Andre's CellML Debugging Tips Documentation

Release 0.1

David Nickerson

May 05, 2016

Contents

1 Version Control **3**

2 `xmllint` **5**

3 OpenCOR **7**

4 CellML2C **9**

5 CellML2Dot **11**

6 Model flattening **15**
6.1 Version Converter 15
6.2 Model Compaction 15

7 Introduction **17**

Increasingly I am being given CellML models that "don't work", for some degree of *don't work*. These can range from invalid XML documents through to valid CellML models that give subtly different results to what is expected. The goal of this document is to collect in one place a description of the tools and techniques I use myself to try and work out what is wrong with the models I receive to help CellML'ers debug some of these problems on their own.

These tips will evolve over time as new tools are developed and the CellML specification itself changes, but the most recent version will always be available from: http://andres-cellml-debugging-tips.readthedocs.org. If you spot any errors or omissions or have suggestions for improvements, please make use of the tracker over on Bitbucket.

Contents:

CHAPTER 1

Version Control

I can't emphasise enough the importance of using a proper version control system for all your work. Any system is better than no system, but I'd obviously recommend the Physiome Repository. There is some documentation that will help you get started using that repository, but you might also want to look into GitHub, Bitbucket, Google Code, SourceForget, etc...

CHAPTER 2

xmllint

`xmllint` is a command line tool that is usually available on most *nix machines, including OS X (http://xmlsoft.org/xmllint.html). It is part of libxml2 and if its not already on your system you can download binaries for most platforms.

There are two main things that you might want to use `xmllint` for. Checking that a CellML document is valid XML and creating a nicely formatted version of the document. You can achieve both of these with the command:

```
$> xmllint --format <file.xml>
```

which will check that `file.xml` is a valid XML document and print a nicely formatted version of the document to the terminal if it is valid. If your document is quite large, you probably don't want to print it to the terminal as it might obscure some useful warnings. In that case you should do something like:

```
$> xmllint --format <file.xml> > <new-file.xml>
```

which will do the same as above, but now if you have a valid XML document the formatted XML document will be in `new-file.xml`.

CHAPTER 3

OpenCOR

OpenCOR is an open source cross-platform modelling environment that supports CellML. As OpenCOR develops it will hopefully serve as a one-stop-shop for model debugging, as well as all its other useful functions. If there are specific ideas or suggestions you have for the developers, I encourage you to contact them.

This tutorial has some examples that demonstrate how to use OpenCOR and the Physiome Repository that might be useful to work through if you are new to the tools.

CHAPTER 4

CellML2C

CellML2C is a test application that is part of the testing framework of the CellML API. OpenCOR uses essentially the same code when it generates code to perform a simulation, but currently the useful debugging information is not presented to the user. Similarly, PMR2 makes use of the same code to generate the various format procedural codes from a model exposure, but that is only available for exposures. OpenCMISS also uses this code.

To have access to CellML2C you need to build the CellML API yourself, which is not too difficult.

CellML2C is executed as follows:

```
$> CellML2C <model URL>
```

Any URL can be provided, including URLs to remote files such as the model repository and local files. For example, with this `test model`:

```
$> CellML2C path/to/valid-test-model.xml
```

Depending on your platform, you may need to play a bit to get relative local URLs to correctly work with CellML2C and models with imports. Executing the above command (for ABI users) which will result in the following `C` code being generated:

```
/* Model is correctly constrained.
 * No equations needed Newton-Raphson evaluation.
 * The rate and state arrays need 0 entries.
 * The algebraic variables array needs 0 entries.
 * The constant array needs 2 entries.
 * Variable storage is as follows:
 * * Target sin in component sin
 * * * Variable type: constant
 * * * Variable index: 1
 * * * Variable storage: CONSTANTS[1]
 * * Target x in component sin
 * * * Variable type: constant
 * * * Variable index: 0
 * * * Variable storage: CONSTANTS[0]
 */
void SetupFixedConstants(double* CONSTANTS, double* RATES, double* STATES)
{
/* Constant x */
CONSTANTS[0] = 1.0;
/* Constant actual_sin */
CONSTANTS[1] =  sin(CONSTANTS[0]);
}
void EvaluateVariables(double VOI, double* CONSTANTS, double* RATES, double* STATES, double* ALGEBRA
{
```

```
}
void ComputeRates(double VOI, double* STATES, double* RATES, double* CONSTANTS, double* ALGEBRAIC)
{
}
```

and having produced some code, CellML2C is letting you know that your model is most-likely complete (but it does not rule out errors in variable connections, units, etc.). It is sometimes useful to check the first *comment* block in the generated code which describes the variables in the model which are present in the generated code. If variables you are expecting to be state variables show up as `constant` or a model parameter shows up as a `state` variable, then it might suggest something to look into.

If your model is not complete, CellML2C will be unable to generate procedural code from the model. In this case you will receive some potentially useful information that might help debug the model. When you get an error message from CellML2C such as `Model is unsuitably constrained` you'll also get a list of all the variables that are known about and their status. For example, with this `incomplete test model`:

```
$> CellML2C /path/to/underconstrained-test-model.xml
/* Model is unsuitably constrained (i.e. would need capabilities beyond those of the CCGS to solve).
 * The status of variables at time of error follows...
 * *  Undefined: sin
 * *  Undefined: x
 */
```

When you get an error like this, it is best to start at the bottom of the list of undefined variables and work your way up. For exa

- the model is underconstrained (such as in the above example) - this is usually due to a variable missing an `initial_value` attribute or perhaps you missed out the equation that defines that variable;
- the model is overconstrained - this is usually caused by having an `initial_value` specified for a variable which is defined as an algebraic variable in the procedural code (e.g., the `sin` variable in the above example); or
- some combination of the above.

These can help get things working, but it can be painful digging through your model(s) to find the errors.

CHAPTER 5

CellML2Dot

CellML2Dot is a little utility I wrote to generate `dot` files from CellML models which can then be used by Graphviz and compatible graph/network software to visualise model hierarchies and variable connections. This can be useful when trying to track down problems with your model. It will also do some model validation which might be helpful.

Executing CellML2Dot without any arguments gives the usage information:

```
$> CellML2Dot
ERROR(main): Missing input file URI
CellML2Dot 0.1 - revision 223M - Fri Jul 11 10:32:17 NZST 2014
Copyright (C) 2007 David Nickerson.
This is free software; see the source for copying conditions. There
is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Usage: CellML2Dot [options] <input file>
Perform a simple check of the given CellML file and write out dot files.
Available options:
  --help             Display help and exit.
  --version          Display version information and exit.
  --quiet            Turn off all printing to terminal.
  --debug            Mainly for development, more occurances more output.
  --dot-file-base=<file>  Base name for dot format files.
  --no-component-hierarchy  Do not create the component hierarchy graph.
  --no-variable-connections Do not create the variable connections graph.
  --no-flat-model Do not create the flattened model graph (CellML 1.1 only).
```

The options are relatively straightforward, but have a play to tweak as you need. Common usage is:

```
$> CellML2Dot --dot-file-base=/tmp/model- path/to/sine/sin_approximations_import.xml
```

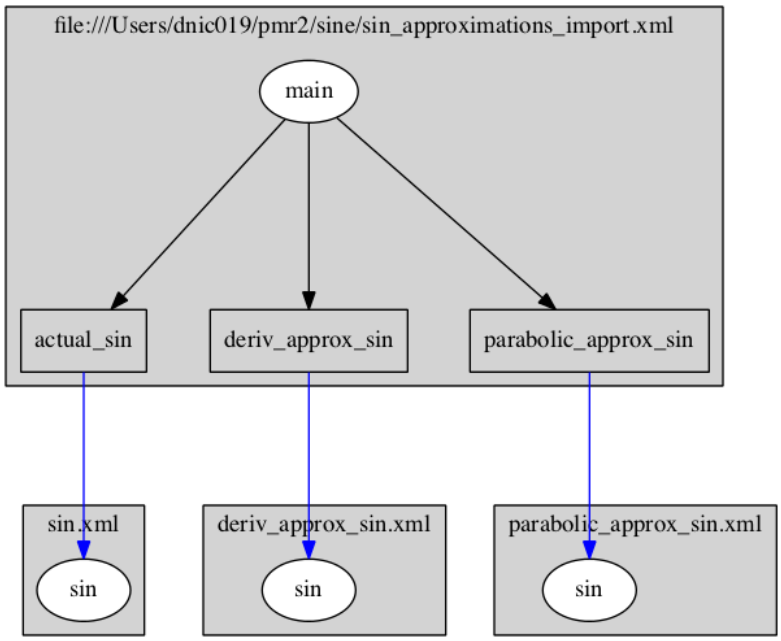
which will produce the three files:

- `/tmp/model-component-hierarchy.dot` - the actual CellML component hierarchy;
- `/tmp/model-flattened-model.dot` - the flattened model graph (components); and
- `/tmp/model-flattened-variable-connections.dot` - the flattened model with all the variable connections (this is the most complicated graph to visualise).

Graphviz has a number of different tools for creating diagrams from the graph descriptions in the DOT format. The standard `dot` tool works quite well for the component hiearchy graphs. For example:

```
$> dot -Tpng -O model-component-hierarchy.dot
```

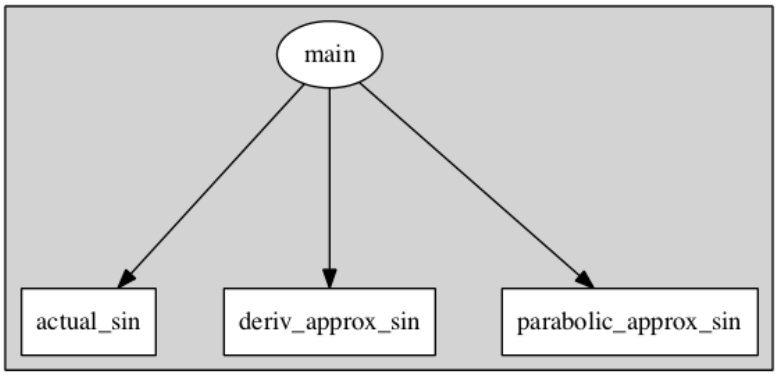
produces the image:



where you can see the top-level model imports components (blue arrows) from the child models via relative URLs. The black arrows indicate the encapsulation hierarchy. Then:

```
$> dot -Tpng -O model-flattened-model.dot
```

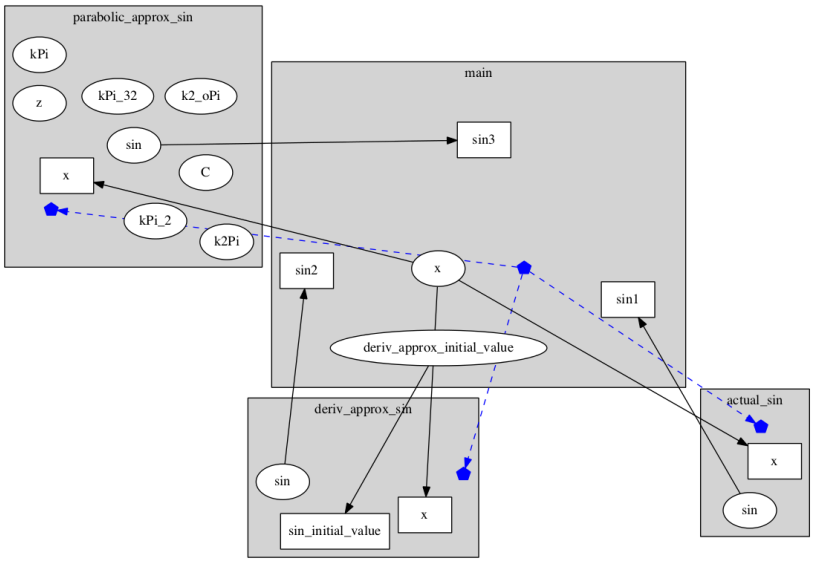
shows the flatened model's encapsulation hierarchy:



The variable connection graph is normally a bit model complicated for detailed models, so `dot` doesn't always give a useful image. In that case it is worth playing with `fdp`:

```
$> fdp -Tpng -O model-flattened-variable-connections.dot
```

produces the image:



where the blue line is showing the component encapsulation hierarchy. For really complicated models, these diagrams can end up being a real mess. To help with that, I have Perl script which lets you extract specific variables of interest and just look at the connections for those variables. It may work for you.

CHAPTER 6

Model flattening

flattenCellmlModel is a little utility which provides two different algorithms for flattening CellML 1.1 model hierarchies into CellML 1.0 models. These are described in detail below.

6.1 Version Converter

The `VersionConverter` algorithm is Jonathan Cooper's original model flattening algorithm updated to work with newer versions of the CellML API. It is primarily designed to flatten model hierarchies into a single CellML model document whiole maintaining the modular structure of the original source model.

This algorithm is used by calling `flattenCellmlModel` with the `model` flattening mode:

```
$> flattenCellmlModel model <source model URL> [output file]
```

if no *output file* is specified the generated model (if flattening is successful) will be written to the terminal - which might obscure any useful model debugging messages that are also printed to the terminal. So you would usually specify an output file:

```
$> flattenCellmlModel model path/to/model.xml flat-model.xml
```

Now you should easily see the *debug* output in the terminal and the flattened model is written to `flat-model.xml`, if the model was successfully flattened. If this hasn't helped debug the problems with your model, but you do get a flattened model, then you might want to play with the flattened model in some of the other tools to see if that helps.

6.2 Model Compaction

Model compaction is a model flattening algorithm that I specifically developed to help debug issues in hierarchical CellML models that none of the existing tools were able to locate or inform me about. The model compaction algorithm also tries to address some of the unsupported features of the version converter algorithm and produce an accurate representation of the mathematical model without worrying about the modularity of the source model. The compacted model will consist of two components. The first component will contain all the variables defined in the model being compacted, with names altered to be unique within the component. The second component will contain all the variables, math, and initial_value's required to fully define the model (if the source model is successfully compacted). Units will all be converted to their canonical representation in the generated model, and in some places the code tries to ensure compatible units are used. See the issues for some of the known issues when dealing with units.

This algorithm is used by calling `flattenCellmlModel` with the `variables` flattening mode (because the focus is on the variables and their relationships rather than the model itself):

```
$> flattenCellmlModel variables <source model URL> [output file]
```

As above, specifying an output file is highly recommended. Since the model compaction algorithm was developed primarily as a model debugging tool, a lot of useful information is presented to the user in the terminal following the (attempted) model compaction. If the source model is successfully compacted you will still get some information that might help debug any issues in your source model, but any errors are highlighted in the `Model Compaction Report` at the end of the terminal output. Also as above, taking the flattened model produced here and putting it through some of the other tools might help locate issues in your model(s).

CHAPTER 7

Introduction

The tools are listed in a fairly arbitrary order. I would generally first use `xmllint` to check that a model is valid XML (after giving it a once-over by eye to check for any glaring errors). The next step depends on what sort of model it is and what errors are being seen (usually in OpenCMISS or CMISS). For hierarchical CellML 1.1 models, the model compaction algorith in *Model flattening* is quite useful (and being actively developed, so if you spot any errors or improvements be sure to let me know). *OpenCOR* is a tool that is rapidly evolving and well worth checking your models in as it will often be one of the first tools readily available with new features (and its trivial to install under Windows, OS X, and Linux!).

For readers based in the ABI, I have made the non-standard commmand line tools described here available on the local server at: `/hpc/dnic019/cellml-debugging/bin/`. When I remember I'll try to keep them updated, but if I forget please don't hesitate to remind me. There are also some test models available at: `/hpc/dnic019/cellml-debugging/cellml-models/`.