Biological Modelling Using CellML and MATLAB

Adam Reeve[1], Alan Garny[2], Andrew K. Miller[1] and Randall D. Britten[1,*]

[1]*Auckland Bioengineering Institute, The University of Auckland, Auckland, New Zealand*

[2]*Department of Physiology, Anatomy and Genetics, Sherrington Building, Parks Road, Oxford OX1 3PT, UK*

Abstract: CellML is an XML-based file format for describing mathematical models, with the aim of simplifying the creation, exchange and reuse of models. There are a number of software tools for modelling biological processes that can read and write CellML files. In this paper, we describe the ability of the OpenCell modelling environment to export CellML files to a number of different programming languages. We give an example of how OpenCell can be used together with MATLAB, a popular language for mathematical modelling, in cardiac cellular electrophysiology. Limitations and areas for future improvement in code export from OpenCell are also addressed.

Keywords: CellML, MATLAB, export, conversion, modelling, electrophysiology.

INTRODUCTION

The aim of the CellML project is to improve the process of constructing mathematical models by making it easier to access, reuse and customise previously published models and to build new models. CellML is an open file format for describing mathematical models, based on the eXtensible Markup Language (XML) [1] and MathML [2]. It is able to describe a broad range of mathematical models; the CellML model repository (http://models.cellml.org/) currently includes electrophysiological, biochemical pathway, mechanical and other model types [3]. There is a modular structure to CellML models that allows reusing and building on previously constructed models [4]. CellML files can be edited with either a text editor, or with a specialised CellML editing tool. One popular CellML tool is OpenCell (http://www.opencellproject.org/), which was previously known as the Physiome CellML Environment (PCEnv). Features from the Cellular Open Resource (COR) tool [5] are being merged into the PCEnv code and PCEnv has been renamed to reflect this combining of tools. Some of these features have been added to the OpenCell code, and this process is currently ongoing. For a thorough list of tools that use CellML, see Garny *et al.* [6].

OpenCell is a Free and Open Source cross-platform software package released under the Mozilla Public License. The OpenCell code is written in JavaScript and utilises Mozilla technologies including XUL and XPCOM. For reading, manipulating and solving CellML models, OpenCell uses the CellML Application Programming Interface (API; http://www.cellml.org/tools/api/).

The CellML API allows rapid construction of CellML based tools without having to replicate previously developed functionality. It consists of a core interface that is required by all applications working with CellML models, and a number of optional services that can be enabled when building the API. These services have a wide range of functions including validating, annotating and solving CellML models. A detailed description of the CellML API is provided by Miller *et al.* [7].

Code Export from CellML

The ability to generate programming code from a CellML file has a number of useful applications. People have different preferred computational environments, and the ability to download code generated from a CellML file allows the use of their preferred environment but also provides all the benefits of using CellML. Models in the CellML repository have a curation status that indicates whether a model accurately reproduces the results from the published paper it is based on, and will also list other problems such as unit inconsistencies if they exist. If a model is obtained from the CellML repository and has been curated and validated, users can be confident that the model will function as intended without having to manually write and test code using the published paper [3].

Although OpenCell is a fully featured modelling environment that allows models to be constructed, solved and plotted, users may find that they require the use of tools available in another modelling or programming environment. In this situation, to be able to generate code from a CellML model is also important.

Previous versions of OpenCell provided the option of exporting code to the C programming language, and recently the ability to export code to MATLAB and Python has been added. Users can now also provide their own XML-based language definition files to add support for new languages or customise the export to an already supported language. A simplified example of a language definition file for MATLAB is given in Listing 1. The functionality for exporting models is implemented using the CeLEDS Exporter service in the CellML API [7]. Because this service

*Address correspondence to this author at the Auckland Bioengineering Institute, The University of Auckland, Auckland, New Zealand; Tel: +64 9 373 7599; Fax: +64 9 367 7157; E-mail: r.britten@auckland.ac.nz

is designed to be able to export code for a number of different types of models to nearly any programming language, there are constraints on the structure of the generated code, and the code generated is often not as clean and simple as if it had been written by hand. An alternative for the user would be to do custom software development using the CellML API to work directly with the relevant CellML models; however, this requires a significant investment of time. Code export offers a more easily accessible alternative. Furthermore, the addition of customisable export also provides an advantage in that users can tailor the code export to their specific requirements. To illustrate the export of a CellML model to MATLAB (Version 7.9, The MathWorks Inc., Natick, Massachusetts), the following simple differential algebraic equation (DAE) will be considered:

$$\frac{dy}{dt} = -y$$

$$d = y \times c$$

where $y(0) = 1$ and $c = 2$.

Apart from the variable of integration (in this example, t), all variables in the exported code are stored in one of the `STATES`, `RATES`, `ALGEBRAIC` or `CONSTANTS` arrays, where each array corresponds to different types of variables. In this example, y is a state variable as it has a rate of change specified, and dy/dt is the rate variable that corresponds to the state variable y. c is a constant as it does not depend on the other variables, and d is an algebraic variable as it depends directly on the values of y and c rather than being defined by a differential equation.

The generated code consists of five main functions. The first function is the primary M-file function. This calls the `initConsts` function to initialize constants and state variables then calls `ode15s`, a MATLAB ordinary differential equation (ODE) solver, to solve for the state variables, which uses the `computeRates` function to determine the rates of change of state variables. Following this, any decoupled algebraic variables are solved for by calling the `computeAlgebraic` function and finally a plot is produced of the solution.

In this example the `initConsts` function initialises the value of the state variable y, STATES(:,1), to 1, and specifies the value of the constant c, CONSTANTS(:,1), as 2:

```
function [STATES, CONSTANTS] = initConsts()
    CONSTANTS = []; STATES = [];
    STATES(:,1) = 1;
    CONSTANTS(:,1) = 2;
end
```

The `computeRates` function is called by the ODE solver and is passed the value of the variable of integration and the values of all state variables and constants. It returns an array containing the calculated rates. Rate variables may depend on algebraic variables, in which case the algebraic variable will be required to be computed within this function. In this example the rate of change of y, `RATES(:,1)`, is calculated:

```
function RATES = computeRates(VOI,
                 STATES, CONSTANTS)
    STATES = STATES'; RATES = [];
    ALGEBRAIC = [0];
    RATES(:,1) =  - STATES(:,1);
    RATES = RATES';
end
```

The ODE solver returns a two-dimensional array of state variables. The columns of this array correspond to different state variables, and each row corresponds to a point in the integration domain. The `computeAlgebraic` function uses the state variables calculated over the entire integration domain to determine any remaining algebraic variables using element-wise vector operations from MATLAB. Algebraic variables previously calculated within `computeRates` must also be recalculated to obtain their values at the correct points in time. In the example code the algebraic variable d, `ALGEBRAIC(:,1)`, is specified:

```
function ALGEBRAIC =
             computeAlgebraic(CONSTANTS,
             STATES, VOI)
    ALGEBRAIC = zeros(length(VOI),1);
    ALGEBRAIC(:,1) =
STATES(:,1).*CONSTANTS(:,1);
end
```

All variable names are defined using a specified string. For the state variables in this example, the string is "STATES(:,%)", where the % is replaced by an array index. The same naming convention is used throughout the code, so in the `initConsts` and `computeRates` functions, state, rate and algebraic variables are accessed by specifying a column of the `STATES`, `RATES` or `ALGEBRAIC` array. However, within these functions, the arrays are simply a row vector, so the operations are only performed on scalar values. This naming scheme is a limitation of the current code export and it would be preferable to access variables by their name rather than through an array index. In order to identify the array indices for different variables the exported code generates arrays of strings in the `createLegends` function, which is also used to generate a legend for the plotted solution. These arrays are named `LEGEND_STATES`, `LEGEND_ALGEBRAIC`, and so on. The value of each item in the `LEGEND_STATES` array will be a string containing the name of the corresponding variable in the `STATES` array.

In the previous example, the algebraic variable d was decoupled from the DAE system, and could be solved for separately once the state variable y had been determined at all points in the time domain. OpenCell also has experimental support for solving index-1 differential algebraic equations where the algebraic variables are coupled with the state variables. An example of such a system is given below, which has the analytic solution $y = e^t$ and $c = d = y/2$:

$$\frac{dy}{dt} = c + d$$

$$\frac{dc}{dt} = \frac{y}{2}$$

$$c \times d = \frac{y^2}{4}$$

where $y(0) = 1$ and $c(0) = 0.5$ and $d(0) = 0.5$.

In MATLAB the recommended method for solving this system is to use a mass matrix, with the equations in the form $M \cdot y' = f(t,y)$, where M is a matrix multiplying the vector of rates, y'. The above example can be solved using the following MATLAB code:

```
function [t,y] = solveDAE()
  M = [1, 0, 0; 0, 1, 0; 0, 0, 0];
  y0 = [1; 0.5; 0.5];
  tspan = [0, 1];
  options = odeset('Mass',M);
  [t,y] = ode15s(@f,tspan,y0,options);

  function r = f(t, Y)
    y = Y(1); c = Y(2); d = Y(3);
    r = [c + d;
         y/2;
         (y^2)/4 – c*d];
  end
end
```

When generating MATLAB code from a CellML model using OpenCell, the structure of the generated code is similar to that for solving an ODE, however within the `computeRates` and `computeAlgebraic` functions, a `rootfind` function is called to solve for the algebraic variable *d*. There may be more than one `rootfind` function required for a model, so they are numbered. The `computeRates` and `rootfind_0` functions for the above example are given below:

```
function RATES = computeRates(VOI,
                STATES, CONSTANTS)
    STATES = STATES'; RATES = [];
    ALGEBRAIC = [0];
    RATES(:,2) = STATES(:,1)./2.0;
    ALGEBRAIC = rootfind_0(VOI,
    CONSTANTS, STATES, ALGEBRAIC);
    RATES(:,1) =
    STATES(:,2)+ALGEBRAIC(:,1);
    RATES = RATES';
end

function ALGEBRAIC = rootfind_0(VOI,
    CONSTANTS, STATES, ALGEBRAIC_IN)
    ALGEBRAIC = ALGEBRAIC_IN;
    global initialGuess;
    if (length(initialGuess) ~= 1),
initialGuess = ones(1,1) * 0.1;, end
        residualfn =
       @(algebraicCandidate)residualSN_0
       (algebraicCandidate,
 ALGEBRAIC, VOI, CONSTANTS, STATES);
        ALGEBRAIC(:,1) =
fsolve(residualfn, initialGuess, options);
        initialGuess = ALGEBRAIC(:,1);
    end
end

function resid =
        residualSN_0(algebraicCandidate,
        ALGEBRAIC, VOI, CONSTANTS,
 STATES)
    ALGEBRAIC(:,1) = algebraicCandidate;
    resid =
(STATES(:,2).*ALGEBRAIC(:,1))-
((STATES(:,1).*STATES(:,1))./4.0);
end
```

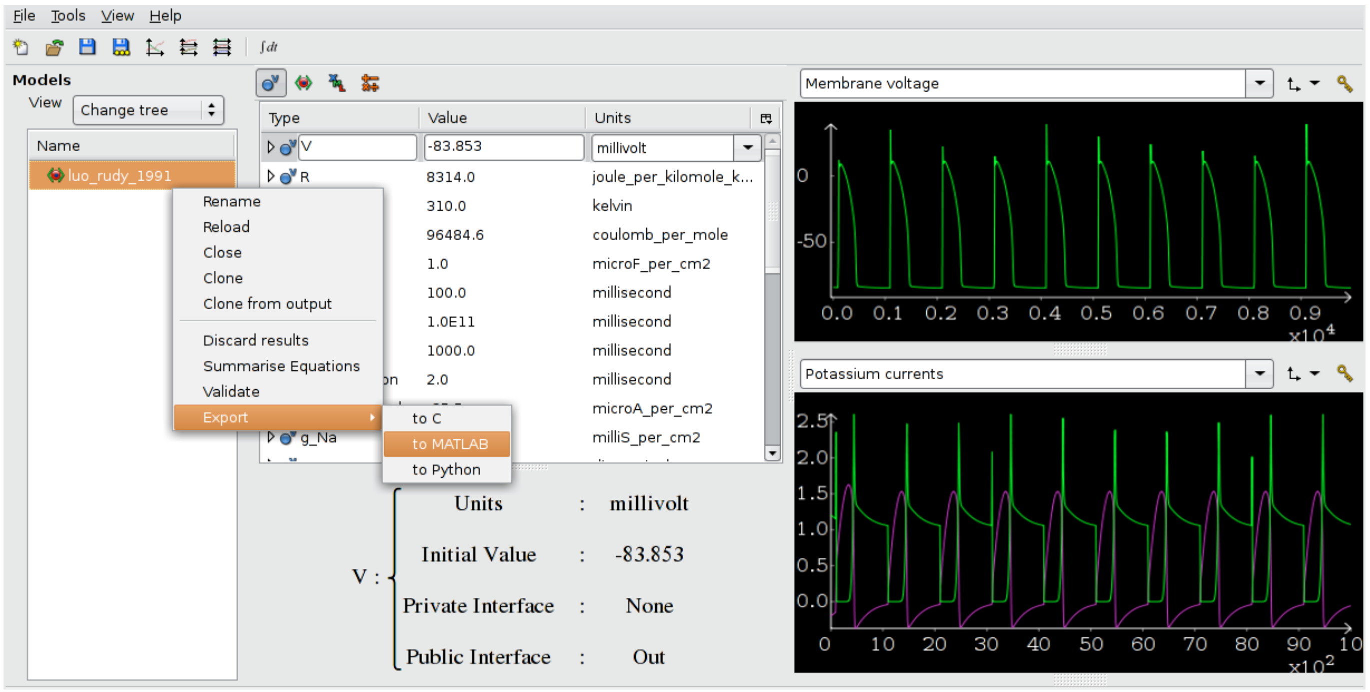
The `rootfind_0` function, in this example, has been simplified to account for only scalar values of the algebraic variables. In the actual exported code the same function is also called from `calculateAlgebraic` using algebraic variables defined at all integration points. `rootfind_0` calls MATLAB's `fsolve` to solve a nonlinear system of equations which is defined in the `residualSN_0` function.

Cardiac Cell Modelling Example

MATLAB is currently a popular tool for cardiac modelling, and even if OpenCell or another CellML based modelling environment is not used, modellers should publish a complete description of their models in a standard format such as CellML to ensure that others have access to their models.

Modellers using MATLAB, or another environment, should consider adding OpenCell to their suite of tools, since it provides an excellent environment for cardiac modelling. The CellML model repository already contains a large number of cardiac models. Fig. (**1**) shows OpenCell editing one such model, Luo and Rudy's 1991 model of the ventricular cardiac action potential [8]. The figure also illustrates the code export menu that is available when right clicking on the model name.

One situation where a user of OpenCell may wish to export their model to MATLAB is to make use of the signal processing tools available in MATLAB. In a hypothetical example, a user may wish to model the effect of measuring the cardiac action potential by convolving the voltage signal from the Luo and Rudy model with the impulse response of a system representing the measurement apparatus used. To accomplish this using OpenCell and MATLAB is straightforward. Right clicking on the model in the left hand side model pane of OpenCell brings up a menu which contains an export option and allows selecting from a range of programming languages. Selecting MATLAB generates MATLAB code which can then be saved to a file.



Type	Value	Units
V	-83.853	millivolt
R	8314.0	joule_per_kilomole_k...
	310.0	kelvin
	96484.6	coulomb_per_mole
	1.0	microF_per_cm2
	100.0	millisecond
	1.0E11	millisecond
	1000.0	millisecond
on	2.0	millisecond
		microA_per_cm2
g_Na		milliS_per_cm2

V :

Units	:	millivolt
Initial Value	:	-83.853
Private Interface	:	None
Public Interface	:	Out

Fig. (1). A screenshot of OpenCell editing Luo and Rudy's 1991 model of the cardiac action potential. A list of variables is shown in the middle panel and plots of variables against time are shown in the right panel. To the left is the menu for exporting a model to C, Python or MATLAB code, which has been opened by right clicking on the model name.

By default, the MATLAB code will return a variable of integration (`VOI`) array with irregular spacing as the solver dynamically adjusts the step size. In order to define variables with regular spacing in time, a line in the exported code, "`tspan = [04000];`" was changed to "`tspan = [0:1:4000];`".

Switching to MATLAB, the exported code can then be run which will return four arrays, `VOI`, `STATES`, `ALGEBRAIC` and `CONSTANTS`. By examining the `LEGEND_STATES` array from the exported MATLAB code it can be seen that the first index of the `STATES` array is the membrane voltage. Ideally, there would be a simpler way to identify variables without having to read the exported code. A fifth-order Butterworth filter was constructed using MATLAB's `butter` function and was applied to the voltage signal using the `filter` function. The MATLAB code used is given below, where `fs` is the sampling frequency in Hz, `fc` is the cut-off frequency in Hz, `b` and `a` are the filter coefficients, `order` is the filter order, `Vm` is the membrane voltage calculated from the model and `Vmf` is the filtered membrane voltage. The frequency and phase response of the filter are shown in Fig. (**2A**), and the membrane voltage signal before and after filtering is shown in Fig. (**2B**).

```
[VOI, STATES, ALGEBRAIC, CONSTANTS] =
luo_rudy();
Vm = STATES(:,1);
% Design filter
fs = 1000;
fc = 10;
order = 5;
[b,a] = butter(order,2*fc/fs);
Vmf = filter(b, a, Vm);
freqz(b,a,[],fs)
figure; plot(VOI, [Vm Vmf])
```

DISCUSSION

The main limitations of code export from OpenCell have already been illustrated. The naming of variables within the code is one issue which needs to be improved, and the way the code is generated limits the structure of the exported code and can prevent the use of preferred solution methods, as seen in the MATLAB example of an index-1 DAE where it would be preferable to use MATLAB's mass matrix method. These are two areas which will be focussed on in future releases of the CellML API and OpenCell. A possible solution to the problem of variable naming is defining variable names as the corresponding array index. Rather than accessing the membrane voltage using "`STATES(:,1)`", we would first define "`membrane_voltage = 1`" and similarly for all other variables, then use "`STATES(:,membrane_voltage)`" to access the value of the membrane voltage.

CONCLUSION

The use of CellML facilitates simpler model development and exchange; however, modellers may prefer to use a different environment for model construction or may wish to take advantage of features available in another environment rather than use a dedicated CellML based application such as OpenCell. In these situations, the code export tool in OpenCell allows generation of code in MATLAB or another programming language, as well as providing the ability for users to customise the code export for their own needs.

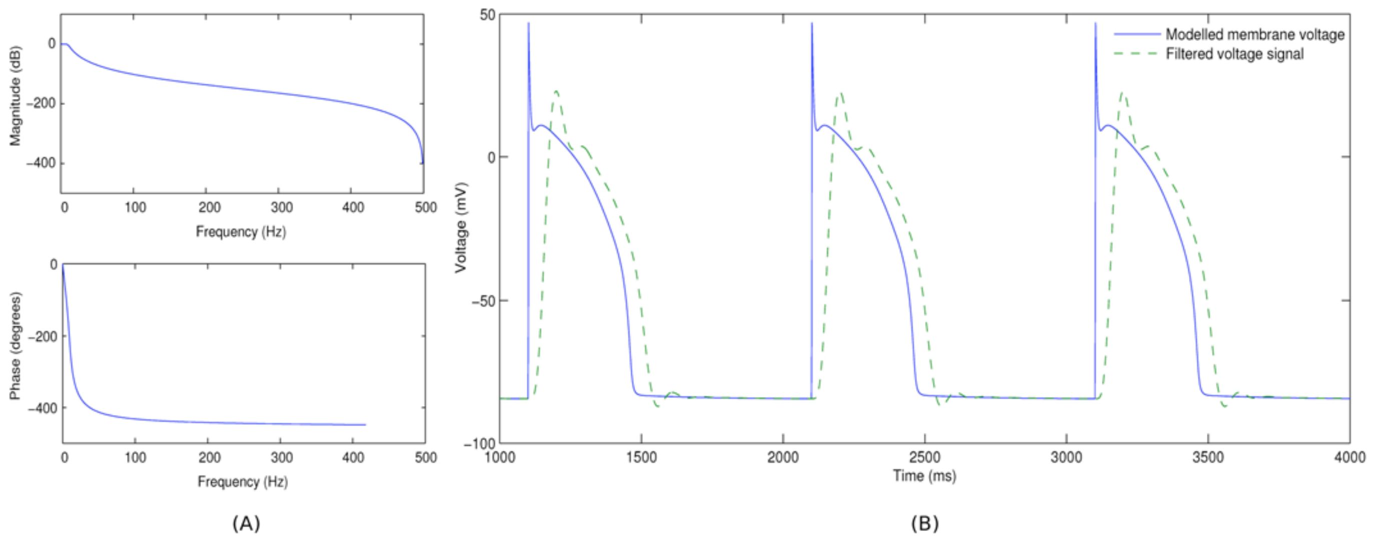


Fig. (2). Filtering of the cardiac action potential. (**A**) shows the magnitude and phase response of the fifth-order Bessel function used to filter the membrane voltage signal, and (**B**) shows the modelled cardiac action potential before and after filtering.

CODE LISTINGS

Listing 1

A condensed version of the language definition file for the MATLAB programming language. Mal and CCGS refer to components of the CellML API that are used for code generation.

```
<?xmlversion="1.0"encoding="ISO-8859-1"?>
<language
xmlns="http://www.cellml.org/CeLEDS/1.0#"
xmlns:mal="http://www.cellml.org/CeLEDS/MaLaES/1.0#"
xmlns:ccgs="http://www.cellml.org/CeLEDS/CCGS/1.0#">
<title>MATLAB</title>
<mal:dictionary>
<mal:mappingkeyname="abs"precedence="H">abs(#expr1)</mal:mapping>
<mal:mappingkeyname="arccos"precedence="H">acos(#expr1)</mal:mapping>
  ...
  ...
</mal:dictionary>
<ccgs:dictionary>
<ccgs:mappingkeyname="constantPattern">CONSTANTS(:,%)</ccgs:mapping>
<ccgs:mappingkeyname="stateVariableNamePattern">STATES(:,%)</ccgs:mapping>
  ...
  ...
</ccgs:dictionary>
<dictionary>
<mappingkeyname="preStateCount">% There are a total of </mapping>
<mappingkeyname="postStateCount"> entries in each of the rate and state variable arrays.</mapping>
  ...
  ...
</dictionary>
<extrafunctions>
<functionsearchname="arbitrary_log"><![CDATA[% Compute a logarithm to any base
function x = arbitrary_log(a, base)
    x = log(a) ./ log(base);
end
]]></function>
   ...
   ...
</extrafunctions>
</language>
```

ACKNOWLEDGEMENTS

The authors would like to thank the Maurice Wilkins Centre for Molecular Biodiscovery and the VPH NoE project(#223920; supported by the European Commission, DG Information Society, through the Seventh Framework Programme of Information and Communication Technologies) for financial contributions to the CellML project.

AG is funded through the preDiCT and euHeart projects (#224381 and #224495, respectively) which are also supported by the European Commission, DG Information Society, through the Seventh Framework Programme of Information and Communication Technologies.

Part of AKM's work was completed in the course of his PhD research, funded by a NZ Tertiary Education Commission (TEC) Top Achievers Doctoral Scholarship.

CONFLICT OF INTEREST

None to declare.

REFERENCES

[1] Bray T, Paoli J, Sperberg-McQueen CM, Maler E, Yergeau F. Extensible Markup Language (XML) 1.0 (5th ed). W3C Recommendation 2008. Available from http://www.w3.org/TR/REC-xml/

[2] Carlisle D, Ion P, Miner R, Poppelier N. Mathematical markup language (MathML) version 2.0. W3C Recommendation 2003. Available from http://www.w3.org/TR/MathML/

[3] Lloyd CM, Lawson JR, Hunter PJ, Nielsen PF. The CellML model repository. Bioinformatics 2008; 24(18): 2122-3.

[4] Cuellar AA, Lloyd CM, Nielsen PF, Bullivant DP, Nickerson DP, Hunter PJ. An overview of CellML 1.1, a biological model description language. Simulation 2003; 79(12): 740-7.

[5] Garny A, Noble D, Hunter PJ, Kohl P. Cellular Open Resource (COR): current status and future directions. Philos Trans A Math Phys Eng Sci 2009; 367(1895): 1885-905.

[6] Garny A, Nickerson DP, Cooper J, *et al*. CellML and associated tools and techniques. Philos Trans A Math Phys Eng Sci 2008; 366(1878): 3017-43.

[7] Miller A, Marsh J, Reeve A, *et al*. An overview of the CellML API and its implementation. (submitted)

[8] Luo CH, Rudy Y. A model of the ventricular cardiac action potential. Depolarization, repolarization, and their interaction. Circ Res 1991; 68: 1501-26.

Received: November 30, 2009 Revised: February 28, 2010 Accepted: March 09, 2010