OpenCOR Tutorial

Release 0.18.0-alpha

Peter Hunter

Oct 30, 2019

CONTENTS

Note: This tutorial originated from a translation from a single Word document in July 2015. Aspects of formatting and presentation may need further work. For reference, the original tutorial is available here: `OpenCOR-Tutorial-v17.pdf`.

The current tutorial has now progressed well beyond the original version and we recommend using this online version or the PDF available via ReadTheDocs.

This tutorial shows you how to install and run the OpenCOR[1] software [APJ15], to author and edit CellML models[2] [DPPJ03] and to use the Physiome Model Repository (PMR)[3] [eal11]. We start by giving a brief background on the VPH-Physiome project. We then create a simple model, save it as a CellML file and run model simulations. We next try opening existing CellML models, both from a local directory and from the Physiome Model Repository. The various features of CellML[4] and OpenCOR are then explained in the context of increasingly complex biological models. A simple linear 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, encapsulation groups and connections. More complex potassium and sodium ion channel models are then developed and subsequently imported into the Hodgkin-Huxley 1952 squid axon neural model using the CellML model import facility. The Noble 1962 model of a cardiac cell action potential is used to illustrate importing of units and parameters. The tutorial finishes with sections on model annotation and the facilities available on the CellML website and the Physiome Model Repository to support model development, including the links to bioinformatic databases. There is a strong emphasis in the tutorial on establishing 'best practice' in the creation of CellML models and using the PMR resources, particularly in relation to modular approaches (model hierarchies) and model annotation.

Note: This tutorial relies on readers having some background in algebra and calculus, but tries to explain all mathematical concepts beyond this, along with the physical principles, as they are needed for the development of CellML models.[5]

---

[1] OpenCOR is an open source, freely available, C++ desktop application written by Alan Garny at INRIA with funding support from the Auckland Bioengineering Institute (http://www.abi.auckland.ac.nz) and the NIH-funded Virtual Physiological Rat (VPR) project led by Dan Beard at the University of Michigan (http://virtualrat.org).

[2] For an overview and the background of CellML see http://www.cellml.org. This project is led by Poul Nielsen and David (Andre) Nickerson at the Auckland (University) Bioengineering Institute (ABI).

[3] https://models.physiomeproject.org. The PMR project is led by Tommy Yu at the ABI.

[4] For details on the specifications of CellML1.0 see http://www.cellml.org/specifications/cellml_1.0.

[5] Please send any errors discovered or suggested improvements to p.hunter@auckland.ac.nz.

CONTENTS 1

BACKGROUND TO THE VPH-PHYSIOME PROJECT

To be of benefit to applications in healthcare, organ and whole organism physiology needs to be understood at both a systems level and in terms of subcellular function and tissue properties. Understanding a re-entrant arrhythmia in the heart, for example, depends on knowledge of not only numerous cellular ionic current mechanisms and signal transduction pathways, but also larger scale myocardial tissue structure and the spatial variation in protein expression. As reductionist biomedical science succeeds in elucidating ever more detail at the molecular level, it is increasingly difficult for physiologists to relate integrated whole organ function to underlying biophysically detailed mechanisms that exploit this molecular knowledge. Multi-scale computational modelling is used by engineers and physicists to design and analyse mechanical, electrical and chemical engineering systems. Similar approaches could benefit the understanding of physiological systems. To address these challenges and to take advantage of bioengineering approaches to modelling anatomy and physiology, the International Union of Physiological Sciences (IUPS) formed the Physiome Project in 1997 as an international collaboration to provide a computational framework for understanding human physiology[1].

1.1 Primary Goals

One of the primary goals of the Physiome Project [PJ04] has been to promote the development of standards for the exchange of information between models. The first of these standards, dealing with time varying but spatially lumped processes, is CellML [VarYY]. The second (dealing with spatially and time varying processes) is FieldML [CPJ09][P13][2]. A further goal of the Physiome Project has been the development of open source tools for creating and visualizing standards-based models and running model simulations. OpenCOR is the latest in a series of software projects aimed at providing a modelling environment for CellML models. Similar tools exist for FieldML models.

Following the publication of the STEP[3] (*Strategy for a European Physiome*) Roadmap in 2006, the European Commission in 2007 initiated the Virtual Physiological Human (VPH) project [ea13]. A related US initiative by the Interagency Modeling and Analysis Group (IMAG) began in 2003[4]. These projects and similar

---

[1] www.iups.org. The IUPS President, Denis Noble from Oxford University, and Jim Bassingthwaighte from the University of Washington in Seattle have been two of the driving forces behind the Physiome Project. Peter Hunter from the University of Auckland was appointed Chair of the newly created Physiome Commission of the IUPS in 2000. The IUPS Physiome Committee, formed in 2008, was co-chaired by Peter Hunter and Sasha Popel (JHU) and is now chaired by Andrew McCulloch from UCSD. The UK Wellcome Trust provided initial support for the Physiome Project through the Heart Physiome grant awarded in 2004 to David Paterson, Denis Noble and Peter Hunter.

[2] CellML began as a joint public-private initiative in 1998 with funding by the US company Physiome Sciences (CEO Jeremy Levin), before being launched under IUPS as a fully open source project in 1999.

[3] The STEP report, led by Marco Viceconte (University of Sheffield, UK), is available at www.europhysiome.org/roadmap.

[4] This coordinates various US Governmental funding agencies involved in multi-scale bioengineering modeling research including NIH, NSF, NASA, the Dept of Energy (DoE), the Dept of Defense (DoD), the US Dept of Agriculture and the Dept of Veteran Affairs. See www.nibib.nih.gov/Research/MultiScaleModeling/IMAG. Grace Peng of NHBIB leads the IMAG group.

initiatives are now coordinated and are collectively referred to here as the 'VPH-Physiome' project[5]. The VPH-Institute[6] was formed in 2012 as a virtual organisation to providing strategic leadership, initially in Europe but now globally, for the VPH-Physiome Project.

---

[5] Other significant contributions to the VPH-Physiome project have come from Yoshi Kurachi in Japan (www.physiome.jp), Stig Omholt in Norway (www.ntnu) and Chae-Hun Leem in Korea (www.physiome.or.kr).

[6] www.vph-institute.org. Formed in 2012, the inaugural Director was Marco Viceconti. The current Director is Adriano Henney. The inaugural and current President of the VPH-Institute is Denis Noble.

TWO

INSTALL AND LAUNCH OPENCOR

Download OpenCOR from www.opencor.ws. Versions are available for Windows, OS X and Linux[1]. Note that some aspects of this tutorial require OpenCOR snapshot 2017-02-10 (or newer). 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 Fig. 2.1(a).
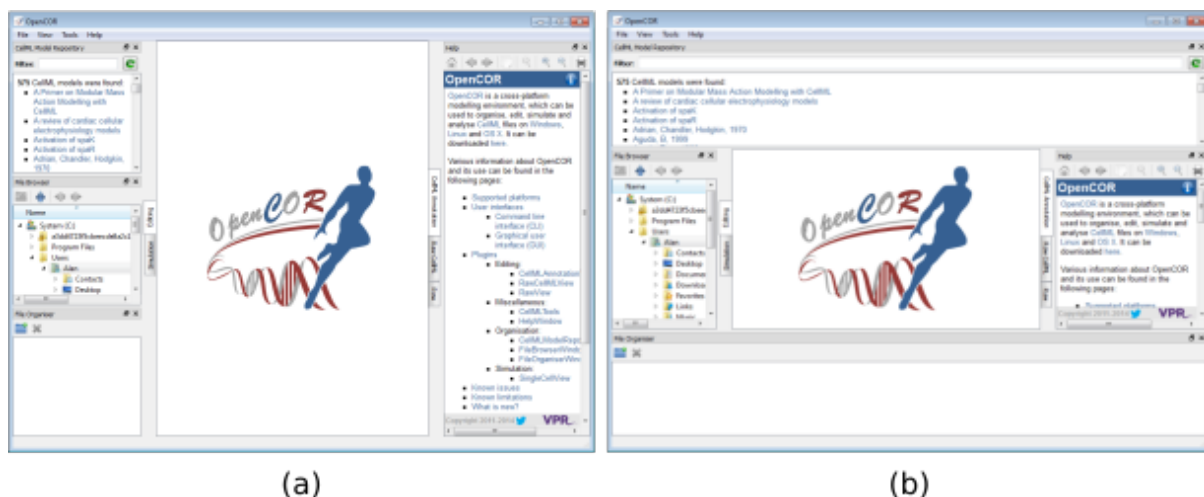

Fig. 2.1: OpenCOR application (a) Default positioning of dockable windows. (b) An alternative configuration achieved by dragging and dropping the dockable windows.

2.1 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. The key combination `Control-spacebar` removes (for less clutter) or restores these two side panels[2].

Any of the subpanels (*Physiome Model Repository*, *File Browser*, and *File Organiser*) can be closed with the top right delete button, and then restored from the *View .. Windows ..* menu. Files can be dragged and dropped into the File Organiser to create a local directory structure for your files.

---

[1] http://opencor.ws/user/supportedPlatforms.html

[2] ⌘ `-spacebar` being the equivalent on OS X.

2.2 Plugins

OpenCOR has a plugin architecture and can be used with or without a range of modules. These can be viewed under the *Tools* menu. By default they are all included, as shown in Fig. 2.2. Information about developing plugins for OpenCOR is also available.
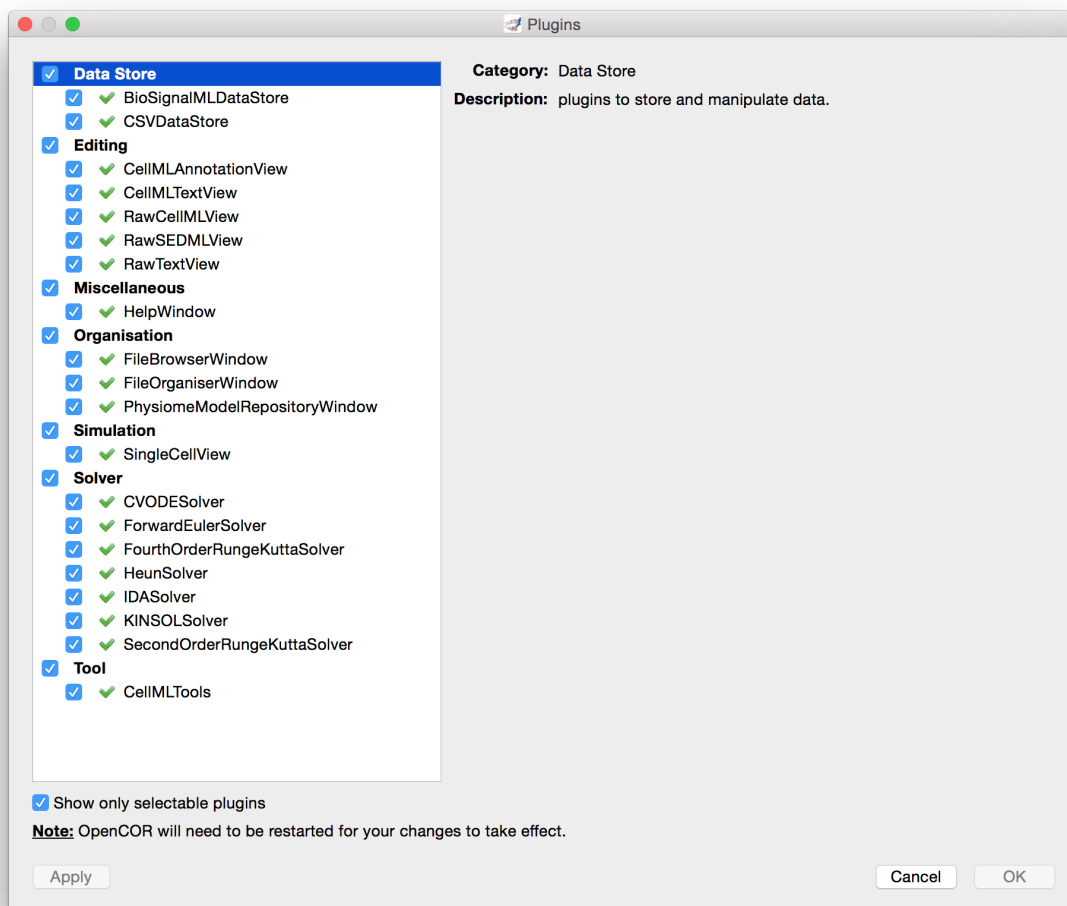

Fig. 2.2: OpenCOR tools menu showing the plugins that are selectable. Untick the box on the bottom left to show all plugins.

CREATE AND RUN A SIMPLE CELLML MODEL: EDITING AND SIMULATION

In this example we create a simple CellML model and run it. The model is the Van der Pol oscillator[1] defined by the second order equation

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

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

$$\frac{\mathrm{dx}}{\mathrm{dt}} = y \tag{3.1}$$

$$\frac{\mathrm{dy}}{\mathrm{dt}} = \mu\left(1 - x^2\right)y - x \tag{3.2}$$

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

With the central pane in *Editing* mode (e.g. *CellML Text* view), create a new CellML file: *File → New → CellML File* and then type in the following lines of code after deleting the three lines that indicate where the code should go:

```
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};
        // These are the ODEs
        ode(x,t)=y;
        ode(y,t)=mu*(1{dimensionless}-sqr(x))*y-x;
    enddef;
enddef;
```

Things to note[3] 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 order[4] ODE in *x* and *t*

 iv. the use of the squaring function *sqr(x)* for x^2, and

---

[1] http://en.wikipedia.org/wiki/Van_der_Pol_oscillator

[2] Equations (3.1) and (3.2) are equations that are implemented directly in OpenCOR.

[3] For more on the *CellML Text* view see http://opencor.ws/user/plugins/editing/CellMLTextView.html.

[4] 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.

v. the use of '//' to indicate a comment.

A partial list of mathematical functions available for OpenCOR is:

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

Table 1. Partial 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 Fig. 3.1.

Note that CellML is a declarative language[5] (unlike say C, Fortran or Matlab, which are procedural languages) and therefore the order of statements 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.
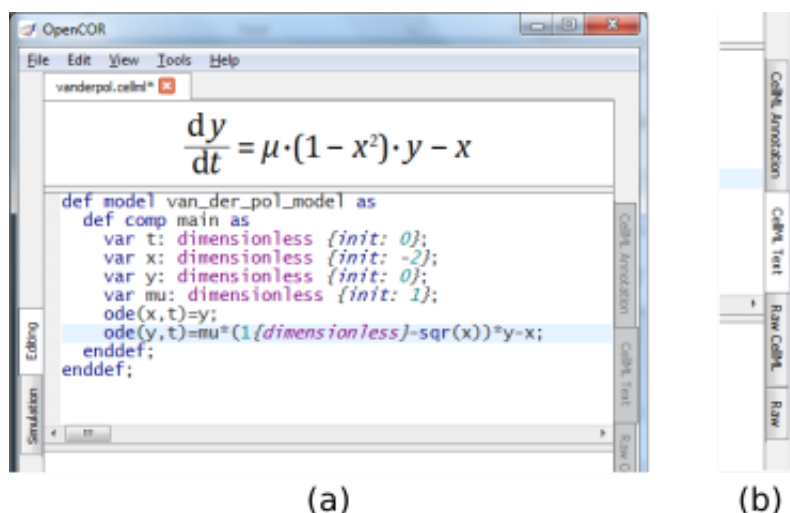

Fig. 3.1: (a) Positioning the cursor over an equation and clicking (shown by the highlighted line) renders the maths. (b) Once the model has been successfully saved, the *CellML Text* view tab becomes white rather than grey. The right hand tabs provide different views of the CellML code.

Now save the code to a local folder using *Save* under the *File* menu (*File → Save*) (or 'CTRL-S') and choosing *.cellml* as the file format[6]. 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 and is intentionally verbose to ensure that the meaning is always unambiguous (note that positioning the cursor over part of the code shows the maths in this view also); and another is the *Raw* view. Notice that 'CTRL-T' in the *Raw CellML* view performs validation tests on the CellML model. The *CellML Text* view provides a much more convenient format for entering and editing the CellML model.

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

---

[5] Note also that the mathematical expressions in CellML are based on MathML – see http://www.w3.org/Math/

[6] Note that .cellml is not strictly required but is best practice.

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.

3.1 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.

3.2 Solvers Panel

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

- Name - this is used to set the solver algorithm. It will be set by default to be the most appropriate solver for the equations you are solving. OpenCOR allows you to change this to another solver appropriate to the type of equations you are solving if you choose to. For example, CVODE for ODE (ordinary differential equation) problems, IDA for DAE (differential algebraic equation) problems, KINSOL for NLA (non-linear algebraic) problems[7].
- Other parameters for the chosen solver – e.g. *Maximum step*, *Maximum 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 problem[8].

3.3 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 parameter.

3.4 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. The icons alongside them have the following meanings:

---

[7] Other solvers include forward Euler, Heun and Runga-Kutta solvers (RK2 and RK4).

[8] 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.

Ⓒ	Editable constant	Ⓢ	Editable state variable
Ⓒ	Computed constant	Ⓡ	Rate variable
Ⓥ	Variable of integration	Ⓐ	Algebraic quantity

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 *Plot Against Variable of Integration* – in this case *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, select *y* and select *Plot Against* then *Main* 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, including the time taken, are displayed in the bottom panel. This can be hidden by dragging down on the bar just above the panel. Fig. 3.2 shows the results. Use the *interval delay* wheel to slow down the plotting if you want to watch the solution evolve. You can also pause the simulation at any time by clicking on the *Run* control and if you change a parameter during the pause, the simulation will continue (when you click the *Run* control button again) with the new parameter.

Note that the values shown for the various parameters are the values they have at the end of the solution run. To restore these to their initial values, use the *Reset parameters* (↻) button. To clear the graphical output traces, click on the *Clear simulation data* (🗑) button.

The top two graphical output panels are showing the time-dependent solution of the *x* and *y* variables. The bottom panel shows how *y* varies as a function of *x*. This is called the solution in state space and it is often useful to analyse the state space solution to capture the key characteristics of the equations being solved.
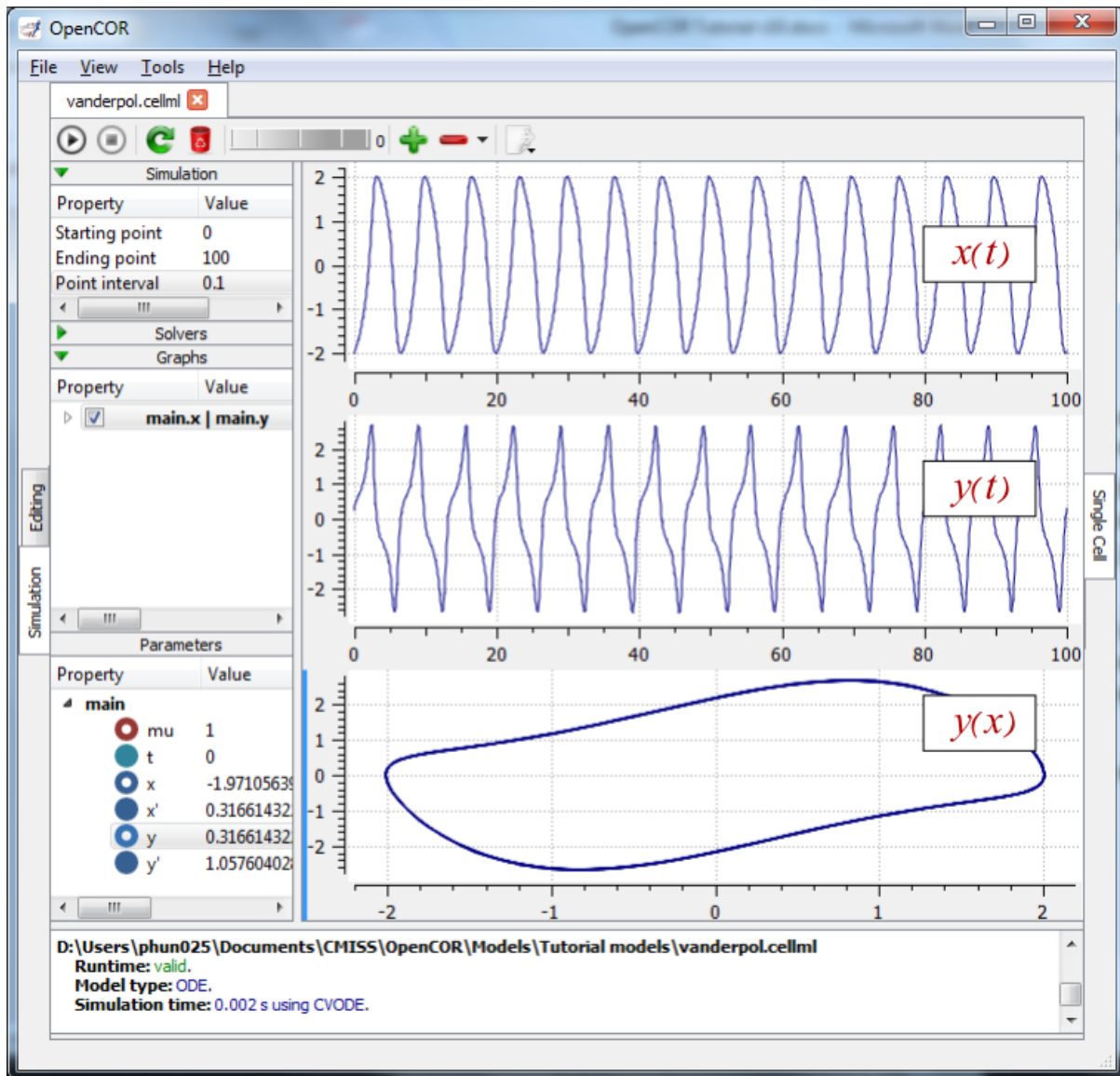
Fig. 3.2: Graphical output from OpenCOR. The top window is *x(t)*, the middle is *y(t)* and the bottom is *y(x)*. The *Graphs* panel shows that *y(x)* is being plotted on the graph output window highlighted by the LH blue line. The window at the very bottom provides runtime information on the type of equation being solved and the simulation time (2ms in this case). The computed variables shown in the left hand panel are at the values they have at the end of the simulation.

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 in future versions of OpenCOR.

You can move the graphical output traces around with 'left click and drag' and you can change the horizontal or vertical scale with 'right click and drag'. Holding the SHIFT key down while clicking on a graphical output panel allows you to interrogate the solution at any point. Right clicking on a panel provides zoom facilities.

Note: The simulation described above can also be loaded and run directly in OpenCOR using this link.

The various plugins used by OpenCOR can be viewed under the Tools menu. A French language version of OpenCOR is also available under the *Tools* menu. An option under the *File* menu allows a file to be locked (also 'CTRL-L'). To indicate that the file is locked, the background colour switches to pink in the *CellML Text* and *Raw CellML* views and a lock symbol appears on the filename tab. Note that OpenCOR text is case sensitive.

FOUR

OPEN AN EXISTING CELLML FILE FROM A LOCAL DIRECTORY OR THE PHYSIOME MODEL REPOSITORY

Go to the *File* menu and select *Open...* (*File → Open*). Browse to the folder that contains your existing 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 Fig. 2.1(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*, *Physiome 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 *Physiome Model Repository* (PMR) (e.g. Chen, Popel, 2007) opens a new browser window with that model (PMR is covered in more detail in Section 13). You can either load this model directly into OpenCOR or create an identical copy (clone) of the model in your local directory. Note that PMR contains *workspaces* and *exposures*. Workspaces are online environments for the collaborative development of models (e.g. by geographically dispersed groups) and can have password protected access. Exposures are workspaces that are exposed for public view and mostly contain models from peer-reviewed journal publications. There are about 600 exposures based on journal papers and covering many areas of cell processes and other ODE/algebraic models, but these are currently being supplemented with reusable protein-based models – see discussion in a Section 13.

To load a model directly into OpenCOR, click on the right-most of the two buttons in Fig. 4.1 - this lists the CellML models in that exposure - and then click on the model you want. Clicking on the left hand button copies the PMR workspace to a local directory that you specify. This is useful if you want to use that model as a template for a new one you are creating.
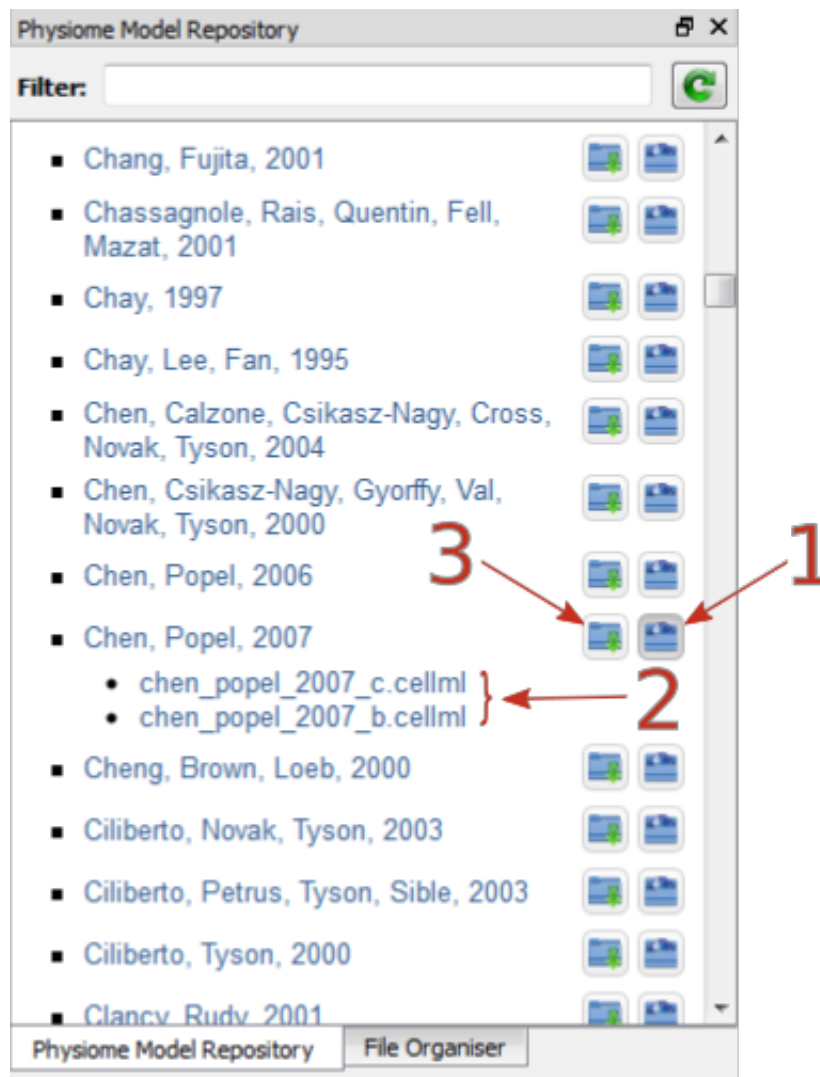
Fig. 4.1: The Physiome Model Repository (PMR) window listing all PMR models. These can be opened from within OpenCOR using the two buttons to the right of a model, as explained below.

In the PMR window (Fig. 4.1) the buttons on the right-hand side [**1**] lists all the CellML files for this model. Clicking on one of those [**2**] uploads the model into OpenCOR. The left-hand buttons [**3**] copies the PMR workspace to a local directory.

A SIMPLE FIRST ORDER ODE

The simplest example of a first order ODE is

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -ay + b$$

with the solution

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

where $y(0)$ or y_0, the value of $y(t)$ at $t = 0$, is the *initial condition*. The final steady state solution as $t \to \infty$ is $y(\ t|_\infty) = y_\infty = \frac{b}{a}$ (see Figure 6). Note that $t = \tau = \frac{1}{a}$ is called the *time constant* of the exponential decay, and that

Fig. 5.1: Solution of 1st order equation.

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

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 ($y(\infty)$) values[1].

Choosing parameters $a = \tau = 1; b = 2$ 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 is shown in Fig. 5.2(a) for these parameters (a decaying exponential) and in Fig. 5.2(b) for parameters $a = 1; b = 5$ and $y(0) = 2$ (an inverted decaying exponential). Note the simulation panel with *Ending point*=10, *Point interval*=0.1. Try putting $a = -1$.

---

[1] It is often convenient to write a first order equation as $\tau \frac{\mathrm{d}y}{\mathrm{d}t} = -y + y_\infty$, so that its solution is expressed in terms of time constant τ, initial condition y_0 and steady state solution y_∞ as: $y(t) = y_\infty + (y_0 - y_\infty) .e^{-\frac{t}{\tau}}$.

(a)   (b)

Fig. 5.2: OpenCOR output $y(t)$ for the simple ODE model with parameters (a) $a = 1; b = 2$ and $y(0) = 5$ (OpenCOR link), and (b) $a = 1; b = 5$ and $y(0) = 2$. The red arrow indicates the point at which the trace reaches the time constant τ (e^{-1} or $\approx 37\%$ of the difference between the initial and final solution values). The black arrows indicate the initial and final (steady state) solutions. Note that the parameters on the left have been reset to their initial values for this figure - normally they would be at their final solution values.

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

The exponential decay curve shown on the left in Fig. 5.2 is a common feature of many models and in the case of radioactive decay (for example) is a statement that the **rate of decay** ($-\frac{dy}{dt}$) is proportional to the **current amount of substance** (y). This is illustrated on the NZ\$100 note (should you be lucky enough to possess one), shown in Figure 8.
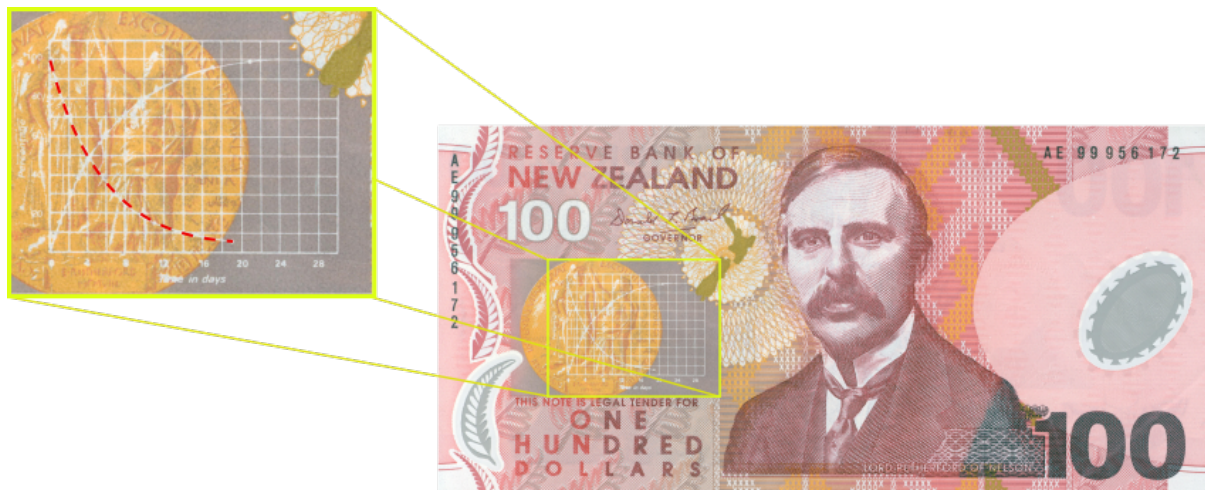
Fig. 5.3: The **exponential curve** representing the naturally occurring radioactive decay explained by the New Zealand Noble laureate Sir Ernest Rutherford - best known for 'splitting the atom'. This may be the only bank note depicting the mathematical solution of a first order ODE.

THE LORENZ ATTRACTOR

An example of a third order ODE system (i.e. three 1st order equations) is the *Lorenz equations*[1].

This system has three equations:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \sigma\left(y - x\right)$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = x\left(\rho - z\right) - y$$

$$\frac{\mathrm{d}z}{\mathrm{d}t} = xy - \beta z$$

where σ, ρ and β are parameters.

The *CellML Text* code entered for these equations is shown in Fig. 6.1 with parameters

$\sigma = 10$, $\rho = 28$, $\beta = 8/3 = 2.66667$

and initial conditions

$x\left(0\right) = y\left(0\right) = z\left(0\right) = 1$.

Solutions for $x\left(t\right)$, $y\left(x\right)$ and $z\left(x\right)$, corresponding to the time integration parameters shown on the LHS, are shown in Fig. 6.2. Note that this system exhibits 'chaotic dynamics' with small changes in the initial conditions leading to quite different solution paths.

This example illustrates the value of OpenCOR's ability to plot variables as they are computed. Use the *Simulation Delay* wheel to slow down the plotting by a factor of about 5-10,000 - in order to follow the solution as it spirals in ever widening trajectories around the left hand wing of the attractor before coming close to the origin that then sends it off to the right hand wing of the attractor.
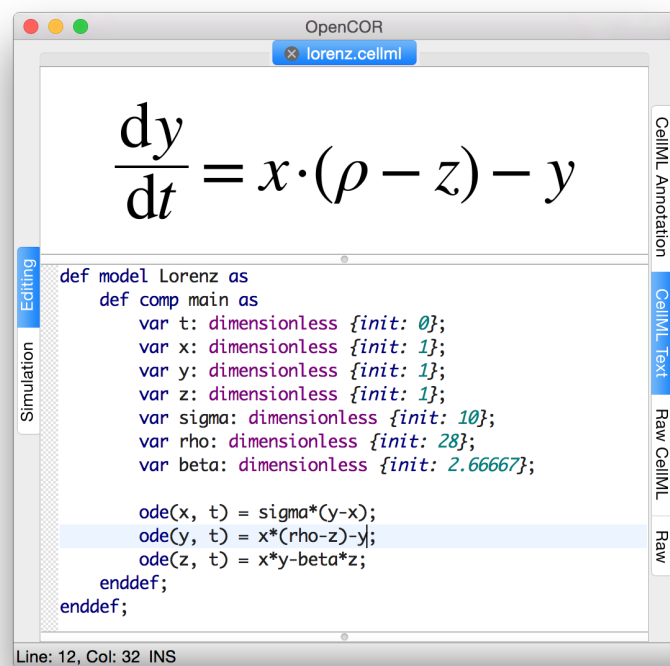
Fig. 6.1: *CellML Text* code for the Lorenz equations.

---

[1] http://en.wikipedia.org/wiki/Lorenz_system
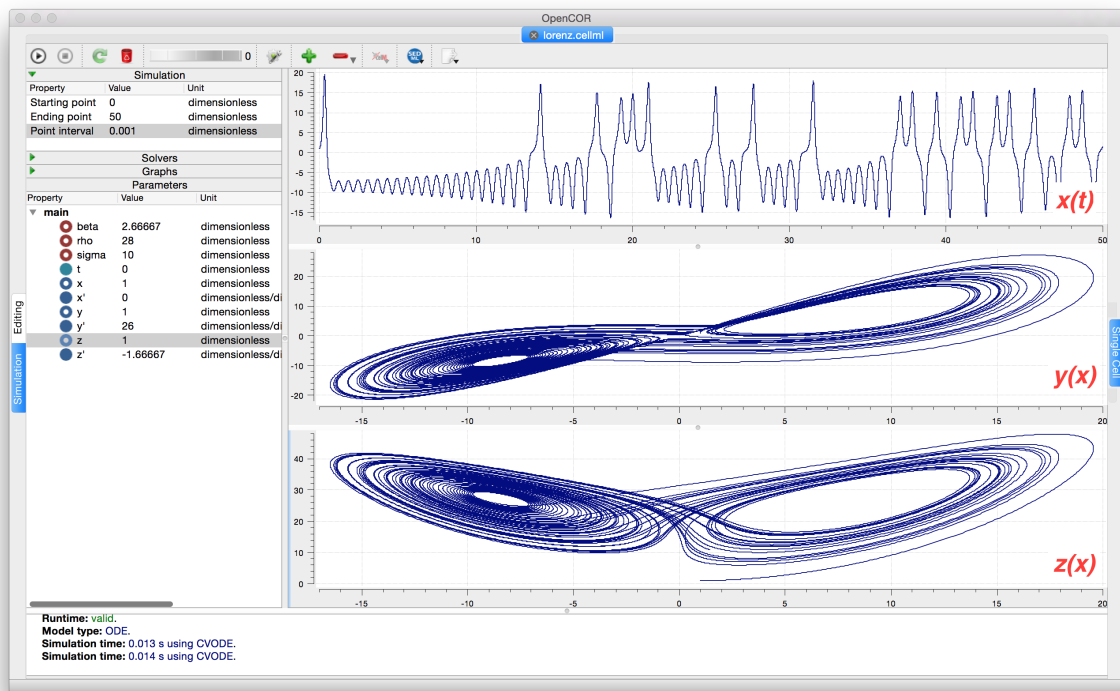
Fig. 6.2: Solutions of the Lorenz equations. Note that the parameters on the left have been reset to their initial values for this figure – normally they would be at their final solution values.

Solutions to the Lorenz equations are organised by the 2D 'Lorenz manifold'. This surface has a very beautiful shape and has become an art form - even rendered in crochet![2] (See Fig. 6.3).

Fig. 6.3: The crocheted Lorenz manifold made by Professors Hinke Osinga and Bernd Krauskopf of the Mathematics Department at the University of Auckland, New Zealand.

Note: The simulation presented in Fig. 6.2 can be loaded direction into OpenCOR using this link.

6.1 Exercise for the reader

Another example of intriguing and unpredictable behaviour from a simple deterministic ODE system is the 'blue sky catastrophe' model [JH02] defined by the following equations:

$$\frac{dx}{dt} = y$$

$$\frac{dy}{dt} = x - x^3 - 0.25y + A\sin t$$

---

[2] http://www.math.auckland.ac.nz/~hinke/crochet/

with parameter $A = 0.2645$ and initial conditions $x(0) = 0.9$, $y(0) = 0.4$. Run to $t = 500$ with $\Delta t = 0.01$ and plot $x(t)$ and $y(x)$ (OpenCOR link). Also try with $A = 0.265$ to see how sensitive the solution is to small changes in parameter values.

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 Huxley [AAF52] to describe the gating behaviour of an ion channel (see also next three sections). Before we describe the gating behaviour of an ion channel, however, we need to explain the concepts of the 'Nernst potential' and channel conductance.

An ion channel is a protein or protein complex embedded in the bilipid membrane surrounding a cell and containing a pore through which an ion Y^+ (or Y^-) can pass when the channel is open. If the concentration of this ion is $[Y^+]_o$ outside the cell and $[Y^+]_i$ inside the cell, the force driving an ion through the pore is calculated from the change in *entropy*.

Entropy S ($J.K^{-1}$) is a measure of the number of microstates available to a system, as defined by Boltzmann's equation $S = k_B \ln W$, where W is the number of ways of arranging a given distribution of microstates of a system and k_B is Boltzmann's constant[1]. The driving force for ion movement is the dispersal of energy into a more probable distribution (see Fig. 7.1; cf. the second law of thermodynamics[2]).
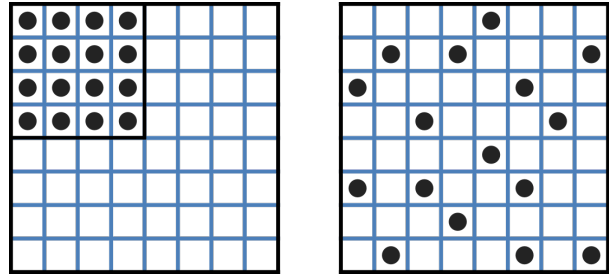
The energy change Δq associated with this change of entropy ΔS at temperature T is $\Delta q = T\Delta S$ (J).

For a given volume of fluid the number of microstates W available to a solute (and hence the entropy of the solute) at a high concentration is less than that for a low concentration[3]. The energy difference driving ion movement from a high ion concentration $[Y^+]_i$ (lower entropy) to a lower ion concentration $[Y^+]_o$ (higher entropy) is therefore

$$\Delta q = T\Delta S = k_B T \left(\ln [Y^+]_o - \ln [Y^+]_i \right) = k_B T \ln \frac{[Y^+]_o}{[Y^+]_i} \ (J.ion^{-1})$$

or

$$\Delta Q = RT \ln \frac{[Y^+]_o}{[Y^+]_i} \ (J.mol^{-1}).$$

$$R = k_B N_A \approx$$
$$1.34x10^{-23}(J.K^{-1})\text{x}6.02x10^{23}(mol^{-1}) \approx$$

Fig. 7.1: Distribution of microstates in a system [J97]. The 16 particles in a confined region (left) have only one possible arrangement (W = 1) and therefore zero entropy ($k_B\ln W = 0$). When the barrier is removed and the number of possible locations for each particle increases 4x (right), the number of possible arrangements for the 16 particles increases by 416 and the increase in entropy is therefore $ln(416)$ or $16ln4$. The thermal energy (temperature) of the previously confined particles on the left has been redistributed in space to achieve a more probable (higher entropy) state. If we now added more particles to the container on the right, the concentration would increase and the entropy would decrease.

---

[1] The Brownian motion of individual molecules has energy $k_B T$ (J), where the Boltzmann constant k_B is approximately $1.34x10^{-23}$ ($J.K^{-1}$). At 25°C, or 298K, $k_B T = 4.10^{-21}$ (J) is the minimum amount of energy to contain a 'bit' of information at that temperature.

[2] The *first law of thermodynamics* states that energy is conserved, and the *second law* (that natural processes are accompanied by an increase in entropy of the universe) deals with the distribution of energy in space.

[3] At infinitely high concentration the specified volume is jammed packed with solute and the entropy is zero.

$8.4 (J.mol^{-1}K^{-1})$ is the 'universal gas constant'[4]. At 25°C (298K), RT $\approx$ $2.5kJ.mol^{-1}$.

Every positively charged ion that crosses the membrane raises the potential difference and produces an electrostatic driving force that opposes the entropic force (see Fig. 7.2). To move an electron of charge e ($\approx 1.6x10^{-19}$ C) through a voltage change of $\Delta\phi$ (V) requires energy $e\Delta\phi$ (J) and therefore the energy needed to move an ion Y^+ of valence $z=1$ (the number of charges per ion) through a voltage change of $\Delta\phi$ is $ze\Delta\phi$ ($J.ion^{-1}$) or $zeN_A\Delta\phi$ ($J.mol^{-1}$). Using Faraday's constant $F = eN_A$, where $F \approx 0.96x10^5 C.mol^{-1}$, the change in energy density at the macroscopic scale is $zF\Delta\phi$ ($J.mol^{-1}$).

Fig. 7.2: The balance between entropic and electrostatic forces determines the Nernst potential.

No further movement of ions takes place when the force for entropy driven ion movement exactly equals the opposing electrostatic driving force associated with charge movement:

$$zF\Delta\phi = RT \ln \frac{[Y^+]_o}{[Y^+]_i} \; (J.mol^{-1}) \text{ or } \Delta\phi = E_Y = \frac{RT}{zF} \ln \frac{[Y^+]_o}{[Y^+]_i} \; (J.C^{-1} \text{ or V})$$

where E_Y is the 'equilibrium' or 'Nernst' potential for Y^+. At 25°C (298K), $\frac{RT}{F} = \frac{2.5x10^3}{0.96x10^5}(J.C^{-1}) \approx 25mV$.

For an open channel the electrochemical current flow is driven by the open channel conductance g_Y times the difference between the transmembrane voltage V and the Nernst potential for that ion:

$i_Y = g_Y (V - E_Y)$.

This defines a linear current-voltage relation ('Ohms law') as shown in Fig. 7.3. The gates to be discussed below modify this open channel conductance.
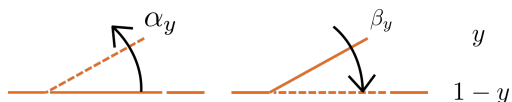

Fig. 7.3: Open channel linear current-voltage relation

Fig. 7.4: Ion channel gating kinetics. y is the fraction of gates in the open state. _y and _y are the rate constants for opening and closing, respectively.

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 Fig. 7.4):

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

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

The solution to this ODE is

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

Fig. 7.5: Transient behaviour for one gate (left) and gates in series (right). Note that the right hand graph has an initial S-shaped increase, reflecting the multiple gates in series.

---

[4] N_A is Avogadro's number ($6.023x10^{23}$) and is the scaling factor between molecular and macroscopic processes. Boltzmann's constant k_B and electron charge e operate at the atomic/molecular scale. Their effect at the physiological scale is via the universal gas constant $R = k_BN_A$ and Faraday's constant $F = eN_A$.
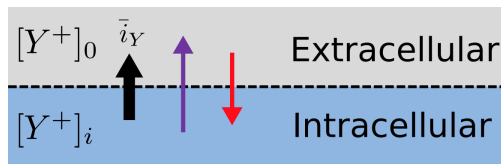
The constant A can be interpreted as $A = y(0) - \frac{\alpha_y}{\alpha_y + \beta_y}$ as in the previous example and, with $y(0) = 0$ (i.e. all gates initially shut), the solution looks like Fig. 7.5(a).

The experimental data obtained by Hodgkin and Huxley for the squid axon, however, indicated that the initial current flow began more slowly (Fig. 7.5(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

$$i_Y = i_Y y^\gamma = y^\gamma g_Y (V - E_Y)$$

where $i_Y = g_Y (V - E_Y)$ is the steady state current through the open gate.

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

Note that the decision to deal with units in CellML, rather than just ignoring them or insisting that all equations are represented in dimensionless form, was made in order to be able to check the physical consistency of all terms in each equation[5].

There are seven base physical quantities defined by the *International d'Unités* (SI)[6]. 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)

- **luminous intensity** (candela).

All other units are derived from these seven. Additional derived units that CellML defines intrinsically (with their dependence on previously defined units) are: **Hz** (s^{-1}); **Newton**, N ($kg.m.s^{-1}$); **Joule**, J ($N.m$); **Pascal**, Pa ($N.m^{-2}$); **Watt**, W ($J.s^{-1}$); **Volt**, V ($W.A^{-1}$); **Siemen**, S ($A.V^{-1}$); **Ohm**, Ω ($V.A^{-1}$); **Coulomb**, C ($s.A$); **Farad**, F ($C.V^{-1}$); **Weber**, Wb ($V.s$); and **Henry**, H ($Wb.A^{-1}$). Multiples and fractions of these are defined as follows:

	Prefix		deca	hecto	kilo	mega	giga	tera	peta	exa	zetta	yotta
Multi-ples	Sym-bol		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}
	Prefix		deci	centi	milli	mi-cro	nano	pico	femto	atto	zepto	yocto
Frac-tions	Sym-bol		d	c	m		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}

Units for this model, with multiples and fractions, are illustrated in the following *CellML Text* code:

```
1  def model first_order_model as
2      def unit millisec as
3          unit second {pref: milli};
4      enddef;
```

(continues on next page)

---

[5] It is well accepted in engineering analysis that thinking about and dealing with units is a key aspect of modelling. Taking the ratio of dimensionally consistent terms provides non-dimensional numbers which can be used to decide when a term in an equation can be omitted in the interests of modelling simplicity. We investigate this idea further in a later section.

[6] http://en.wikipedia.org/wiki/International_System_of_Units

```
5      def unit per_millisec as
6          unit second {pref: milli, expo: -1};
7       enddef;
8      def unit millivolt as
9          unit volt {pref: milli};
10     enddef;
11     def unit microA_per_cm2 as
12         unit ampere {pref: micro};
13         unit metre {pref: centi, expo: -2};
14     enddef;
15     def unit milliS_per_cm2 as
16         unit siemens {pref: milli};
17         unit metre {pref: centi, expo: -2};
18     enddef;
19     def comp ion_channel as
20         var V: millivolt {init: 0};
21         var t: millisec {init: 0};
22         var y: dimensionless {init: 0};
23         var E_y: millivolt {init: -85};
24         var i_y: microA_per_cm2;
25         var g_y: milliS_per_cm2 {init: 36};
26         var gamma: dimensionless {init: 4};
27         var alpha_y: per_millisec {init: 1};
28         var beta_y: per_millisec {init: 2};
29         ode(y, t) = alpha_y*(1{dimensionless}-y)-beta_y*y;
30         i_y = g_y*pow(y, gamma)*(V-E_y);
31     enddef;
32 enddef;
```

Line 2: Define units for time as millisecs

Line 5: Define per_millisec units

Line 8: Define units for voltage as millivolts

Line 11: Define units for current as microAmps per cm^2

Line 15: Define units for conductance as milliSiemens per cm^2

Lines 20-28: Define units and initial conditions for variables

Line 29: Define ODE for gating variable y

Line 30: Define channel current

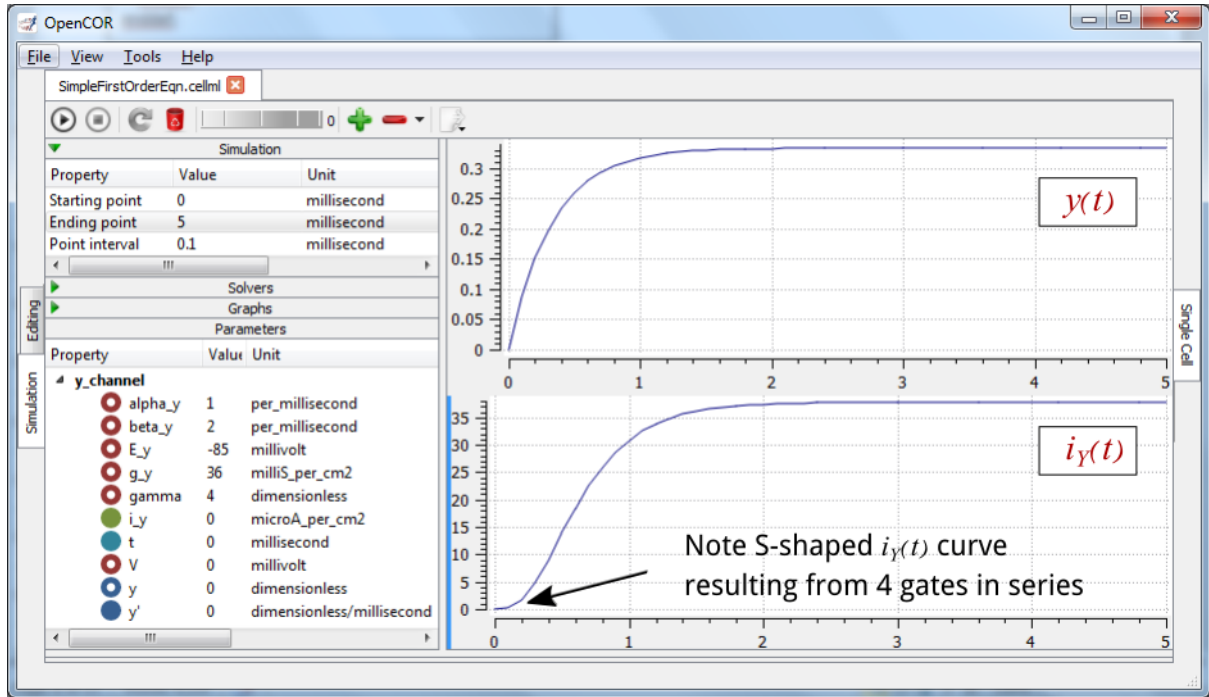The solution of these equations for the parameters indicated above is illustrated in Fig. 7.6.

Fig. 7.6: The behaviour of an ion channel with $\gamma = 4$ gates transitioning from the closed to the open state at a membrane voltage $V = 0$ (OpenCOR link). The opening and closing rate constants are $\alpha_y = 1$ ms^{-1} and $\beta_y = 2$ ms^{-1}. The ion channel has an open conductance of $g_Y = 36$ mS.cm^{-2} and an equilibrium potential of $E_Y = -85$ mV. The upper transient is the response $y(t)$ for each gate and the lower trace is the current through the channel. Note the slow start to the current trace in comparison with the single gate transient $y(t)$.

The model of a gated ion channel presented here is used in the next two sections for the neural potassium and sodium channels and then in Section 11 for cardiac ion channels. The gates make the channel conductance time dependent and, as we will see in the next section, the experimentally observed voltage dependence of the gating rate constants α_y and β_y means that the channel conductance (including the open channel conductance) is voltage dependent. For a partially open channel ($y < 1$), the steady state conductance is $(y_\infty)^\gamma \cdot g_Y$, where $y_\infty = \frac{\alpha_y}{\alpha_y + \beta_y}$. Moreover the gating time constants $\tau = \frac{1}{\alpha_y + \beta_y}$ are therefore also voltage dependent. Both of these voltage dependent factors of ion channel gating are important in explaining channel properties, as we show now for the neural potassium and sodium ion channels.

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

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

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 ($z = 1$ since one +ve charge on K^+) is

$$E_k = \tfrac{\text{RT}}{\text{zF}} \, ln \tfrac{[K^+]_o}{[K^+]_i} = 25 \, ln \tfrac{3}{90} = -85 \text{mV}.$$

As noted above, 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 potential (voltage) difference between internal and external ion charges is exactly matched by the entropic driving force from the ion concentration difference. $n^4 \bar{g}_K$ is the channel conductance.

The gating kinetics are described (as before) by

$$\tfrac{\text{dn}}{\text{dt}} = \alpha_n \left(1 - n \right) - \beta_n . \text{n}$$

with time constant $\tau_n = \tfrac{1}{\alpha_n + \beta_n}$ (see *A simple first order ODE*).

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 Fig. 8.1) as follows[1]:

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

Note that under steady state conditions when $t \to \infty$ and

$$\tfrac{\text{dn}}{\text{dt}} \quad \to \quad 0, \quad n|_{t=\infty} \quad = \quad n_\infty \quad = \quad \tfrac{\alpha_n}{\alpha_n + \beta_n}.$$

The voltage dependence of the steady state channel conductance is then

$$g_{\text{SS}} = \left(\tfrac{\alpha_n}{\alpha_n + \beta_n} \right)^4 . \bar{g}_Y.$$

(see Fig. 8.1). The steady state current-voltage relation for the channel is illustrated in Fig. 8.2.
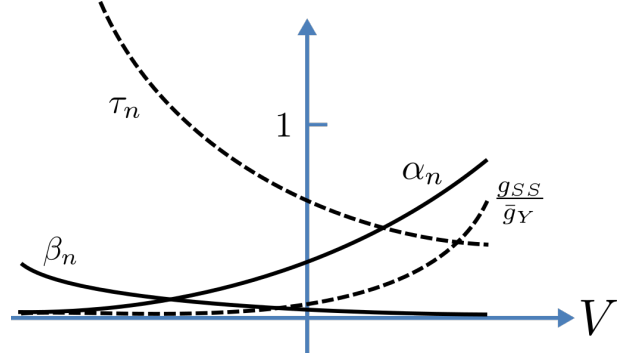
Fig. 8.1: Voltage dependence of rate constants α_n and β_n (ms^{-1}), time constant τ_n (ms) and relative conductance $\tfrac{g_{SS}}{\bar{g}_Y}$.

---

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

These equations are captured with OpenCOR *CellML Text* view (together with the previous unit definitions) below. But first we need to explain some further CellML concepts.

We introduced CellML units above. We now need to introduce three more CellML constructs: components, connections (mappings between components) and groups. For completeness we also show one other construct in Fig. 8.3, imports, that will be used later in *A model of the nerve action potential: Introducing CellML imports*.

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 [DPPJ03]. 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 below.

The *CellML Text* code for the potassium ion channel model is as follows[2]:

Fig. 8.2: The steady-state current-voltage relation for the potassium channel.
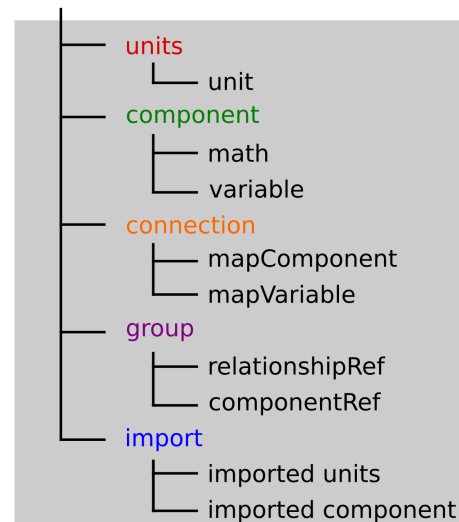
Fig. 8.3: Key entities in a CellML model.

Potassium_ion_channel.cellml

```
1   def model potassium_ion_channel as
2      def unit millisec as
3         unit second {pref: milli};
4      enddef;
5      def unit per_millisec as
6       ↳ unit second {pref: milli, expo: -1};
7      enddef;
8      def unit millivolt as
9         unit volt {pref: milli};
10     enddef;
11     def unit per_millivolt as
12        unit millivolt {expo: -1};
13     enddef;
14     def unit per_millivolt_millisec as
15        unit per_millivolt;
16        unit per_millisec;
17     enddef;
18     def unit microA_per_cm2 as
19        unit ampere {pref: micro};
20      ↳ unit metre {pref: centi, expo: -2};
21     enddef;
22     def unit milliS_per_cm2 as
23        unit siemens {pref: milli};
24      ↳ unit metre {pref: centi, expo: -2};
25     enddef;
26     def unit mM as
27        unit mole {pref: milli};
28     enddef;
29     def comp environment as
```
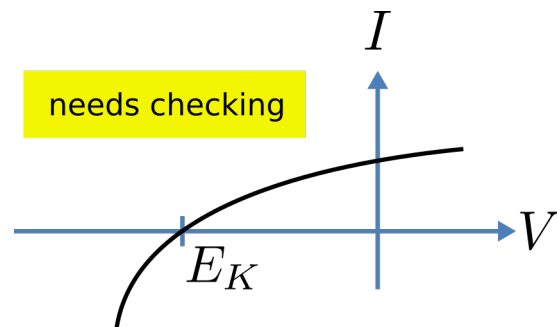
(continues on next page)

---

[2] From here on we use a coloured background to identify code blocks that relate to a particular CellML construct: units, components, mappings and encapsulation groups and later imports.

```
30          var V: millivolt { pub: out};
31          var t: millisec {pub: out};
32          V = sel
33          case (t > 5
   {millisec}) and (t < 15 {millisec}):
34              -85.0 {millivolt};
35          otherwise:
36              0.0 {millivolt};
37          endsel;
38      enddef;
39      def group as encapsulation for
40          comp potassium_channel incl
41              comp potassium_channel_n_gate;
42          endcomp;
43      enddef;
44      def comp potassium_channel as
45          var
   V: millivolt {pub: in , priv: out};
46
   var t: millisec {pub: in, priv: out};
47          var n: dimensionless {priv: in};
48
     var i_K: microA_per_cm2 {pub: out};
49
     var g_K: milliS_per_cm2 {init: 36};
50          var Ko: mM {init: 3};
51          var Ki: mM {init: 90};
52          var RTF: millivolt {init: 25};
53          var E_K: millivolt;
54          var K_conductance:
   milliS_per_cm2 {pub: out};
55          E_K=RTF*ln(Ko/Ki);
56          K_conductance
   = g_K*pow(n, 4{dimensionless});
57          i_K = K_conductance*(V-E_K);
58      enddef;
59      def comp potassium_channel_n_gate as
60          var V: millivolt {pub: in};
61          var t: millisec {pub: in};
62          var n: dimensionless
   {init: 0.325, pub: out};
63          var alpha_n: per_millisec;
64          var beta_n: per_millisec;
65          alpha_n = 0.01{per_
   millivolt_millisec}*(V+10{millivolt})
66              /(exp((V+10{millivolt}
   )/10{millivolt})-1{dimensionless});
67          beta_n = 0.125
   {per_millisec}*exp(V/80{millivolt});
68          ode(n, t) = alpha_
   n*(1{dimensionless}-n)-beta_n*n;
69      enddef;
70      def map between
   environment and potassium_channel for
71          vars V and V;
72          vars t and t;
73      enddef;
74
   def map between potassium_channel and
75          potassium_channel_n_gate for
76          vars V and V;
```

<div style="text-align:center">(continued from previous page)</div>

```
77          vars t and t;
78          vars n and n;
79      enddef;
80  enddef;
```

Lines 2-28: Define units.

Lines 29-38: Define component 'environment'.

Lines 32-37: Define **voltage step**.

Lines 39-43: Define encapsulation of 'n_gate'.

Lines 44-58: Define component 'potassium_channel'.

Lines 59-69: Define component 'potassium_channel_n_gate'.

Lines 70-79: Define mappings between components for variables that are shared between these components.

Note that several other features have been added:

- the event control *select case* which indicates that the voltage is specified to jump from 0 mV to -85 mV at t = 5 ms then back to 0 mV at t = 15 ms. This is only used here in order to test the K channel model; when the potassium_channel component is later imported into a neuron model, **the environment component is not imported**.

- the use of encapsulation to embed the **potassium_channel_n_gate** inside the **potassium_channel**. This avoids the need to establish mappings from **environment** to **potassium_channel_n_gate** since the gate component is entirely within the channel component.

- the use of $\{pub : in\}$ and $\{pub : out\}$ to indicate which variables are either supplied as inputs to a component or produced as outputs from a component[3]. 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.

We now use OpenCOR, with *Ending point* 40 and *Point interval* 0.1, to solve the equations for the potassium channel under a voltage step condition in which the membrane voltage is clamped initially at 0mV and then stepped down to -85mV for 10ms before being returned to 0mV. At 0mV, the steady state value of the n gate is $n_\infty = \frac{\alpha_n}{\alpha_n + \beta_n} = 0.324$ and, at -85mV, $n_\infty = 0.945$.

The voltage traces are shown at the top of Figure 21. The n-gate response, shown next, is to open further from its partially open value of $n = 0.324$ at 0mV and then plateau at an almost fully open state of $n = 0.945$ at the Nernst potential -85mV before closing again as the voltage is stepped back to 0mV. Note that the gate opening behaviour (set by the voltage dependence of the α_n opening rate constant) is faster than the closing behaviour (set by the

---

[3] 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.

voltage dependence of the β_n closing rate constant). The channel conductance (= $n^4 \bar{g}_K$) is shown next – note the initial s-shaped conductance increase caused by the n^4 (four gates in series) effect on conductance. Finally the channel current i_K = conductance x $(V - E_K)$ is shown at the bottom. Because the voltage is clamped at the Nernst potential (-85mV) during the period when the gate is opening, there is no current flow, but when the voltage is stepped back to 0mV, the open gates begin to close and the conductance declines but now there is a voltage gradient to drive an outward (positive) current flow through the partially open channel – albeit brief since the channel is closing.
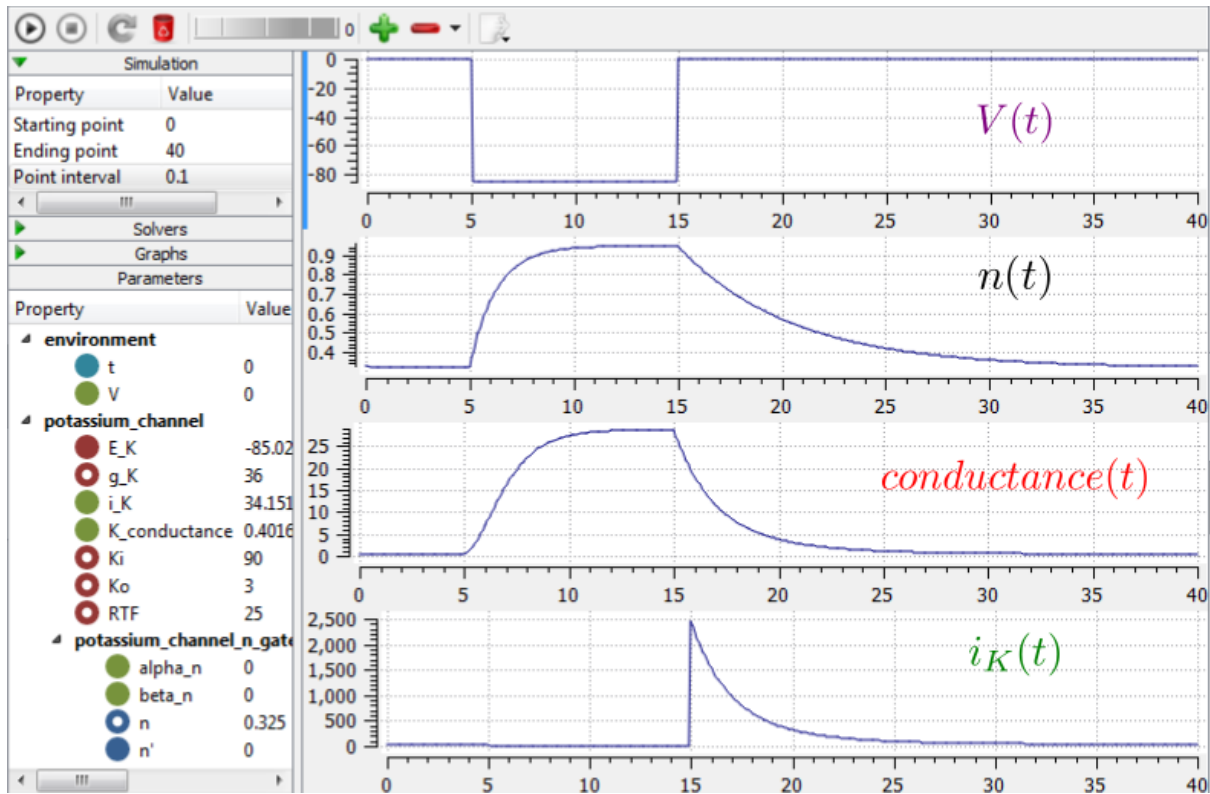

Fig. 8.4: Kinetics of the potassium channel gates for a voltage step from 0mV to -85mV (OpenCOR link). The voltage clamp step is shown at the top, then the n gate first order response, then the channel conductance, then the channel current. Notice how the conductance is slightly slower to turn on (due to the four gates in series) but fast to inactivate. Current only flows when there is a non-zero conductance and a non-zero voltage gradient. This is called the 'tail current'.

Note that the *CellML Text* code above includes the Nernst equation with its dependence on the concentrations $[K^+]_i$= 90mM and $[K^+]_o$= 3mM. Try raising the external potassium concentration to $[K^+]_o$= 10mM – you will then see the Nernst potential increase from -85mV to -55mV and a negative (inward) current flowing during the period when the membrane voltage is clamped to -85mV. The cell is now in a 'hyperpolarised' state because the potential is less than the equilibrium potential.

Note that you can change a model parameter such as $[K^+]_o$ either by changing the value in the left hand *Parameters* window (which leaves the file unchanged) or by editing the *CellML Text* code (which does change the file when you save from *CellML Text* view – which you have to do to see the effect of that change.

This potassium channel model will be used later, along with a sodium channel model and a leakage channel model, to form the Hodgkin-Huxley neuron model, where the membrane ion channel current flows are coupled to the equations governing current flow along the axon to generate an action potential.

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,

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

where $\bar{g}_{\text{Na}} = 120$ mS.cm^{-2}, and with $\left[\text{Na}^+ \right]_i = 30$mM and $\left[\text{Na}^+ \right]_o = 140$mM, the Nernst potential for the sodium channel (z=1) is

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

The gating kinetics are described by

$$\frac{\text{dm}}{\text{dt}} = \alpha_m \left(1 - m \right) - \beta_m . \text{m}; \frac{\text{dh}}{\text{dt}} = \alpha_h \left(1 - h \right) - \beta_h . \text{h}$$

where the voltage dependence of these four rate constants is determined experimentally to be[1]

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

Before we construct a CellML model of the sodium channel, we first introduce some further CellML concepts that help deal with the complexity of biological models: 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'.

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

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 above:

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 Fig. 9.1).

---

[1] The HH paper used $\alpha_m = \frac{0.1(v+25)}{e^{\frac{(v+25)}{10}} - 1}; \beta_m = 4 e^{\frac{v}{18}}; \alpha_h = 0.07 e^{\frac{v}{20}}; \beta_h = \frac{1}{e^{\frac{(v+30)}{10}} + 1};$.
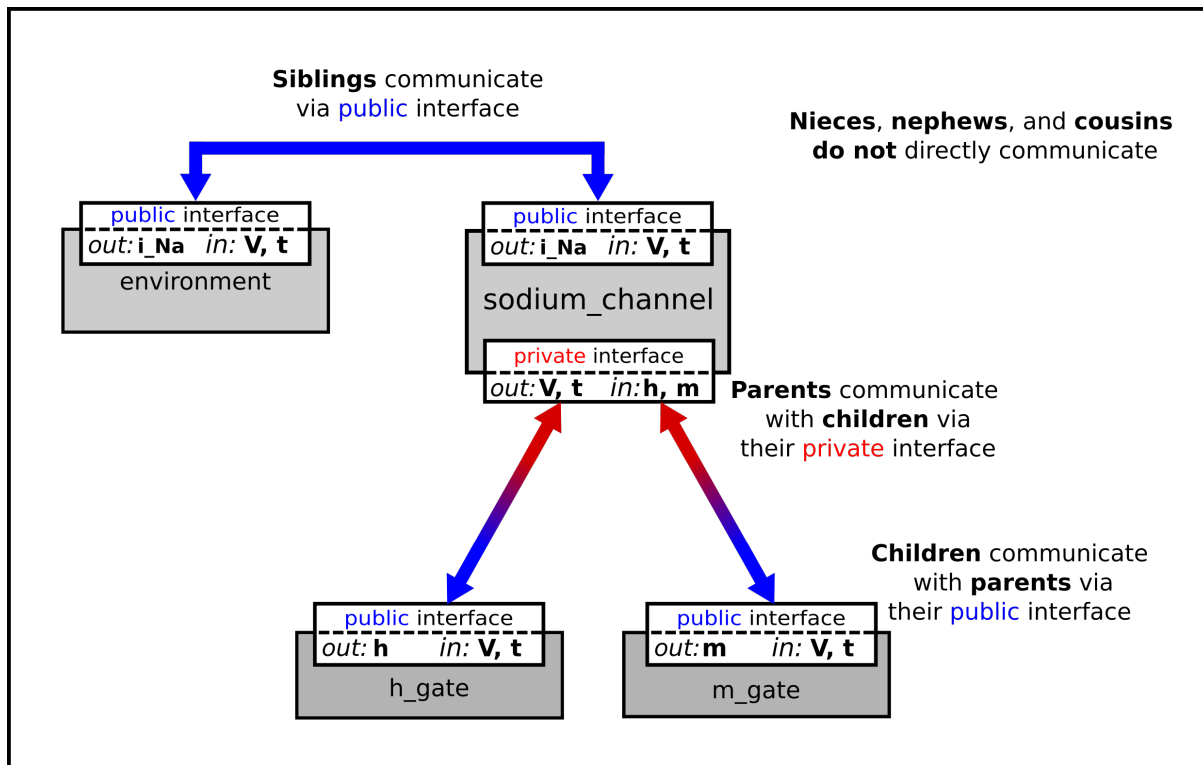
Fig. 9.1: Children talk to each other as siblings, and to their parents, via *public interfaces*. But the outside world can only talk to children through their parents via a *private interface*. Note that the siblings **m_gate** and **h_gate** could talk via a *public interface* but only if a mapping is established between them (not needed here).

The OpenCOR *CellML Text* for the HH sodium ion channel is given below.

```
Sodium_ion_channel.cellml
```

```
def model sodium_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 unit mM as
      unit mole {pref: milli};
```

```
    enddef;
    def comp environment as
        var V: millivolt {pub: out};
        var t: millisec {pub: out};
        V = sel
        case (t > 5 {millisec}) and (t < 15 {millisec}):
            -20.0 {millivolt};
        otherwise:
            -85.0 {millivolt};
        endsel;
    enddef;
    def group as encapsulation for
        comp sodium_channel incl
            comp sodium_channel_m_gate;
            comp sodium_channel_h_gate;
        endcomp;
    enddef;
    def comp sodium_channel as
        var V: millivolt {pub: in, priv: out};
        var t: millisec {pub: in, priv: out };
        var m: dimensionless {priv: in};
        var h: dimensionless {priv: in};
        var g_Na: milliS_per_cm2 {init: 120};
        var i_Na: microA_per_cm2 {pub: out};
        var Nao: mM {init: 140};
        var Nai: mM {init: 30};
        var RTF: millivolt {init: 25};
        var E_Na: millivolt;
        var Na_conductance: milliS_per_cm2 {pub: out};

        E_Na=RTF*ln(Nao/Nai);
        Na_conductance = g_Na*pow(m, 3{dimensionless})*h;
        i_Na= Na_conductance*(V-E_Na);
    enddef;
    def comp sodium_channel_m_gate as
        var V: millivolt {pub: in};
        var t: millisec {pub: in};
        var alpha_m: per_millisec;
        var beta_m: per_millisec;
        var m: dimensionless {init: 0.05, pub: out};
        alpha_m = 0.1{per_millivolt_millisec}*(V+25{millivolt})
            /(exp((V+25{millivolt})/10{millivolt})-1{dimensionless});
        beta_m = 4{per_millisec}*exp(V/18{millivolt});
        ode(m, t) = alpha_m*(1{dimensionless}-m)-beta_m*m;
    enddef;
    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;
    def map between environment and sodium_channel 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;
enddef;
```

The results of the OpenCOR computation, with *Ending point* 40 and *Point interval* 0.1, are shown in Fig. 9.2 with plots $V(t)$, $m(t)$, $h(t)$, $g_{\text{Na}}(t)$ and $i_{\text{Na}}(t)$ for voltage steps from (a) -85mV to -20mV, (b) -85mV to 0mV and (c) -85mV to 20mV. There are several things to note:

i. The kinetics of the m-gate are much faster than the h-gate.

ii. The opening behaviour is faster as the voltage is stepped to higher values since $\tau = \frac{1}{\alpha_n + \beta_n}$ reduces with increasing V (see Fig. 8.1).

iii. 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 Fig. 8.4 which is only reduced back to zero by stepping the voltage back to its resting value – i.e. *deactivating* it.

iv. 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.

v. The large sodium current i_{Na} is an inward current and hence negative.

Note that the bottom trace does not quite line up at t=0 because the values shown on the axes are computed automatically and hence can take more or less space depending on their magnitude.
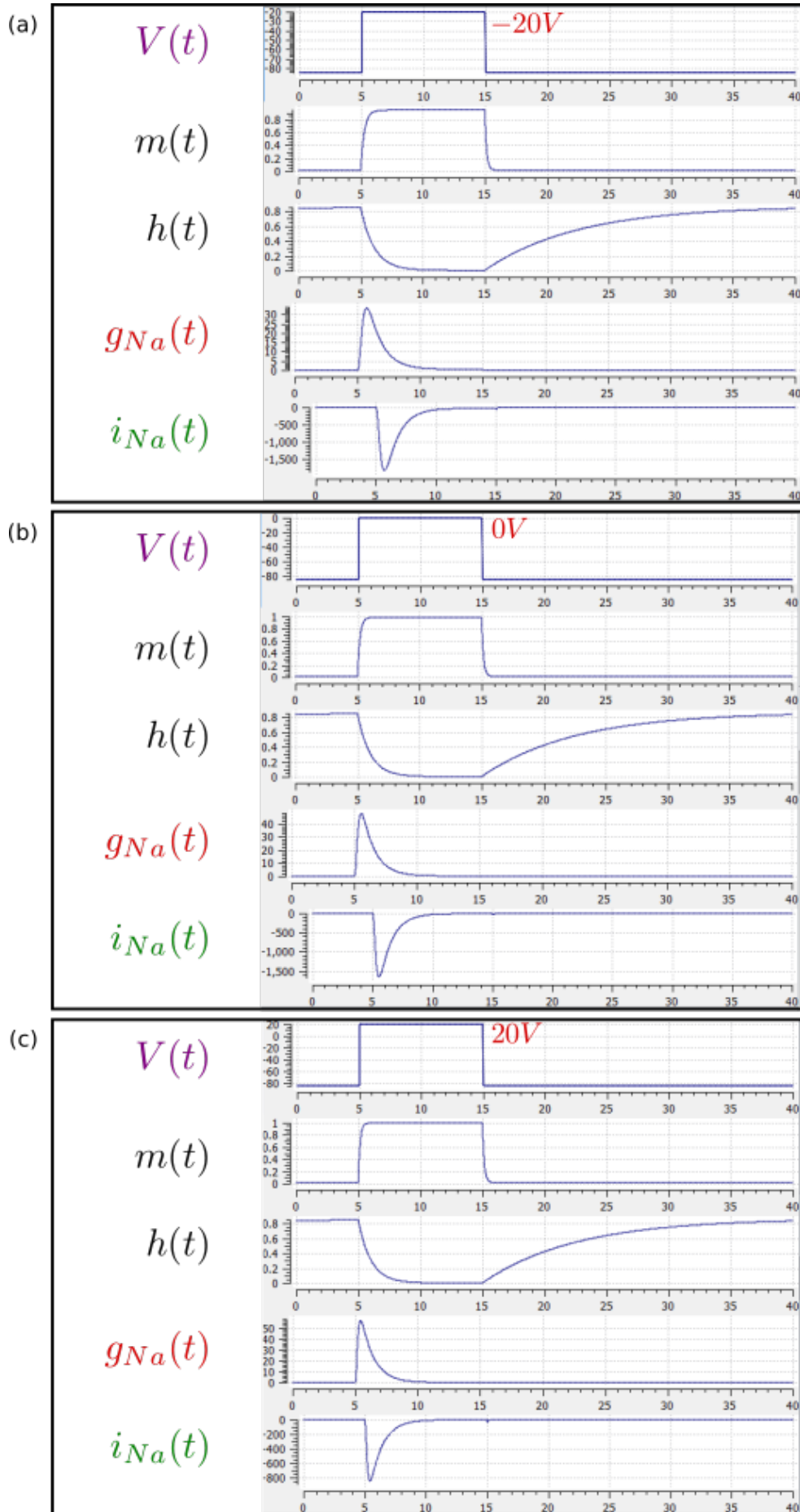
Fig. 9.2: Kinetics of the sodium channel gates for voltage steps to (a) -20mV, (b) 0mV (OpenCOR link), and (c) 20mV.

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 1952 [AAF52] 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"*.

10.1 Cable equation

The *cable equation* was developed in 1890[1] 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 Fig. 10.1), 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 ($\Omega.m^{-1}$), Ohm's law gives $-\frac{\partial V}{\partial x} = r_a i_a$. But if the 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 ($\approx 1\mu F/cm^2$) 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
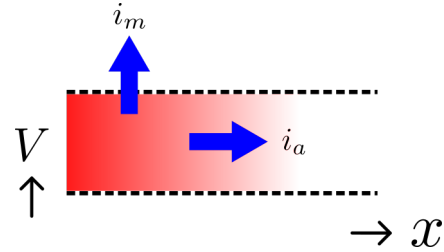

Fig. 10.1: Current flow in a leaky cable.

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

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

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

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

10.2 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
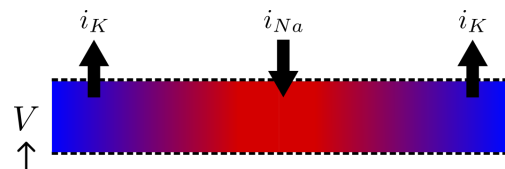
---

[1] http://en.wikipedia.org/wiki/Cable_theory

Fig. 10.2: Current flow in a neuron.

outward movement of potassium ions through a membrane 'potassium channel', giving the **outward** current i_K (see Fig. 10.2). A further small leak current $i_L = g_L (V - E_L)$ associated with chloride and other ions is also included.

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 Fig. 10.3).

We can neglect[2] the term $(-\frac{1}{r_a} \frac{\partial^2 V}{\partial x^2})$ (the rate of change of axial current along the cable) for the present models since we assume the whole cell is clamped with an axially uniform potential. We can therefore obtain the membrane potential V by integrating the first order ODE

$$\frac{\mathrm{d}V}{\mathrm{d}t} = -\left(i_{\mathrm{Na}} + i_K + i_L\right)/C_m.$$
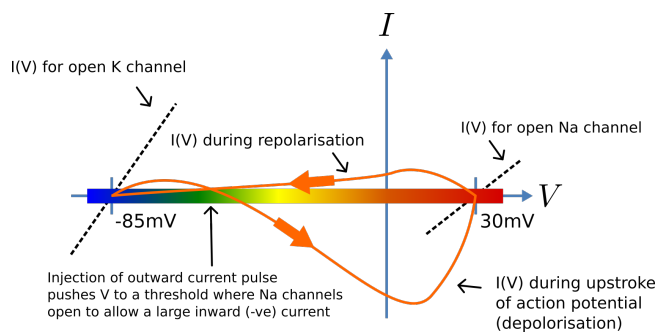

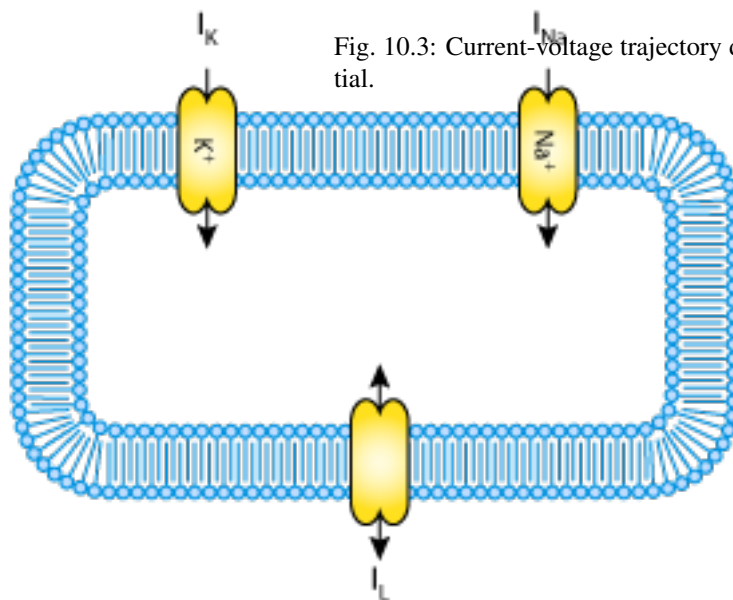Fig. 10.3: Current-voltage trajectory during an action potential.

Fig. 10.4: 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 channel 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 (Fig. 10.5).

---

[2] This term is needed when determining the propagation of the action potential, including its wave speed.
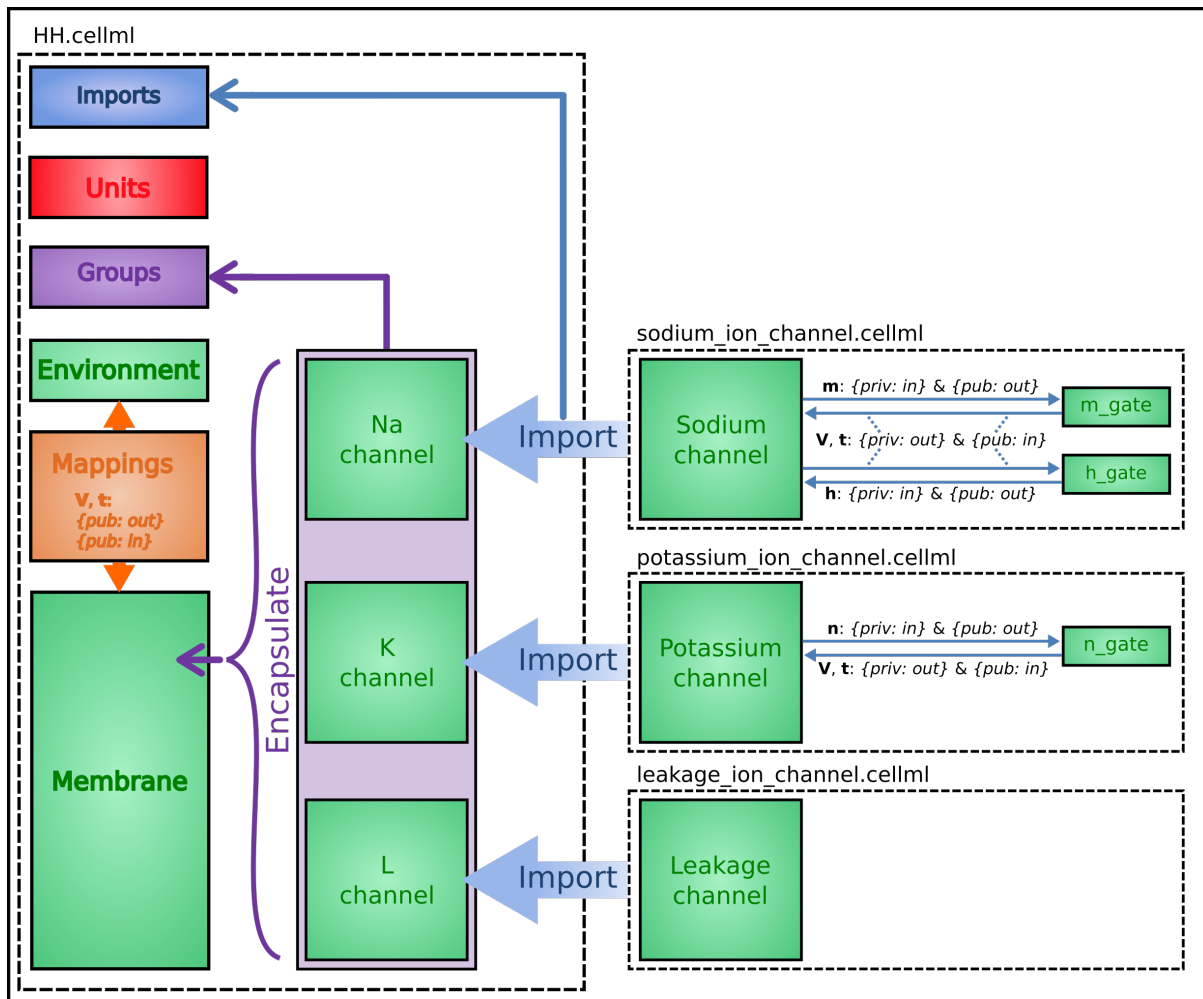

Fig. 10.5: Overall structure of the HH CellML model showing the encapsulation hierarchy (purple), the CellML model imports (blue) and the other key parts (units, components, and mappings) of the top level CellML model.

The HH model is the top level model. The *CellML Text* code for the HH model, together with the leakage_channel model, is given below. The imported potassium_ion_channel model and sodium_ion_channel model are unchanged from the previous sections

HH.cellml

```
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;
   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;
   def group as encapsulation for
      comp membrane incl
         comp Na_channel;
         comp K_channel;
         comp L_channel;
      endcomp;
   enddef;
   def comp environment as
      var V: millivolt {init: -85, pub: out};
      var t: millisec {pub: out};
   enddef;
   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;
   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;
enddef;
```

Leakage_ion_channel

```
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};
    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;
```

Note that the *CellML Text* code for the potassium channel is *Potassium_ion_channel.cellml* and for the sodium channel is *Sodium_ion_channel.cellml*.

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

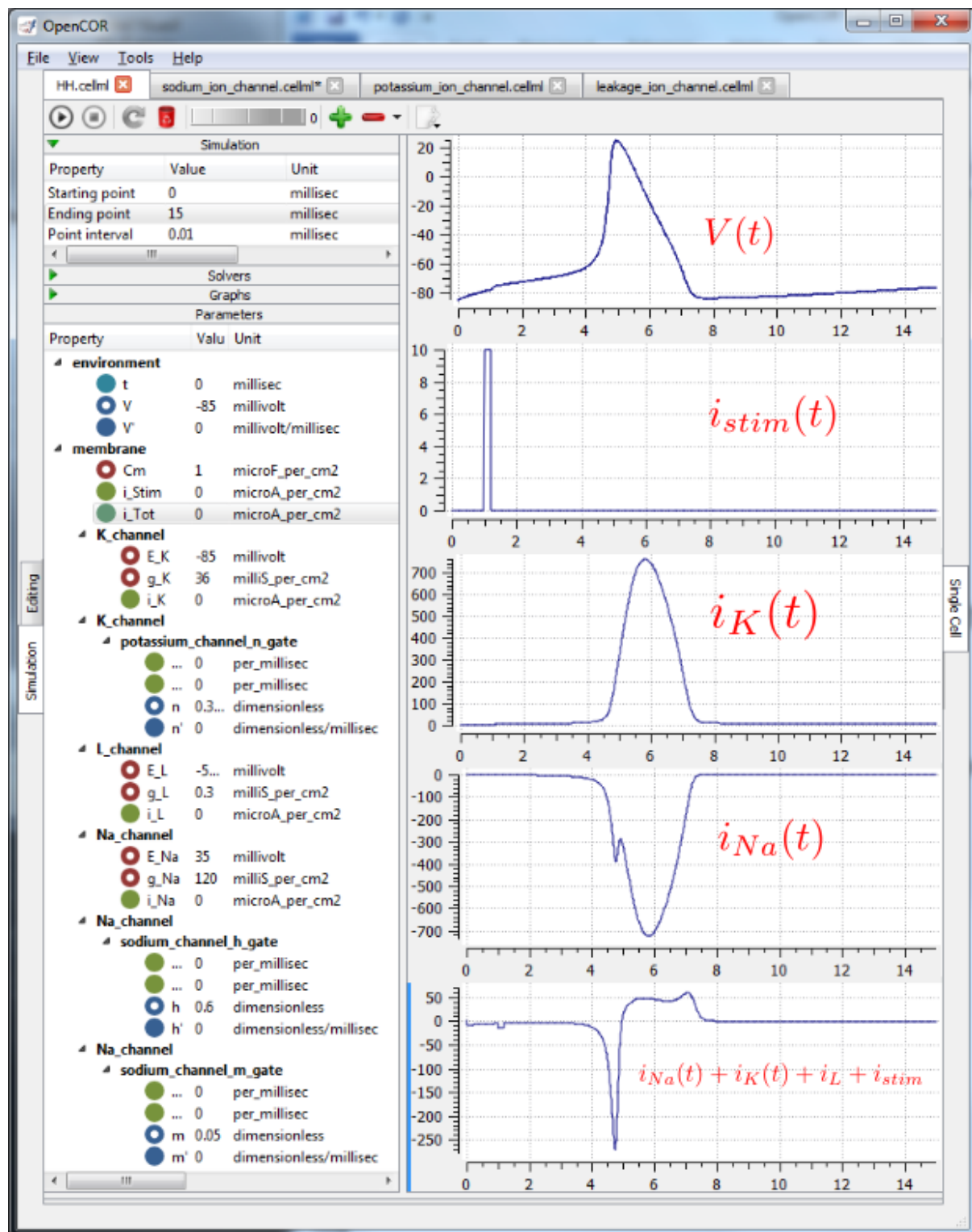The results generated by the HH model are shown in Fig. 10.6.

Fig. 10.6: 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.

10.2.1 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 Fig. 10.7 . 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). An asterisk appears next to the name of a file when a change has been made and the file has not been saved. The asterisk disappears when the file is saved.
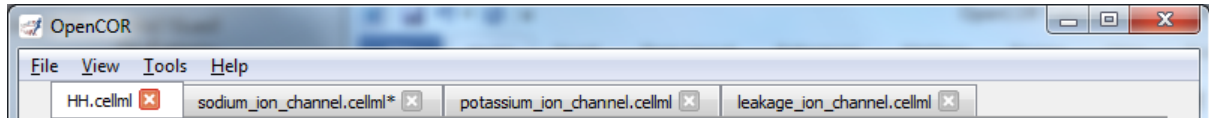
Fig. 10.7: The HH.cellml model and its three sub-models are available under separate tabs in OpenCOR.

ELEVEN

A MODEL OF THE CARDIAC ACTION POTENTIAL: IMPORTING UNITS AND PARAMETERS

We now examine the Noble 1962 model [D62] that applied the Hodgkin-Huxley approach to cardiac cells and thereby initiated the development of a long line of cardiac cell models that, in their human cell formulation, are now used clinically and are the most sophisticated models of any cell type. It was the incorporation of these models into whole heart bioengineering models that initiated the Physiome Project. We also illustrate the use of imported units and imported parameter sets.

Cardiac cells have similar gradients of potassium and sodium ions that operate in a similar way to neurons (as do all electrically active cells). There is one major difference, however, in the potassium channel that holds the cells in their resting state at -85mV (HH neuron) or -100mV (cardiac Purkinje cells). This difference is illustrated in Fig. 11.1(a). When the membrane potential is raised above the equilibrium potential for potassium, the cardiac channel conductance shown by the dashed line drops to nearly zero – i.e. it is an *inward rectifier* since it rectifies ('cuts off') the outward current that otherwise would have flowed through the channel at that potential. This is an evolutionary adaptation of the potassium channel to avoid loss of potassium ions out of the cell during the long plateau phase of the cardiac action potential (Fig. 11.1(b)) needed to give the heart time to contract. This evolutionary change saves the additional energy that would otherwise be needed to pump potassium ions back into the cell, but this Faustian "pact with the devil" is also the reason the heart is so susceptible to conduction failure (more on this later). To explain his data on Purkinje cells Noble [D62] postulated the existence of two inward rectifier potassium channels, one with a conductance g_{K1} that showed voltage dependence but no significant time dependence and another with conductance g_{K2} that showed less severe rectification with time dependent gating similar to the HH four-gated potassium channel.
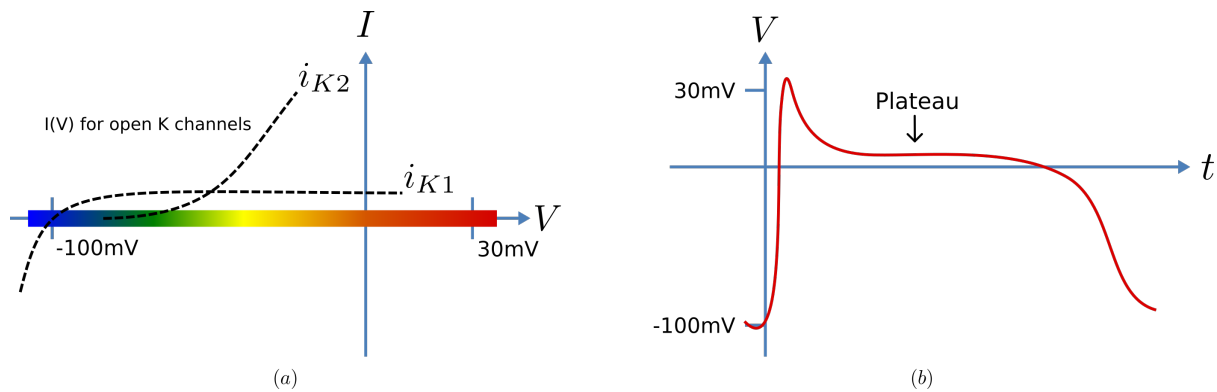

Fig. 11.1: Current-voltage relations (a) around the equilibrium potentials for the potassium and sodium channels in cardiac cells. The sodium channel is similar to the one in neurons but the two potassium channels have an inward rectifying property that stops leakage of potassium ions out of the cell when the membrane potential (illustrated in (b)) is high during the plateau phase of the cardiac action potential.

To model the cardiac action potential in Purkinje fibres (a cardiac cell specialised for rapid conduction from the atrio-ventricular node to the apical ventricular myocardial tissue), Noble [D62] proposed two potassium channels (one of these being the inwardly rectifying potassium channel described above and the other called the delayed potassium channel), one sodium channel (very similar to the HH neuronal sodium channel) and one leakage channel (also similar to the HH one).

The equations for these are as follows: (as for the HH model, time is in ms, voltages are in mV, concentrations are in mM, conductances are in mS, currents are in μA and capacitance is in μF).

Inward rectifying i_{K1} potassium channel (voltage dependent only)

$$i_{K1} = g_{K1}\left(V - E_K\right), \text{ with } E_K = \frac{RT}{zF} ln \frac{\left[K^+\right]_o}{\left[K^+\right]_i} = 25ln\frac{2.5}{140} = -100\text{mV}.$$

$$g_{K1} = 1.2e^{\frac{-(V+90)}{50}} + 0.015e^{\frac{(V+90)}{60}}$$

Inward rectifying i_{K2} potassium channel (voltage and time dependent)[1]

$$i_{K2} = g_{K2}\left(V - E_K\right)$$

$$g_{K2} = 1.2n^4$$

$$\frac{dn}{dt} = \alpha_n\left(1 - n\right) - \beta_n.n, \text{ where } \alpha_n = \frac{-0.0001\left(V + 50\right)}{e^{\frac{-(V+50)}{10}} - 1} \text{ and } \beta_n = 0.002e^{\frac{-(V+90)}{80}}.$$

Note that the rate constants here reflect a much slower onset of the time dependent change in conductance than in the HH potassium channel.

Sodium channel

$$i_{Na} = \left(g_{Na} + 140\right)\left(V - E_{Na}\right), \text{ with } E_{Na} = \frac{RT}{zF} ln \frac{\left[Na^+\right]_o}{\left[Na^+\right]_i} = 25ln\frac{140}{30} = 35\text{mV}.$$

$$g_{Na} = m^3 h.g_{Na\_max} \text{ where } g_{Na\_max} = 400\text{mS}.$$

$$\frac{dm}{dt} = \alpha_m\left(1 - m\right) - \beta_m.m, \text{ where } \alpha_m = \frac{-0.1\left(V + 48\right)}{e^{\frac{-(V+48)}{15}} - 1} \text{ and } \beta_m = \frac{0.12\left(V + 8\right)}{e^{\frac{(V+8)}{5}} - 1}$$

$$\frac{dh}{dt} = \alpha_h\left(1 - h\right) - \beta_h.h, \text{ where } \alpha_h = 0.17e^{\frac{-(V+90)}{20}} \text{ and } \beta_h = \frac{1}{1 + e^{\frac{-(V+42)}{10}}}$$

Leakage channel

$$i_{\text{leak}} = g_L\left(V - E_L\right), \text{ with } E_L = -60mV \text{ and } g_L = 0.075\text{mS}.$$

Membrane equation[2]

$$\frac{dV}{dt} = -\left(i_{Na} + i_{K1} + i_{K2} + i_{\text{leak}}\right)/C_m \text{ where } C_m = 12\mu\text{F}.$$

Fig. 11.2 shows the structure of the model, including separate files for units, parameters, and the three ion channels (the two potassium channels are lumped together). We include the Nernst equations dependence on potassium and sodium ion concentrations in order to demonstrate the use of parameter values, defined in a separate parameters file, that are read in at the top (whole cell model) level and passed down to the individual ion channel models.

---

[1] The second inwardly rectifying channel model was later replaced with two currents and , so that modern cardiac cell models do not include but they do include the inward rectifier (see later section).

[2] The Purkinje fibre membrane capacitance is 12 times higher than that found for squid axon. The use of μF ensures unit consistency with ms, mV and A since F is equivalent to C.V^{-1} or s.A.V^{-1} and therefore A/ F or A/(ms. A. mV^{-1}) on the RHS matches mV/ms on the LHS).
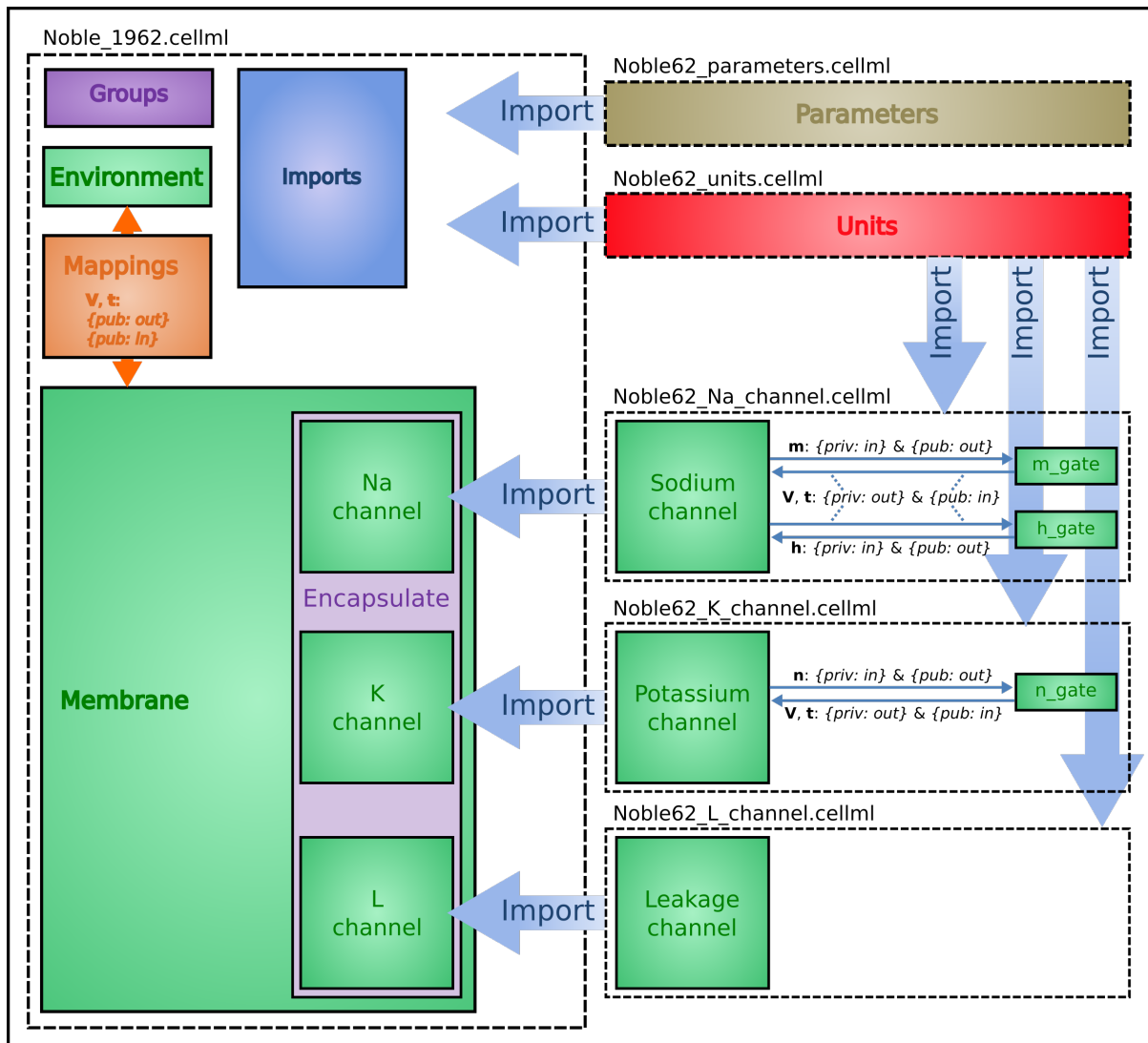
Fig. 11.2: Overall structure of the Noble62 CellML model showing the encapsulation hierarchy (purple), the CellML model imports (blue) and the other key parts (units, components & mappings) of the top level CellML model. Note that the overall structure of the Noble62 model differs from that of the earlier HH model in that all units are defined in a units file and imported where needed (shown by the import arrows). Also the ion concentration parameters are defined in a parameters file and imported into the top level file but passed down to the modules that use them via the mappings.

The CellML Text code for all six files is shown on the following two pages. The arrows indicate the imports (appropriately colour coded for units, components, and parameters).

Graphical outputs from solution of the Noble 1962 model with OpenCOR for 5000ms are shown in Fig. 11.2. Interpretation of the model outputs is given in the Fig. 11.2 legend. The Noble62 model was developed further by Noble and others to include additional sodium and potassium channels, calcium channels (needed for excitation-contraction coupling), chloride channels and various ion exchange mechanisms (Na/Ca, Na/H), co-transporters (Na/Cl, K/Cl) and energy (ATP)-dependent pumps (Na/K, Ca) needed to model the observed beat by beat changes in intracellular ion concentrations. These are discussed further in Section 15.

Note: The downloadable links below are links to the raw text file that may be used for copying and pasting into OpenCOR. For the underlying CellML file that is suitable for opening with OpenCOR from disk obtain the xml file.

Raw text: `Noble_1962.txt`, XML file: Noble_1962.cellml.

```
def model Noble_1962 as
   def import using "Noble62_Na_channel.xml" for
      comp Na_channel using comp sodium_channel;
   enddef;
   def import using "Noble62_K_channel.xml" for
      comp K_channel using comp potassium_channel;
   enddef;
   def import using "Noble62_L_channel.xml" for
      comp L_channel using comp leakage_channel;
   enddef;
   def import using "Noble62_units.xml" for
      unit mV using unit mV;
      unit ms using unit ms;
      unit nanoF using unit nanoF;
      unit nanoA using unit nanoA;
   enddef;
   def import using "Noble62_parameters.xml" for
      comp parameters using comp parameters;
   enddef;
   def map between parameters and membrane for
      vars Ki and Ki;
      vars Ko and Ko;
      vars Nai and Nai;
      vars Nao and Nao;
   enddef;
   def comp environment as
      var t: ms {init: 0, pub: out};
   enddef;
   def group as encapsulation for
      comp membrane incl
         comp Na_channel;
         comp K_channel;
         comp L_channel;
      endcomp;
   enddef;
   def comp membrane as
      var V: mV {init: -85, pub: out, priv: out};
      var t: ms {pub: in, priv: out};
      var Cm: nanoF {init: 12000};
      var Ki: mM {pub: in, priv: out};
      var Ko: mM {pub: in, priv: out};
      var Nai: mM {pub: in, priv: out};
      var Nao: mM {pub: in, priv: out};
      var i_Na: nanoA {pub: out, priv: in};
      var i_K: nanoA {pub: out, priv: in};
      var i_L: nanoA {pub: out, priv: in};
      ode(V, t) = -(i_Na+i_K+i_L)/Cm;
   enddef;
   def map between environment and membrane for
      vars t and t;
   enddef;
   def map between membrane and Na_channel for
      vars V and V;
      vars t and t;
      vars Nai and Nai;
      vars Nao and Nao;
      vars i_Na and i_Na;
   enddef;
   def map between membrane and K_channel for
      vars V and V;
      vars t and t;
      vars Ki and Ki;
```

```
      vars Ko and Ko;
      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;
enddef;
```

Raw text: `Noble62_units.txt`, XML file Noble62_units.cellml.

```
def model Noble62_units as
   def unit ms as
      unit second {pref: milli};
   enddef;
   def unit per_ms as
      unit second {pref: milli, expo: -1};
   enddef;
   def unit mV as
      unit volt {pref: milli};
   enddef;
   def unit mM as
      unit mole {pref: milli};
   enddef;
   def unit per_mV as
      unit volt {pref: milli, expo: -1};
   enddef;
   def unit per_mV_ms as
      unit mV {expo: -1};
      unit ms {expo: -1};
   enddef;
   def unit microS as
      unit siemens {pref: micro};
   enddef;
   def unit nanoF as
      unit farad {pref: nano};
   enddef;
   def unit nanoA as
      unit ampere {pref: nano};
   enddef;
enddef;
```

Raw text: `Noble62_parameters.txt`, XML file Noble62_parameters.cellml.

```
def model Noble62_parameters as
   def import using "Noble62_units.xml" for
      unit mM using unit mM;
   enddef;
   def comp parameters as
      var Ki: mM {init: 140, pub: out};
      var Ko: mM {init: 2.5, pub: out};
      var Nai: mM {init: 30, pub: out};
      var Nao: mM {init: 140, pub: out};
   enddef;
enddef;
```

Raw text: `Noble62_Na_channel.txt`, XML file Noble62_Na_channel.cellml.

```
def model sodium_ion_channel as
   def import using "Noble62_units.xml" for
      unit mV using unit mV;
```

```
      unit ms using unit ms;
      unit mM using unit mM;
      unit per_ms using unit per_ms;
      unit per_mV using unit per_mV;
      unit per_mV_ms using unit per_mV_ms;
      unit microS using unit microS;
      unit nanoA using unit nanoA;
   enddef;
   def group as encapsulation for
      comp sodium_channel incl
      comp sodium_channel_m_gate;
      comp sodium_channel_h_gate;
   endcomp;
   enddef;
   def comp sodium_channel as
      var V: mV {pub: in, priv: out};
      var t: ms {pub: in, priv: out};
      var g_Na_max: microS {init: 400000};
      var g_Na: microS;
      var E_Na: mV;
      var m: dimensionless {priv: in};
      var h: dimensionless {priv: in};
      var Nai: mM {pub: in};
      var Nao: mM {pub: in};
      var RTF: mV {init: 25};
      var i_Na: nanoA {pub: out};
      E_Na = RTF*ln(Nao/Nai);
      g_Na = pow(m, 3{dimensionless})*h*g_Na_max;
      i_Na = (g_Na+140{microS})*(V-E_Na);
   enddef;
   def comp sodium_channel_m_gate as
      var V: mV {pub: in};
      var t: ms {pub: in};
      var m: dimensionless {init: 0.01, pub: out};
      var alpha_m: per_ms;
      var beta_m: per_ms;
      alpha_m = -0.10{per_mV_ms}*(V+48{mV})
        /(exp(-(V+48{mV})/15{mV})-1{dimensionless});
      beta_m = 0.12{per_mV_ms}*(V+8{mV})
        /(exp((V+8{mV})/5{mV})-1{dimensionless});
      ode(m, t)=alpha_m*(1{dimensionless}-m)-beta_m*m;
   enddef;
   def comp sodium_channel_h_gate as
      var V: mV {pub: in};
      var t: ms {pub: in};
      var h: dimensionless {init: 0.8, pub: out};
      var alpha_h: per_ms;
      var beta_h: per_ms;
      alpha_h = 0.17{per_ms}*exp(-(V+90{mV})/20{mV});
      beta_h = 1.00{per_ms}
        /(1{dimensionless}+exp(-(V+42{mV})/10{mV}));
      ode(h, t) = alpha_h*(1{dimensionless}-h)-beta_h*h;
   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;
enddef;
```

Raw text: `Noble62_K_channel.txt`, XML file Noble62_K_channel.cellml.

```
def model potassium_ion_channel as
   def import using "Noble62_units.xml" for
      unit mV using unit mV;
      unit ms using unit ms;
      unit mM using unit mM;
      unit per_ms using unit per_ms;
      unit per_mV using unit per_mV;
      unit per_mV_ms using unit per_mV_ms;
      unit microS using unit microS;
      unit nanoA using unit nanoA;
   enddef;
   def group as encapsulation for
      comp potassium_channel incl
         comp potassium_channel_n_gate;
      endcomp;
   enddef;
   def comp potassium_channel as
      var V: mV {pub: in, priv: out};
      var t: ms {pub: in, priv: out};
      var n: dimensionless {priv: in};
      var Ki: mM {pub: in};
      var Ko: mM {pub: in};
      var RTF: mV {init: 25};
      var E_K: mV;
      var g_K1: microS;
      var g_K2: microS;
      var i_K: nanoA {pub: out};
      E_K = RTF*ln(Ko/Ki);
      g_K1 = 1200{microS}*exp(-(V+90{mV})/50{mV})
        +15{microS}*exp((V+90{mV})/60{mV});
      g_K2 = 1200{microS}*pow(n, 4{dimensionless});
      i_K = (g_K1+g_K2)*(V-E_K);
   enddef;
   def comp potassium_channel_n_gate as
      var V: mV {pub: in};
      var t: ms {pub: in};
      var n: dimensionless {init: 0.01, pub: out};
      var alpha_n: per_ms;
      var beta_n: per_ms;
      alpha_n = -0.0001{per_mV_ms}*(V+50{mV})
        /(exp(-(V+50{mV})/10{mV})-1{dimensionless});
      beta_n = 0.0020{per_ms}*exp(-(V+90{mV})/80{mV});
      ode(n,t)= alpha_n*(1{dimensionless}-n)-beta_n*n;
   enddef;
   def map between environment
     and potassium_channel 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;
```

Raw text: `Noble62_L_channel.txt`, XML file [Noble62_L_channel.cellml](#).

```
def model leakage_ion_channel as
    def import using "Noble62_units.xml" for
        unit mV using unit mV;
        unit ms using unit ms;
        unit microS using unit microS;
        unit nanoA using unit nanoA;
    enddef;
    def comp leakage_channel as
        var V: mV {pub: in};
        var g_L: microS {init: 75};
        var E_L: mV {init: -60};
        var i_L: nanoA {pub: out};
        i_L = g_L*(V-E_L);
    enddef;
enddef;
```
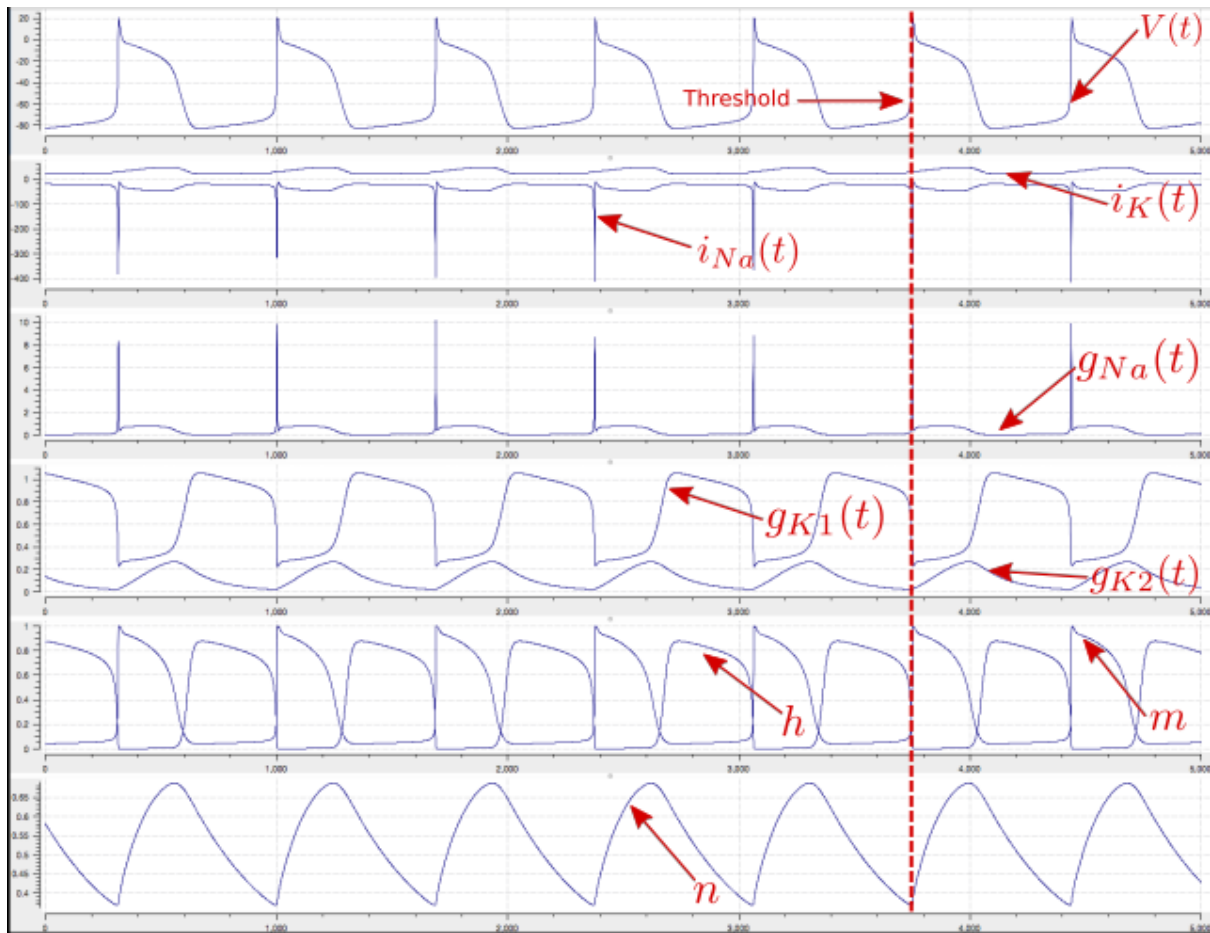
Fig. 11.3: Output from the Noble62 model (OpenCOR link). Top panel is $V(t)$, the cardiac action potential. The next panel has the two membrane ion channel currents $i_{Na}(t)$ and $i_K(t)$. Note that $i_{Na}(t)$ has a very brief downward (i.e. inward current) spike that is triggered when the membrane voltage reaches about -70mV. This is caused by the huge increase in sodium channel conductance $g_{Na}(t)$ shown in the panel below associated with the simultaneous opening of the *m*-gate and closing of the *h*-gate (5th panel down). The resting state of about -80mV in the top panel is set by the potassium equilibrium (Nernst) potential via the open potassium channels. As can be seen from the 4th and bottom panels, it is the closing of the time-dependent potassium *n*-gate and the corresponding decline of potassium conductance that, with a small background leakage current $i_L(t)$, leads to the membrane potential rising from -80mV to the threshold for activation of the sodium channel (note the dotted red line showing the point when *n(t)* reaches a minimum). Later cardiac cell models include additional ion channels that directly affect the heart rate by controlling this rise.

We have now covered all existing features of CellML and OpenCOR. But, most importantly, you have learned 'best practice' for building CellML models, including encapsulation of sub-components and a modular approach in which units, parameters and model components are defined in separate files that are imported into a composite model.

CODE GENERATION

It is sometimes required to export CellML models to various procedural formats to make use of a given model with existing tools. OpenCOR currently uses the CellML Language Export Definition Service provided by the CellML API to achieve this (see this article for details). This service takes an XML file containing a conversion definition and uses that to export a CellML model to the defined format.

The OpenCOR distribution packages include definition files for C, Fortran 77, Python, and Matlab. These definition files are available in the formats folder of your OpenCOR installation or can be downloaded and used directly using the previous links.

The C and Fortran code generated using these definition files contain functions suitable for inclusion in DAE/ODE simulation codes. Whereas the Python and Matlab code generated are complete scripts that use standard Python or Matlab methods to actually perform an *default* simulation. The default simulation is probably not what is needed, so the generated code can be modified or reused to meet the specific usage requirements.

12.1 Exporting CellML to code

The steps to generate code from OpenCOR are given below.

1. Load the desired CellML model into OpenCOR (both CellML 1.0 and 1.1 models can be used)

2. From the OpenCOR menu, choose *Tools → CellML File Export To → User-Defined Format*.

3. The first file selection dialog is to provide the conversion definition file (as above).

4. The second file selection dialog is to provide the file to save the generated code to.

This conversion can also be performed using OpenCOR as a command line client. In this case the command is:

```
$ ./OpenCOR -c CellMLTools::export myfile.cellml myformat.xml
```

or for a remote model:

```
$ ./OpenCOR -c CellMLTools::export http://mydomain.com/myfile.cellml myformat.xml
```

where myformat.xml can be one of the standard definition files described above.

12.2 Generated code in PMR

The Physiome Model Repository uses the same code generation service from the CellML API to generate code in the above formats for all exposures containing CellML models. These are available from the *Generated Code* view for CellML models. See here for an example.

MODEL ANNOTATION

One of the most powerful features of CellML is its ability to import models. This means that complex models can be built up by combining previously defined models. There is a potential problem with this process, however, since the imported models (often developed by completely different modellers) may represent the same biological or biophysical entity with different expressions. The potassium channel model in *A model of the potassium channel: Introducing CellML components and connections*, for example, represents the intracellular concentration of potassium as 'Ki' (see the *CellML Text* code *Potassium_ion_channel.cellml*) but another model involving the intracellular potassium concentration may use a different expression.

The solution to this dilemma is to annotate the CellML variables with names from controlled vocabularies that have been agreed upon by the relevant scientific community. In this case we may simply want to annotate Ki as '*the concentration of potassium in the cytosol*'. This expression, however, refers to three distinct entities: *concentration*, *potassium* and *cytosol*. We might also want to specify that we are referring to the cytosol of a neuron . . . and that the neuron comes from a particular part of a giant squid (the experimental animal used by Hodgkin and Huxley). Annotations can clearly get very complicated!

What comes to our rescue here is that most scientific communities have developed controlled vocabularies together with the relationships between the terms of that vocabulary – called **ontologies**. Furthermore relationships can always be expressed in the form subject-predicate-object. E.g. Ki is-the-concentration-of potassium is one relationship and potassium in-the cytosol is another. Each object can become the subject of another expression. We could continue, for example, with cytosol of-the neuron, neuron of-the squid and so on. The terms s-the-concentration-of, in-the and of-the are the predicates and these semantically rich expressions too have to come from controlled vocabularies. Each of these subject-predicate-object expressions is called an RDF **triple** and the World Wide Web consortium[1] has established a framework called the *Resource Description Framework* (RDF[2]) to support these.

CellML models therefore contain two parts, one dealing with **syntax** (the MathML definition of the models together with the structure of components, connections, groups, units, etc) as discussed in previous sections, and one dealing with **semantics** (the meanings of the terms used in the models) discussed in this section[3]. This latter is also referred to as *metadata* – i.e. data about data.

In the CellML metadata specification[4] the first RDF *subject* of a triple is a CellML element (e.g. a variable such as 'Ki'), the RDF *predicate* is chosen from the Biomodels Biological Qualifiers[5] list, and the RDF *object* is a URI (the string of characters used to identify the name of a resource[6]). Establishing these RDF links to biological and biophysical meaning is the goal of annotation.

Note the different types of subject/object used in the RDF triples: *the concentration* is a biophysical entity, *potassium* is a chemical entity, *the cytosol* is an anatomical entity. In fact, to cover all the terminology used in the models, CellML uses five separate ontologies:

- ChEBI (Chemical Entities of Biological Interest) www.ebi.ac.uk/chebi

- GO (Gene Ontology) www.geneontology.org

- FMA (Foundation Model of Anatomy) fma.biostr.washington.edu/projects/fm/

---

[1] Referred to as W3C – see www.w3.org

[2] www.w3.org/RDF

[3] For details on the annotation plugin see http://opencor.ws/user/plugins/editing/CellMLAnnotationView.html

[4] See http://www.cellml.org/specifications/metadata/ and http://www.cellml.org/specifications/metadata/mcdraft

[5] http://co.mbine.org/standards/qualifiers

[6] http://en.wikipedia.org/wiki/Uniform_resource_identifier

- Cell type ontology code.google.com/p/cell-ontology

- OPB sbp.bhi.washington.edu/projects/the-ontology-of-physics-for-biology-opb

These ontologies are available through OpenCOR's annotation facilities as explained below.

If we now go back to the potassium ion channel CellML model and, under *Editing*, click on *CellML Annotation*, the various elements of the model (Units, Components, Variables, Groups and Connections) are displayed (see Fig. 13.1). If you right click on any of them a popup menu will appear, which you can use to expand/collapse all the child nodes, as well as remove the metadata associated with the current CellML element or the whole CellML file. Expanding *Components* lists all the components and their variables. To annotate the potassium channel component, select it and specify a *Qualifier* from the list displayed:

Fig. 13.1: Clicking on *CellML Annotation* lists the CellML components with their variables ready for annotation.

```
bio:encodes,          bio:isPropertyOf
bio:hasPart,          bio:isVersionOf
bio:hasProperty,      bio:occursIn
bio:hasVersion,       bio:hasTaxon
bio:is,               model:is
bio:isDescribedBy,    model:isDerivedFrom
bio:isEncodedBy,      model:isDescribedBy
bio:isHomologTo,      model:isInstanceOf
bio:isPartOf,         model:hasInstance
```

If you do not know which qualifier to use, click on the ⬤ button to get some information about the current qualifier (you must be connected to the internet) and go through the list of qualifiers until you find the one that best suits your needs. Here, we will say that you want to use bio:isVersionOf. Fig. 13.2 shows the information displayed about this qualifier.

Fig. 13.2: The qualifiers are displayed from the top right menu. Clicking on the most appropriate one (bio:isVersionOf) gives more information about this qualifier in the bottom panel.

Now you need to retrieve some possible ontological terms to describe the *potassium_channel* component. For this you must enter a search term, which in our case is 'potassium channel' (note that regular expressions are supported[7]). This returns 24 possible ontological terms as shown in Fig. 13.3. The *voltage-gated potassium channel complex* is the most appropriate. Clicking on the GO identifier link shown provides more information about this term (see Fig. 13.4).

---

[7] http://en.wikipedia.org/wiki/Regular_expression
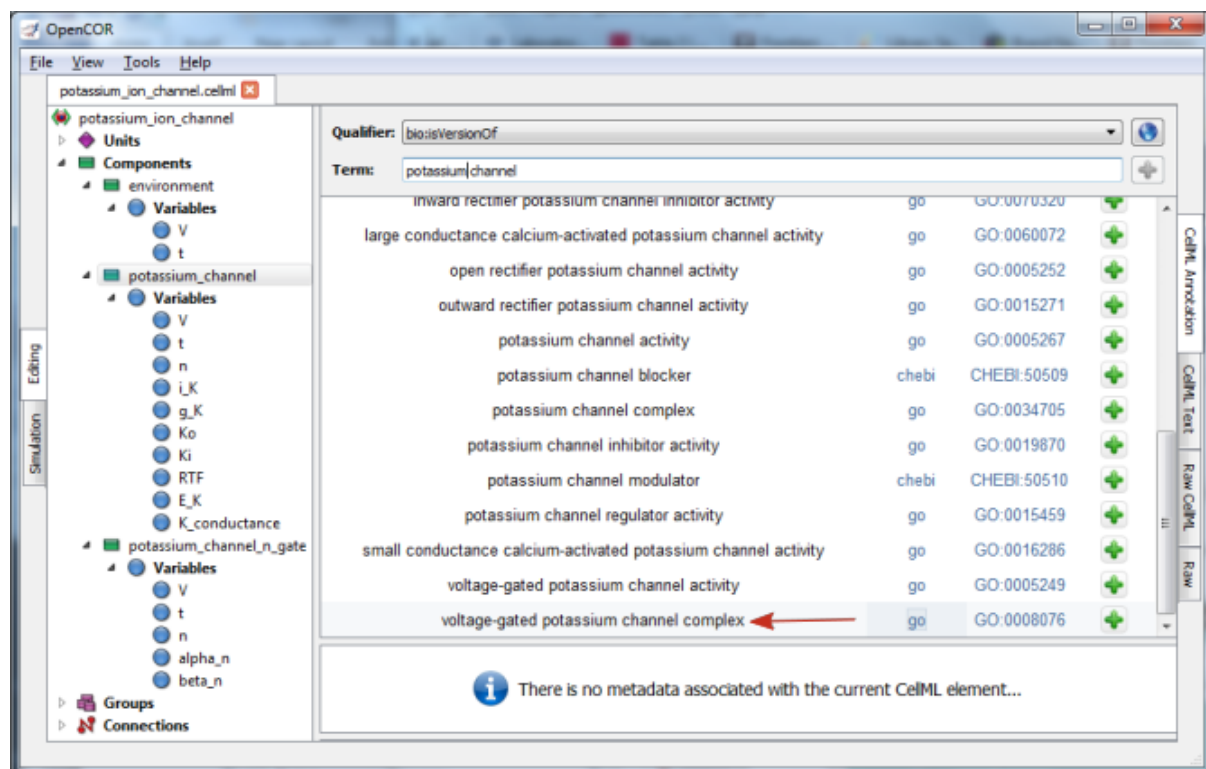
Fig. 13.3: The ontological terms listed when 'potassium channel' is entered into the search box next to *Term*.
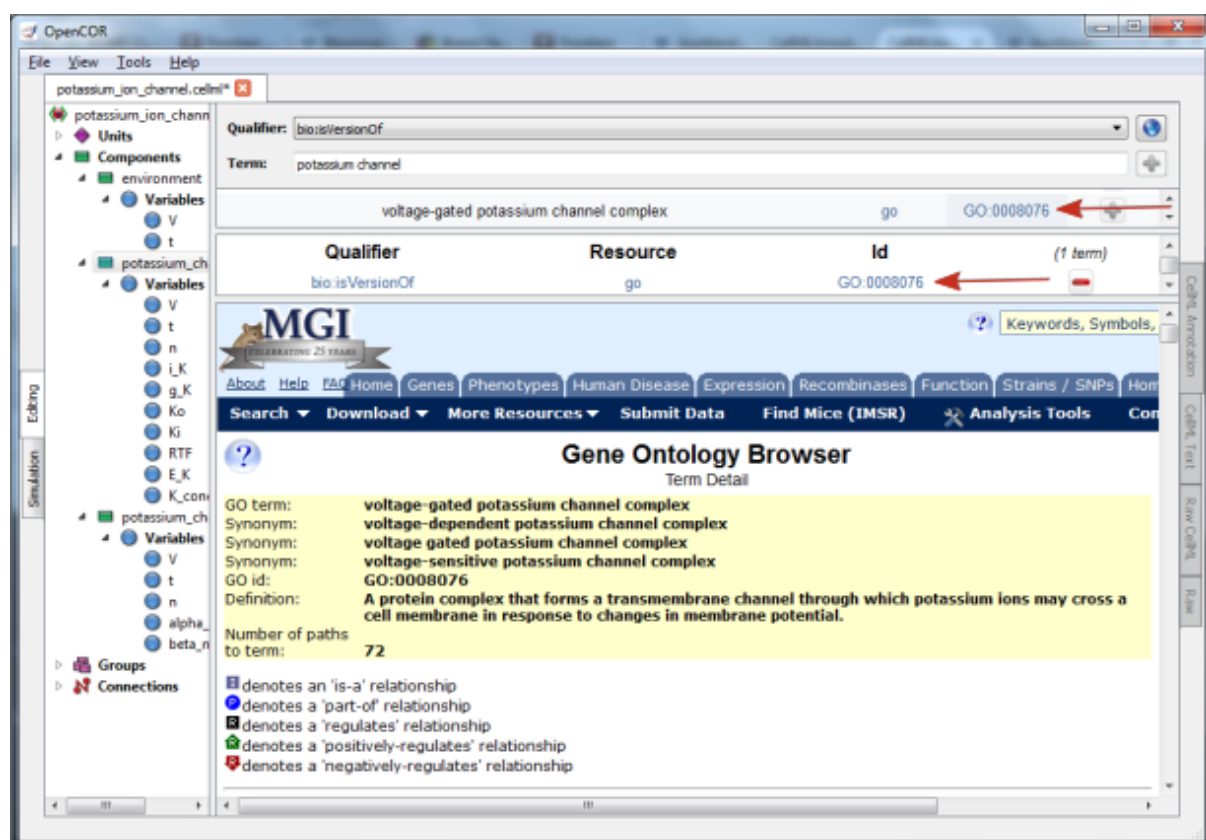
Fig. 13.4: The qualifier, resource & ID information in the middle panel appears when you click on the ✚ button next to the selected term in Fig.32. GO identifier details are listed when either of the arrowed links are clicked.

Now, assuming that you are happy with your choice of ontological term, you can associate it with the *potassium_channel* component by clicking on its corresponding ➕ button which then displays the qualifier, resource and ID information in the middle panel as shown in Fig. 13.3. If you make a mistake, this can be removed by clicking on the ➖ button.

The first level annotation of the *potassium_channel* component has now been achieved. The content of the three terms in the RDF triple are shown in Fig. 13.5, along with the annotation for the variables *Ki* and *Ko*.
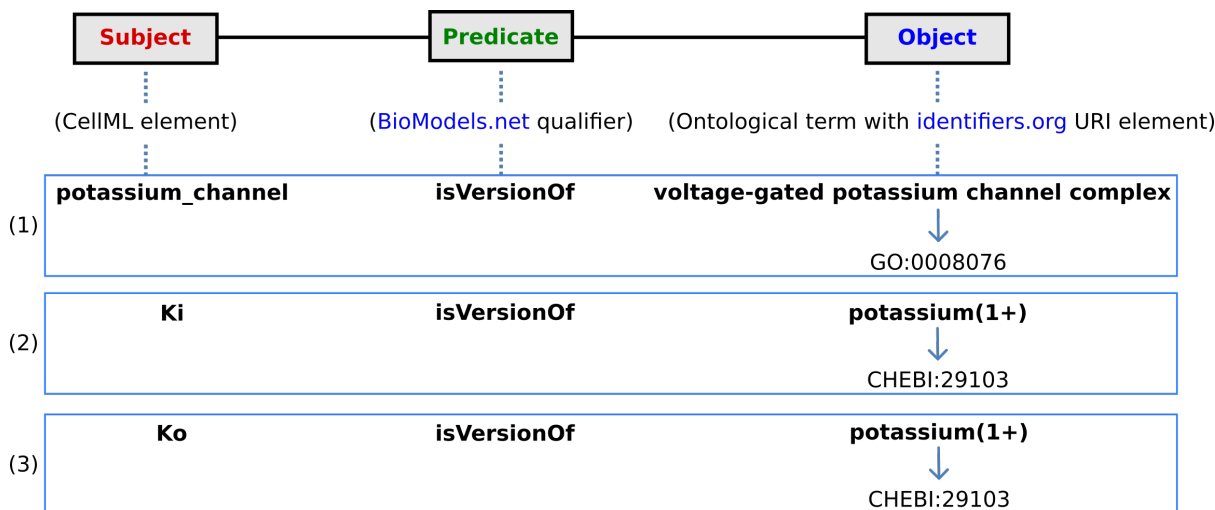
Fig. 13.5: The RDF triple used in CellML metadata to link a CellML element (component or variable) with an ontological term from one of the five ontologies accessed via identifiers.org, using a predicate qualifier from BioModels.net. The three examples of annotated CellML model elements shown are for (1) the *potassium_channel* component (this points to a GO identifier), (2) the variable *Ki*, and (3) the variable *Ko*. These two variables are defined within the *potassium_channel* component of the model and point to CHEBI identifiers. A further annotation is needed to identify the cellular location of those variables (since one is intracellular and one is extracellular).

```
def comp {id_000000001} 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 {id_000000002} Ki: mM {init: 90};
   var {id_000000003} Ko: mM {init: 3};
   var RTF: millivolt {init: 25};
   var E_K: millivolt;
   var K_conductance: milliS_per_cm2 {pub: out};

   E_K = RTF*ln(Ko/Ki);
   K_conductance = g_K*pow(n, 4{dimensionless});
   i_K = K_conductance*(V-E_K);
enddef;
```

When saved (the *CellML Annotation* tag will appear un-grayed), the result of these annotations is to add metadata to the CellML file. If you switch to the *CellML Text* view you will see that the elements that have been annotated appear with ID numbers, as shown above. These point to the corresponding metadata contained in the CellML file for this model and are displayed under the qualifier-resource-Id headings in the annotation window when you click on the element in the editing window.

Note that the three annotations added above are all biological annotations. Many of the other components and variables in the CellML potassium channel model deal with biophysical entities and these require the use of the OPB ontology (yet to be implemented in OpenCOR). The use of composite annotations is also being developed[8], such

---

[8] This is a project being carried out at the University of Washington, Seattle, using an annotation tool called SEMGEN (...).

as "Ki is-the concentration of potassium in-the cytosol of-the neuron of-the giant-squid", where *concentration*, *potassium*, *cytosol*, *neuron* and *giant-squid* are defined by the ontologies OPB, ChEBI, GO, FMA and a species ontology, respectively.

THE PHYSIOME MODEL REPOSITORY AND THE LINK TO BIOINFORMATICS

The Physiome Model Repository (PMR) [LCPF08] is the main online repository for the IUPS Physiome Project, providing version and access controlled repositories, called *workspaces*, for users to store their data. Currently there are over 700 public workspaces and many private workspaces in the repository. PMR also provides a mechanism to create persistent access to specific revisions of a workspace, termed *exposures*. Exposure plugins are available for specific types of data (e.g. CellML or FieldML documents) which enable customizable views of the data when browsing the repository via a web browser, or an application accessing the repository's content via web services.

More complete documentation describing how to use PMR is available in the PMR documentation: https://models. physiomeproject.org/docs.

The CellML models on models.physiomeproject.org are listed under 20 categories, shown below: (numbers of exposures in each category are given besides the bar graph, correct as at early 2016)

Browse by category

Category	Count
Calcium Dynamics	140
Cardiovascular Circulation	60
Cell Cycle	38
Cell Migration	2
Circadian Rhythms	22
Electrophysiology	230
Endocrine	60
Excitation-Contraction Coupling	22
Gene Regulation	12
Hepatology	29
Immunology	55
Ion transport	13
Mechanical Constitutive Laws	19
Metabolism	86
Myofilament Mechanics	22
Neurobiology	33
pH regulation	2
PKPD	11
Signal Transduction	120
Synthetic Biology	6

Note that searching of models can be done anywhere on the site using the search box on the upper right hand corner. An important benefit of ensuring that the models on the PMR are annotated is that models can then be retrieved by a web-search using any of the annotated terms in the models.

To illustrate the features of PMR, click on the Hund, Rudy 2004 (Basic) model in the alphabetic listing of models under the *Electrophysiology* category.

Fig. 14.1: The Physiome Model Repository exposure page for the basic Hund-Rudy 2004 model.

The section labelled 'Model Structure' contains the journal paper abstract and often a diagram of the model[1]. This is shown for the Hund-Rudy 2004 model in Fig. 14.2. This model, with over 22 separate protein model components, is also a good example of why it is important to build models from modular components [CMEJ08], and in particular the individual ion channels for electrophysiology models.

---

[1] These are currently hand drawn SVG diagrams but the plan is to automatically generate them from the model annotation and also (at some stage!) to animate them as the model is executed.
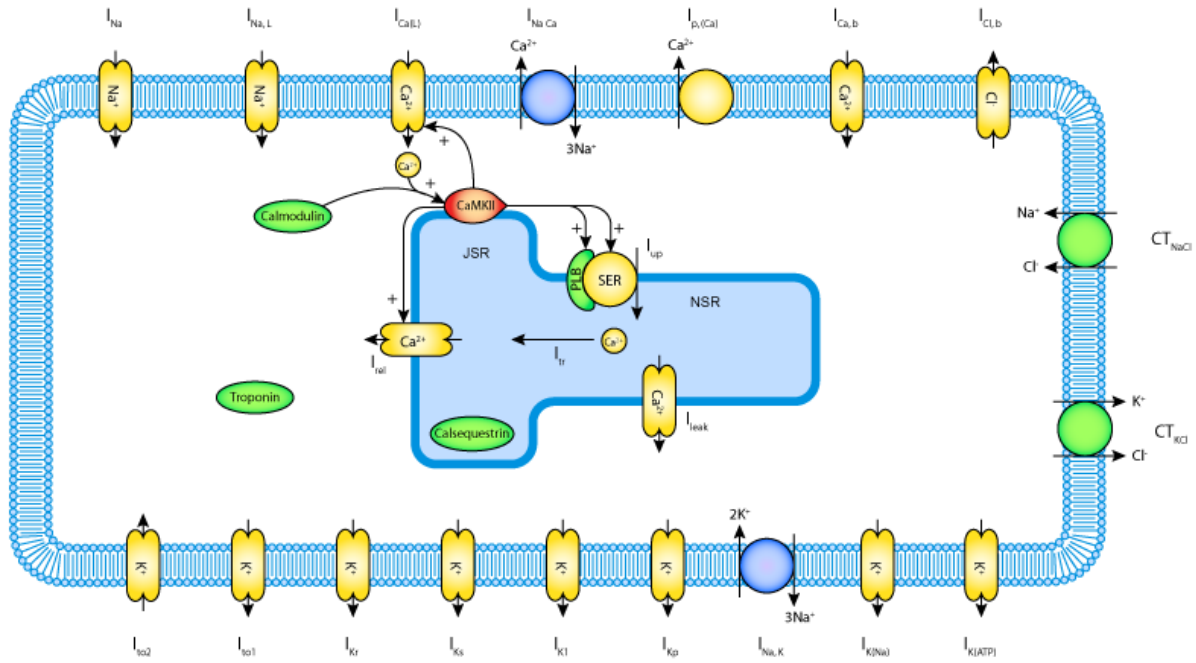
Fig. 14.2: A diagrammatic representation of the Hund-Rudy 2004 model.

There is a list of 'Views Available' for the CellML model on the right hand side of the exposure page. The function of each of these views is as follows:

Views Available

Documentation - Takes you to the main exposure page.

Model Metadata - Lists metadata including authors, title, journal, Pubmed ID and model annotations.

Model Curation - Provides the curation status of the model. Note: this is soon to be updated.

Mathematics - Displays all the mathematical equations contained in the model.

Generated Code - Various codes (C, C-IDA, F77, MATLAB or Python) generated from the model.

Cite this model - Provides details on how to cite use of the CellML model.

Source view - Gives a full listing of the XML code for the model.

Launch with OpenCOR - Opens the model (or simulation experiment) in OpenCOR.

Note that CellML models are available under a Creative Commons Attribution 3.0 Unported License[2]. This means that you are free to:

- Share — copy and redistribute the material in any medium or format
- Adapt — remix, transform, and build upon the material

for any purpose, including commercial use.

The next stage of content development for PMR is to provide a list of the modular components of these models each with their own exposure. For example, models for each of the individual ion channels used in the publication-based electrophysiological models will be available as standalone models that can then be imported as appropriate into a new composite model. Similarly for enzymes in metabolic pathways and signalling complexes in signalling pathways, etc. Some examples of these protein modules are:

Sodium/hydrogen exchanger 3 https://models.physiomeproject.org/e/236/

Thiazide-sensitive Na-Cl cotransporter https://models.physiomeproject.org/e/231/

---

[2] https://creativecommons.org/licenses/by/3.0/

Sodium/glucose cotransporter 1 https://models.physiomeproject.org/e/232/

Sodium/glucose cotransporter 2 https://models.physiomeproject.org/e/233/

Note that in each case, as well as the CellML-encoded mathematical model, links are provided (see Fig. 14.3) to the UniProt Knowledgebase for that protein, and to the Foundational Model of Anatomy (FMA) ontology (via the EMBLE-EBI Ontology Lookup Service) for information about tissue regions relevant to the expression of that protein (e.g. *Proximal convoluted tubule*, *Apical plasma membrane*; *Epithelial cell of proximal tubule*; *Proximal straight tubule*). Similar facilities are available for SMBL-encoded biochemical reaction models through the Biomodels database [AYY].

Fig. 14.3: The PMR workspace for the Thiazide-sensitive Na-Cl cotransporter. Bioinformatic data for this model is accessed via the links under the headings highlight by the arrows and include **Protein** (labelled A) and the model **Location** (labelled B). Other information is as already described for the Hund-Rudy 2004 model.

USING PMR WITH OPENCOR

In addition to the *PMR* window for browsing public exposures directly in OpenCOR (*PMR window*) OpenCOR has the ability for users to directly create and access their workspaces in PMR.

Note: It is a feature of PMR that all data is persistent and permanent. As such, any workspaces created on the main instance of PMR (https://models.physiomeproject.org/) can not be deleted. For the purposes of teaching, we have an alternate instance of PMR (https://teaching.physiomeproject.org/) which is periodically cleared out and synschronised from the main instance. Using the teaching instance allows you to play around without the worry of things being permanent.

1. Register for a user account on the teaching instance of PMR.

In order to make use of the teaching instance of PMR, you must first have an account for that instance of the repository. For teaching purposes it is best to register a new account. This can be done by first opening this link in your browser: https://teaching.physiomeproject.org. Then click the *Log in* button on (shown in Fig. 15.1) then the *registration form* link.

Fig. 15.1: The log in button for the teaching instance of PMR.

After filling in the names and email fields and clicking *Register* you will receive an email inviting you to confirm and set a password. Once that is completed you can then log in. Clicking on *My Workspaces* will take you to a listing of all your workspaces and provides access to the the *Workspace creation form*.

2. Create a new workspace, in this example the title 'Test workspace' has been used.

15.1 The PMR Workspaces window

A window labelled *PMR workspaces* is available in OpenCOR (see Fig. 15.2). If it is not currently visible it can be selected via *View → Windows → PMR workspaces* (or perhaps the Ctrl-space shortcut).
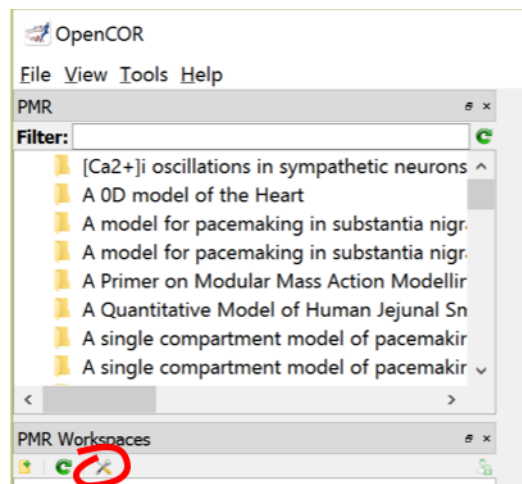
Fig. 15.2: PMR workspace shown on the left hand panel in OpenCOR. The preferences button is highlighted.

3. Set preferences.

Clicking the preferences button (Fig. 15.2) presents a *Preferences* dialog box with three settings: PMR instance, Name and Email. For the current purpose choose *https://teaching.physiomeproject.org* for the first and enter your name and email. These are used to identify you as the author of changes you submit back to the repository (view an example history).

4. Log into PMR from OpenCOR.

Before you can view private information or submit changes to PMR you must first log in to PMR from OpenCOR and grant OpenCOR permission to use your account. You accomplish this by clicking on the top right button in the *PMR Workspaces* window and then logging in with your new user name and password (created in step 1). Then grant access for OpenCOR to gain access to your PMR workspaces. The PMR workspaces window will then show all your workspaces, which should currently consist of the new workspace created in step 2. Note that using the same top right button you can log off - and when you next authenticate you will again be asked to grant access but this time without needing to login with your password.

Right clicking on the workspace name brings up a list of options for that workspace, the first being to view the workspace within PMR (in the web browser). Another option allows you to make a local copy of a workspace on your local disk - this will create a copy of the workspace on your local computer in which you are able to make changes.

5. Make a local copy of your *test workspace*

Using the *Make Local Workspace Copy. . .* option from the right-click menu on the workspace you created in step 2, clone the workspace to your PC. When doing this you will need to provide the folder in which you want to store the workspace contents - make sure you remember where this folder is!

6. Save a CellML model to your workspace.

A CellML file opened in OpenCOR (choose any model you have access to) can be saved (*File → Save As. . .*) to the folder you created for the cloned workspace. Once you have saved a model you will see the file appear under the workspace's folder in the *PMR Workspaces* window. Note that the file appears under the workspace with a red patch on the logo indicating the the file is not yet flagged to upload. To upload the file to PMR, you need to choose *Synchronise Workspace With PMR. . .* from the right-click menu on the workspace folder. This will ask you to provide a description of the change you would like to submit to PMR, and display all the differences you will be synchronising. When you now click the *OK* button, the changes will actually be submitted to PMR and you will see the file appear on the refreshed browser window. The file icon in the PMR Workspaces window will be shown without the red or green patch. Fig. 15.3 shows two CellML files that have been uploaded to PMR.
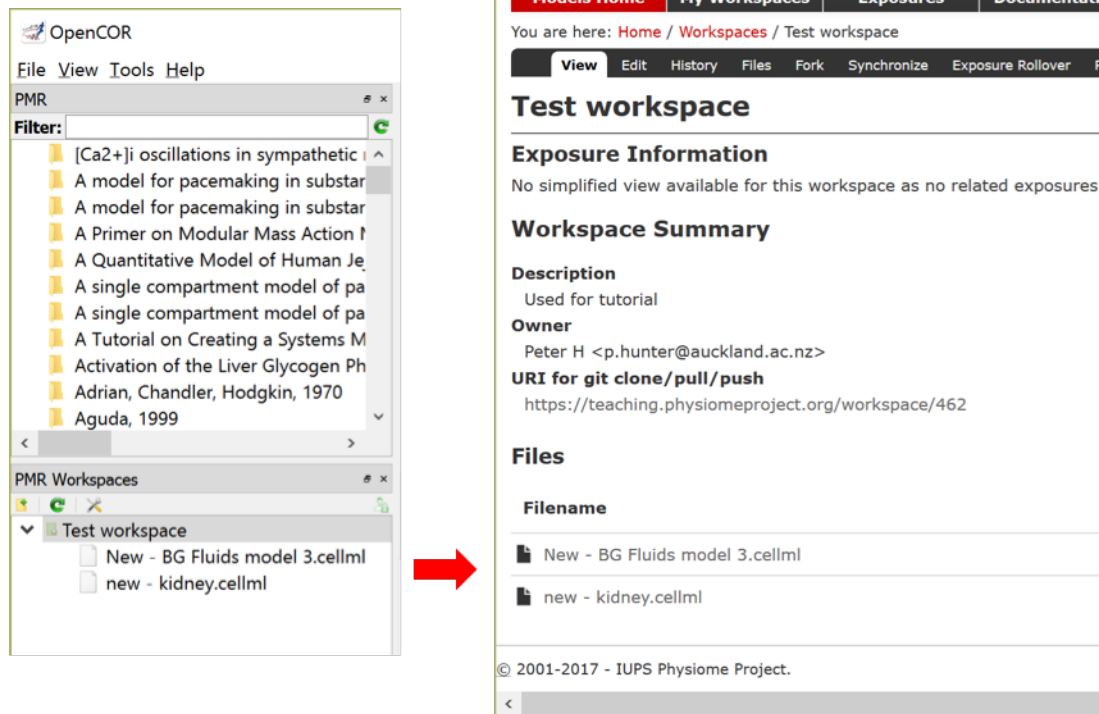
Fig. 15.3: Two CellML files (`New - BG Fluids model 3.cellml` and `new - kidney.cellml`) have been uploaded from OpenCOR to PMR and can be seen in the PMR workspace on the browser window on the right.

SED-ML, FUNCTIONAL CURATION AND WEB LAB

In the same way that CellML models can be defined unambiguously, and shared easily, in a machine-readable format, there is a need to do the same thing with 'protocols' - i.e. to define what you have to do to replicate/simulate an experiment, and to analyse the results. An XML standard for this called SED-ML[1] is being developed by the COMBINE community and preliminary support for SED-ML has been implemented in OpenCOR in order to allow precise and reproducible control over the OpenCOR simulation and graphical output (e.g., see Fig. 11.3).

The recent versions of OpenCOR (since early 2016) support exporting the *Simulation view* configuration to a SED-ML file, which can then be read back into OpenCOR to reproduce a given simulation experiment, illustrated in Fig. 16.1.
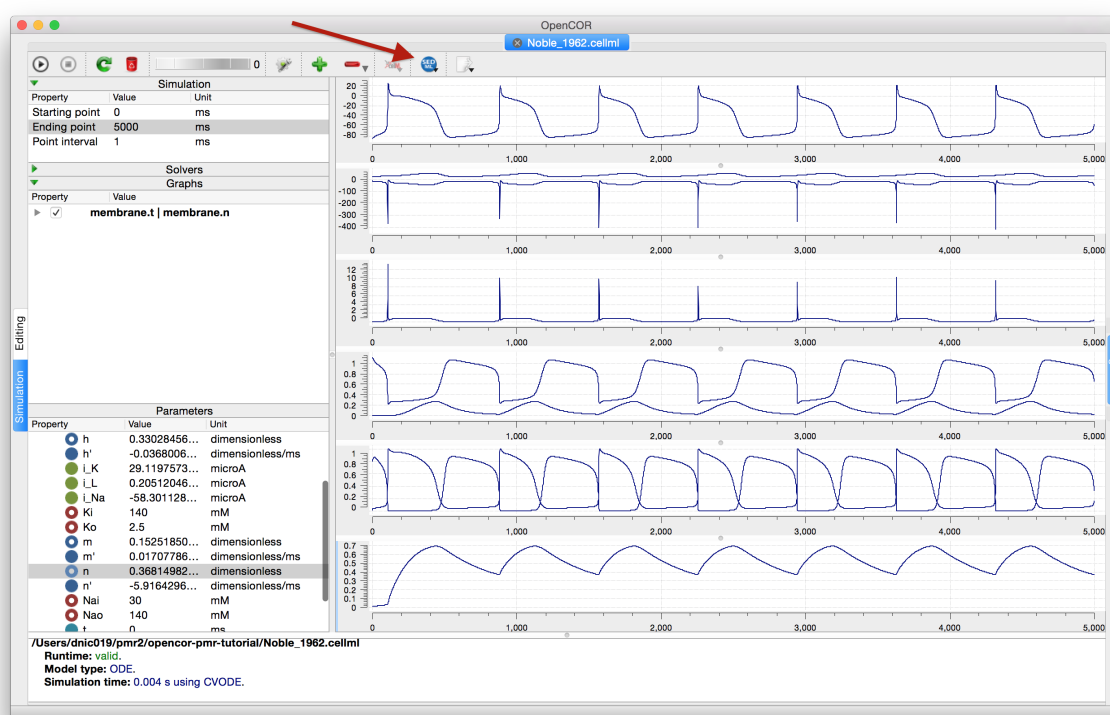
Fig. 16.1: Once you are happy with the configuration of the *Simulation view* in OpenCOR, clicking the SED-ML button (highlighted) will prompt for a file to save the SED-ML document to. This document can be loaded back into OpenCOR to reproduce the simulation, or shared with collaborators so they can reproduce the simulation.

Support for SED-ML will also facilitate the curation of models according to their functional behaviour under a range of experimental scenarios. The key idea behind functional curation is that, when mathematical and computational models are being developed, a primary goal should be the continuous comparison of those models against experimental data. When computational models are being re-used in new studies, it is similarly important to

---

[1] The 'Simulation Experiment Description Markup Language': sed-ml.org

check that they behave appropriately in the new situation to which you're applying them. To achieve this goal, a pre-requisite is to be able to replicate in-silico precisely the same protocols used in an experiment of interest. A language for describing rich 'virtual experiment' protocols and software for running these on compatible models is being developed in the Computational Biology Group at Oxford University[2]. An online system called Web Lab[3] is also being developed that supports definition of experimental protocols for cardiac electrophysiology, and allows any CellML model to be tested under these protocols [CJ15]. This enables comparison of the behaviours of cellular models under different experimental protocols: both to characterise a model's behaviour, and comparing hypotheses by seeing how different models react under the same protocol (Fig. 16.2 adapted from [CJ15]).
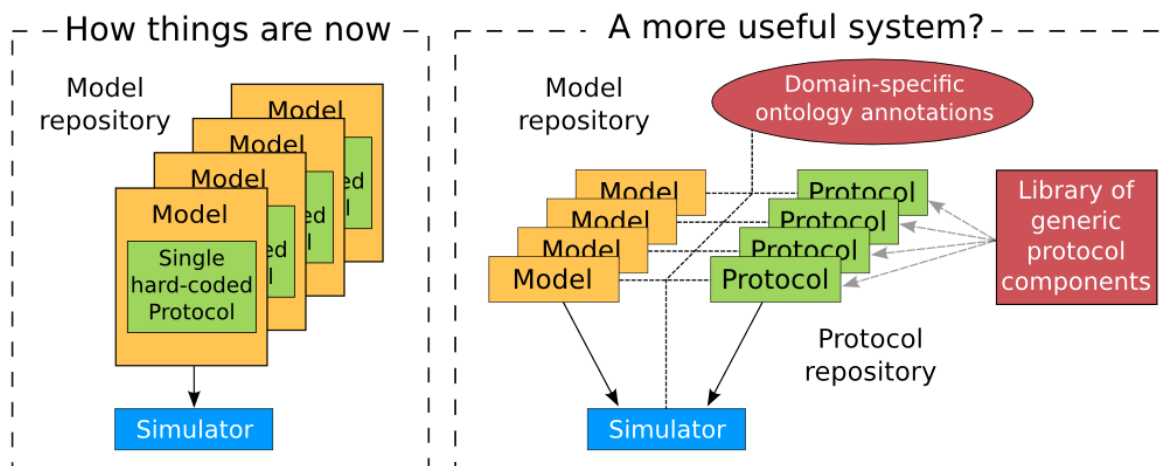
Fig. 16.2: A schematic of the way we organise model and protocol descriptions. Web Lab provides an interface to a Model/Protocol Simulator, storing and displaying the results for cardiac electrophysiology models.

The Web Lab website provides tools for comparing how two different cardiac electrophysiology models behave under the same experimental protocols. Note that Web Lab demonstration for CellML models of cardiac electrophysiology is a prototype for a more general approach to defining simulation protocols for all CellML models.

---

[2] travis.cs.ox.ac.uk/FunctionalCuration/about.html This initiative is led by Jonathan Cooper and Gary Mirams.
[3] travis.cs.ox.ac.uk/FunctionalCuration.

USING OPENCOR WITH PYTHON (BETA)

CellML provides a good technology to create, describe, and share definitions of mathematical models. *SED-ML* similarly provides a good technology to share reproducible descriptions of simulation experiments. Whenever possible, it is best to make use of these standard formats to ensure the models and simulations are Findable, Accessible, Interoperable, and Reusable.

Often in research projects, however, it is not always possible to describe the model and/or simulation that you need to perform in these declarative formats. It also doesn't make sense to try and standardise extensions or modifications in such standards for potentially short-lived, one-off, research studies. Thus having access to a flexible scripting environment that works in concert with a standards-based tool like OpenCOR allows users to make use of standards when possible but with the flexibility to adapt as needed. OpenCOR supports this through the integration of a Python interpreter within the OpenCOR application.

Python-enabled versions of OpenCOR are now relatively mature, but still undergoing extensive user testing and implementation review. As such, this functionality is only available in special snapshot releases of OpenCOR available from: https://github.com/dbrnz/opencor/releases. In this part of the tutorial we are going to be using the *20 September 2019* snapshot of the Python-enabled OpenCOR. This particular release is distributed with the following Python packages and their dependencies: numpy, scipy, and matplotlib.

Contents

17.1 Installation and setup

Python-enabled OpenCOR release can be installed as per the standard *installation instructions*. As this is an early release of the new functionality, it is best to use one of the compressed archive releases which you can extract locally rather than overwriting the stable system install. Once you have a Python-enabled release of OpenCOR your main OpenCOR window should look similar to that shown in Fig. 17.1.
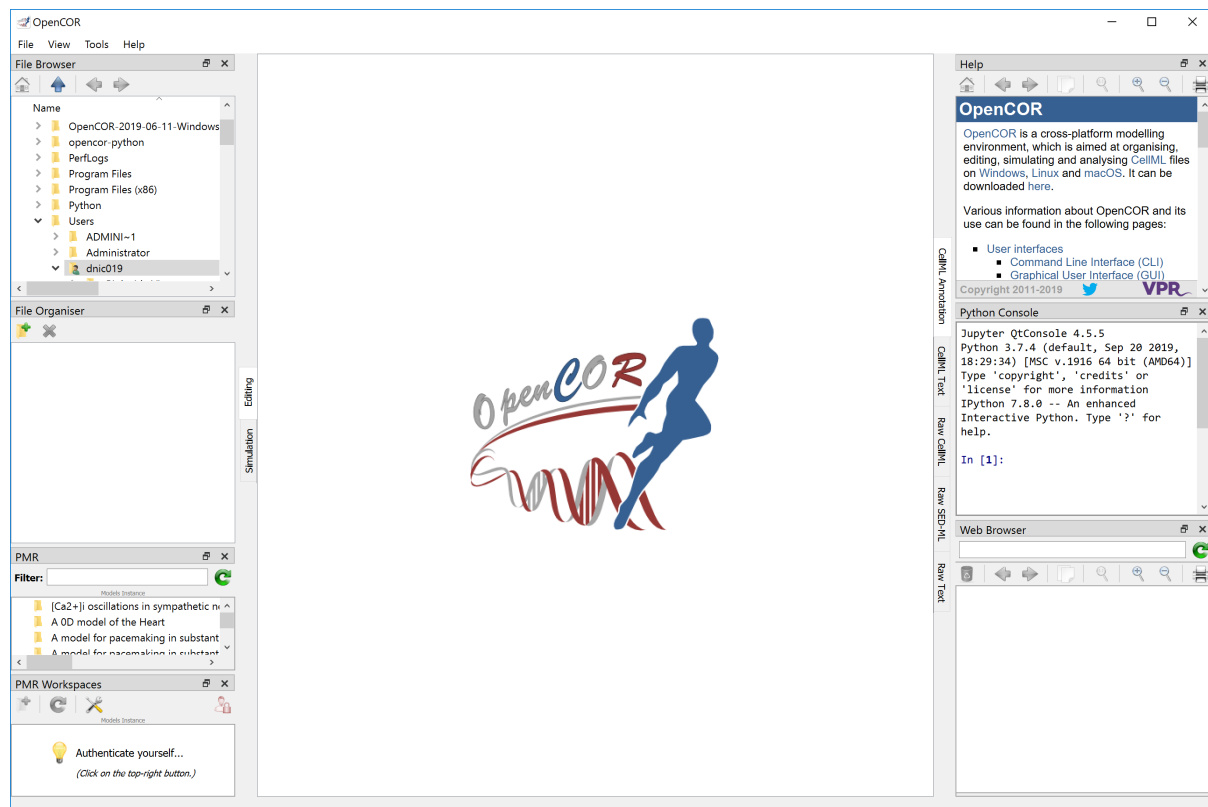
Fig. 17.1: OpenCOR application with default positioning of dockable windows including the Python console (right-side, middle). As described in *Install and Launch OpenCOR* the dockable windows can be rearranged as desired to suit your preferred layout.

17.1.1 Command line usage

In Python-enabled versions of OpenCOR the Python interpreter is embedded within the OpenCOR application. Which means that in order to access the OpenCOR functionality you must use that Python within the OpenCOR application rather than, for example, importing OpenCOR into your system Python. The Python console available in the OpenCOR graphical user interface handles this for you allowing a seamless user experience. However, often with Python-scripted simulation workflows it is nice to have the ability to run in a headless or batch mode. As such, Python-enabled versions of OpenCOR come with some command line scripts to help provide the user avoid the issues of making sure their Python scripts run using the correct Python interpreter.

In the top-level folder of your Python-enabled OpenCOR installation there is a script named `run_python` which will depend on your operating system - on Windows for example, it is called `run_python.bat`. Running this script without providing a Python script to execute will give you a standard Python console using the Python embedded inside the OpenCOR application:

```
C:\Users\andre\OpenCOR-2019-09-20-Windows> run_python.bat
Python 3.7.4 (default, Sep 20 2019, 18:29:34) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Providing a Python script will result in that script being interpreted by the interpreter embedded in the OpenCOR application:

```
C:\Users\andre\OpenCOR-2019-09-20-Windows> run_python.bat hello_world.py
Hello World!
```

Command line arguments can be provided as usual following the script to be executed.

> **Warning:** Due to the use of a Python interpreter embedded in a graphical user interface, there can be some weirdness when trying to use UI toolkits from the command line, for example using `matplotlib`. This works within the OpenCOR graphical user interface, but will fail when running from the command line. Hence, it is best to currently use the command line version when working in a truly headless manner without the need for a graphical user interface.

17.1.2 Jupyter notebooks

There is another mode to make use of the Python-enabled versions of OpenCOR and that is to access this functionality via Jupyter notebooks. This is enabled via the `run_jupyter` helper script.

Todo: Write this section on Jupyter notebooks and OpenCOR.

17.1.3 Installing packages

As described above, the Python interpreter lives inside the OpenCOR application – making it difficult to access in order to install packages or modules that are not distributed with the Python-enabled versions of OpenCOR. To install packages using `pip` combined with the interactive Python console in the OpenCOR graphical user interface is the way to go here, as shown below.

```
Jupyter QtConsole 4.5.5
Python 3.7.4 (default, Sep 20 2019, 18:29:34) [MSC v.1916 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.8.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: !pip install [options] package
```

17.2 Basic usage

The scope and capabilities of the Python interface to OpenCOR is still being refined, but here we focus on use of the capabilities relevant to performing simulation experiments. Here we walk through the basic usage of using Python to interact with OpenCOR in performing a simulation experiment.

As with any Python script, we must first import the OpenCOR module to expose the functionality that we desire.

```python
import OpenCOR as oc
```

The main object that we are interested in dealing with is OpenCOR's representation of a simulation. OpenCOR is able to generate a default simulation for a CellML model or to load a SED-ML document which defines the simulation experiment in detail. As the exposed simulation features are not yet complete, it is best to load a SED-ML document giving full control over the simulation settings. The output plots defined in the SED-ML will also be used when running the Python code via the interactive Python console in OpenCOR, but will be disregarded when running via the command line mode.

```python
# for a local file
simulation = oc.openSimulation('path/to/cellml/or/sedml')

# OR for loading a remote file, e.g., from the model repository:
simulation = oc.openRemoteSimulation('URL/of/cellml/or/sedml')

# OR if using the OpenCOR GUI and models are already loaded
simulation = oc.simulation()    # The model in the currently active tab
```

For a given simulation, the `data` object houses all the relevant information and pointers to the OpenCOR internal data representations.

```
data = simulation.data()
```

And the data object allows us to define the interval of interest for this simulation experiment.

```
data.setStartingPoint(start)
data.setEndingPoint(end)
data.setPointInterval(pointInterval)
```

As in the OpenCOR graphical user interface, constant parameters and initial values for the state variables can also be set via the Python interface OpenCOR provides. When address specific variables in a model, they are mapped to Python dictionaries using key's comprising of `component_name/variable_name`. This provides a method to uniquely identify all variables in a model.

```python
# Set constant parameter values
data.constants()['key'] = value

# Set initial value for state variables
data.states()['key'] = value
```

Once you have the simulation defined that you would like to perform, it can be executed with the following.

```
simulation.run()
```

If you are using the OpenCOR graphical user interface and have define plots for the current simulation experiment, then these will be displayed as usual during the execution of the simulation. The simulation results can also be used directly in the Python script as shown below.

```python
# Access simulation results
results = simulation.results()

# grab a specific state variable results
r1 = results.states()['key'].values()  # Numpy array

# grab a specific algebraic variable results
r2 = results.algebraic()['key'].values()  # Numpy array

# access the full datastore representation of the simulation results
ds = results.dataStore()
# the dictionary or all result variables in the simulation
variables = ds.voiAndVariables()

# grab a the results for a given variable
r3 = variables['key'].values()   # Python list of values
```

When continuing a simulation from an existing state, the default behaviour is to continue from the current state. The system can be reset to the initial state as shown below. As with using the OpenCOR graphical user interface, this includes resetting any parameters or initial values that you may have set via the GUI or the Python interface.

```python
# Reset things if needed when re-running
simulation.resetParameters()
# clear any existing results
simulation.clearResults()
```

17.2.1 Interactive example

In this example, we use the *simple ODE model* introduced earlier in the tutorial. We will be using the Python console in the OpenCOR graphical user interface, working with the SED-ML loaded directly from the Physiome

Model Repository. As we are using the OpenCOR application, you should see the user interface updating in response the to various Python commands. The following commands should be copy-pasted one at a time into the Python console to observe the behaviour.

```python
import OpenCOR as oc

simulation = oc.openRemoteSimulation('https://models.physiomeproject.org/workspace/
↪25d/rawfile/60ac9389285471a704f2f4be6e1a8ba5cbf45d1a/Firstorder.sedml')
data = simulation.data()
data.setStartingPoint(0)
data.setEndingPoint(10)
data.setPointInterval(0.1)
simulation.run()

# reset
simulation.resetParameters()
simulation.clearResults()

# change parameter values
data.constants()['main/b'] = 5
data.states()['main/y'] = 2
simulation.run()

# look at the simulation results
results = simulation.results()
y = results.states()['main/y'].values()   # Numpy array
print(y)

ds = results.dataStore()
variables = ds.voiAndVariables()
y = variables['main/y'].values()    # Python list of values
print(y)

a = variables['main/a'].values()
print(a)
```

In working through this example, you should be able to reproduce the results as seen in Fig. 5.2.

17.3 OpenCOR, CellML, and TensorFlow

TensorFlow is a popular end-to-end open source machine learning platform in Python. Together with the Python-enabled OpenCOR capabilities and CellML itself, this opens up a new world of application of machine learning in computational physiology. This is a very new application that we are still actively developing, but here we give a brief demonstration that might help show what could be achieved.

17.3.1 Getting prepared

The first step is to ensure that you have TensorFlow installed. As described above, Python packages need to be installed in the Python embedded inside OpenCOR. We are using here TensorFlow version 1.15, which can be installed using the OpenCOR Python console with the following command. (TensorFlow 2.0 will not work with this demonstration.)

```
In [1]: !pip install tensorflow==1.15
```

We have prepared a couple of Python scripts that you can use for this demonstration. The first is `MPL.py`, which is a TensorFlow-based script to construct a simple MLP (fully-connected feed-forward network or MultiLayer Perceptron) and trains it with a given dataset. The second is `train-tf-model.py`, which will first generate a set of training data using the O'Hara & Rudy cardiac electrophysiology model, which has been encoded in the

CellML format as an extension of this model in the Physiome Model Repository. Both files should be downloaded into the same folder on your local machine.

Finally, in the OpenCOR Python console we need to make sure the plotting happens in-place rather than trying to bring windows. This is done by exectuing the following command in the OpenCOR Python console.

```
In [1]: %matplotlib inline
```

17.3.2 Training a machine learning model

The `train-tf-model.py` script is the one that contains the definition of the workflow we are demonstrating here. It is easiest to open this file in your preferred Python editor and follow through the script, with the comments attempting to explain what is happening.

This script can be run in the OpenCOR Python console by first making sure the console is looking at the correct folder,

```
In [1]: %cd path/to/folder/with/downloaded/scripts
```

and then running the training script as follows.

```
In [1]: %run train-tf-model.py
```

All going well, this should result in something similar to Fig. 17.2.
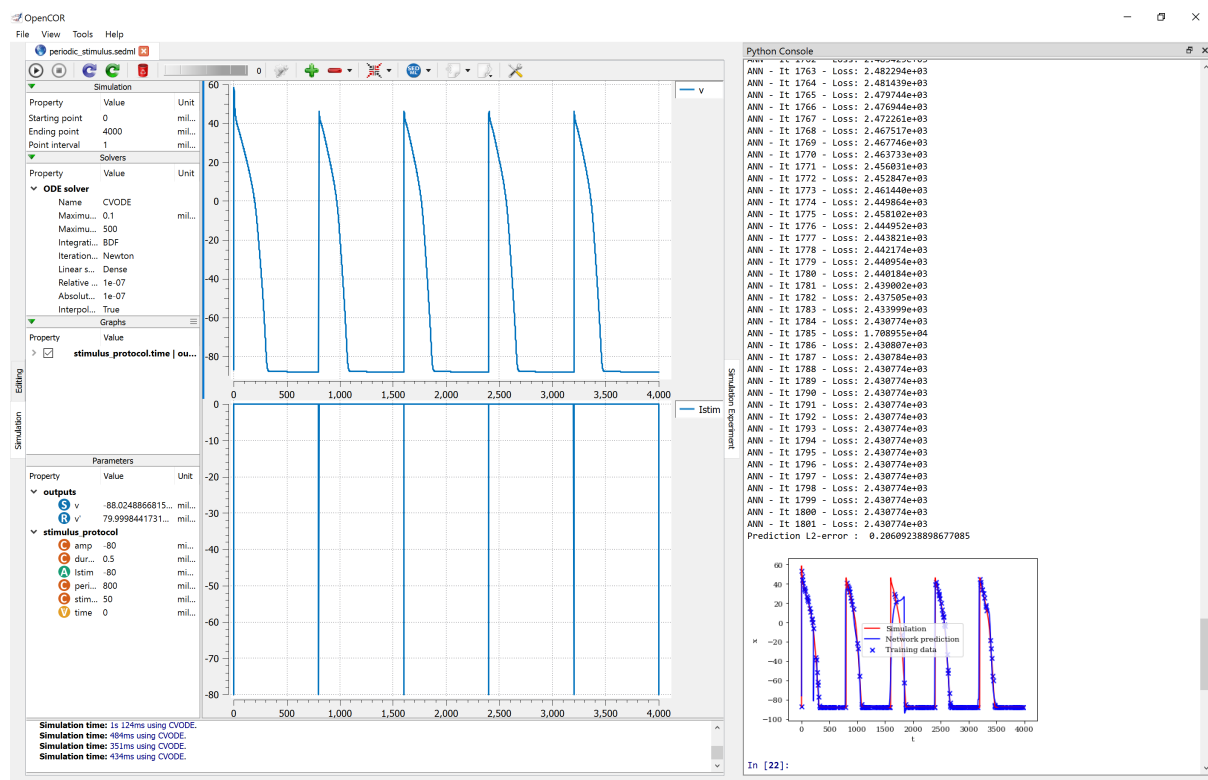

Fig. 17.2: The result of training a TensorFlow machine learning model using data from a simulation of a CellML model in OpenCOR and then comparing the ML-model predictions to the actual simulation results.

You should now be able to play around with the training script to see what happens as you change, for example, the stimulation period or simulation duration.

SPEED COMPARISONS WITH MATLAB

Solution speed is important for complex computational models and here we compare the performance of Open-COR with MATLAB[1]. Nine representative CellML models were chosen from the PMR model repository. For the MATLAB tests we used the MATLAB code, generated automatically from CellML, that is available on the PMR site. These comparisons are based on using the default solvers (listed below) available in the two packages.

18.1 Testing environment

- MacBook Pro (Retina, Mid 2012).

- Processor: 2.6 GHz Intel Core i7.

- Memory: 16 GB 1600 MHz DDR3.

- Operating system: OS X Yosemite 10.10.3.

18.2 OpenCOR

- Version: 0.4.1.

- Solver: CVODE with its default settings, except for its Maximum step parameter, which is set to the model's stimulation duration, if needed.

18.3 MATLAB

- Version: R2013a.

- Solver: ode15s (i.e. a solver suitable for stiff problems and which has low to medium order of accuracy) with both its RelTol and AbsTol parameters set to 1e-7 and its MaxStep parameter set to the stimulation duration, if needed.

18.4 Testing protocol

- Run a model for a given simulation duration.

- Generate simulation data every milliseconds.

- Only keep track of all the simulation data (i.e. no graphical output).

- Run a model 7 times, discard the 2 slowest runs (to account for unpredictable slowdowns of the testing machine) and average the resulting computational times.

---

[1] www.mathworks.com/products/matlab

- Computational times are obtained directly from OpenCOR and MATLAB (through a couple of calls to cputime in the case of MATLAB).

18.5 Results

CellML model (from PMR on 18/6/2015)	Dura-tion (s)	OpenCOR time (s)	MATLAB time (s)	Time ratio (MAT-LAB/OpenCOR)
Bondarenko et al. 2004	10	1.16	140.14	121
Courtemanche et al. 1998	100	0.998	45.720	46
Faber & Rudy 2000	50	0.717	29.010	40
Garny et al. 2003	100	0.996	48.180	48
Luo & Rudy 1991	200	0.666	70.070	105
Noble 1962	1000	1.42	310.02	218
Noble et al. 1998	100	0.834	42.010	50
Nygren et al. 1998	100	0.824	31.370	38
ten Tusscher & Panfilov 2006	100	0.969	59.080	61

[*]The value of membrane.stim_end was increased so as to get action potentials for the duration of the simulation

18.6 Conclusions

For this range of tests, OpenCOR is between 38 and 218 times faster than MATLAB. A more extensive evaluation of these results is available on GitHub[2].

---

[2] https://github.com/opencor/speedcomparison. These tests were carried out by Alan Garny.

REFERENCES

Todo:

- Colour background of *CellML Text*

- Annotate screen shots with svg for same look and feel

- *CellML Text* code is not highlighted for all display situations, currently only in environments that are using an adapted version of pygments

- Tidy up citations and BiBTeX source (possibly use Zotero to manage?)

- Make horizontal line for footnotes only visible in html output

- Check external references markup

- Consider a more suitable theme (may require changes to an existing one to get a good result)

- Must check over output (and models) from screenshots to make sure that it matches the current release of OpenCOR, especially against running experiments for the first time.

BIBLIOGRAPHY

[APJ15]   Garny A. and Hunter P.J. Opencor: a modular and interoperable approach to computational biology. *Frontiers in Physiology*, 2015.

[AYY]   Non A. Www.biomodels.org <http://www.biomodels.org>. YYYY.

[AAF52]   Hodgkin AL and Huxley AF. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117:500–544, 1952.

[CPJ09]   Christie R. Nielsen P.M.F. Blackett S. Bradley C. and Hunter P.J. Fieldml: concepts and implementation. *Philosophical Transactions of the Royal Society (London)*, A367(1895):1869–1884, 2009.

[CMEJ08]   Hunter PJ Cooling M and Crampin EJ. Modeling biological modularity with cellml. *IET Systems Biology*, 2:73–79, 2008.

[CJ15]   Waltemath D. Cooper J, Vik JO. A call for virtual experiments: accelerating the scientific process. *Progress in Biophysics and Molecular Biolog*, 117:99–106, 2015.

[D62]   Noble D. A modification of the hodgkin-huxley equations applicable to purkinje fibre action and pacemaker potentials. *Journal of Physiology*, 160:317–352, 1962.

[DPPJ03]   Cuellar A.A. Lloyd C.M. Nielsen P.F. Halstead M.D.B. Bullivant D.P. Nickerson D.P. and Hunter P.J. An overview of cellml 1.1, a biological model description language. *SIMULATION: Transactions of the Society for Modeling and Simulation*, 79(12):740–747, 2003.

[ea13]   Hunter P.J. et al. A vision and strategy for the virtual physiological human: 2012 update. *Interface Focus*, 2013.

[eal11]   Yu T. et al. The physiome model repository 2. *Bioinformatics*, 27:743–744, 2011.

[J97]   Wigglesworth J. Energy and life. *Taylor & Francis Ltd*, 1997.

[JH02]   Thompson JMT and Stewart HB. Nonlinear dynamics and chaos. *Wiley*, 2002.

[LCPF08]   Hunter PJ Lloyd CM, Lawson JR and Nielsen PF. The cellml model repository. *Bioinformatics*, 24:2122–2123, 2008.

[P13]   Britten R.D. Christie G.R. Little C. Miller A.K. Bradley C. Wu A. Yu T. Hunter P.J. Nielsen P. Fieldml, a proposed open standard for the physiome project for mathematical model representation. *Med. Biol. Eng. Comput.*, 51(11):1191–1207, 2013.

[PJ04]   Hunter P.J. The iups physiome project: a framework for computational physiology. *Progress in Biophysics and Molecular Biology*, 85:551–569, 2004.

[VarYY]   Various. See www.cellml.org/about/publications for a more extensive list of publications on cellml and opencor. *Various*, YYYY.