

This tutorial shows you how to install and run the OpenCOR software, and to author and edit CellML models¹. We start by creating a simple model from scratch, saving it as a CellML file and running model simulations. We next try opening existing CellML models, both from a local directory and from the Physiome Model Repository. The various features of CellML² and OpenCOR are then explained in the context of increasingly complex biological models. A simple first order ODE model and a nonlinear third order model are introduced. Ion channel gating models are used to introduce the way that CellML handles units, components and connections. More sophisticated potassium and sodium ion channel models are then described as models that can subsequently be imported into the Hodgkin-Huxley 1952 squid axon neural model using the CellML model import facility.

Contents

	page
1. Installing and launching OpenCOR	1
2. Creating and running a simple CellML model: editing and simulation	2
3. Opening an existing CellML file	5
4. A simple first order ODE	6
5. A Lorentz attractor	7
6. A model of ion channel gating and current: Introducing CellML units	8
7. A model of the potassium channel: Introducing CellML components and connections	10
8. A model of the sodium channel: Introducing CellML encapsulation and interfaces	13
9. A model of the nerve action potential: Introducing CellML imports	17

1. Installing and launching OpenCOR

Download OpenCOR from www.opencor.ws. Versions are available for Windows, Mac and Linux. Create a shortcut to the executable (found in the bin directory) on your desktop and click on this to launch OpenCOR. A window will appear that looks like Figure 1(a).

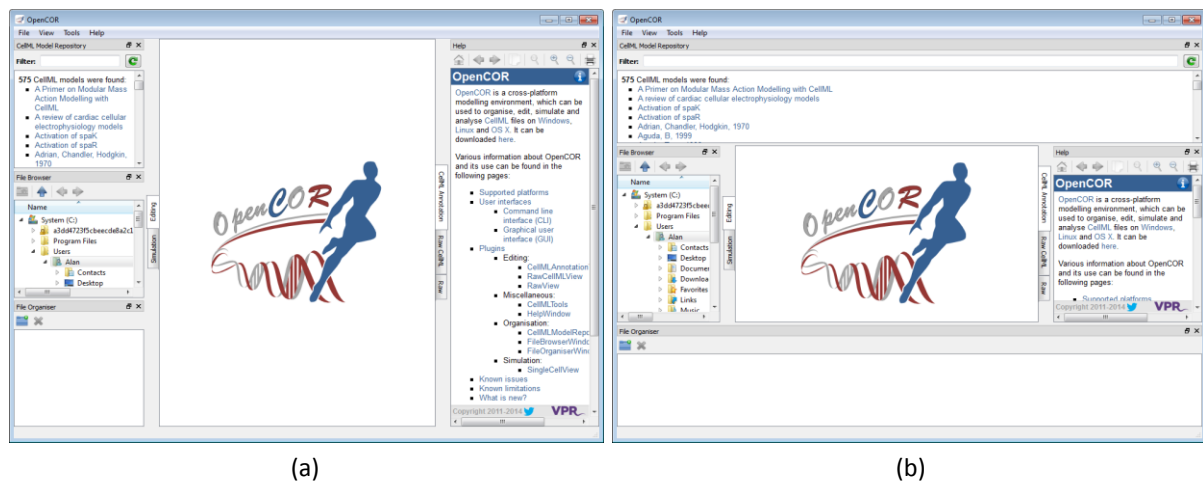


Figure 1. (a) Default positioning of dockable windows. (b) An alternative configuration achieved by dragging and dropping the dockable windows.

The central area is used to interact with files. By default, no files are open, hence the OpenCOR logo is shown instead. To the sides, there are dockable windows, which provide additional features. Those windows can be dragged and dropped to the top or bottom of the central area as shown in Figure 1(b) or they can be individually undocked or closed. All closed panels can be re-displayed by enabling them in the *View* menu, or by using the *Tools* menu *Reset All* option. Clicking on 'CTRL' & 'spacebar' on the Windows version, removes (for less clutter) or restores these two side panels.

¹ For an overview and the background of CellML see www.cellml.org.

² For details on the specifications of CellML1.0 see www.cellml.org/specifications/cellml_1.0.

2. Creating and running a simple CellML model: editing and simulation

In this example we create a simple CellML model from scratch and run it. The model is the Van der Pol oscillator³ defined by the second order equation⁴

$$\frac{d^2x}{dt^2} - \mu(1 - x^2)\frac{dx}{dt} + x = 0$$

with initial conditions $x = -2$; $\frac{dx}{dt} = 0$. The parameter μ controls the magnitude of the damping term. To create a CellML model we convert this to two first order equations by defining the velocity $\frac{dx}{dt}$ as a new variable y :

$$\begin{aligned}\frac{dx}{dt} &= y \\ \frac{dy}{dt} &= \mu(1 - x^2)y - x\end{aligned}$$

The initial conditions are now $x = -2$; $y = 0$.

Under the *File* menu and *New*, click on *CellML 1.0 File* then type in the following lines of code:

```
def model van_der_pol_model as
  def comp main as
    var t: dimensionless {init: 0};
    var x: dimensionless {init: -2};
    var y: dimensionless {init: 0};
    var mu: dimensionless {init: 1};

    ode(x,t)=y;
    ode(y,t)=mu*(1{dimensionless}-sqr(x))*y-x;
  enddef;
enddef;
```

Things to note⁵ are: (i) the closing semicolon at the end of each line (apart from the first two *def* statements that are opening a CellML construct); (ii) the need to indicate dimensions for each variable and constant (all dimensionless in this example – but more on dimensions later); (iii) the use of *ode(x,t)* to indicate a first order⁶ ODE in x and t , and (iv) the use of the convenient squaring function *sqr(x)* for x^2 .

A partial list of mathematical functions available for OpenCOR is:

x^2	<i>sqr(x)</i>	\sqrt{x}	<i>sqrt(x)</i>	$\ln x$	<i>ln(x)</i>	$\log_{10} x$	<i>log(x)</i>	e^x	<i>exp(x)</i>	x^a	<i>pow(x,a)</i>
$\sin x$	<i>sin(x)</i>	$\cos x$	<i>cos(x)</i>	$\tan x$	<i>tan(x)</i>	$\csc x$	<i>csc(x)</i>	$\sec x$	<i>sec(x)</i>	$\cot x$	<i>cot(x)</i>
$\sin^{-1} x$	<i>asin(x)</i>	$\cos^{-1} x$	<i>acos(x)</i>	$\tan^{-1} x$	<i>atan(x)</i>	$\csc^{-1} x$	<i>acsc(x)</i>	$\sec^{-1} x$	<i>asec(x)</i>	$\cot^{-1} x$	<i>acot(x)</i>
$\sinh x$	<i>sinh(x)</i>	$\cosh x$	<i>cosh(x)</i>	$\tanh x$	<i>tanh(x)</i>	$\operatorname{csch} x$	<i>csch(x)</i>	$\operatorname{sech} x$	<i>sech(x)</i>	$\operatorname{coth} x$	<i>coth(x)</i>
$\sinh^{-1} x$	<i>asinh(x)</i>	$\cosh^{-1} x$	<i>acosh(x)</i>	$\tanh^{-1} x$	<i>atanh(x)</i>	$\operatorname{csch}^{-1} x$	<i>acsch(x)</i>	$\operatorname{sech}^{-1} x$	<i>asech(x)</i>	$\operatorname{coth}^{-1} x$	<i>acoth(x)</i>

Table 1. The list of mathematical functions available for coding in OpenCOR.

Positioning the cursor over either of the ODEs renders the maths in standard form above the code as shown in Figure 2(a).

³ en.wikipedia.org/wiki/Van_der_Pol_oscillator

⁴ Note that gray boxes are used to indicate equations that are implemented directly in OpenCOR.

⁵ For more on the CellML Text view see opencor.ws/user/plugins/editing/CellMLTextView.html.

⁶ Note that a more elaborated version of this is '*ode(x, t, 1{dimensionless})*' and a 2nd order ODE can be specified as '*ode(x, t, 2{dimensionless})*'. 1st order is assumed as the default.

Note that CellML is a declarative language (unlike say C, Fortran or Matlab, which are procedural languages) and the order of statements therefore does not affect the solution. For example, the order of the ODEs could equally well be

```
ode(y,t)=mu*(1{dimensionless}-sqr(x))*y-x;
ode(x,t)=y;
```

The significance of this will become apparent later when we import several CellML models to create a composite model. This means that models can be treated as reusable modules.

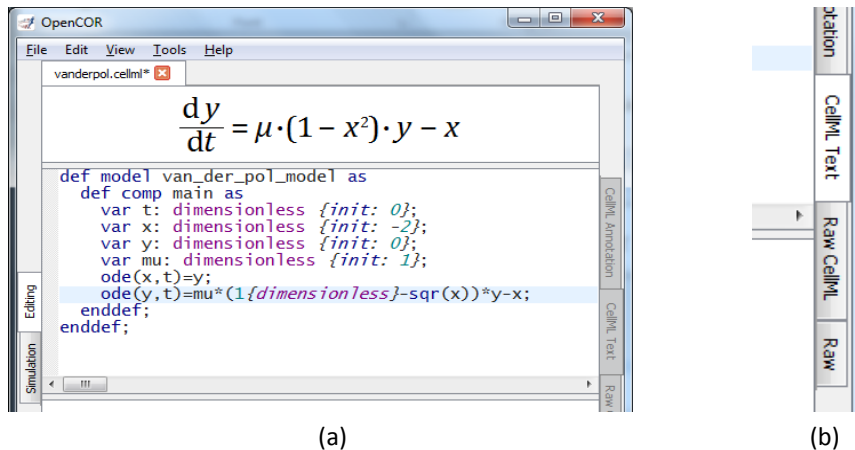


Figure 2. (a) Positioning the cursor over an equation (shown by the highlighted line) renders the maths. (b) Once the model has been saved, the RH tabs provide different views of the CellML code.

Now save the code using *Save* under the *File* menu (or CTRL-S) and choosing *.cellml* as the file format⁷. With the CellML model saved various views, accessed via the tabs on the right hand edge of the window, become available. One is the *CellML Text* view (the view used to enter the code above); another is the *Raw CellML* view that displays the way the model is stored (note that positioning the cursor over part of the code shows the maths in this view also); and another is the *Raw* view.

With the equations and initial conditions defined, we are ready to run the model. To do this, click on the *Simulation* tab on the left hand edge of the window. You will see three main areas - at the left hand side of the window are the *Simulation*, *Solvers*, *Graphs* and *Parameters* panels, which are explained below. At the right hand side is the graphical output window, and running along the bottom of the window is a status area, where status messages are displayed.

Simulation panel

This area is used to set up the simulation settings.

- Starting point - the value of the variable of integration (often time) at which the simulation will begin. Leave this at 0.
- Ending point - the point at which the simulation will end. Set to 100.
- Point interval - the interval between data points on the variable of integration. Set to 0.1.

Just above the *Simulation panel* are controls for running the simulation. These are:

Run (⏮), Pause (⏸), Reset parameters (↺), Clear simulation data (🗑), Interval delay (⏱), Add(+) / Subtract(-) graphical output windows and Output solution to a CSV file (📄).

For this model, we suggest that you create three graphical output windows using the + button.

Solvers panel

This area is used to configure the solver that will run the simulation.

- Name - this is used to set the solver algorithm. OpenCOR will only allow you to choose solvers appropriate to the type of problem you are simulating. For example, CVODE for ODE

⁷ Note that '.cellml' is not strictly required but is best practice.

(ordinary differential equation) problems, IDA for DAE (differential algebraic equation) problems, KINSOL for NLA (non-linear algebraic) problems⁸.

- Other parameters for the chosen solver - eg. maximum step size, number of steps, and tolerance settings for CVODE and IDA. For more information on the solver parameters, please refer to the documentation for the particular solver.

Note: these can all be left at their default values for our simple demo problem⁹.

Graphs panel

This shows what parameters are being plotted once these have been defined in the *Parameters panel*. These can be selected/deselected by clicking in the box next to a parameter¹⁰.

Parameters panel

This panel lists all the model parameters, and allows you to select one or more to plot against the variable of integration or another parameter in the graphical output windows. OpenCOR supports graphing of any parameter against any other. All variables from the model are listed here, arranged by the components in which they appear, and in alphabetical order. Parameters are displayed with their variable name, their value, and their units. Right clicking on a parameter provides the options for displaying that parameter in the currently selected graphical output window. With the cursor highlighting the top graphical output window (a blue line appears next to it), select x then the variable of integration t in order to plot $x(t)$. Now move the cursor to the second graphical output window and select y then t to plot $y(t)$. Finally select the bottom graphical output window and select y then x to plot $y(x)$.

Now Click on the *Run* control. You will see a progress bar running along the bottom of the status window. Status messages about the successful simulation will be displayed. Use the *interval delay* wheel to slow down the plotting.

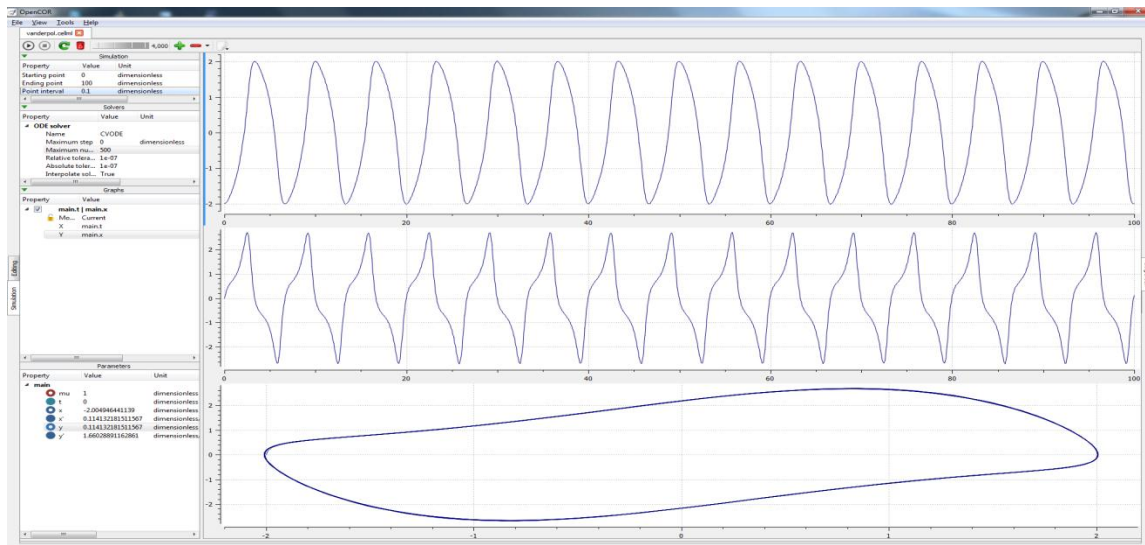


Figure 3. Graphical output from OpenCOR. The top window is $x(t)$, the middle is $y(t)$ and the bottom is $y(x)$.


To obtain numerical values for all variables (i.e. $x(t)$ and $y(t)$), click on the *CSV file* button (). You will be asked to enter a filename and type (use .csv). Opening this file (e.g. with Microsoft Excel) provides access to the numerical values. Other output types (e.g. BiosignalML) will be available shortly.

⁸ Other solvers include forward Euler, Heun and Runge-Kutta solvers (RK2 and RK4).

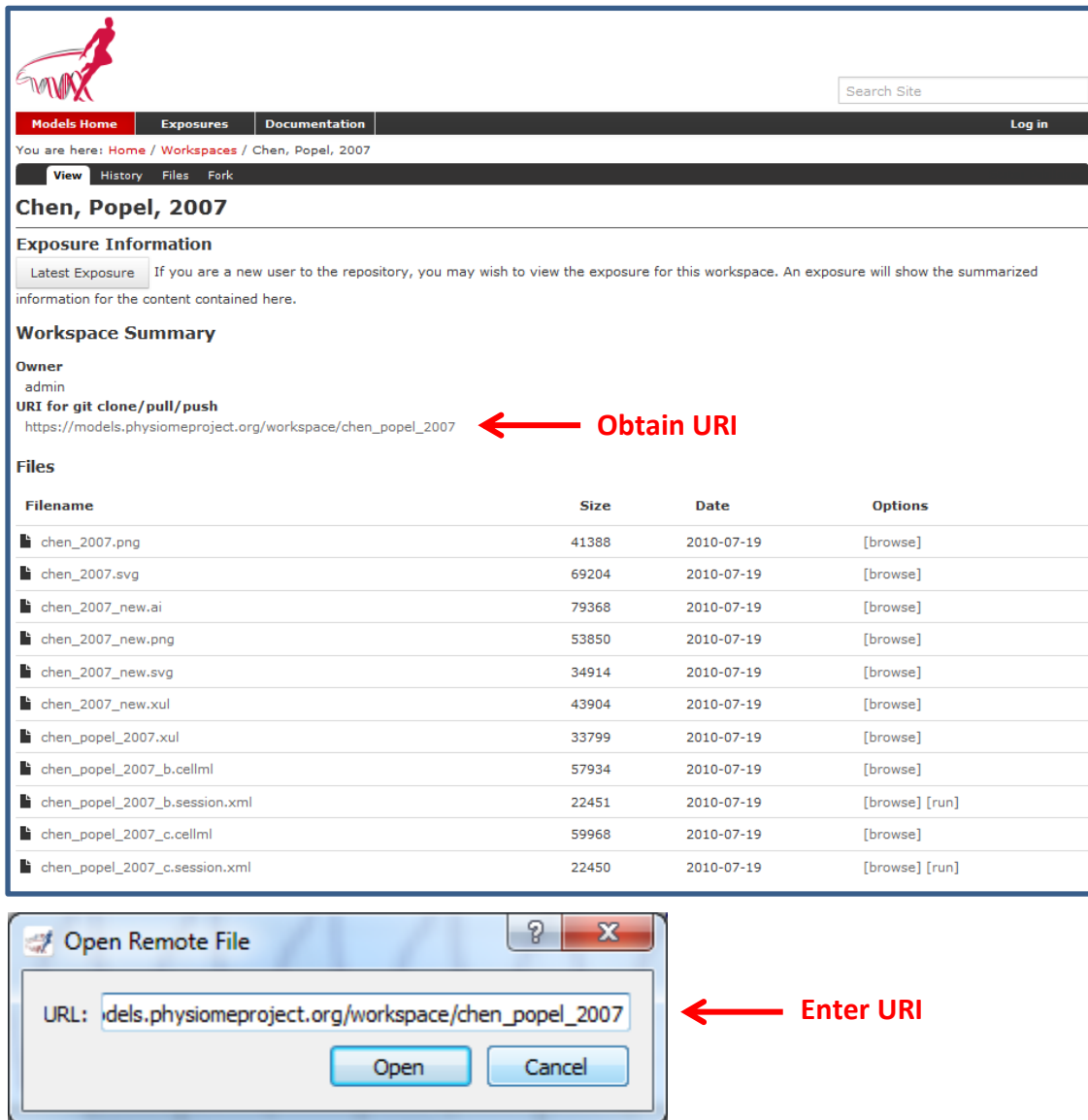
⁹ Note that a model that requires a stimulus protocol should have the maximum step value of the CVODE solver set to the length of the stimulus.

¹⁰ Two simulations can be compared by 'locking' a graph – see later.

3. Opening an existing CellML file

Go to the *File* menu and select *Open...*. Browse to the folder containing models and select one. Note that this brings up a new tabbed window and you can have any number of CellML models open at the same time in order to quickly move between them. A model can be removed from this list by clicking on  next to the CellML model name.

You can also access models from the left hand panel in Figure 1(a). If this panel is not currently visible, use 'CTRL-spacebar' to make it reappear. Models can then be accessed from any one of the three subdivisions of this panel – *File Browser*, *CellML Model Repository* or *File Organiser*. For a file under *File Browser* or *File Organiser*, either double-click it or 'drag&drop' it over the central workspace to open that model. Clicking on a model in the *CellML Model Repository*, opens a new browser window with that model. To open this model in OpenCOR, copy the URI (see Figure 4) into the text box that appears under *Open Remote...* in the *File* menu.



The top part of the image shows a web browser window displaying the CellML Model Repository page for a workspace named 'Chen, Popel, 2007'. The page includes a search bar, navigation tabs (Models Home, Exposures, Documentation), and a breadcrumb trail. The 'Exposure Information' section contains a 'Latest Exposure' button and a description. The 'Workspace Summary' section lists the owner as 'admin' and provides a URI for git clone/pull/push: `https://models.physionemproject.org/workspace/chen_popel_2007`. A red arrow points to this URI with the label 'Obtain URI'. Below this is a table of files in the workspace.

Filename	Size	Date	Options
chen_2007.png	41388	2010-07-19	[browse]
chen_2007.svg	69204	2010-07-19	[browse]
chen_2007_new.ai	79368	2010-07-19	[browse]
chen_2007_new.png	53850	2010-07-19	[browse]
chen_2007_new.svg	34914	2010-07-19	[browse]
chen_2007_new.xul	43904	2010-07-19	[browse]
chen_popel_2007.xul	33799	2010-07-19	[browse]
chen_popel_2007_b.cellml	57934	2010-07-19	[browse]
chen_popel_2007_b.session.xml	22451	2010-07-19	[browse] [run]
chen_popel_2007_c.cellml	59968	2010-07-19	[browse]
chen_popel_2007_c.session.xml	22450	2010-07-19	[browse] [run]

The bottom part of the image shows a dialog box titled 'Open Remote File'. It has a text input field labeled 'URL:' containing the same URI as the one in the browser window: `https://models.physionemproject.org/workspace/chen_popel_2007`. A red arrow points to this text field with the label 'Enter URI'. At the bottom of the dialog are 'Open' and 'Cancel' buttons.

Figure 4. Browser window (top) opened from within OpenCOR and showing a model in the CellML model repository. The red arrow points to the URI for that model. Copying this and inserting it into the text box that appears under *Open Remote...* in the *File* menu (bottom), opens the model in OpenCOR.

4. A simple first order ODE

The simplest example of a first order ODE is

$$\frac{dy}{dt} = -ay + b$$

with the solution

$$y(t) = \frac{b}{a} + \left(y(0) - \frac{b}{a}\right) \cdot e^{-at},$$

where $y(0)$, the value of $y(t)$ at $t = 0$, is the *initial condition* and as $t \rightarrow \infty$, $y(t) \rightarrow \frac{b}{a}$ (see Figure 5).

Note that $t = \tau = \frac{1}{a}$ is called the *time constant* of the exponential decay, and that

$$y(\tau) = \frac{b}{a} + \left(y(0) - \frac{b}{a}\right) \cdot e^{-1}.$$

At $t = \tau$, $y(t)$ has therefore fallen to $\frac{1}{e}$ (or about 37%) of the difference between the initial ($y(0)$) and final steady state ($y(\infty) = \frac{b}{a}$) values.

Choosing parameters $a = 1$; $b = 2$ and $y(0) = 5$, the CellML text for this model is

```
def model first_order_model as
  def comp main as
    var t: dimensionless {init: 0};
    var y: dimensionless {init: 5};
    var a: dimensionless {init: 1};
    var b: dimensionless {init: 2};
    ode(y,t)=-a*y+b;
  enddef;
enddef;
```

The solution by OpenCOR (using *Ending point*=10, *Point interval*=0.1) is shown in Figure 6(a) for these parameters and in Figure 6(b) for parameters $a = 1$; $b = 5$ and $y(0) = 2$.

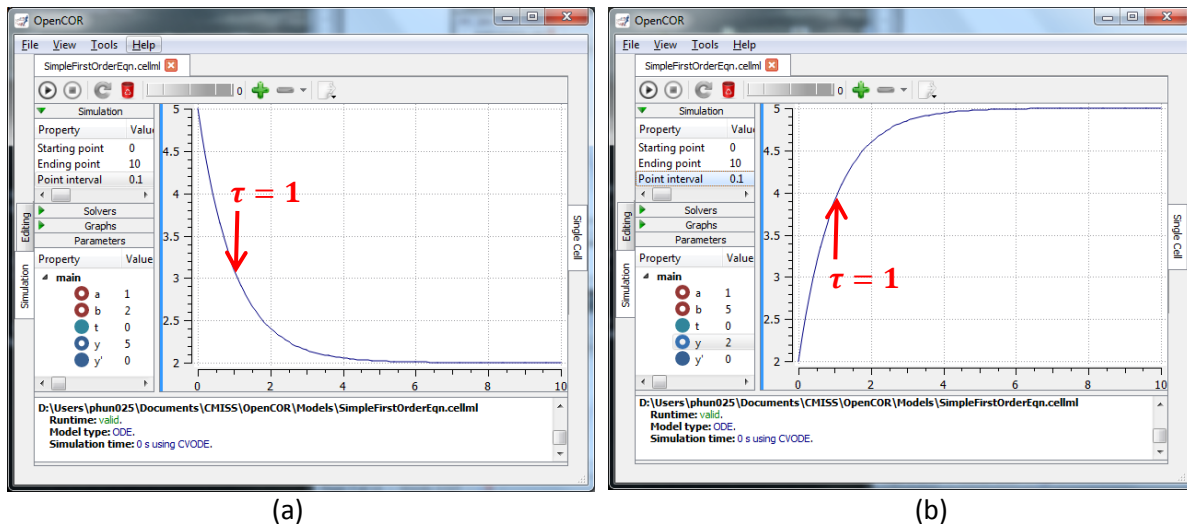


Figure 6. OpenCOR output $y(t)$ for the simple ODE model with parameters (a) $a = 1$; $b = 2$ and $y(0) = 5$, and (b) $a = 1$; $b = 5$ and $y(0) = 2$.

These two solutions have the same exponential time constant ($\tau = \frac{1}{a} = 1$) but different initial and final (steady state) values.

5. A Lorentz attractor

An example of a third order ODE system (i.e. three 1st order equations) is the *Lorentz attractor*¹¹.

This has three equations:

$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = x(\rho - z) - y$$

$$\frac{dz}{dt} = xy - \beta z$$

where σ , ρ and β are parameters.

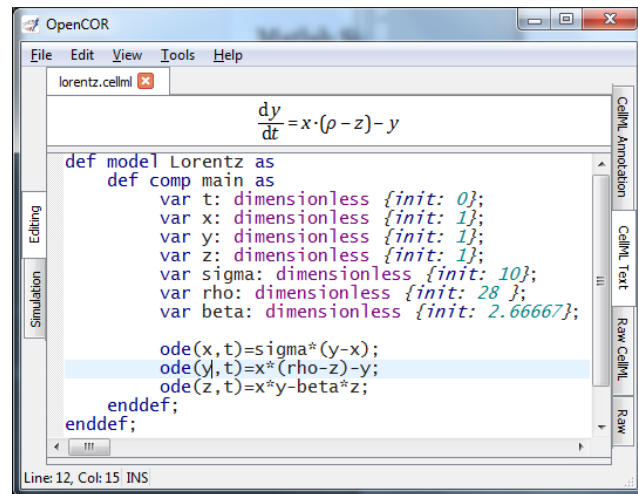
The CellML text code entered for these equations is shown in Figure 7 with parameters

$$\sigma = 10, \rho = 28, \beta = 8/3 = 2.6667$$

and initial conditions

$$x(0) = y(0) = z(0) = 1.$$

Solutions for $x(t)$, $y(x)$ and $z(x)$, corresponding to the time integration parameters shown on the LHS, are shown in Figure 8. Note that this is an example of a ‘chaotic’ system of equations because small changes in the initial conditions lead to quite different solution paths.



```

def model Lorentz as
  def comp main as
    var t: dimensionless {init: 0};
    var x: dimensionless {init: 1};
    var y: dimensionless {init: 1};
    var z: dimensionless {init: 1};
    var sigma: dimensionless {init: 10};
    var rho: dimensionless {init: 28};
    var beta: dimensionless {init: 2.66667};

    ode(x,t)=sigma*(y-x);
    ode(y,t)=x*(rho-z)-y;
    ode(z,t)=x*y-beta*z;
  enddef;
enddef;

```

Figure 7. CellML text code for the Lorentz equations

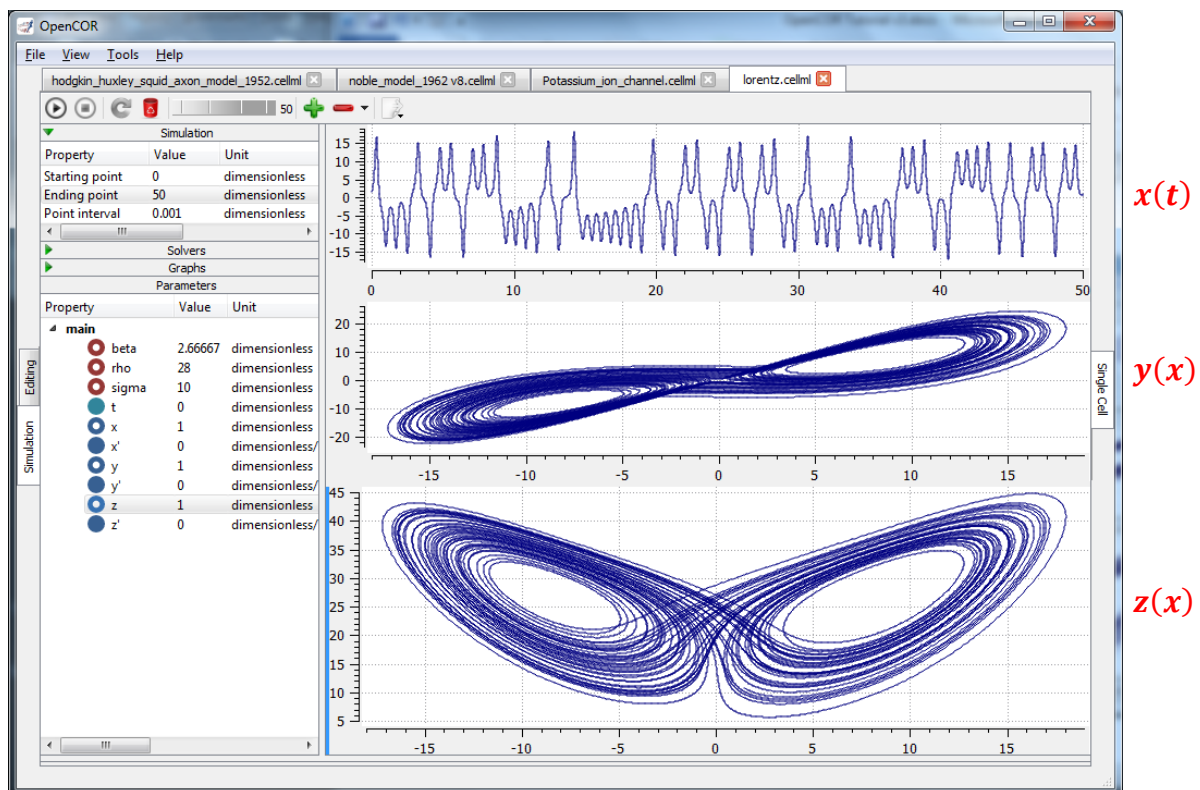


Figure 8. Solutions of the Lorentz equations

This example illustrates the value of OpenCOR’s ability to plot solution variables as they are computed. Use the ‘simulation delay’ wheel to slow down the plotting by a factor of about 50,000 – in order to follow the solution trajectory as it spirals in ever widening trajectories around the left hand ‘attractor’ before reaching a bifurcation point that sends it off to the right hand attractor.

¹¹ http://en.wikipedia.org/wiki/Lorenz_system; <https://www.math.auckland.ac.nz/~hinke/crochet/>

6. A model of ion channel gating and current: Introducing CellML units

A good example of a model based on a first order equation is the one used by Hodgkin and Huxley¹² to describe the gating behaviour of an ion channel (see also next three sections). To describe the time dependent transition between the closed and open states of the channel, Hodgkin and Huxley introduced the idea of channel gates that control the passage of ions through a membrane ion channel. If the fraction of gates that are open is y , the fraction of gates that are closed is $1-y$, and a first order ODE can be used to describe the transition between the two states (see Figure 9):

$$\frac{dy}{dt} = \alpha_y(1-y) - \beta_y \cdot y$$

where α_y is the opening rate and β_y is the closing rate.

The solution to this ODE is

$$y = \frac{\alpha_y}{\alpha_y + \beta_y} + A e^{-(\alpha_y + \beta_y)t}$$

The constant A can be interpreted as $A = y(0) - \frac{\alpha_y}{\alpha_y + \beta_y}$ as in the previous example and, with $y(0) = 0$ (i.e. all gates initially shut), the solution looks like Figure 10(a).

The experimental data obtained by Hodgkin and Huxley for the squid axon, however, indicated that the initial current flow began more slowly (Figure 10(b)) and they modelled this by assuming that the ion channel had γ gates in series so that conduction would only occur when all gates were at least partially open. Since y is the probability of a gate being open, y^γ is the probability of all γ gates being open (since they are assumed to be independent) and the current through the channel is

$$i_y = \bar{i}_y y^\gamma = y^\gamma \bar{g}_y (V - E_y)$$

where $\bar{i}_y = \bar{g}_y (V - E_y)$, the steady state current through the open gate, is governed by Ohm's law. i.e. the current is equal to the channel conductance \bar{g}_y times the driving potential, which in this case is the difference between the membrane potential V (referenced to the external potential) and the Nernst equilibrium potential for that ion, E_y .

We can represent this in OpenCOR with a simple extension of the previous model, but in developing this model we will also demonstrate the way in which CellML deals with units.

There are seven base physical quantities defined by the *Système International d'Unités* (SI)¹³. These are (with their SI units): length (meter or m), time (second or s), amount of substance (mole), temperature (°K), mass (kilogram or kg), current (amp or A) and luminous intensity (candela). All other units are derived from these seven. Additional derived units that CellML defines intrinsically are: Hz (s^{-1}); Newton, N ($kg \cdot m \cdot s^{-2}$); Joule, J ($N \cdot m$); Pascal, Pa ($N \cdot m^{-2}$); Watt, W ($J \cdot s^{-1}$); Volt, V ($W \cdot A^{-1}$); Siemen, S ($A \cdot V^{-1}$); Ohm, Ω ($V \cdot A^{-1}$); Coulomb, C ($s \cdot A$); Farad, F ($C \cdot V^{-1}$); Weber, Wb ($V \cdot s$); Henry, H ($Wb \cdot A^{-1}$). Multiples and fractions of these are defined as follows:

Multiples	Prefix		deca	hecto	kilo	mega	giga	tera	peta	exa	zetta	yotta
	Symbol		da	h	k	M	G	T	P	E	Z	Y
	Factor	10^0	10^1	10^2	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}	10^{21}	10^{24}
Fractions	Prefix		deci	centi	milli	micro	nano	pico	femto	atto	zepto	yocto
	Symbol		d	c	m	μ	n	p	f	a	z	y
	Factor	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-6}	10^{-9}	10^{-12}	10^{-15}	10^{-18}	10^{-21}	10^{-24}



Figure 9. Ion channel gating kinetics. y is the fraction of gates in the open state. α_y and β_y are the rate constants for opening and closing, respectively.

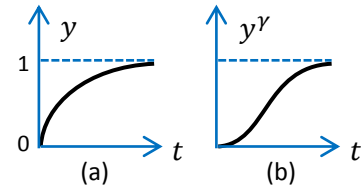


Figure 10. Transient behaviour for one gate (left) and γ gates in series (right).

¹² Hodgkin, A.L. and Huxley, A.F. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117, 500-544, 1952. [PubMed ID: 12991237](https://pubmed.ncbi.nlm.nih.gov/12991237/)

¹³ http://en.wikipedia.org/wiki/International_System_of_Units

These units with multiples and fractions are illustrated in the following CellML text code:

```
def model first_order_model as
  def unit millisecond as
    unit second {pref: milli};
  enddef;

  def unit per_millisecond as
    unit second {pref: milli, expo: -1};
  enddef;

  def unit millivolt as
    unit volt {pref: milli};
  enddef;

  def unit microA_per_cm2 as
    unit ampere {pref: micro};
    unit metre {pref: centi, expo: -2};
  enddef;

  def unit milliS_per_cm2 as
    unit siemens {pref: milli};
    unit metre {pref: centi, expo: -2};
  enddef;

  def comp ion_channel as
    var V: millivolt {init: 0};
    var t: millisecond {init: 0};
    var y: dimensionless {init: 0};
    var E_y: millivolt {init: -85};
    var i_y: microA_per_cm2;
    var g_y: milliS_per_cm2 {init: 36};
    var gamma: dimensionless {init: 4};
    var alpha_y: per_millisecond {init: 1};
    var beta_y: per_millisecond {init: 2};

    ode(y, t) = alpha_y*(1-dimensionless)-y)-beta_y*y;
    i_y = g_y*pow(y, gamma)*(V-E_y);
  enddef;
enddef;
```

← Define units for time as milliseconds

← Define per_millisecond units

← Define units for voltage as millivolts

← Define units for current as microAmps per cm^2

← Define units for conductance as milliSiemens per cm^2

} Define units and initial conditions for variables

← Define ODE for gating variable y

← Define channel current

The solution of these equations for the parameters indicated above is illustrated in Figure 11.

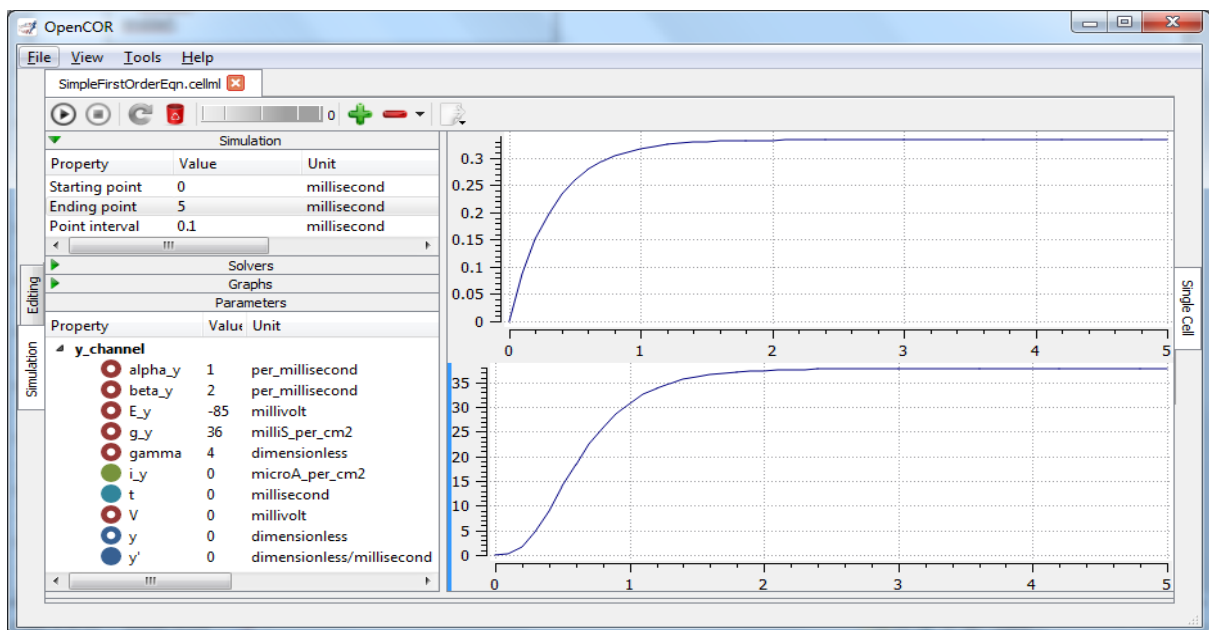


Figure 11. The behaviour of an ion channel with $\gamma = 4$ gates transitioning from the closed to the open state at a membrane voltage $V = 0$. The opening and closing rate constants are $\alpha_y = 1 \text{ ms}^{-1}$ and $\beta_y = 2 \text{ ms}^{-1}$. The ion channel has an open conductance of $\bar{g}_y = 36 \text{ mS} \cdot \text{cm}^{-2}$ and an equilibrium potential of $E_y = -85 \text{ mV}$. The upper transient is the response $y(t)$ for each gate and the lower trace is the current through the channel.

7. A model of the potassium channel: Introducing CellML components and connections

We now deal specifically with the application of the previous model to the Hodgkin and Huxley (HH) potassium channel. Following the convention introduced by Hodgkin and Huxley, the gating variable for the potassium channel is n and the number of gates in series is $\gamma = 4$, therefore

$$i_K = \bar{i}_K n^4 = n^4 \bar{g}_K (V - E_K)$$

where $\bar{g}_K = 36 \text{ mS.cm}^{-2}$, and with intra- and extra-cellular concentrations $[K^+]_i = 90\text{mM}$ and $[K^+]_o = 3\text{mM}$, respectively, the Nernst potential for the potassium channel is

$$E_K = \frac{RT}{F} \ln \frac{[K^+]_o}{[K^+]_i} = 25 \ln \frac{3}{90} = -85\text{mV}.$$

Note that this is called the *equilibrium potential* since it is the potential across the cell membrane when the channel is open but no current is flowing because the electrostatic driving force from the difference between ion charges (inside minus outside) is exactly matched by the osmotic driving force from the ion concentration difference. $n^4 \bar{g}_K$ is the channel conductance.

The gating kinetics are described (as before) by

$$\frac{dn}{dt} = \alpha_n(1 - n) - \beta_n \cdot n \quad \text{with time constant } \tau = \frac{1}{\alpha_n + \beta_n}.$$

The main difference from the gating model in our previous example is that Hodgkin and Huxley found it necessary to make the rate constants functions of the membrane potential V (see Figure 12) as follows¹⁴:

$$\alpha_n = \frac{-0.01(V+65)}{e^{\frac{-(V+65)}{10}} - 1}; \quad \beta_n = 0.125e^{\frac{-(V+75)}{80}}.$$

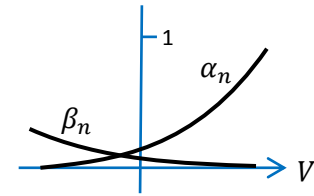


Figure 12. Voltage dependence of rate constants α_n and β_n .

Note that under steady state conditions when $t \rightarrow \infty$ and $\frac{dn}{dt} \rightarrow 0$, $n|_{t=\infty} = n_\infty = \frac{\alpha_n}{\alpha_n + \beta_n}$.

These equations are captured with OpenCOR CellML text (together with the previous unit definitions) on the next page. But first we need to explain some further CellML concepts.

We introduced CellML *units* above. We now need to introduce two more CellML constructs: *components* and *connections* (mappings between components). For completeness we also show two other constructs in Figure 13 that will be used later: *imports* and *groups*.¹⁵

Defining components serves two purposes: it preserves a modular structure for CellML models, and allows these component modules to be imported into other models, as we will illustrate later. For the potassium channel model we define components representing (i) the environment, (ii) the potassium channel conductivity, and (iii) the dynamics of the n -gate.

Since certain variables (t , V and n) are shared between components, we need to also define the component maps as indicated in the CellML text view on the next page. Also note the use of $\{pub:in\}$ and $\{pub:out\}$ to indicate which variables are either supplied as inputs to a component or

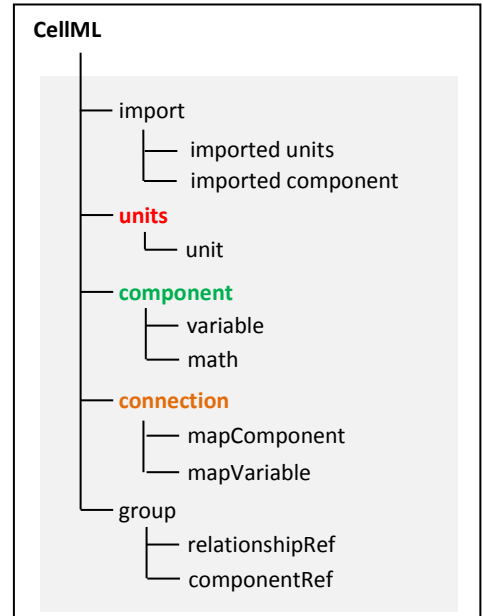


Figure 13. Key entities in a CellML model.

¹⁴ The original expression in the HH paper used $\alpha_n = \frac{0.01(v+10)}{e^{\frac{(v+10)}{10}} - 1}$ and $\beta_n = 0.125e^{\frac{v}{80}}$, where v is defined relative to the resting potential (-75mV) with +ve v corresponding to +ve inward current and $v = -(V + 75)$.

¹⁵ See also Lloyd *et al*, 2004; Lloyd *et al*, 2008.

produced as outputs from a component¹⁶. Any variables not labelled as *in* or *out* are local variables or parameters defined and used only within that component. Public and private interfaces are discussed in more detail in the next section.

The CellML text code for the potassium ion channel model is as follows¹⁷:

def model potassium_ion_channel as

```
def unit millisecond as
  unit second {pref: milli};
enddef;

def unit per_millisecond as
  unit second {pref: milli, expo: -1};
enddef;

def unit millivolt as
  unit volt {pref: milli};
enddef;

def unit microA_per_cm2 as
  unit ampere {pref: micro};
  unit metre {pref: centi, expo: -2};
enddef;

def unit milliS_per_cm2 as
  unit siemens {pref: milli};
  unit metre {pref: centi, expo: -2};
enddef;
```

Define **units**

```
def comp environment as
  var V: millivolt {init: -85, pub: out};
  var t: millisecond {pub: out};
enddef;
```

Define **component 'environment'**

```
def comp potassium_channel as
  var V: millivolt {pub: in};
  var n: dimensionless {pub: in};
  var i_K: microA_per_cm2 {pub: out};
  var g_K: milliS_per_cm2 {init: 36};
  var E_K: millivolt {init: 36};

  i_K = g_K*pow(n, 4{dimensionless})*(V-E_K);
enddef;
```

Define **component 'potassium channel'**

```
def comp potassium_channel_n_gate as
  var V: millivolt {pub: in};
  var t: millisecond {pub: in};
  var n: dimensionless {init: 0.325, pub: out};
  var alpha_n: per_millisecond;
  var beta_n: per_millisecond;

  alpha_n = 0.01{per_millisecond}*exp((V+10{millivolt})/10{millivolt})
    /(exp((V+10{millivolt})/10{millivolt})-1{dimensionless});
  beta_n = 0.125{per_millisecond}*exp(V/80{millivolt});
  ode(n, t) = alpha_n*(1{dimensionless}-n)-beta_n*n;
enddef;
```

Define **component 'potassium channel n gate'**

```
def map between potassium_channel and environment for
  vars V and V;
enddef;
def map between potassium_channel and potassium_channel_n_gate for
  vars n and n;
enddef;
def map between potassium_channel_n_gate and environment for
  vars V and V;
  vars t and t;
enddef;
```

Define **mappings** between **components** for variables that are shared between these components

enddef;

¹⁶ Note that a later version of CellML will remove the terms *in* and *out* since it is now thought that the direction of information flow should not be constrained.

¹⁷ From here on we use a coloured background to identify code blocks that relate to a particular CellML construct: **red** for units, **green** for components, **orange** for mappings and, later, **purple** for encapsulation groups and **blue** for imports.

We now use OpenCOR to solve the equations for the potassium channel under a voltage step condition in which the membrane voltage is clamped initially at -85mV and then stepped up to higher voltage for 10ms before being returned to -85mV. At a membrane voltage of -85mV, the steady state value of the n gate is $n_{\infty} = \frac{\alpha_n}{\alpha_n + \beta_n} = 0.191$. Three cases are shown in Figure 14: steps to -20mV (top set), 0mV (middle set) and +20mV (bottom set). The voltage traces are shown at the top of each set, the n-gate response (by opening from its resting value and then plateauing at the peak step voltage before closing again as the voltage is stepped back to rest) are shown in the middle of each set, and the channel conductance $n^4 \bar{g}_K$ is shown as the bottom trace in each set. Note that the gate closing behaviour is slower than the opening behaviour in all cases and that the opening behaviour is faster as the voltage is stepped to higher values since $\tau = \frac{1}{\alpha_n + \beta_n}$ reduces with V (see Figure 12).

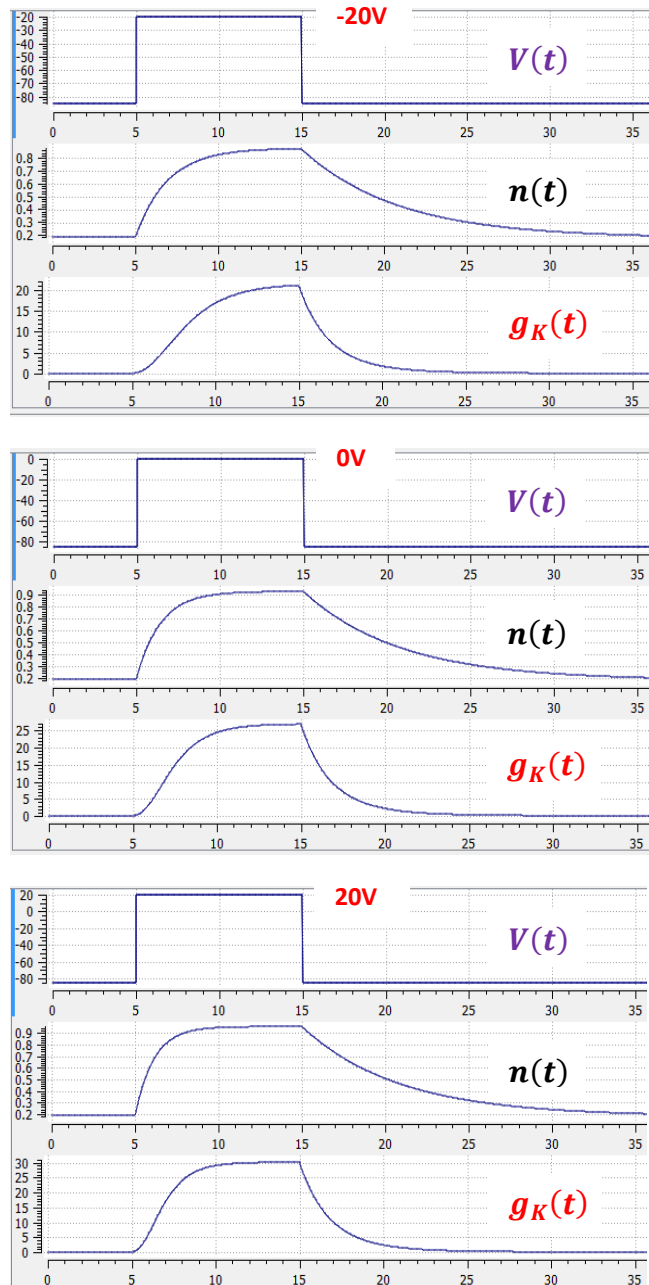


Figure 14. Transient responses to voltage steps. See text for explanation and analysis.

8. A model of the sodium channel: Introducing CellML encapsulation and interfaces

The HH sodium channel has two types of gate, an m gate (of which there are 3) that is initially closed ($m = 0$) before activating and inactivating back to the closed state, and an h gate that is initially open ($h = 1$) before activating and inactivating back to the open state. The short period when both types of gate are open allows a brief window current to pass through the channel. Therefore,

$$i_{Na} = \bar{i}_{Na} m^3 h = m^3 h \cdot \bar{g}_{Na} (V - E_{Na})$$

where $\bar{g}_{Na} = 120 \text{ mS.cm}^{-2}$, and with $[\text{Na}^+]_i = 30\text{mM}$ and $[\text{Na}^+]_o = 140\text{mM}$, the Nernst potential for the sodium channel is

$$E_{Na} = \frac{RT}{F} \ln \frac{[\text{Na}^+]_o}{[\text{Na}^+]_i} = 25 \ln \frac{140}{30} = 35\text{mV}.$$

The gating kinetics are described by

$$\frac{dm}{dt} = \alpha_m(1 - m) - \beta_m \cdot m; \quad \frac{dh}{dt} = \alpha_h(1 - h) - \beta_h \cdot h$$

where the voltage dependence of these four rate constants is determined experimentally to be

$$\alpha_m = \frac{-0.1(V+50)}{e^{\frac{-(V+50)}{10}} - 1}; \quad \beta_m = 4e^{\frac{-(V+75)}{18}}; \quad \alpha_h = 0.07e^{\frac{-(V+75)}{20}}; \quad \beta_h = \frac{1}{e^{\frac{-(V+45)}{10}} + 1}^{18}$$

Before we construct a CellML model of the sodium channel, we first introduce some further CellML concepts: first the use of *encapsulation groups* and *public* and *private interfaces* to control the visibility of information in modular CellML components. To understand encapsulation, it is useful to use the terms ‘parent’, ‘child’ and ‘sibling’.

We define the CellML components **sodium_channel_m_gate** and **sodium_channel_h_gate** below. Each of these components has its own equations (voltage-dependent gates and first order gate kinetics) but they are both parts of one protein – the sodium channel – and it is useful to group them into one **sodium_channel** component as shown on the right:

```
def group as encapsulation for
  comp sodium_channel incl
    comp sodium_channel_m_gate;
    comp sodium_channel_h_gate;
  endcomp;
enddef;
```

We can then talk about the sodium channel as the parent of two children: the m gate and the h gate, which are therefore siblings. A *private interface* allows a parent to talk to its children and a *public interface* allows siblings to talk among themselves and to their parents (see Figure 15).

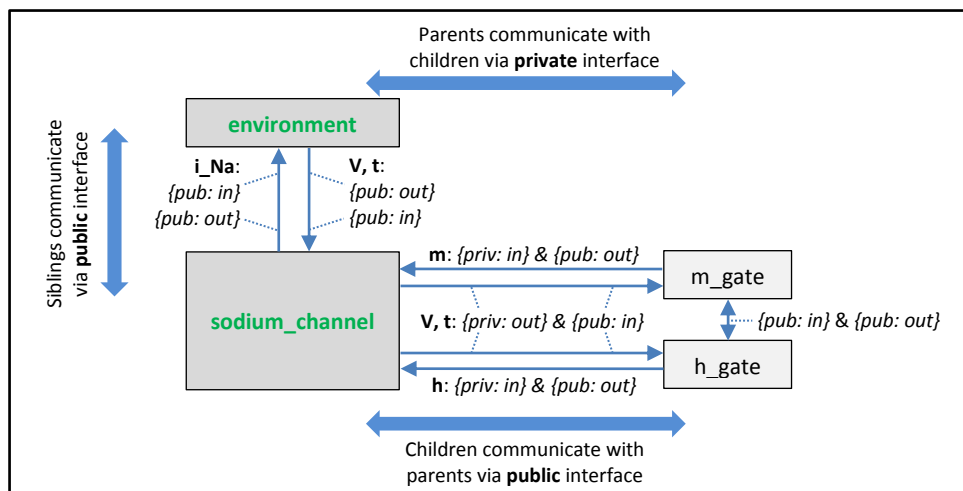


Figure 15. The children talk to each other as siblings, and to their parents, via public interfaces. But the outside world can only talk to children through their parents via a private interface. Note that the siblings m_gate and h_gate can talk via a public interface but only if a mapping is established between them (not needed here).

¹⁸ The HH paper used $\alpha_m = \frac{0.1(v+25)}{e^{\frac{(v+25)}{10}} - 1}$; $\beta_m = 4e^{\frac{v}{18}}$; $\alpha_h = 0.07e^{\frac{v}{20}}$; $\beta_h = \frac{1}{e^{\frac{(v+30)}{10}} + 1}$ (see footnote on p10).

The OpenCOR CellML text for the HH sodium ion channel is given below. Note that as well as *encapsulation group* one other feature has been added: the event control *select case* which indicates that the voltage is specified to jump from -85mV to 35mV at t=5ms then back again at t=15ms. This is only used here in order to test the Na channel model; when the sodium_channel component is later imported into a neuron model, the environment component is not imported.

def model sodium_ion_channel as

```
def unit millisecond as
  unit second {pref: milli};
enddef;
def unit per_millisecond as
  unit second {pref: milli, expo: -1};
enddef;
def unit millivolt as
  unit volt {pref: milli};
enddef;
def unit per_millivolt as
  unit millivolt {expo: -1};
enddef;
def unit per_millivolt_millisecond as
  unit per_millivolt;
  unit per_millisecond;
enddef;
def unit microA_per_cm2 as
  unit ampere {pref: micro};
  unit metre {pref: centi, expo: -2};
enddef;
def unit milliS_per_cm2 as
  unit siemens {pref: milli};
  unit metre {pref: centi, expo: -2};
enddef;
```

Define **units**

```
def comp environment as
  var V: millivolt {init: -85, pub: out};
  var t: millisecond {pub: out};
  V = sel
    case (t > 5 {millisecond}) and (t < 15 {millisecond}):
      35.0 {millivolt};
    otherwise:
      -85.0 {millivolt};
  endsel;
enddef;
```

Define
voltage step

Define **component**
'environment'

```
def group as encapsulation for
  comp sodium_channel incl
    comp sodium_channel_m_gate;
    comp sodium_channel_h_gate;
  endcomp;
enddef;
```

Define **encapsulation**
of **m_gate** and **h_gate**

```
def comp sodium_channel as
  var V: millivolt {pub: in, priv: out};
  var t: millisecond {pub: in, priv: out};
  var m: dimensionless {priv: in};
  var h: dimensionless {priv: in};
  var g_Na: milliS_per_cm2 {init: 120};
  var E_Na: millivolt {init: 35};
  var i_Na: microA_per_cm2 {pub: out};
  i_Na = g_Na*pow(m, 3{dimensionless})*h*(V-E_Na);
enddef;
```

Define **component**
'sodium channel'

```
def comp sodium_channel_m_gate as
  var V: millivolt {pub: in};
  var t: millisecond {pub: in};
  var alpha_m: per_millisecond;
  var beta_m: per_millisecond;
  var m: dimensionless {init: 0.05, pub: out};
  alpha_m = 0.1{per_millivolt_millisecond}*(V+25{millivolt})
    /(exp((V+25{millivolt})/10{millivolt})-1{dimensionless});
  beta_m = 4{per_millisecond}*exp(V/18{millivolt});
  ode(m, t) = alpha_m*(1{dimensionless}-m)-beta_m*m;
enddef;
```

Define **component**
'sodium channel m gate'


```

def comp sodium_channel_h_gate as
  var V: millivolt {pub: in};
  var t: millisec {pub: in};
  var alpha_h: per_millisec;
  var beta_h: per_millisec;
  var h: dimensionless {init: 0.6, pub: out};

  alpha_h = 0.07{per_millisec}*exp(V/20{millivolt});
  beta_h = 1{per_millisec}/(exp((V+30{millivolt})/10{millivolt})+1{dimensionless});

  ode(h, t) = alpha_h*(1{dimensionless}-h)-beta_h*h;
enddef;

```

Define **component**
'sodium channel h gate'

```

def map between sodium_channel and environment for
  vars V and V;
  vars t and t;
enddef;
def map between sodium_channel and sodium_channel_m_gate for
  vars V and V;
  vars t and t;
  vars m and m;
enddef;
def map between sodium_channel and sodium_channel_h_gate for
  vars V and V;
  vars t and t;
  vars h and h;
enddef;

```

Define **mappings** between
components for variables that are
shared between these components

enddef;

The results of the OpenCOR computation are shown in Figure 16 with plots of $V(t)$, $m(t)$, $h(t)$, $g_{Na}(t)$ and $i_{Na}(t)$. There are several things to note:

- (i) The kinetics of the m-gate are much faster than the h-gate.
- (ii) The sodium channel conductance rises (*activates*) and then falls (*inactivates*) under a positive voltage step from rest since the three m-gates turn on but the h-gate turns off and the conductance is a product of these. Compare this with the potassium channel conductance shown in Figure 14 which is only reduced back to zero by stepping the voltage back to its resting value – i.e. *deactivating* it.
- (iii) The only time current i_{Na} flows through the sodium channel is during the brief period when the m-gate is rapidly opening and the much slower h-gate is beginning to close. A small current flows during the reverse voltage step but this is at a time when the h-gate is now firmly off so the magnitude is very small.
- (iv) The large sodium current i_{Na} is an inward current and hence negative.

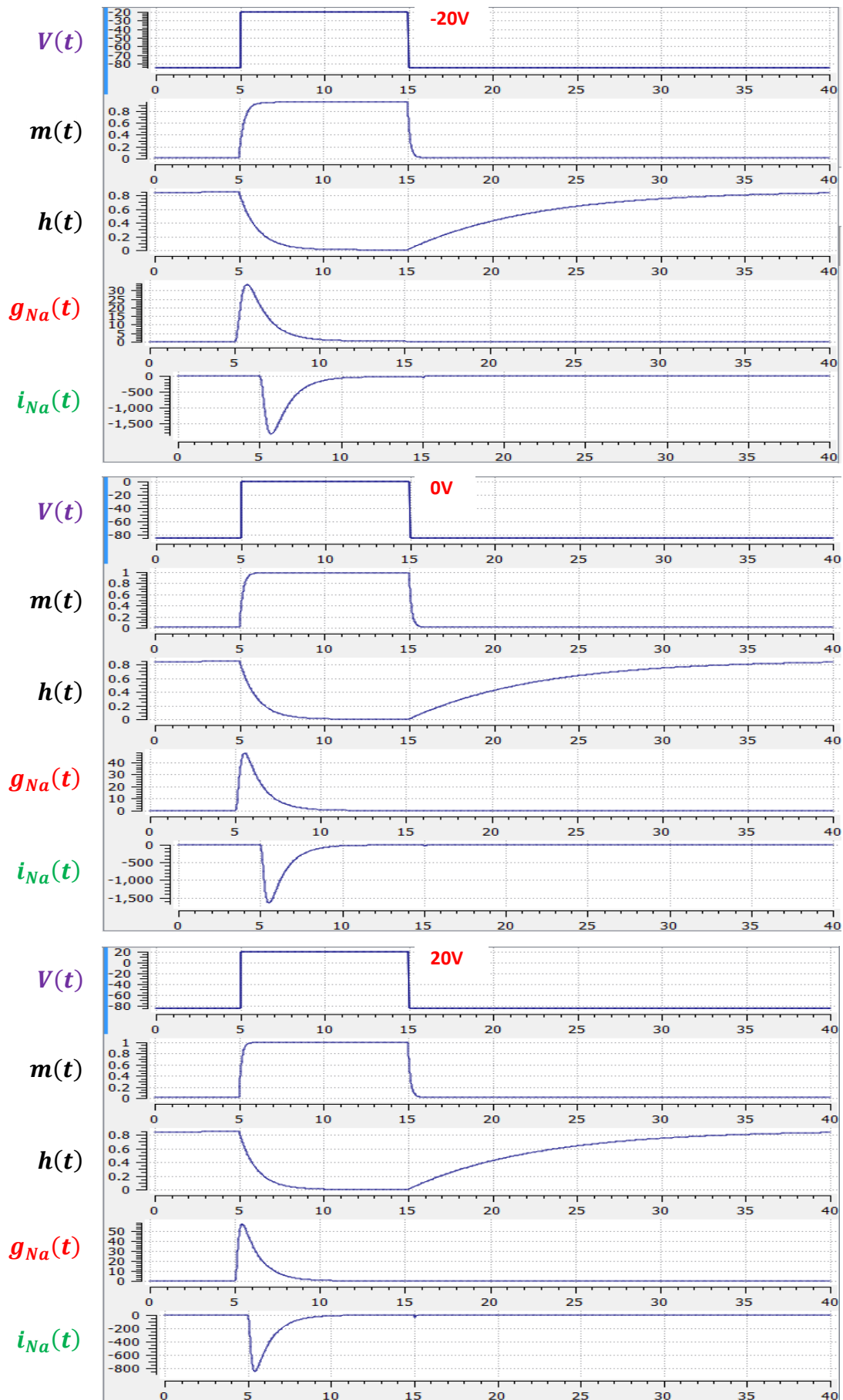


Figure 16. The kinetics of the sodium channel gates. See text for discussion.

9. A model of the nerve action potential: Introducing CellML imports

Here we describe the first (and most famous) model of nerve fibre electrophysiology based on the membrane ion channels that we have discussed in the last two sections. This is the work by Alan Hodgkin and Andrew Huxley in 1952¹⁹ that won them (together with John Eccles) the 1963 Noble prize in Physiology or Medicine for "*their discoveries concerning the ionic mechanisms involved in excitation and inhibition in the peripheral and central portions of the nerve cell membrane*".

Cable equation

The *cable equation* was developed in 1890²⁰ to predict the degradation of an electrical signal passing along the transatlantic cable. It is derived as follows:

If the voltage is raised at the left hand end of the cable (shown by the deep red in Figure 17), a current i_a (A) will flow that depends on the voltage gradient, given by $\frac{\partial V}{\partial x}$ (V.m⁻¹) and the resistance r_a (Ω.m⁻¹), Ohm's law gives $-\frac{\partial V}{\partial x} = r_a i_a$. But if the

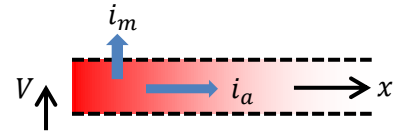


Figure 17. Current flow in a leaky cable.

cable leaks current i_m (A.m⁻¹) per unit length of cable, conservation of current gives $\frac{\partial i_a}{\partial x} = i_m$ and therefore, substituting for i_a , $\frac{\partial}{\partial x} \left(-\frac{1}{r_a} \frac{\partial V}{\partial x} \right) = i_m$. There are two sources of membrane current i_m , one associated with the capacitance C_m ($\approx 1\mu\text{F}/\text{cm}^2$) of the membrane, $C_m \frac{\partial V}{\partial t}$, and one associated with holes or channels in the membrane, i_{leak} . Inserting these into the RHS gives

$$\frac{\partial}{\partial x} \left(-\frac{1}{r_a} \frac{\partial V}{\partial x} \right) = i_m = C_m \frac{\partial V}{\partial t} + i_{leak}$$

Rearranging gives the *cable equation* (for constant r_a):

$$C_m \frac{\partial V}{\partial t} = -\frac{1}{r_a} \frac{\partial^2 V}{\partial x^2} - i_{leak}$$

where all terms represent *current density* (current per membrane area) and have units of $\mu\text{A}/\text{cm}^2$.

Action potentials

The cable equation can be used to model the propagation of an action potential along a neuron or any other excitable cell. The 'leak' current is associated primarily with the inward movement of sodium ions through the membrane 'sodium channel', giving the **inward** membrane current i_{Na} , and the outward movement of potassium ions through a membrane 'potassium channel', giving the **outward** current i_K (see Figure 18). A further small leak current $i_L = g_L(V - E_L)$ associated with chloride and other ions is also included.

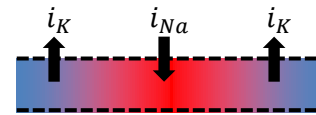


Figure 18. Current flow in a neuron.

When the membrane potential V rises due to axial current flow, the Na channels open and the K channels close, such that the membrane potential moves towards the Nernst potential for sodium. The subsequent decline of the Na channel conductance and the increasing K channel conductance as the voltage drops, rapidly repolarises the membrane to its resting potential of -85mV (see Figure 19).

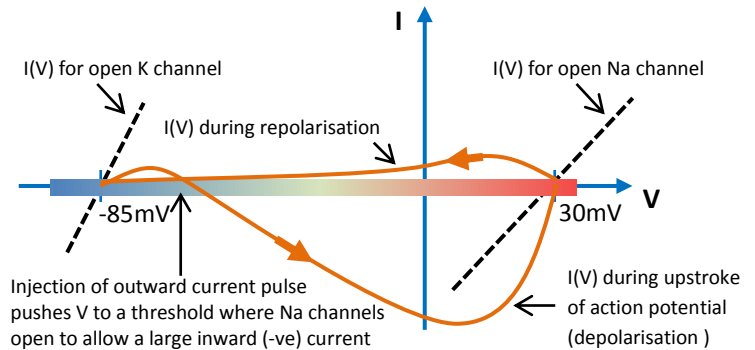


Figure 19. Current-voltage trajectory during an action potential.

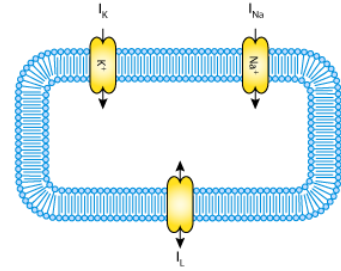
¹⁹ Hodgkin, A.L. and Huxley, A.F. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117, 500-544, 1952. [PubMed ID: 12991237](https://pubmed.ncbi.nlm.nih.gov/12991237/)

²⁰ http://en.wikipedia.org/wiki/Cable_theory

If we neglect the axial current term ($-\frac{1}{r_a} \frac{\partial^2 V}{\partial x^2}$), we can obtain the membrane potential V by integrating the first order ODE,

$$\frac{dV}{dt} = -(i_{Na} + i_K + i_L)/C_m.$$

Figure 20. A schematic cell diagram describing the current flows across the cell bilipid membrane that are captured in the Hodgkin-Huxley model. The membrane ion channels are a sodium (Na^+) channel, a potassium (K^+) channel, and a leakage (L) channel (for chloride and other ions) through which the currents i_{Na} , i_K and i_L flow, respectively.



We use this example to demonstrate the importing feature of CellML. CellML *imports* are used to bring a previously defined CellML model of a component into the new model (in this case the Na and K channel²¹ components defined in the previous two sections, together with a leakage ion channel model specified below). Note that importing a component brings the children components with it along with their connections and units, but it does not bring the siblings of that component with it.

To establish a CellML model of the HH equations we first lay out the model components with their public and private interfaces (Figure 21).

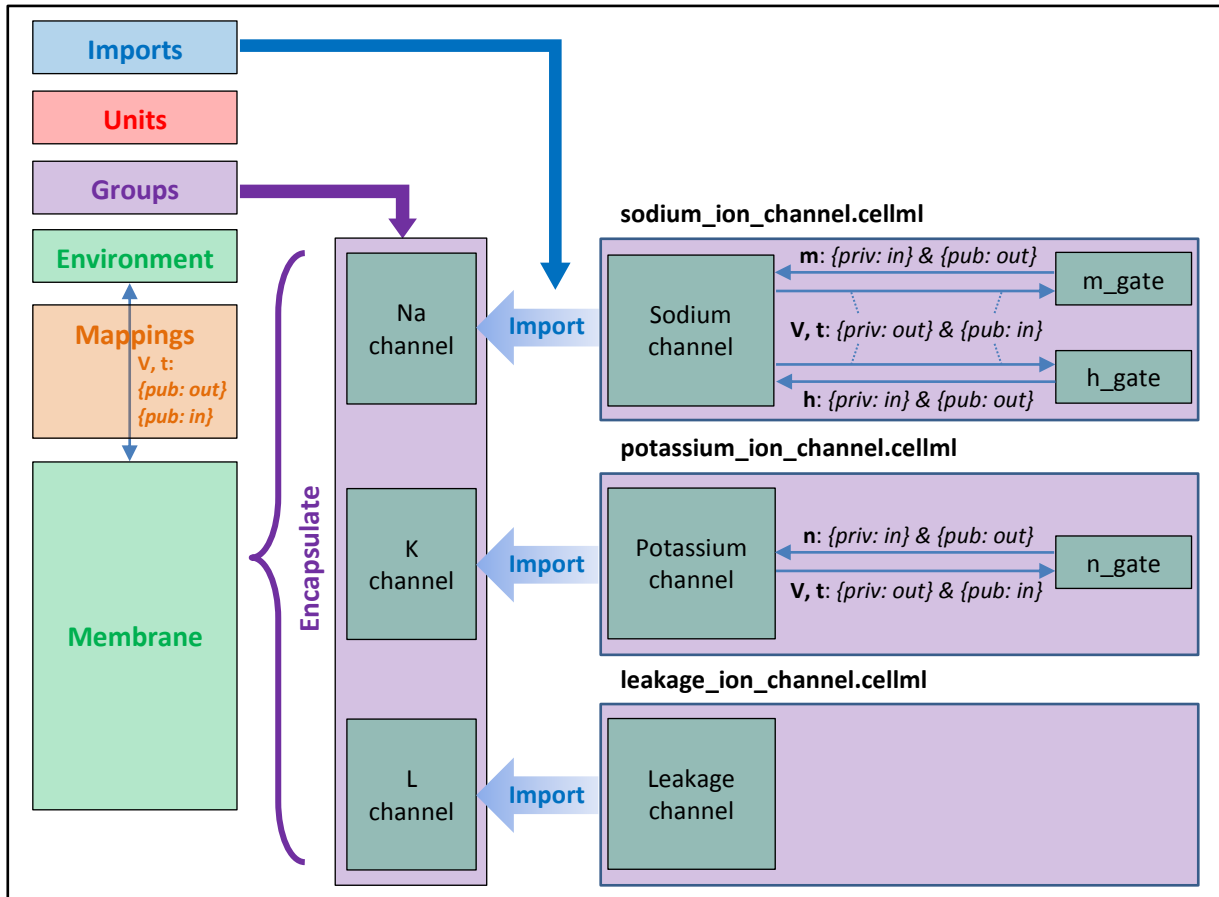


Figure 21. Overall structure of the HH CellML model showing the encapsulation hierarchy (purple), the CellML model imports (blue) and the other key parts (units, components & mappings) of the top level CellML model.

The CellML text code for the K channel, L channel and HH models are given on the next two pages.

²¹ Note that we have updated the K channel model to include the encapsulation of the n-gate into the potassium channel component – see p19.

```
def model potassium_ion_channel as
```

```
  def unit millisec as
    unit second {pref: milli};
  enddef;
  def unit per_millisec as
    unit second {pref: milli, expo: -1};
  enddef;
  def unit millivolt as
    unit volt {pref: milli};
  enddef;
  def unit per_millivolt as
    unit millivolt {expo: -1};
  enddef;
  def unit per_millivolt_millisec as
    unit per_millivolt;
    unit per_millisec;
  enddef;
  def unit microA_per_cm2 as
    unit ampere {pref: micro};
    unit metre {pref: centi, expo: -2};
  enddef;
  def unit milliS_per_cm2 as
    unit siemens {pref: milli};
    unit metre {pref: centi, expo: -2};
  enddef;
```

```
  def comp environment as
    var V: millivolt {init: 0, pub: out};
    var t: millisec {pub: out};
  enddef;
```

```
  def group as encapsulation for
    comp potassium_channel incl
      comp potassium_channel_n_gate;
    endcomp;
  enddef;
```

```
  def comp potassium_channel as
    var V: millivolt {pub: in, priv: out};
    var t: millisec {pub: in, priv: out};
    var n: dimensionless {priv: in};
    var i_K: microA_per_cm2 {pub: out};
    var g_K: milliS_per_cm2 {init: 36};
    var E_K: millivolt {init: -85};
    i_K = g_K*pow(n, 4{dimensionless})*(V-E_K);
  enddef;
```

```
  def comp potassium_channel_n_gate as
    var V: millivolt {pub: in};
    var t: millisec {pub: in};
    var alpha_n: per_millisec;
    var beta_n: per_millisec;
    var n: dimensionless {init: 0.325, pub: out};
    alpha_n = 0.01{per_millivolt_millisec}*(V+10{millivolt})
      /(exp((V+10{millivolt})/10{millivolt})-1{dimensionless});
    beta_n = 0.125{per_millisec}*exp(V/80{millivolt});
    ode(n, t) = alpha_n*(1{dimensionless}-n)-beta_n*n;
  enddef;
```

```
  def map between potassium_channel and environment for
    vars V and V;
    vars t and t;
  enddef;
  def map between
    potassium_channel and potassium_channel_n_gate for
    vars V and V;
    vars t and t;
    vars n and n;
  enddef;
```

```
enddef;
```

```
def model leakage_ion_channel as
```

```
  def unit millisec as
    unit second {pref: milli};
  enddef;
  def unit millivolt as
    unit volt {pref: milli};
  enddef;
  def unit per_millivolt as
    unit millivolt {expo: -1};
  enddef;
  def unit microA_per_cm2 as
    unit ampere {pref: micro};
    unit metre {pref: centi, expo: -2};
  enddef;
  def unit milliS_per_cm2 as
    unit siemens {pref: milli};
    unit metre {pref: centi, expo: -2};
  enddef;
```

```
  def comp environment as
    var V: millivolt {init: 0, pub: out};
    var t: millisec {pub: out};
    ode(V,t) = 1 {dimensionless};
  enddef;
```

```
  def map between leakage_channel and environment for
    vars V and V;
  enddef;
```

```
  def comp leakage_channel as
    var V: millivolt {pub: in};
    var i_L: microA_per_cm2 {pub: out};
    var g_L: milliS_per_cm2 {init: 0.3};
    var E_L: millivolt {init: -54.4};
    i_L = g_L*(V-E_L);
  enddef;
```

```
enddef;
```

def model HH as

```
def import using "sodium_ion_channel.cellml" for
  comp Na_channel using comp sodium_channel;
enddef;
def import using "potassium_ion_channel.cellml" for
  comp K_channel using comp potassium_channel;
enddef;
def import using "leakage_ion_channel.cellml" for
  comp L_channel using comp leakage_channel;
enddef;
```

Imports

```
def unit millisec as
  unit second {pref: milli};
enddef;
def unit millivolt as
  unit volt {pref: milli};
enddef;
def unit microA_per_cm2 as
  unit ampere {pref: micro};
  unit metre {pref: centi, expo: -2};
enddef;
def unit microF_per_cm2 as
  unit farad {pref: micro};
  unit metre {pref: centi, expo: -2};
enddef;
```

Units

```
def group as encapsulation for
  comp all_channels incl
    comp Na_channel;
    comp K_channel;
    comp L_channel;
  endcomp;
enddef;
```

Groups

```
def comp environment as
  var V: millivolt {init: -85, pub: out};
  var t: millisec {pub: out};
enddef;
```

Environment component

```
def map between environment and membrane for
  vars V and V;
  vars t and t;
enddef;
def map between membrane and Na_channel for
  vars V and V;
  vars t and t;
  vars i_Na and i_Na;
enddef;
def map between membrane and K_channel for
  vars V and V;
  vars t and t;
  vars i_K and i_K;
enddef;
def map between membrane and L_channel for
  vars V and V;
  vars i_L and i_L;
enddef;
```

Mappings

```
def comp membrane as
  var V: millivolt {pub: in, priv: out};
  var t: millisec {pub: in, priv: out};
  var i_Na: microA_per_cm2 {pub: out, priv: in};
  var i_K: microA_per_cm2 {pub: out, priv: in};
  var i_L: microA_per_cm2 {pub: out, priv: in};
  var Cm: microF_per_cm2 {init: 1};
  var i_Stim: microA_per_cm2;
  var i_Tot: microA_per_cm2;

  i_Stim = sel
    case (t >= 1{millisec}) and (t <= 1.2{millisec}):
      100{microA_per_cm2};
    otherwise:
      0{microA_per_cm2};
  endsel;

  i_Tot = i_Stim + i_Na + i_K + i_L;
  ode(V,t) = -i_Tot/Cm;
enddef;
```

Membrane component

enddef;

Note that the only units that need to be defined for this top level model are the ones explicitly needed in the membrane component. All the other units, needed for the various imported sub-models are imported along with the imported components.

The results generated by the HH model are shown in Figure 22.

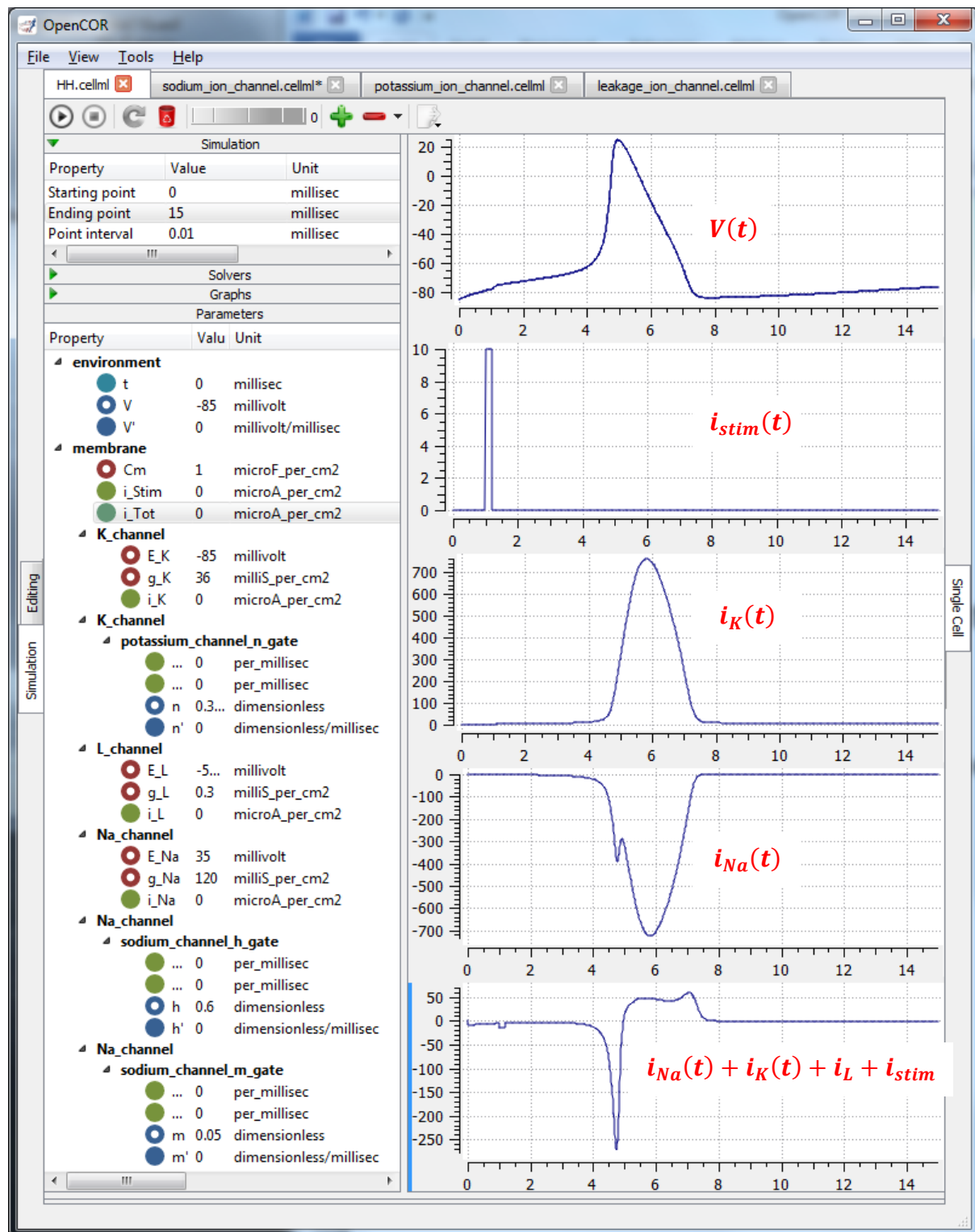


Figure 22. Results from OpenCOR for the Hodgkin Huxley (HH) CellML model. The top panel shows the generated action potential. Note that the stimulus current is not really needed as the background outward leakage current is enough to drive the membrane potential up to the threshold for sodium channel opening.

Important note

It is often convenient to have the sub-models – in this case the `sodium_ion_channel.cellml` model, the `potassium_ion_channel.cellml` model and the `leakage_ion_channel.cellml` model - loaded into OpenCOR at the same time as the high level model (`HH.cellml`), as shown in Figure 23. If you make changes to a model in the CellML text view, you must save the file (*CTRL-S*) before running a new simulation since the simulator works with the saved model. Furthermore, a change to a sub-model will only affect the high level model which imports it if you also save the high level model (or use the *Reload* option under the File menu).

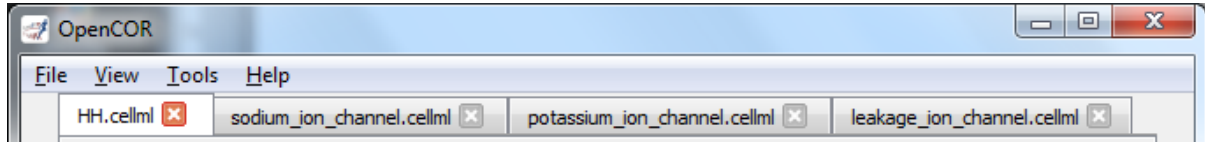


Figure 23. The `HH.cellml` model and its three sub-models are available under separate tabs in OpenCOR.