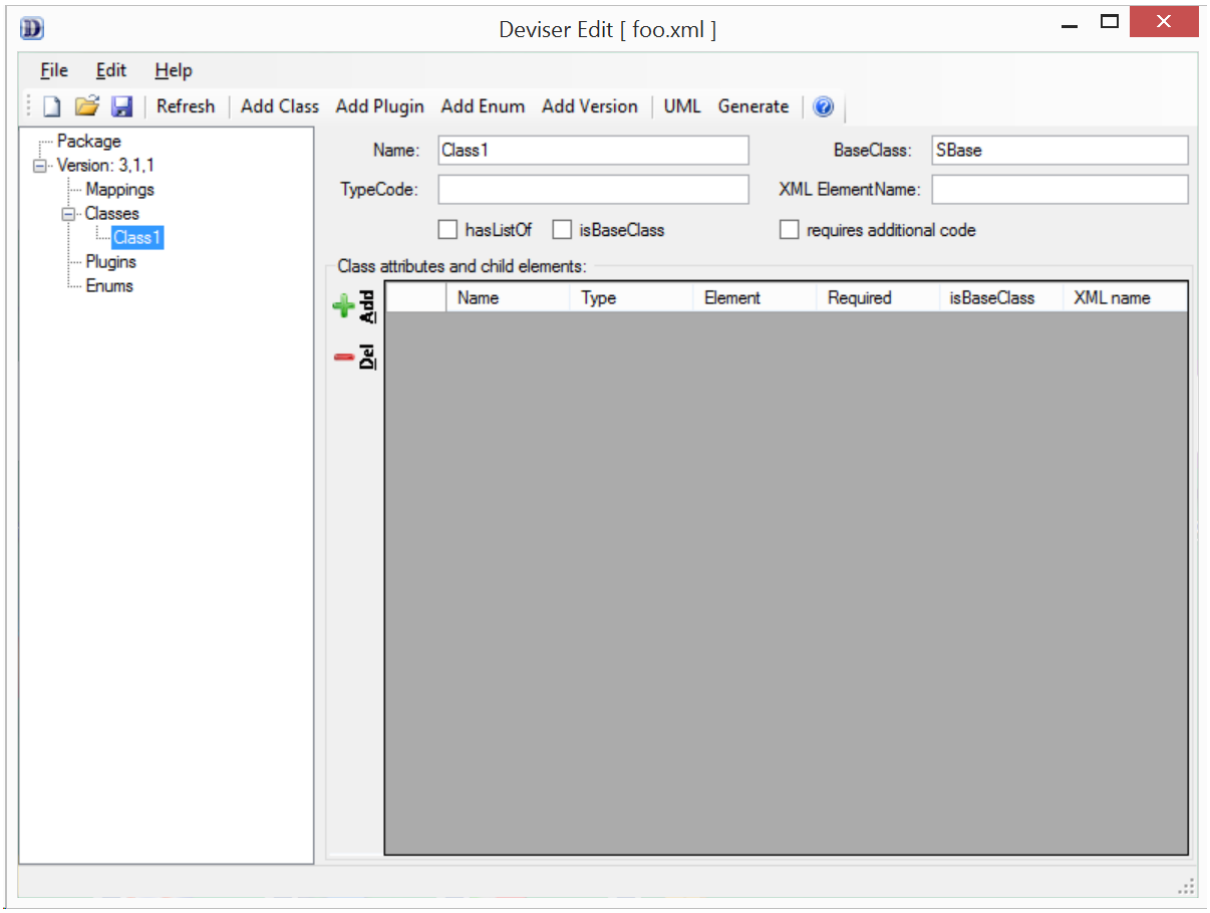


Figure 7 The ‘Add Class’ sheet

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

The **BaseClass** field gives a base class if this class derives from a base. It defaults to SBase for specifying SBML classes as generally these derive from SBase.

The **TypeCode** field is a value that will be used in an enumeration of the types for this package.

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. In most cases there is no need to specify this, as the name displayed corresponds to the class name. An example of where this might be different is if two packages use the same class name and it is necessary to distinguish between these in code. The example in Figure 8 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. Note that if blank this field defaults to the value given in the Name field.

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 ListOfBars class corresponding to the bar class being described.

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

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

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

2.3.3 Adding attributes and child elements

Here we expand on the fields for describing the attributes and child elements of a class as shown in Figure 7.

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

The **Type** field gives the type of the attribute or child. The allowed types for an attribute are the datatypes allowed by SBML. These are (with accepted variations)

SId, SIdRef, string, bool/boolean, double, int/integer, uint/unsigned integer

UnitSId, UnitSIdRef

If the attribute is of an enumeration type defined within the package it should have type

enum

If the object being defined is a child element it can have one of the following types:

element – the object is a single instance of this element

lo_element – the object is a ListOf element

inline_lo_element – the object is a baseClass that may have child elements

The **Element** field provides additional information depending on the type of the object being described. Table 1 describes how this field should be populated.

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

Table 1: The attribute/child element type and appropriate values for the Element field

The **Required** field indicates whether the attribute or child element is mandatory.

The **isBaseClass** field indicates that the child element is a base class and not instantiated directly.

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.

2.3.4 Example 1 - Adding a class with no containing ListOf

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

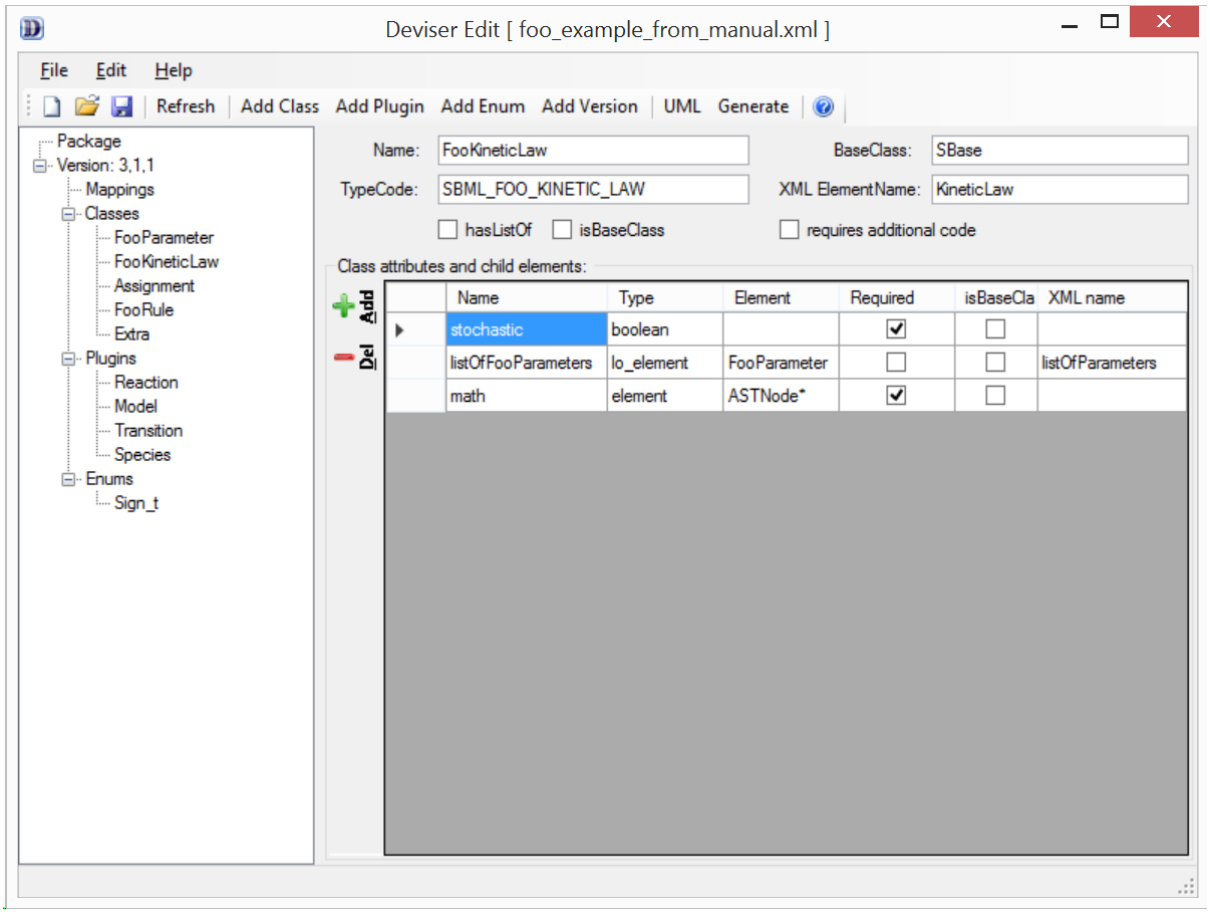


Figure 8 Definition of the FooKineticLaw class.

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'.

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 tick the required checkbox.

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.

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

2.3.5 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.

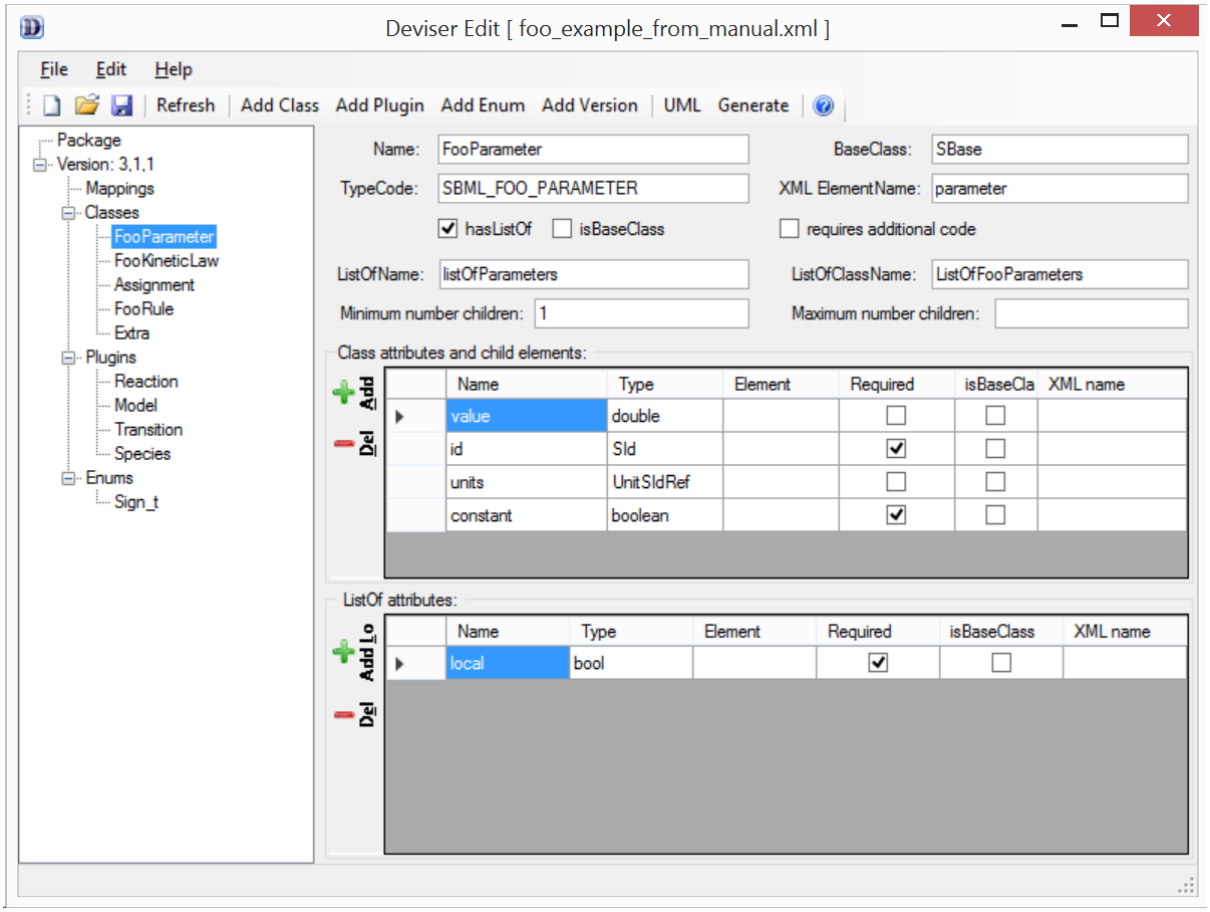


Figure 9 Specifying the FooParameter class.

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

The **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.

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

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

The **Minimum number of children** field is used to indicate the minimum number of child objects of type Bar a ListOfBars expects. Currently in SBML ListOf elements cannot be empty and so must have a minimum of 1 child; which we have indicated in our example.

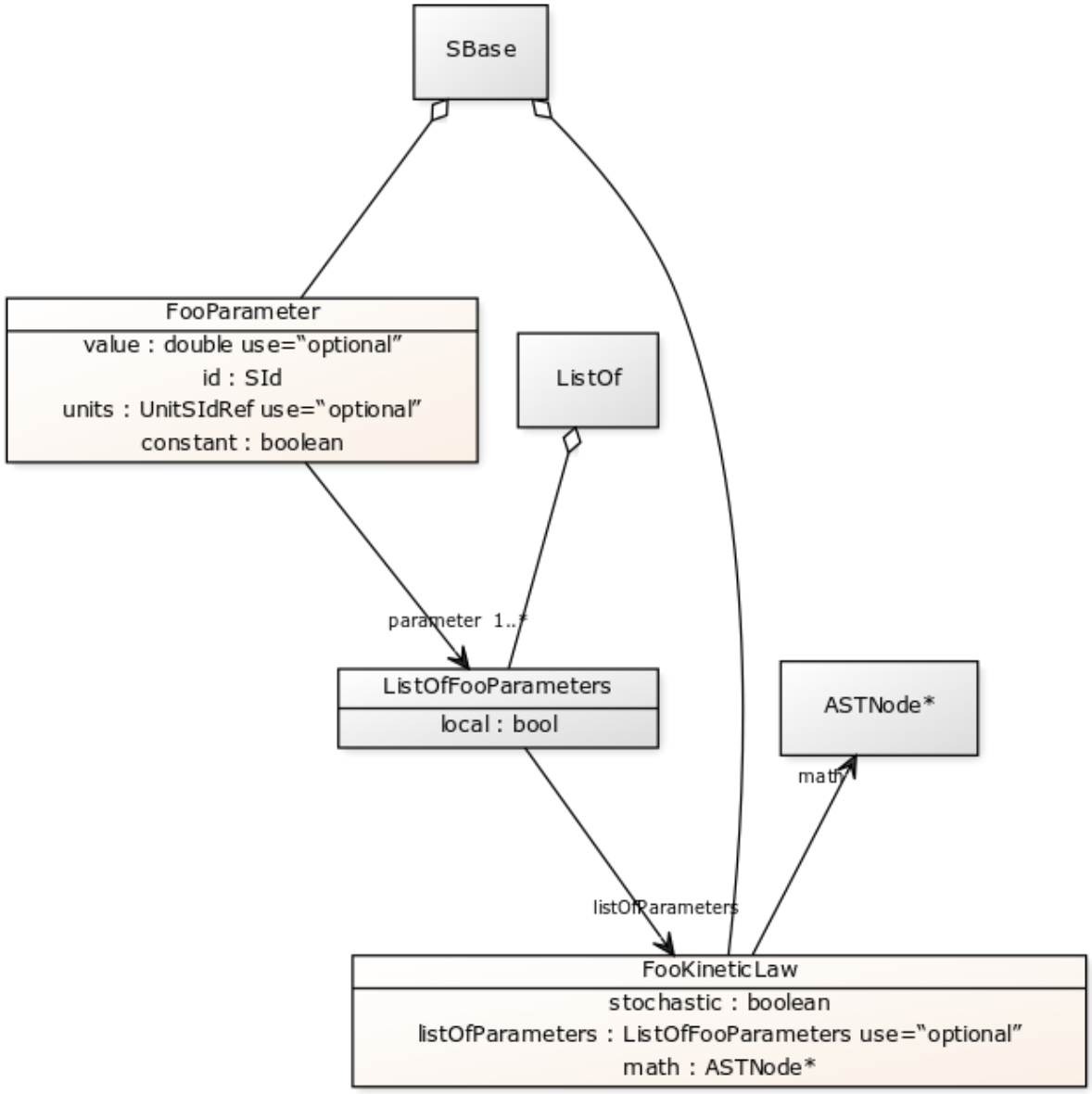
The **Maximum number of children** field is used to indicate the maximum number of child objects a ListOf expects.

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.

A UML diagram of the package 'foo' as described so far in examples 1 and 2 is shown on the next page with the XML representation of the KineticLaw shown in below.

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

SBML snippet of the foo kineticLaw



The UML diagram produced by Deviser following the definition of package 'Foo' in Examples 1 and 2.

2.3.6 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 10).

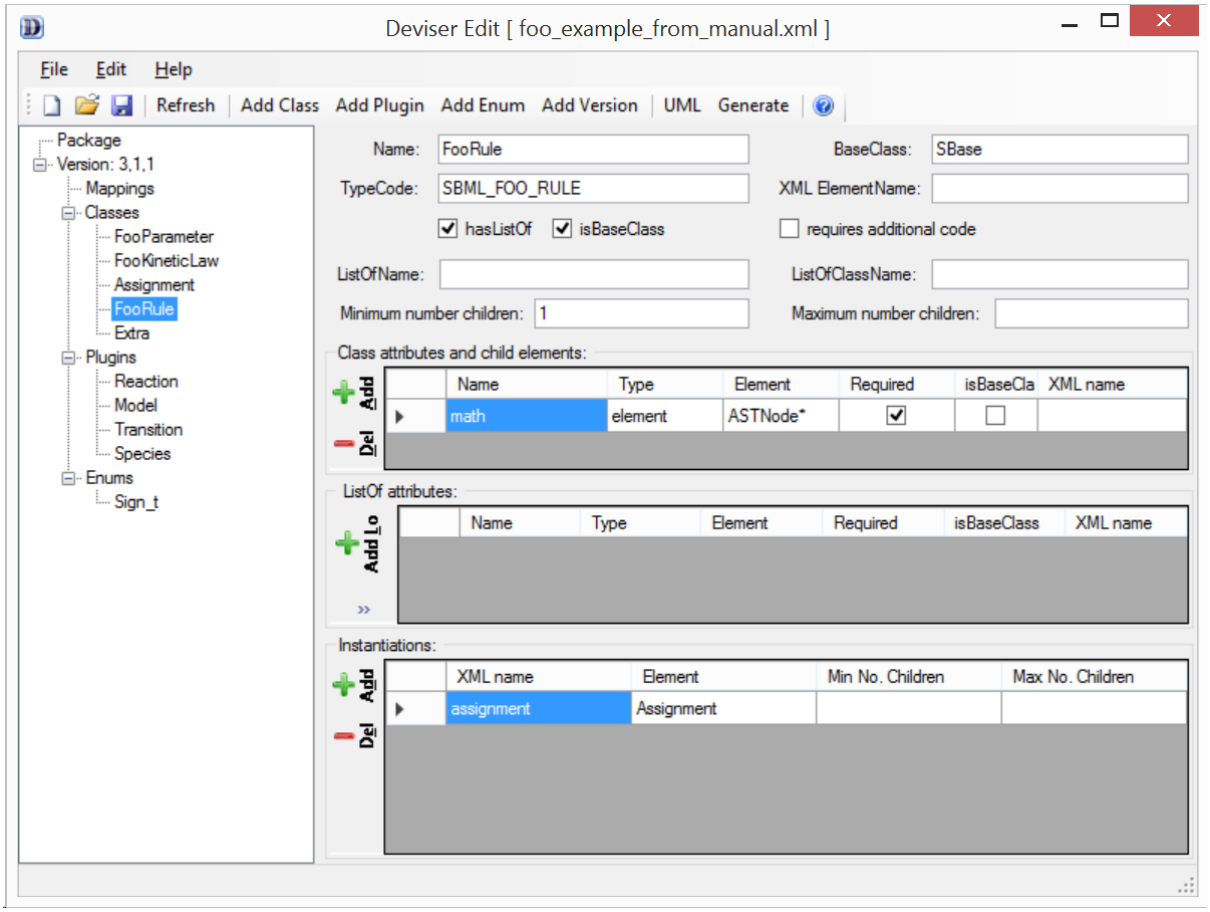


Figure 10 Specifying the base class 'FooRule'.

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

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

The **Instantiations** table allows you to specify the class(es) that will be derived from this base class.

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

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

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

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

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

Here we have specified that the ListOfFooRules may contain objects of type Assignment. We specify Assignment as a new class as in Figure 11.

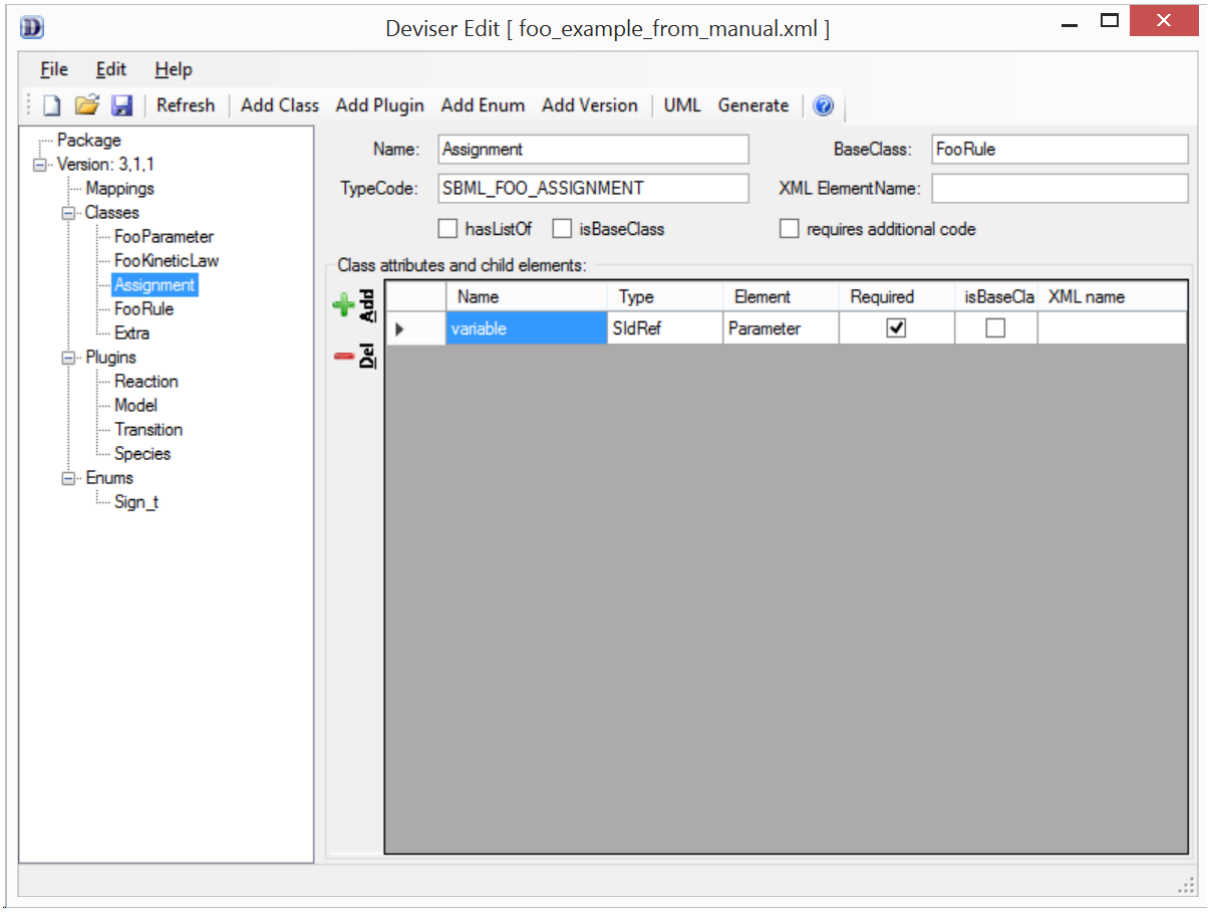


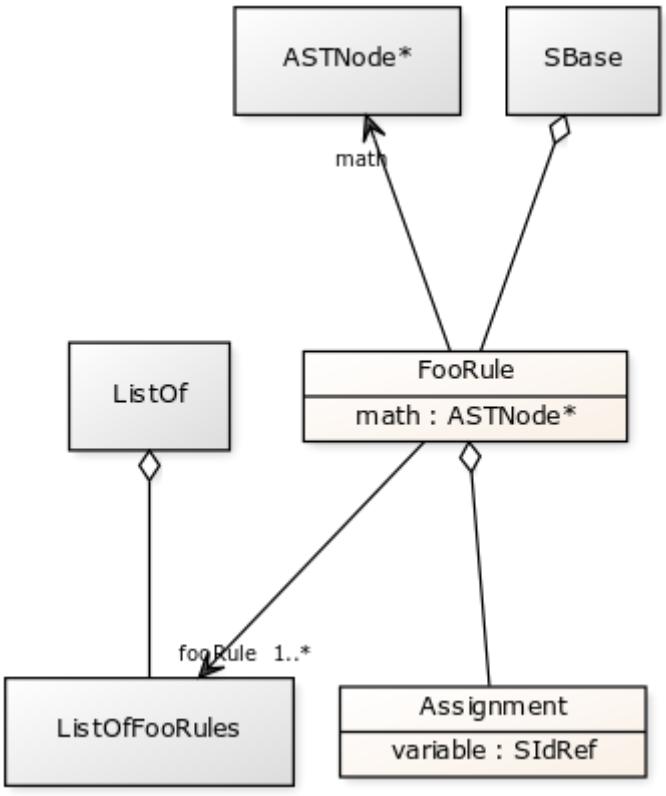
Figure 11 Specifying the class Assignment.

Note on this occasion we have changed the BaseClass to be FooRule.

An SBML snippet of the FooRule specification is shown below with the corresponding UML.

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

An SBML snippet of a listOfFooRules



The UML diagram of the ListOfFooRules.

2.4 Add plugin information

2.4.1 What is a plugin ?

In order to extend SBML Level 3 Core with a package not only is it necessary to define new classes, it is also necessary to attach these elements to an existing point in an SBML model. The simplest case would be that a new element is added to the containing <sbml> element but the point of extension may be much further embedded within the SBML. Here (and indeed within libSBML) we use the term 'plugin' to specify the necessary information that links the new package classes with other classes. Code for any given class in any relevant function then checks whether it has a plugin attached and passes control to the plugin if necessary. Figure 12 shows two plugins that are created 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.

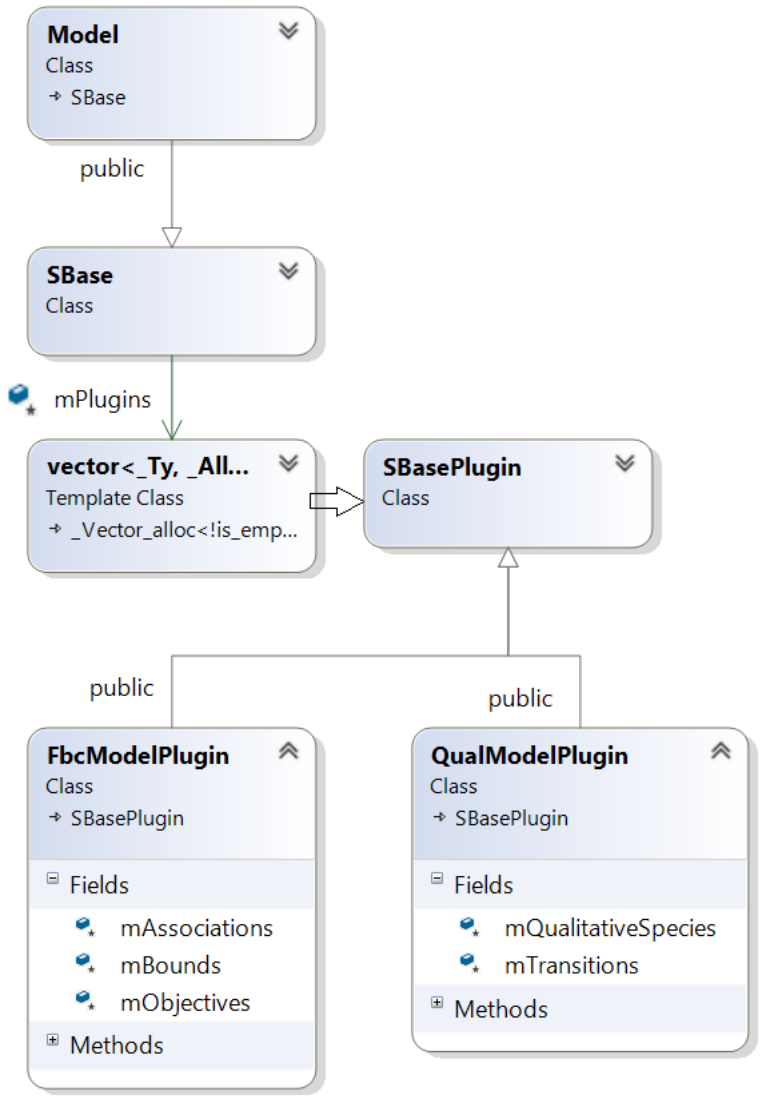


Figure 12 libSBML class hierarchy showing 'plugins' to the Model class

2.4.2 General plugin information

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

Select ‘Add Plugin’ from the toolbar or the ‘Edit’ menu.

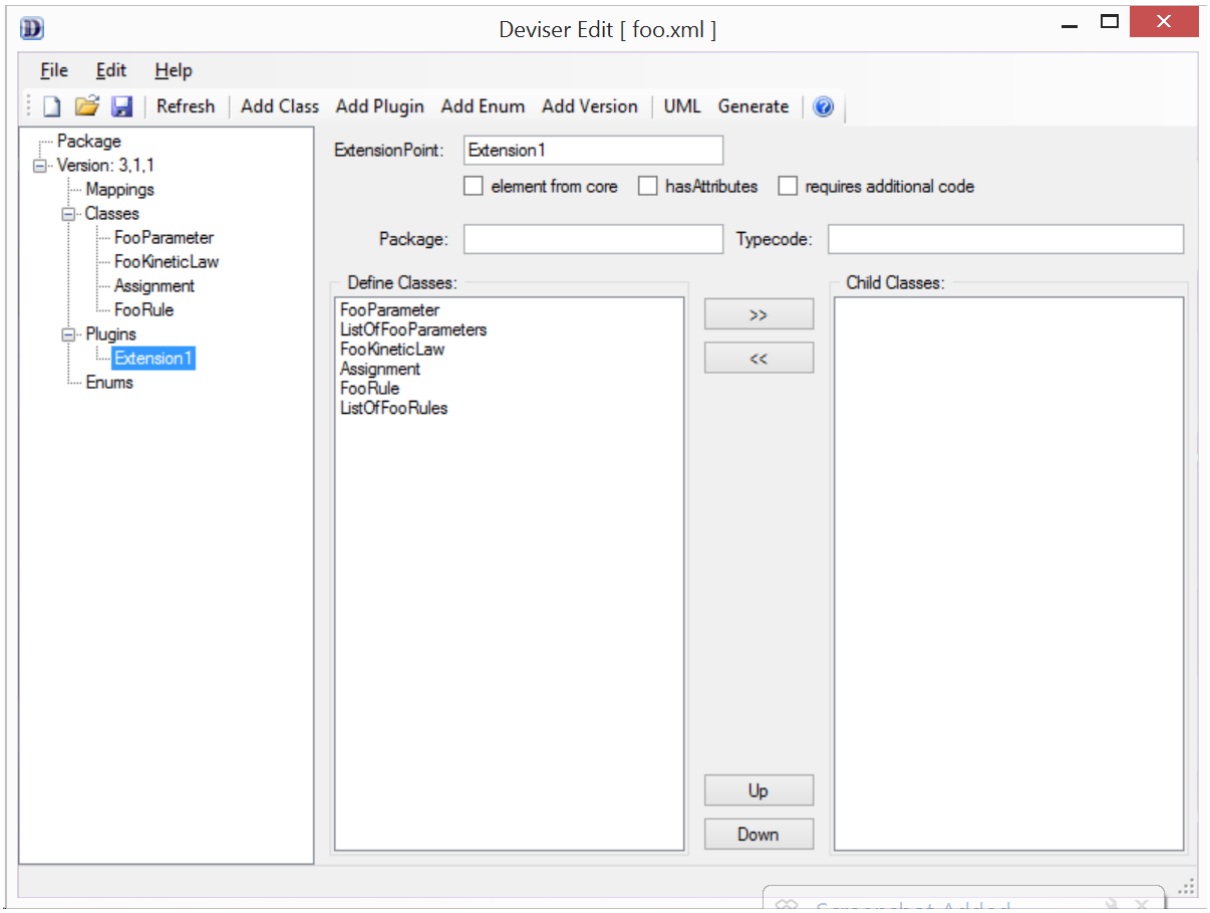


Figure 13 The ‘addPlugin’ sheet.

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

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

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

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

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

2.4.3 Example 4 – Extending a core element

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

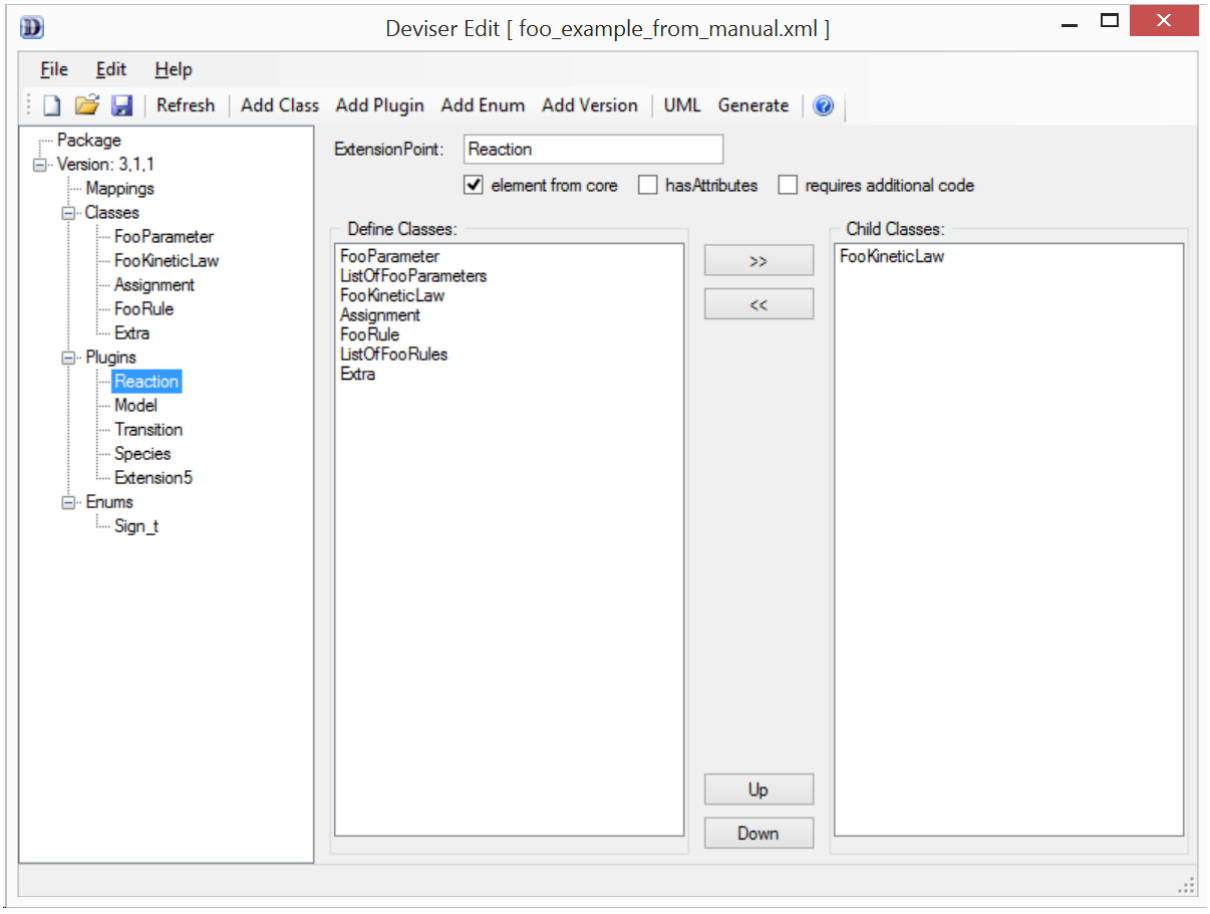


Figure 14 Specifying the extension of SBML Level 3 Core Reaction by package foo.

We fill in the ExtensionPoint with 'Reaction', tick the checkbox to note that the element is from core and highlight FooKineticLaw in the Define classes column and use the arrows to move it to the Child Classes column. Deviser will generate the class FooReactionPlugin.

2.4.4 Example 5 – Extending a core element with attributes only

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

The table **Plugin 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 15). Note that it is not necessary to specify child elements that originate in the package being defined.

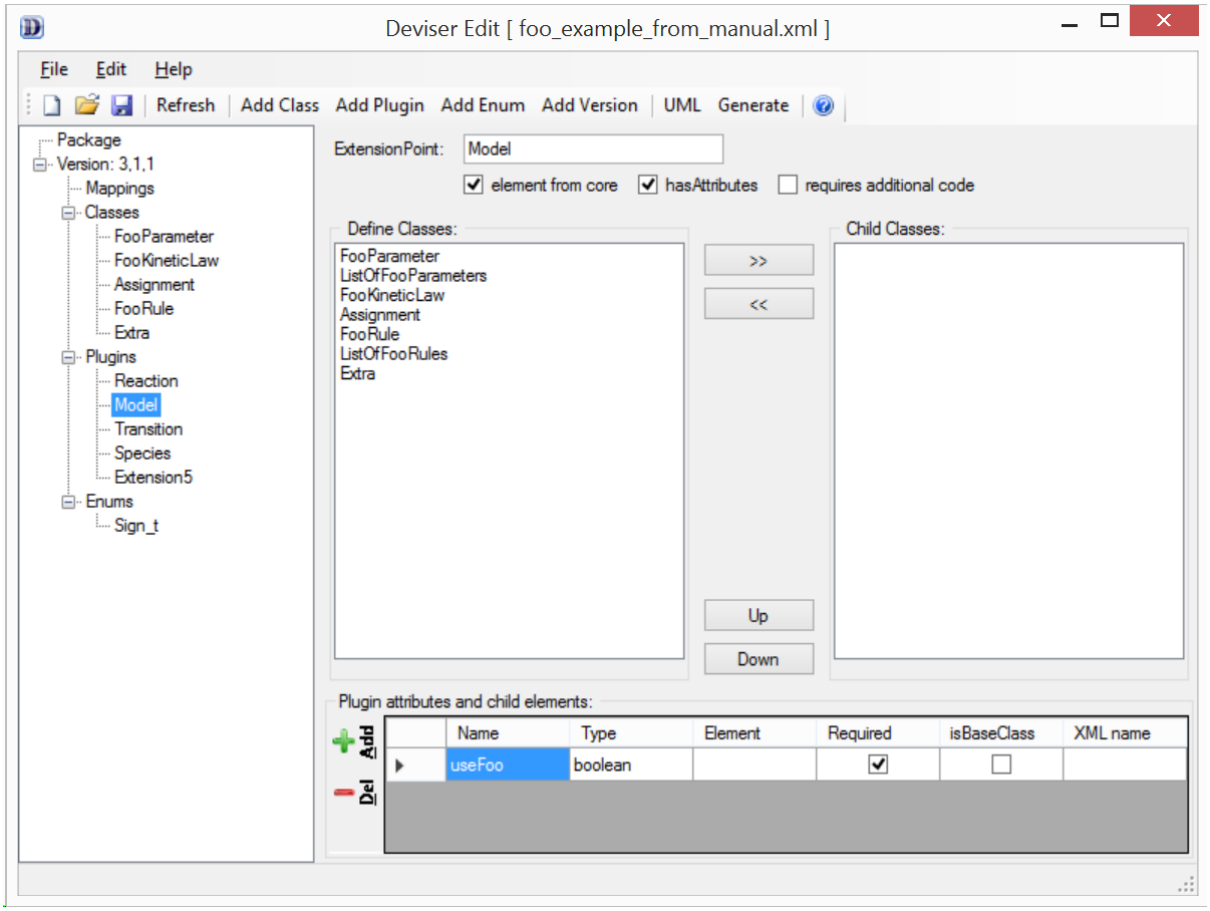


Figure 15 Specifying the extension of SBML Level 3 Core Model by package foo.

2.4.5 Example 6 – Extending a non-core element

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

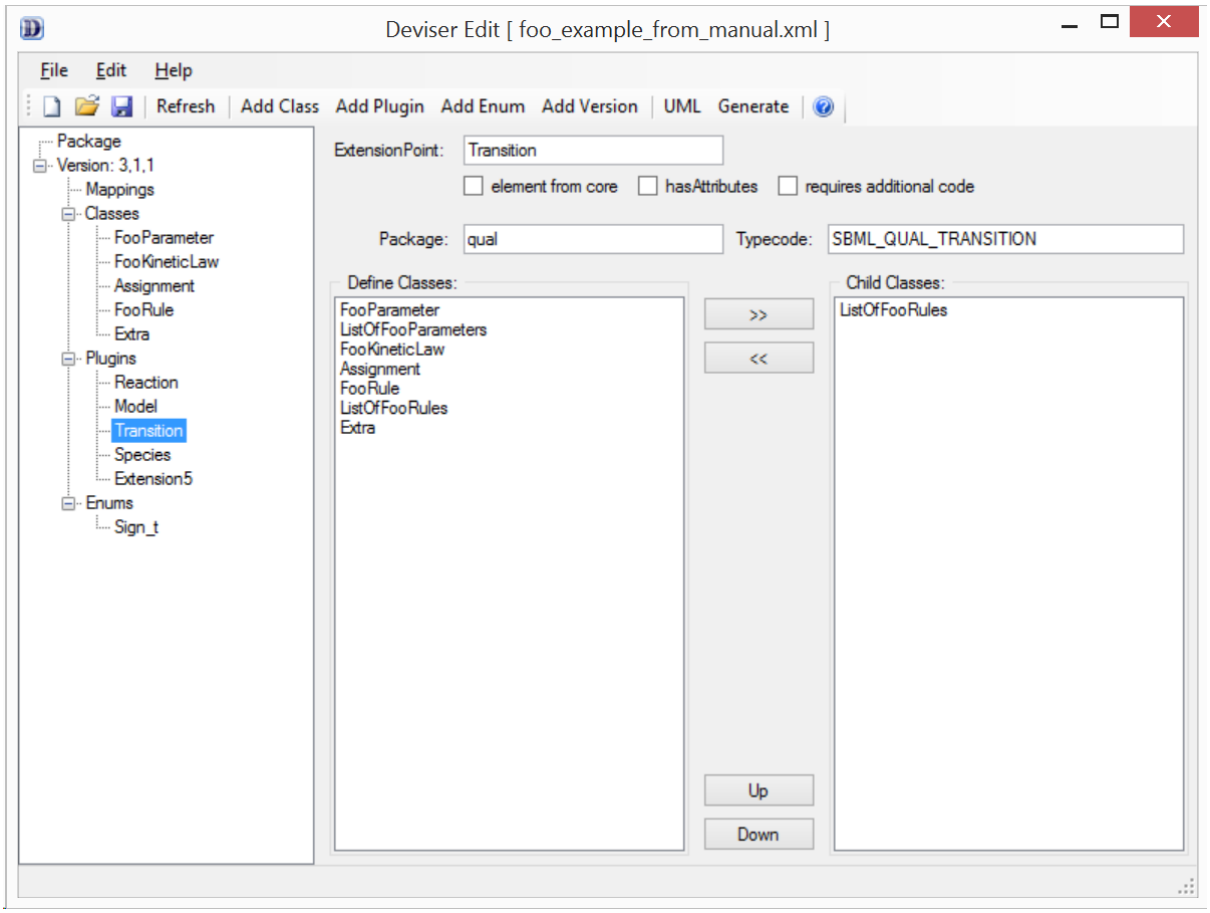


Figure 16 Specifying the extension of SBML Level 3 Qual Transition by package foo.

2.5 Add enum information

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

2.5.1 Example 7 – Adding an enumeration

Assume we have an object ‘extra’ that has an attribute called ‘sign’ which is of an enumeration type ‘Sign_t’. 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_t’ as shown in Figure 17.

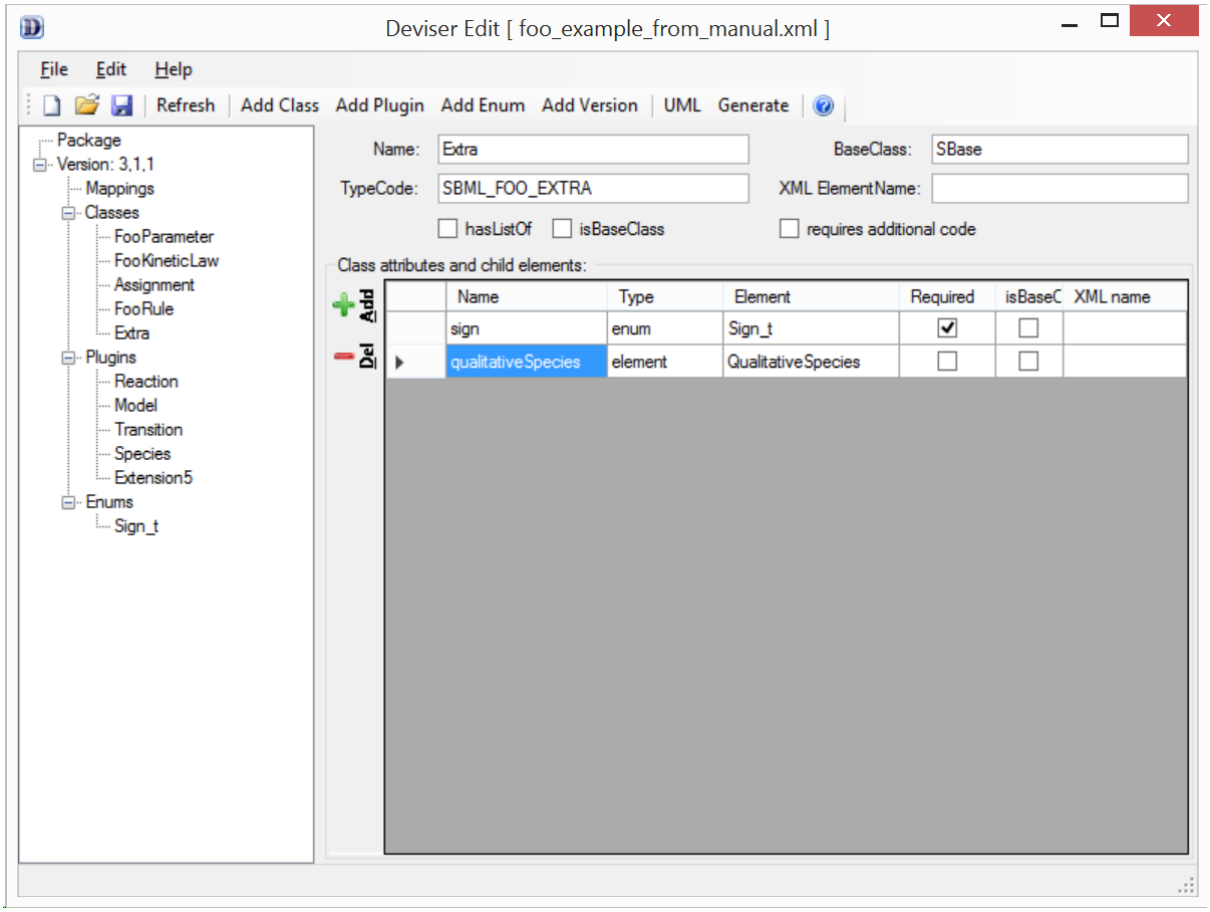


Figure 17 Specifying the Extra class which has an attribute of type enum.

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

The **Name** field is used to declare the name of the enumeration, in this case Sign_t.

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

The **Name** field is the enumeration value.

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

Here (Figure 18) we have specified that the enumeration sign_t has three possible values: ‘positive’, ‘negative’ and ‘neutral. Note the names used reflect the individual values and the package in which they originate.

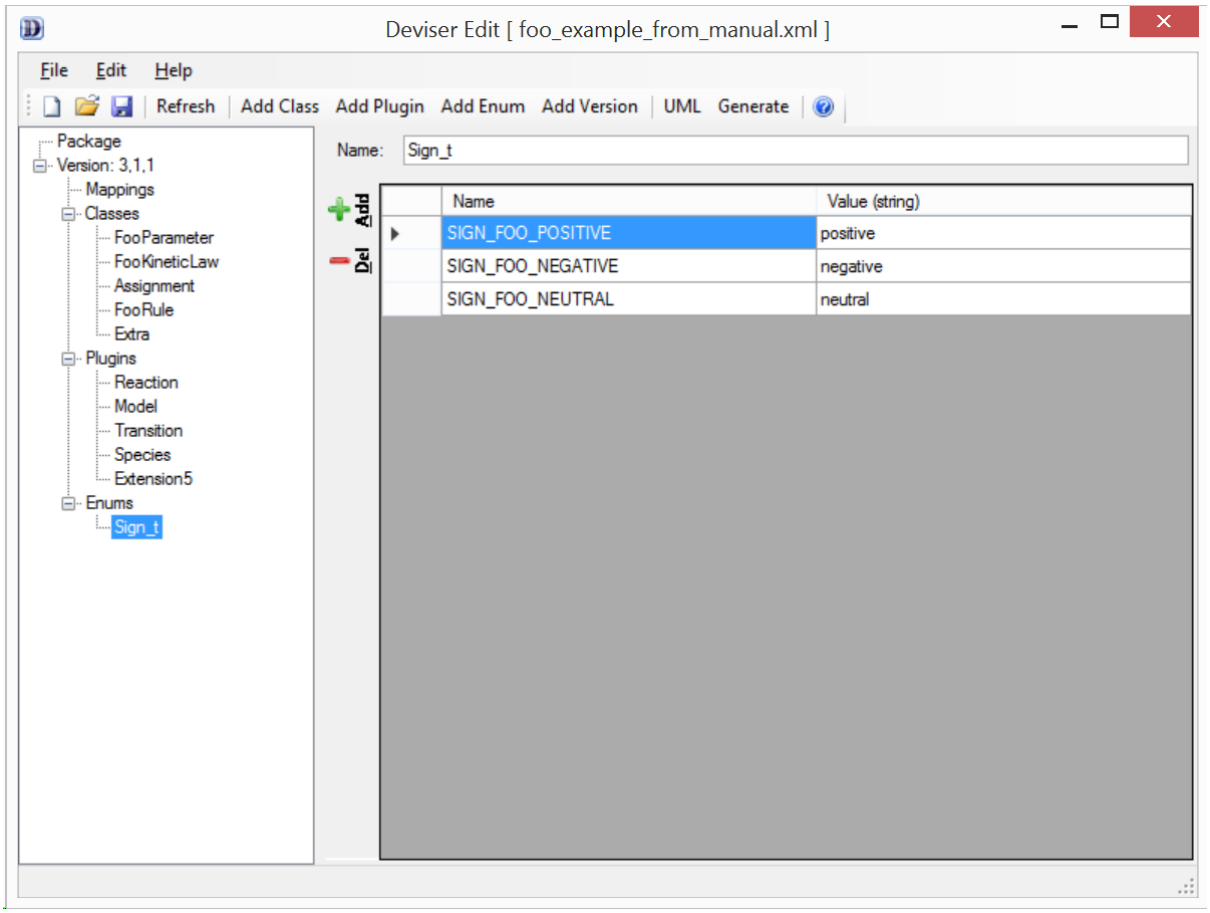


Figure 18 Specifying the Sign_t enumeration.

2.6 Mappings

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

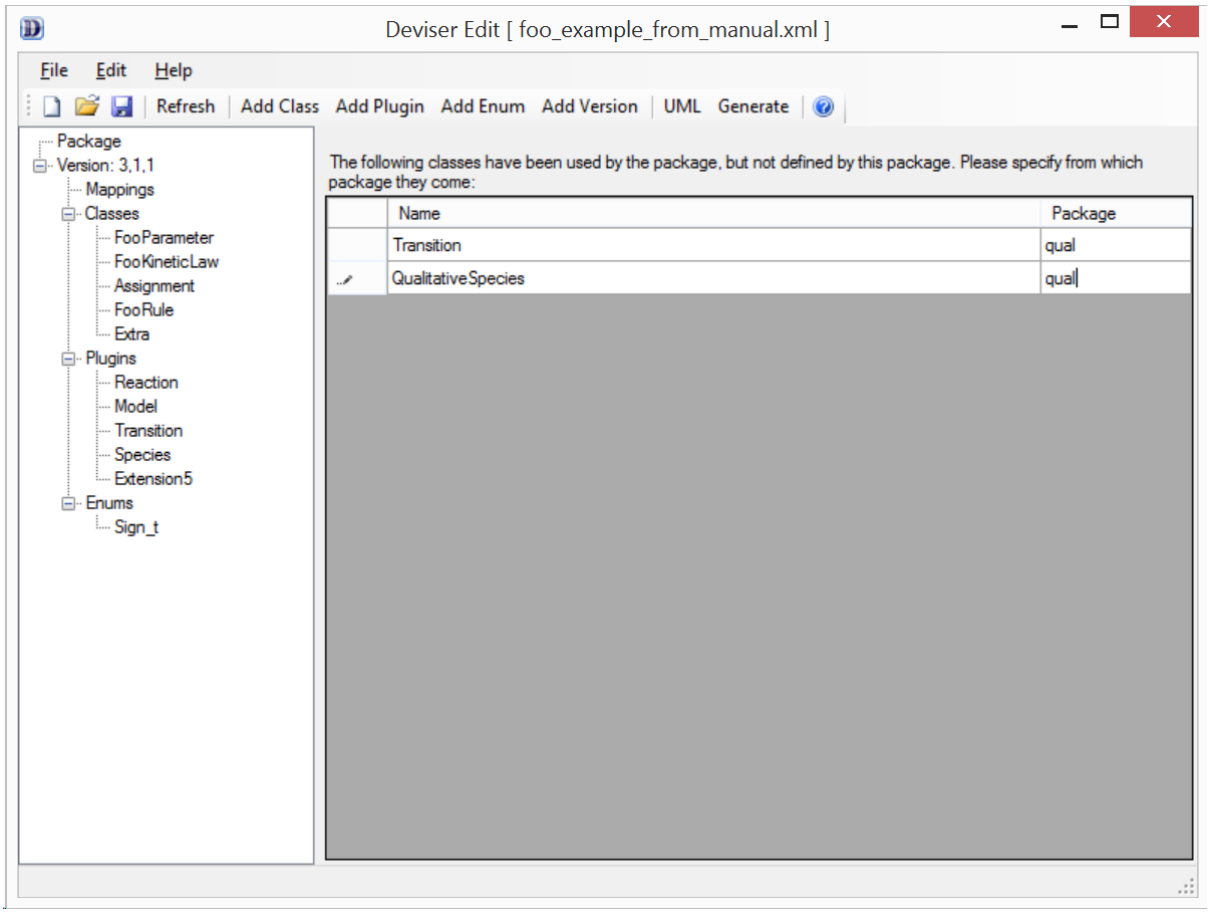


Figure 19 Identifying the origin of classes from other packages

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

2.7 Results

Select ‘Version’ from the tree in the panel on the left hand side. Now that all the classes have been specified these are listed here (see Figure 20) and the ordering can be adjusted.

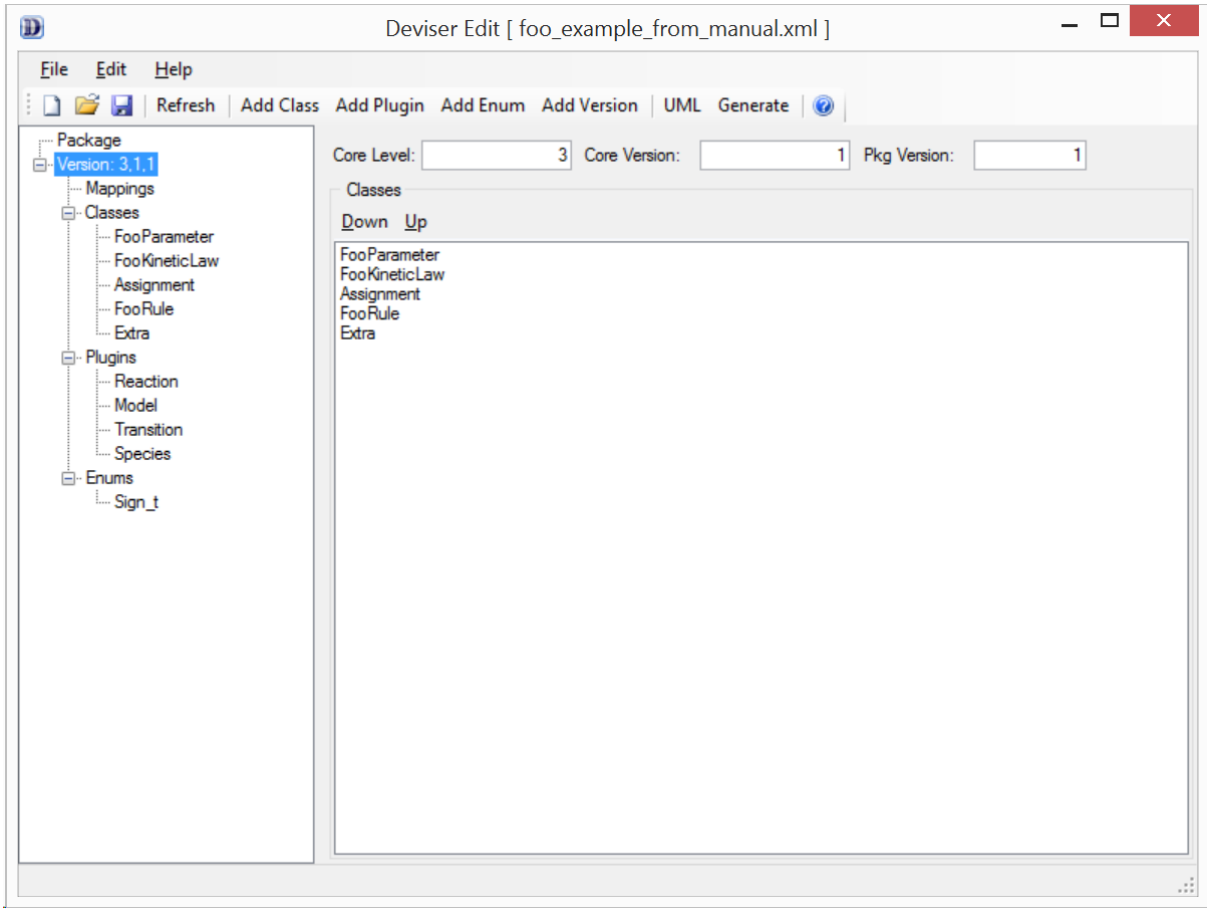


Figure 20 The complete description of the foo package

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

Note this underlying file can be saved at any point and reopened using Deviser Edit.

2.7.1 Validating the description

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

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

A pop-up window will appear with either an error message or a confirmation that everything is consistent.

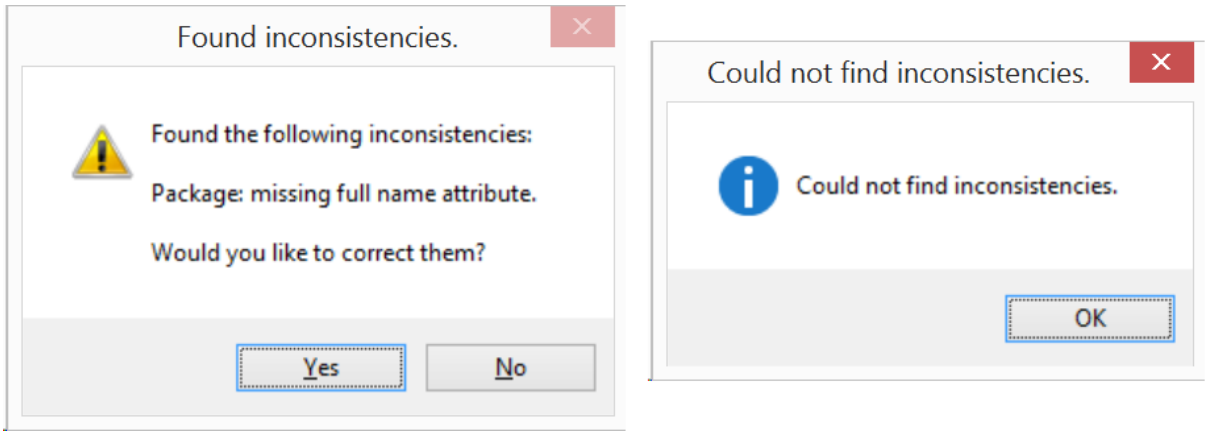


Figure 21 Validating the package description

If errors have been reported and the user selects 'Yes' then these will be corrected.

Fix Errors provides a direct way of validating and then correcting any inconsistencies. Figure 22 shows the result on selecting **Fix Errors** on the same description as the one that produced the inconsistencies reported in Figure 21.

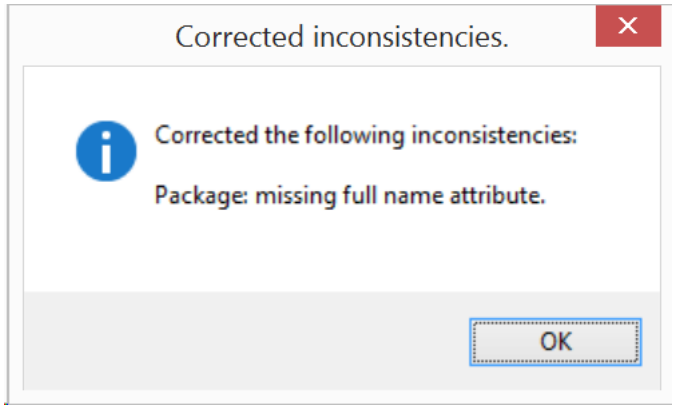


Figure 22 Report following the Fix Errors command

Note it may be necessary to press the **Refresh** button on the toolbar for the 'fixes' to appear.

3 Using Deviser

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

3.1 View UML diagrams.

Once you have specified at least one class you can click on the UML button on the toolbar.

The UML output will appear (the more complex the package the longer it will take) and can be saved.

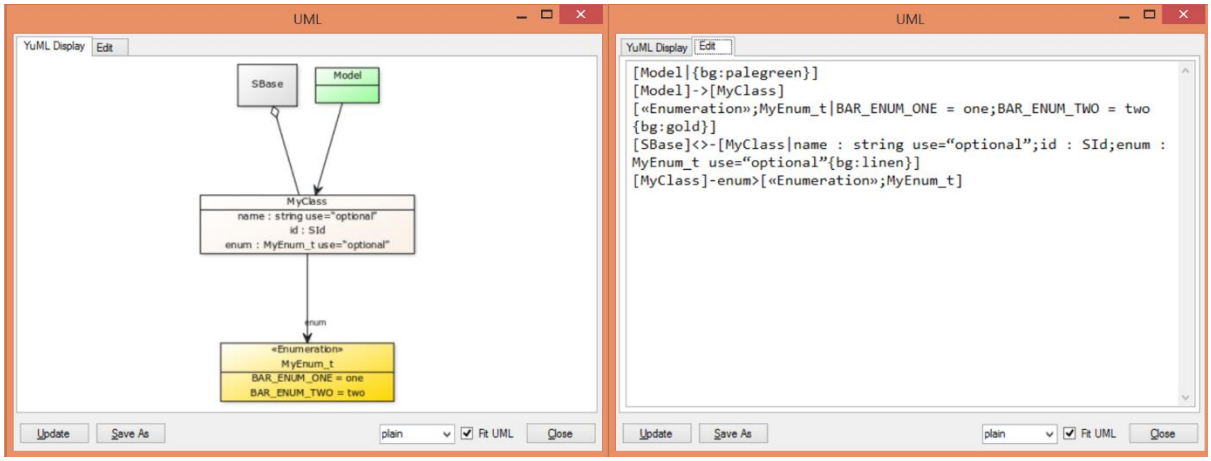


Figure 23 The UML windows

Key:

White	Class from this package
Green	Class from core/other package
Yellow	Enumeration
Grey	Base class

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 specified 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.

The **dropdown combo box** allows a choice of three yUML supported views: plain, nofunky and scruffy.

The **Fit UML** checkbox selects whether to fit the diagram to the viewing window. When deselected the view adds scroll bars to allow the diagram to be viewed.

3.2 Generate libSBML package code.

Click on the 'Generate' button on the toolbar and the Generate window (Figure 24) will appear.

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

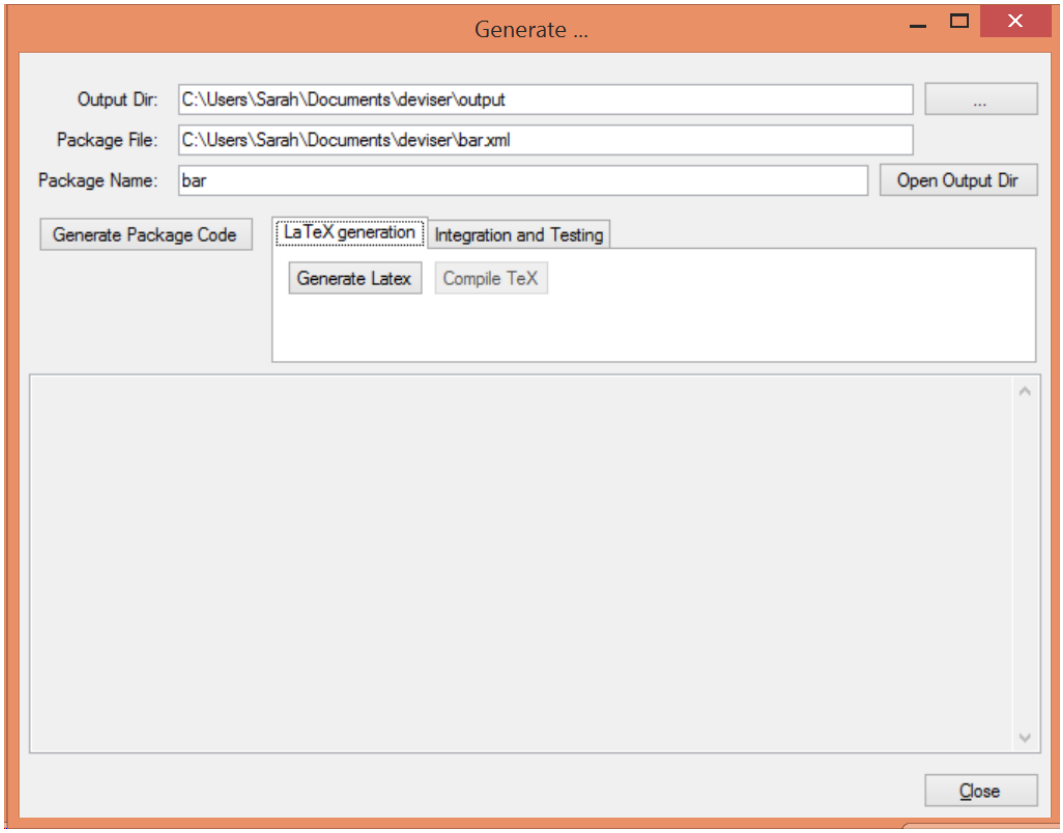


Figure 24 The Generate window

3.3 Generate basic specification documentation.

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

The Generate Latex button will generate three TeX files.

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

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

3.4 Integrate and test the package with libSBML.

Click the **Integration and Testing** tab in the Generate window and a further set of buttons are revealed (Figure 25).

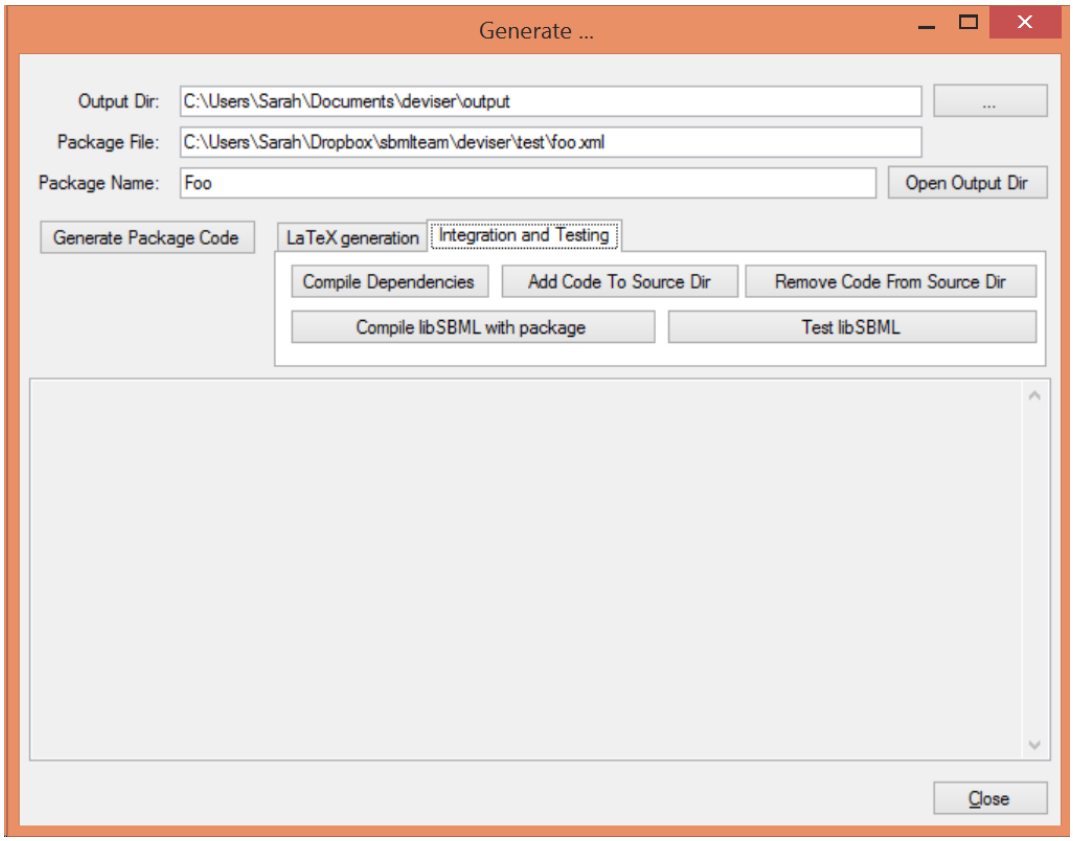


Figure 25 Integration and Testing tab selected on the Generate window.

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

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

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

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

Test libSBML starts an interactive python session with the package loaded to facilitate to manual testing of the code.

3.5 Command line

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

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

This program takes as input a Deviser XML file and either

- --generate (-g) generates the libSBML code for the package
  This generates the complete src package that can be zipped and then unzipped over an existing libSBML src tree.

- --latex (-l) generates a LaTeX scaffold for its specification.
  This generates the adx-validation.tex, macros.tex and body.tex files which can then be integrated into a package pdf using the sbmlpkgspec files.

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

4 Appendix A

```
<?xml version="1.0" encoding="utf-8"?>
<package name="Foo" fullname="Imaginary Package Foo"
        number="2000" offset="1200000" version="1" required="true">
  <versions>
    <pkgVersion level="3" version="1" pkg_version="1">
      <elements>
        <element name="FooParameter" typeCode="SBML_FOO_PARAMETER"
              hasListOf="true" hasChildren="false" hasMath="false"
              childrenOverwriteElementName="false" minNumListOfChildren="1"
              baseClass="SBase" abstract="false" elementName="parameter"
              listOfName="listOfParameters" listOfClassName="ListOfFooParameters">
          <attributes>
            <attribute name="value" required="false" type="double" abstract="false" />
            <attribute name="id" required="true" type="SId" abstract="false" />
            <attribute name="units" required="false" type="UnitSIdRef" abstract="false" />
            <attribute name="constant" required="true" type="boolean" abstract="false" />
          </attributes>
          <listOfAttributes>
            <listOfAttribute name="local" required="true" type="bool" abstract="false" />
          </listOfAttributes>
        </element>
        <element name="FooKineticLaw" typeCode="SBML_FOO_KINETIC_LAW"
              hasListOf="false" hasChildren="true" hasMath="true"
              childrenOverwriteElementName="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"
              hasChildren="false" hasMath="false" childrenOverwriteElementName="false"
              baseClass="FooRule" abstract="false">
          <attributes>
            <attribute name="variable" required="true" type="SIdRef" element="Parameter"
                  abstract="false" />
          </attributes>
        </element>
        <element name="FooRule" typeCode="SBML_FOO_RULE" hasListOf="true"hasChildren="false"
              hasMath="true" childrenOverwriteElementName="false" 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" />
          </concretes>
```

```
      </element>
      <element name="Extra" typeCode="SBML_FOO_EXTRA" hasListOf="false" hasChildren="true"
            hasMath="false" childrenOverwriteElementName="false" baseClass="SBase"
            abstract="false">
        <attributes>
          <attribute name="sign" required="true" type="enum" element="Sign_t"
                abstract="false" />
          <attribute name="qualitativeSpecies" required="false" type="element"
                element="QualitativeSpecies" abstract="false" />
        </attributes>
      </element>
    </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_t">
        <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>
```

5 References

Bornstein, B. J., Keating, S. M., Jouraku, A., & Hucka, M. (2008). LibSBML: An API Library for SBML. *Bioinformatics*, 880-881.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Mahler, E., & Yergeau, F. (2004, February 4). *Extensible markup language (XML) 1.0 3rd edition*. Retrieved from http://www.w3.org/TR/2004/REC-xml-20040204/

Hucka, M., Bergmann, F., Hoops, S., Keating, S., Sahle, S., Schaff, J., . . . Wilkinson, D. (2010). *The Systems Biology Markup Language (SBML): Language Specification for Level 3 Version 1 Core*. Retrieved from http://sbml.org/Documents/Specifications

TeX. Retrieved from http://en.wikipedia.org/wiki/TeX

Unified Modeling Language™ (UML®) Resource Page. Retrieved from http://www.uml.org/