VPH 2014 ABI Software Tutorial

Release 0.1-@VPH2014

Auckland Bioengineering Institute

Sep 18, 2017

Contents

This documentation has been prepared for the session, *"VPH tools from the Auckland Bioengineering Institute"*, presented at the VPH 2014 meeting.

This tutorial will demonstrate some of the tools, techniques and best practices developed at the Auckland Bioenginerring Institute that aid scientists in the development and application of computational models and simulation experiments in their work toward the creation of a virtual physiological human. The *Auckland Physiome Repository* provides a framework for the storage, curation and exchange of data. By using standards suitable to their data, scientists maximise their ability to reuse existing knowledge and enable others to make use of their achievements in novel work. Annotations ensure scientists are able to find existing data and are also able to correctly interpret and apply their own data. These tutorials are designed to help demonstrate and promote practices which will aid attendees in their own work. Attendees are encouraged to raise issues specifically related to their needs with the tutors.

Documentation for the software used in this tutorial is available online, including the most recent version of the tutorial itself. This tutorial guides the particpant through various common computational modelling scenarios faced by scientists working toward the virtual physiologial human. We use these scenarios to achieve scientific outputs using the covered tools and demonstrating practices we believe will help ensure reproducible and reusable science.

When interacting directly with *Mercurial*, this tutorial demonstrates how to work with the repository using TortoiseHg, which provides a Windows explorer integrated system for working with Mercurial repositories.

Note: Brief mention of the equivalent command line versions of the TortoiseHg actions will also be mentioned, so that these ideas can also be used without a graphical client, and on Linux or OS X and similar systems. These will be denoted by boxes like this.

This tutorial requires you to have:

- A Mercurial client such as TortoiseHg or Mercurial installed;
- The *OpenCOR* CellML modelling environment and/or the *MAP* workflow tool installed; and
- Possibly a text editor such as Notepad++ or gedit.

The tutorial makes use of two primary tools, OpenCOR and MAP Client, as well as the model repository. For convenience, documentation for each of these projects has been collated here, corresponding to the versions of the tools used in the tutorial.

Contents:

CHAPTER 1

VPH 2014 - ABI Software Tutorial

This tutorial consists of two independent components that can be worked through in any order. One is the OpenCOR tutorial, which focuses on working with CellML models. The other is the MAP Client workflow tool focussing on moving data through a series of processing steps. Both of these are able to make use of the Auckland Physiome Repository to locate, archive, and share data.

Note: The teaching instance of the repository is a mirror of the main repository site found at http://models.physiomeproject.org/, running the latest development version of *PMR2*.

Any changes you make to the contents of the teaching instance are not permanent, and will be overwritten with the contents of the main repository whenever the teaching instance is upgraded to a new release of PMR2. For this reason, you can feel free to experiment and make mistakes when pushing to the teaching instance. Please subscribe to the cellml-discussion mailing list to receive notifications of when the teaching instance will be refreshed.

See the section *Migrating content to the main repository* for instructions on how to migrate any content from the teaching instance to the main (permanent) Auckland Physiome Repository.

Tutorial outline

OpenCOR

In this part of the tutorial, we take you through a few common scenarios that a modeller might want encounter in their daily work, showing how to complete the required tasks using *OpenCOR* and the *model repository*. In the *Auckland Physiome Repository*, a complete piece of work is stored in a *workspace*. Each workspace is a *Mercurial* repository, which allows the repository to maintain a complete history of all changes made to every file it contains. In this part of the tutorial, we take you through the creation of a new piece of work, which will be stored in a *workspace*. Useful information on working with the repository using Mercurial is available in the *repository documentation*.

A new CellML-based piece of work

In this section we are going to create a new *workspace* into which we will add a CellML model, annotate the model using *OpenCOR*, and simulate the model to check that it produces the expected results. We will be using

the seminal Noble (1962) cardiac cellular electrophysiology model as the demonstration model for this part of the tutorial.

Create a new workspace

You can find instructions for creating a new workspace on the teaching instance repository in the *repository workspaces* documentation. Following those instructions, create a workspace similar to that shown below:

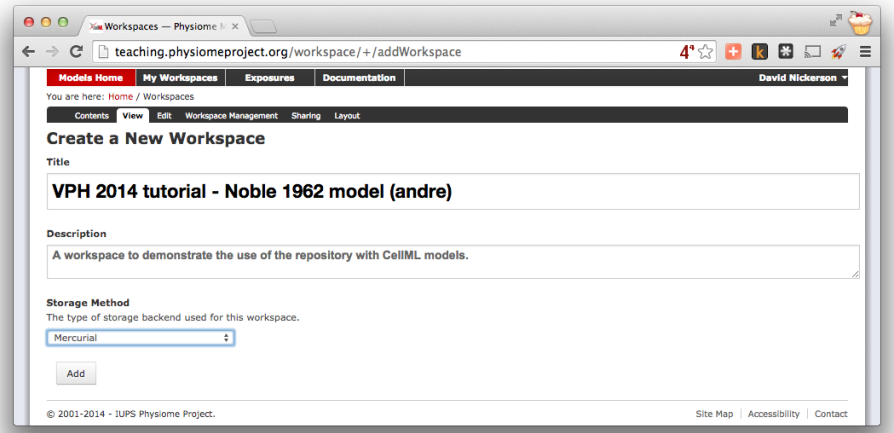


Fig. 1.1: Creating a new workspace to begin a scientific study based on the Noble 1962 cardiac cellular electrophysiology model.

Once you have created the workspace, you will be taken to the workspace listing page. Take particular note of the *URI for mercurial clone/pull/push*, also the same as the current page URL.

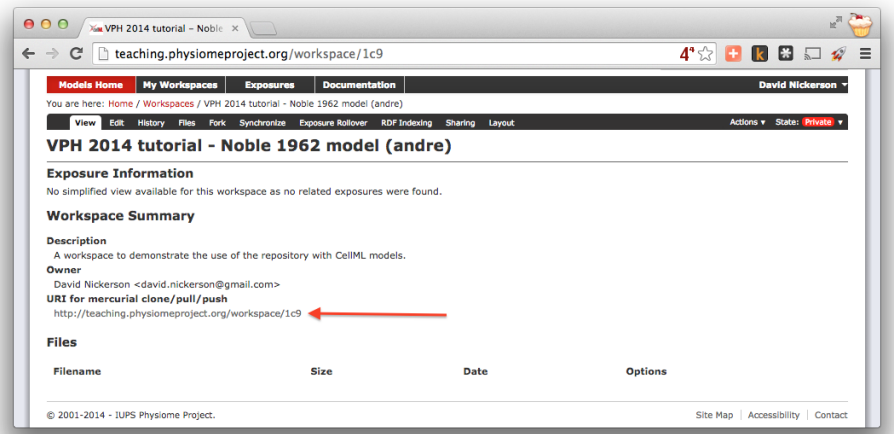
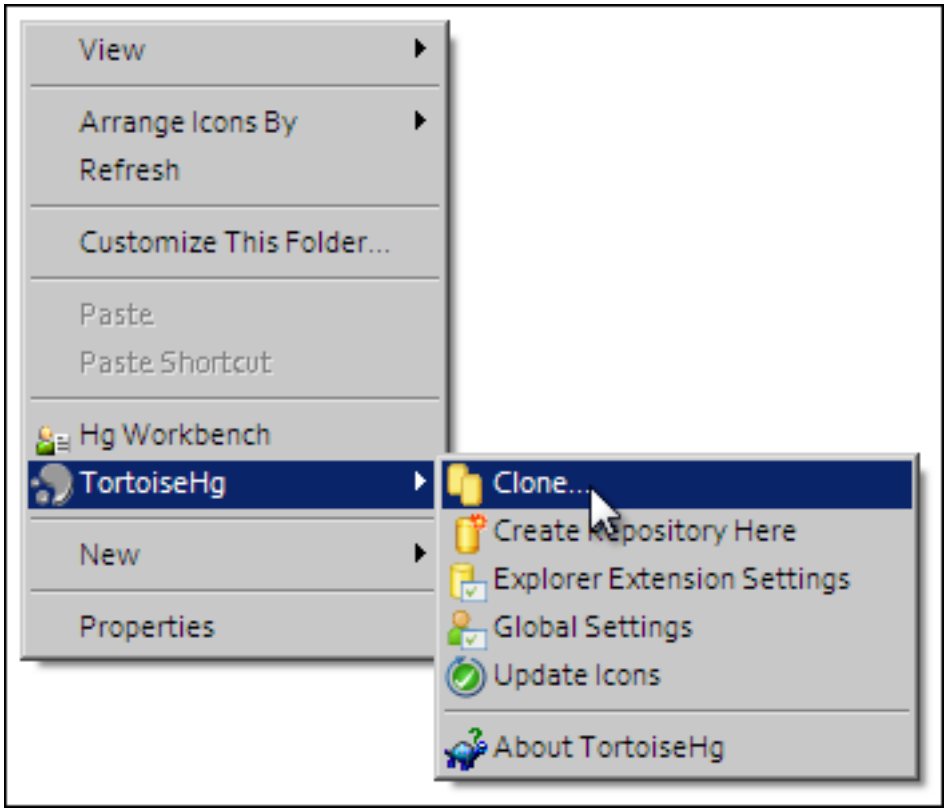


Fig. 1.2: A view of the newly created and empty workspace. Note: the workspace URI is unique to every workspace, so yours will be different to the one shown above.

In order to make changes to your workspace, you have to *clone* it to your own computer. In order to do this, copy the URI for mercurial clone/pull/push as shown above. In Windows explorer, find the folder where you want to create the clone of the workspace. Then, right click to bring up the context menu, and select *TortoiseHG* → *Clone* as shown below:



Paste the copied URL into the *Source:* area and then click the *Clone* button. This will create a folder named after the workspace identifier (a hexadecimal number) that will be empty. The folder will be created inside the folder in which you instigated the clone command.

Command line equivalent

```
hg clone [URI]
```

The repository will be cloned within the current directory of your command line window.

You will need to enter your username and password to clone the workspace, as the workspace will be set to *private* when it is created.

Populate with content

We have prepared a copy of the Noble (1962) model encoded in CellML ready for your use. You can download the model `n62.cellml` and save it into your cloned workspace folder created above. To verify that the model works, you can load it into the *OpenCOR Single Cell view* and simulate the model for *5000 ms*. You can plot the variable *V* in the *membrane* component and you should see results as shown below:

Todo

These images need to be updated if there is time.

As long as your results look similar to the above, everything is working as expected. Now is a good time to add the CellML model to the workspace record. The first step is to choose the *TortoiseHG* → *Add Files...* option from the context menu for your workspace folder (1).

This will bring up the *hg add* dialog box, showing the files which can be added (in this case, only the `n62.cellml` file is available and it is selected by default). Clicking the *Add* button (2) will inform Mercurial that you want to add the selected file to the workspace.

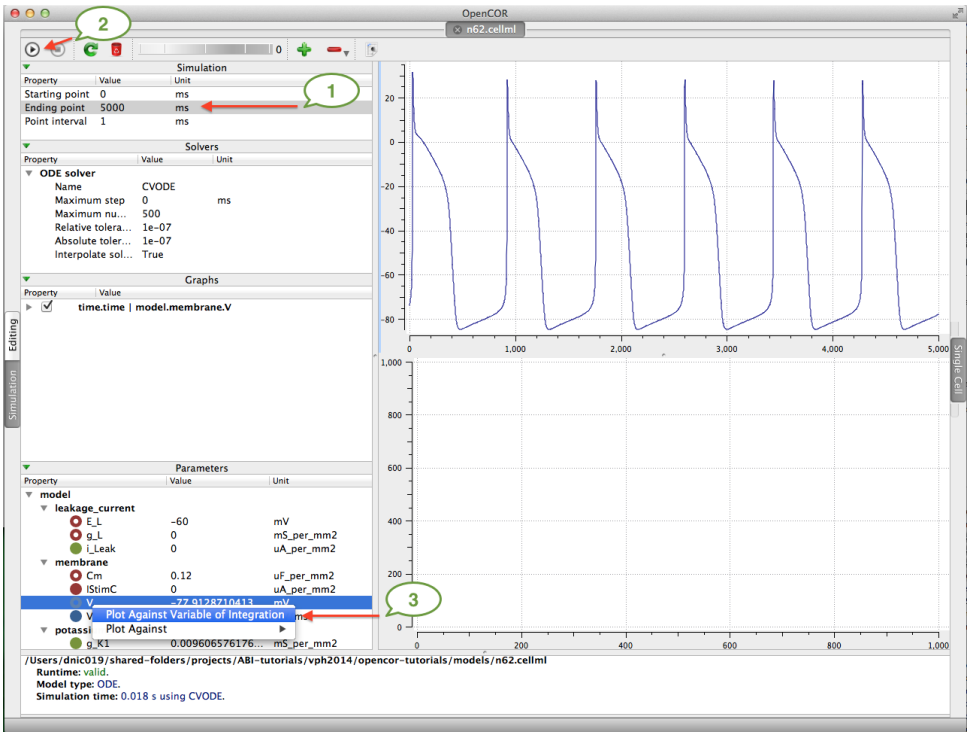
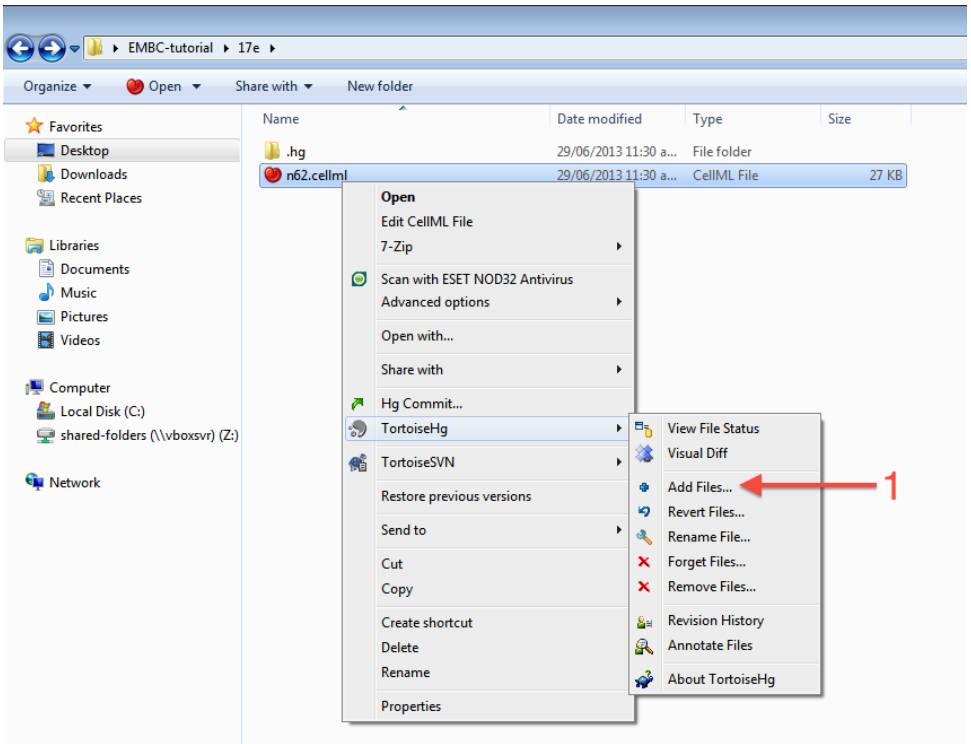
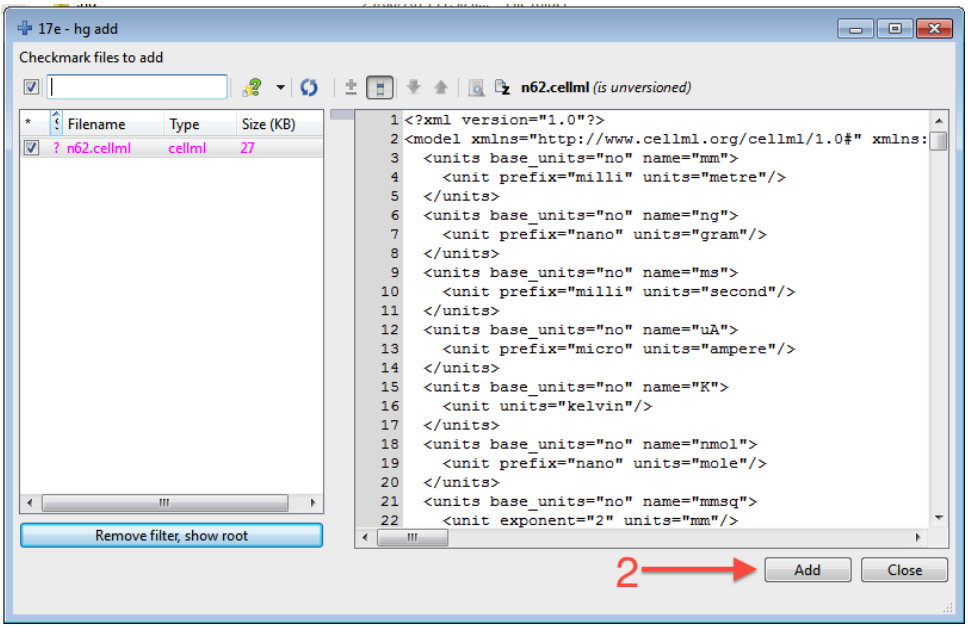
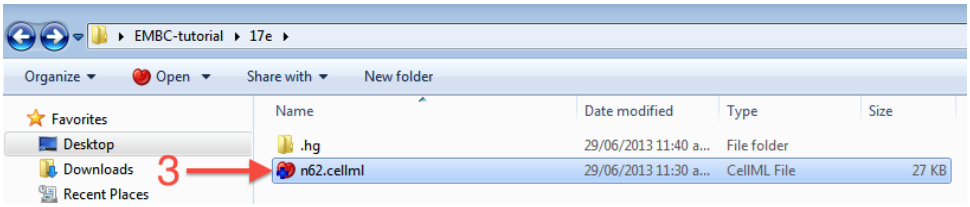


Fig. 1.3: The arrows highlight the *Ending point* which should be set to *5000 ms*, the button to run the simulation, and the variable V to be plotted.





In Windows Explorer, you will see the file icon for the `n62.cellml` model now overlaid with the Mercurial + icon (3) to indicate that you have added the file, but not yet committed it to the workspace.



You can now commit the added file to the workspace by choosing *Hg Commit...* from the context menu in your workspace folder (4).

This will bring up the *commit* dialog, which lets you explore and select all the possible changes in this workspace that you can commit. In this case, there is just the addition of the `n62.cellml` file to be committed. Before committing, a useful log message should be entered - this will help you keep track of the changes you make to the workspace and possibly the reasons for why a given set of changes were made (for example, due to feedback from reviewers). After entering the log message, click the *Commit* button to commit the changes (5). The dialog will stay visible in case you have further changes to commit, but in this case you can just close the dialog.

Once you have successfully committed the change, you will see that the icon for the `n62.cellml` file has now changed to a green tick (6) to indicate that the file is up-to-date with no modifications.

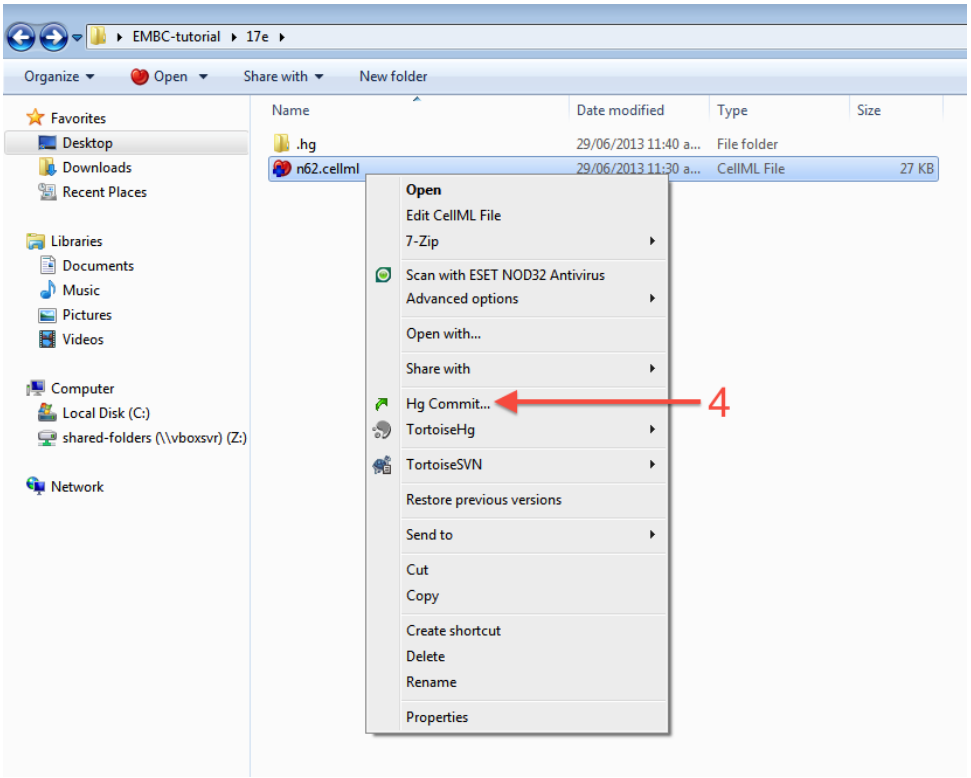
Command line equivalent

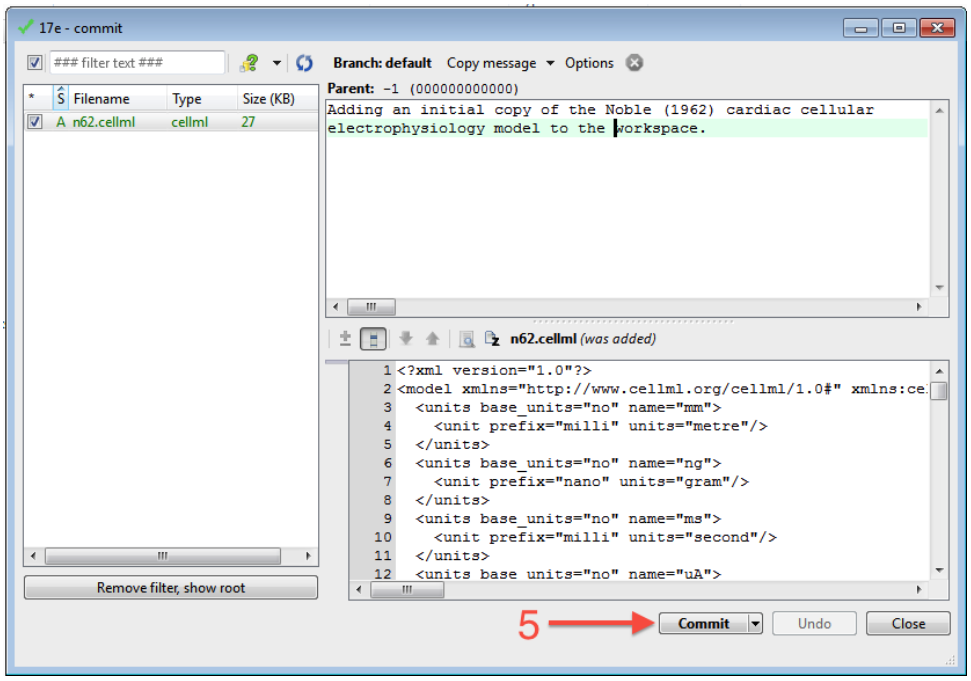
```
hg add n62.cellml
hg commit -m "Adding an initial copy of the Noble (1962) cardiac cellular␣
↪electrophysiology model to the workspace."
```

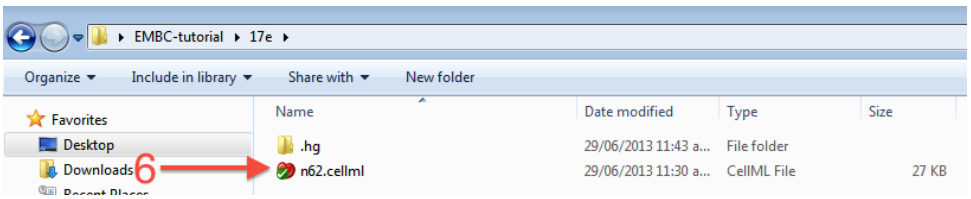
Annotating the model

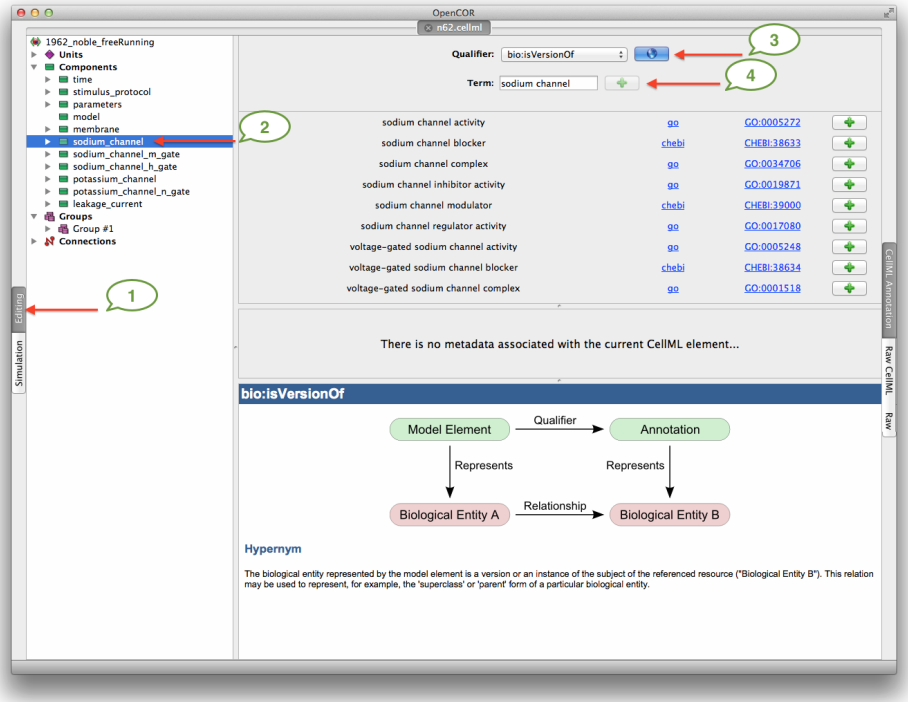
While we have the model open in OpenCOR, we should have a go at annotating some of the objects in the model. Full instructions for this can be found in the *OpenCOR CellML Annotation view*. First, we will follow the *example given in those instructions* for annotating the `sodium_channel` component.

The first step is to switch to the *Editing* mode (1) (make sure that the *CellML Annotation* view is selected) and select the `sodium_channel` component for annotation (2). We will be using the `bio:isVersionOf` as the qualifier for this annotation (3) and searching for terms related to `sodium channel` (4).

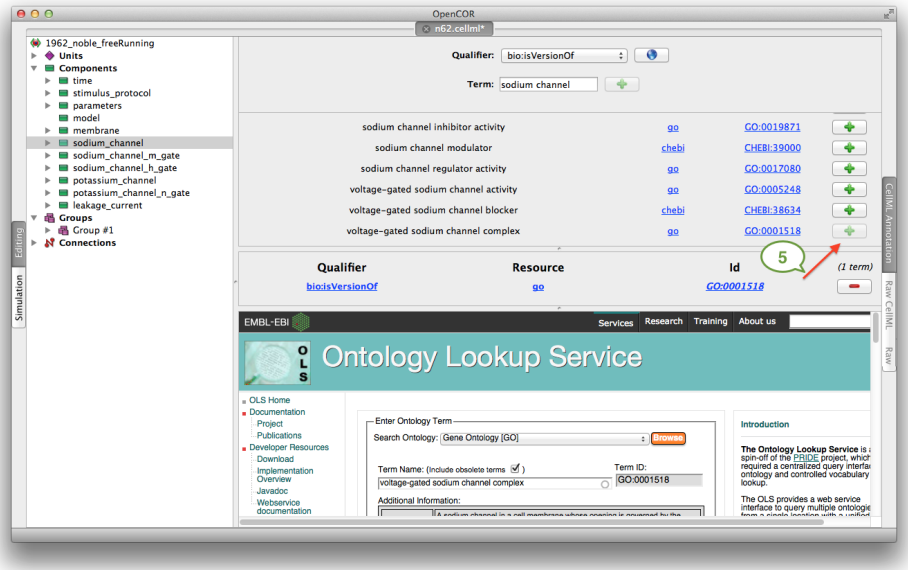

EMBC-tutorial ▸ 17e ▸
Organize
Open
Share with
New folder
Favorites
Desktop
Downloads
Recent Places
Libraries
Documents
Music
Pictures
Videos
Computer
Local Disk (C:)
shared-folders (\\vboxsvr) (Z:)
Network
Name
Date modified
Type
Size
.hg
29/06/2013 11:40 a...
File folder
n62.cellml
29/06/2013 11:30 a...
CellML File
27 KB
Open
Edit CellML File
7-Zip
Scan with ESET NOD32 Antivirus
Advanced options
Open with...
Share with
Hg Commit...
TortoiseHg
TortoiseSVN
Restore previous versions
Send to
Cut
Copy
Create shortcut
Delete
Rename
Properties
4


17e - commit
filter text
Branch: default
Copy message
Options
Filename
Type
Size (KB)
A n62.cellml
cellml
27
Parent: -1 (000000000000)
Adding an initial copy of the Noble (1962) cardiac cellular electrophysiology model to the workspace.
n62.cellml (was added)
<?xml version="1.0"?>
<model xmlns="http://www.cellml.org/cellml/1.0#" xmlns:ce
<units base_units="no" name="mm">
<unit prefix="milli" units="metre"/>
</units>
<units base_units="no" name="ng">
<unit prefix="nano" units="gram"/>
</units>
<units base_units="no" name="ms">
<unit prefix="milli" units="second"/>
</units>
<units base_units="no" name="uA">
Remove filter, show root
5
Commit
Undo
Close


EMBC-tutorial ▸ 17e ▸
Organize
Include in library
Share with
New folder
Favorites
Desktop
Downloads
6
Name
Date modified
Type
Size
.hg
29/06/2013 11:43 a...
File folder
n62.cellml
29/06/2013 11:30 a...
CellML File
27 KB



We can then add desireable terms from the search results by choosing the button beside the term to add to the annotations for the `sodium_channel` component (5).

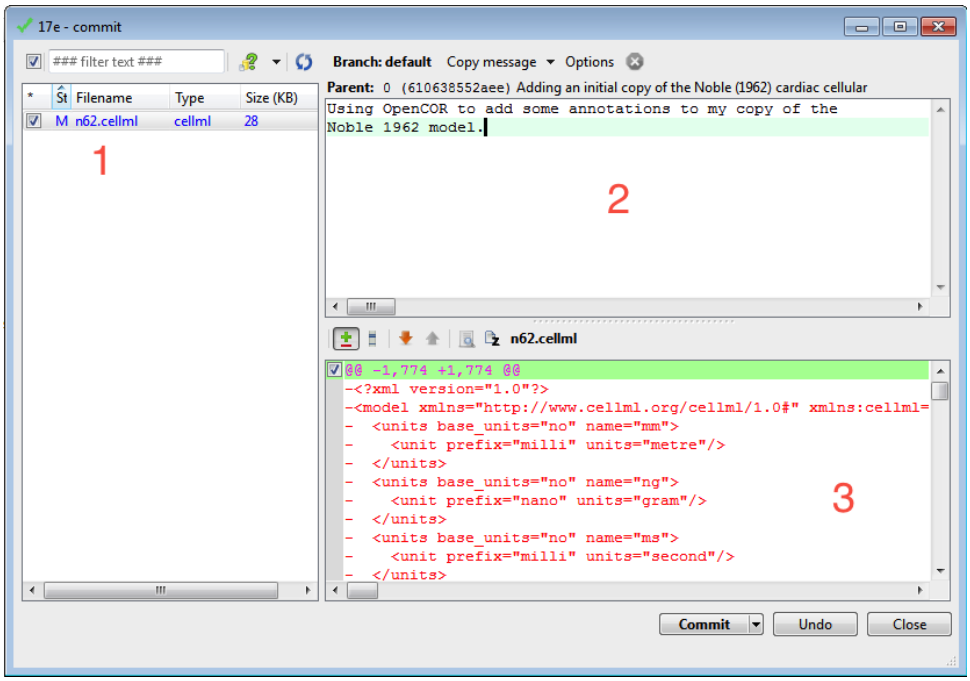


Have a play annotating other variables and components in the model. When done annotating, make sure to save the model (*File* → *Save*). With the CellML model updated, now is a good time to commit the changes to the workspace.

Commit changes

As above, choose *Hg Commit...* from the context menu in your workspace folder to bring up the Mercurial *commit* dialog. This time, you will see that there is one file modified that can be committed, `n62.cellml` (1). As we

mentioned previously, it is important to enter a good log message to keep a record of the changes you make (2), and the changes made to the currently selected file are shown to help remind you as to your changes (3). In this case, OpenCOR has made many changes to the whitespace in the file, as well as adding the RDF annotations at the bottom of the file.



Command line equivalent

```
hg diff
hg commit -m "Using OpenCOR to add some annotations to my copy of the Noble 1962
↪model."
```

Push back to the repository

Having added content and performed some modifications, it is time to *push* the changes back to the model repository, achieved in TortoiseHG with the synchronization action. First, select *TortoiseHG* → *Synchronize* from the context menu for your workspace folder.

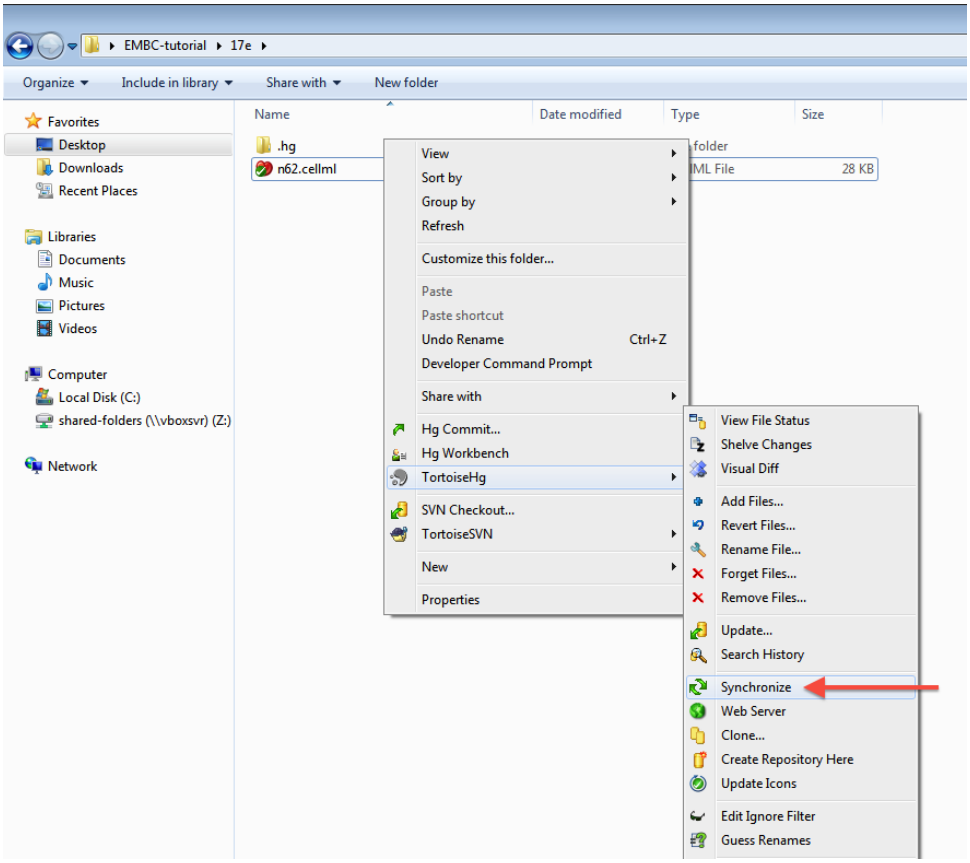
This will bring up the *TortoiseHG Sync* dialog. In this dialog, you will see that by default you will be synchronizing with the workspace on the teaching repository from which you originally created this clone. This is usually what you want to do, but it is possible to synchronize with other Mercurial repositories. In this case, we want to *push* the changes we have made to the model repository, so choose the corresponding action from the toolbar (highlighted below).

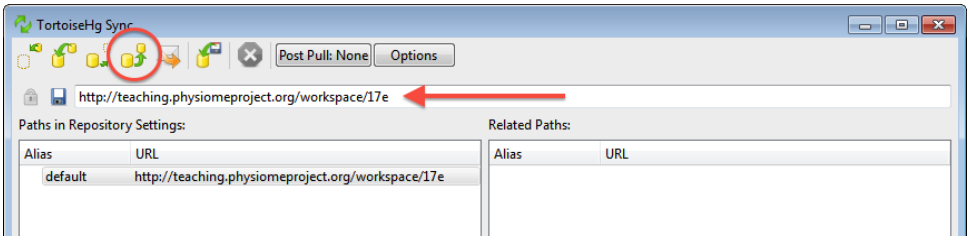
Once you choose the *push* action, you will be asked to confirm that you want to push to your remote repository and then asked for your username and password (these are the credentials you created when registering for an account in the model repository). You will then see a listing of the transaction as your changes are pushed to the repository and a message stating the push has completed.

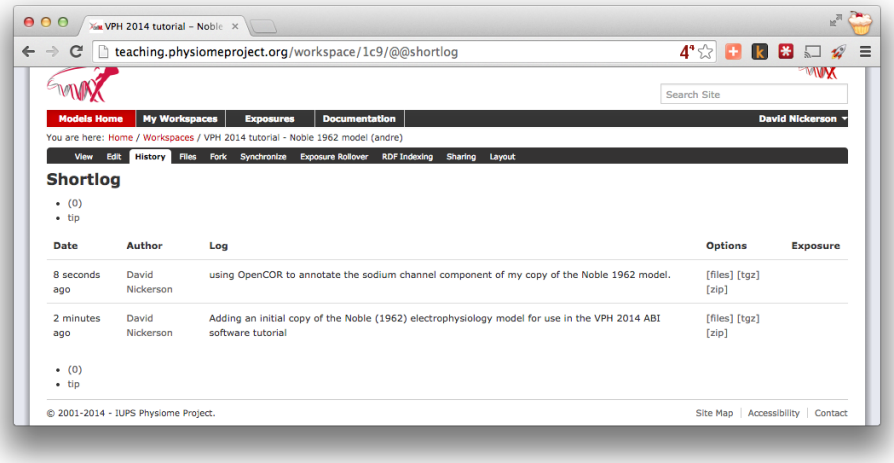
Command line equivalent

```
hg push
```

If you now return to browsing your workspace in your web browser, and refresh the page, you will see that your workspace now has some content - `n62.cellml` - and if you view the workspace history, you will see the log messages that you entered when committing your changes above.


EMBC-tutorial ▸ 17e ▸
Organize
Include in library
Share with
New folder
Favorites
Desktop
Downloads
Recent Places
Libraries
Documents
Music
Pictures
Videos
Computer
Local Disk (C:)
shared-folders (\\vboxsvr) (Z:)
Network
Name
Date modified
Type
Size
.hg
n62.cellml
View
Sort by
Group by
Refresh
Customize this folder...
Paste
Paste shortcut
Undo Rename
Ctrl+Z
Developer Command Prompt
Share with
Hg Commit...
Hg Workbench
TortoiseHg
SVN Checkout...
TortoiseSVN
New
Properties
View File Status
Shelve Changes
Visual Diff
Add Files...
Revert Files...
Rename File...
Forget Files...
Remove Files...
Update...
Search History
Synchronize
Web Server
Clone...
Create Repository Here
Update Icons
Edit Ignore Filter
Guess Renames


TortoiseHg Sync
Post Pull: None
Options
http://teaching.physiomeproject.org/workspace/17e
Paths in Repository Settings:
Related Paths:
Alias
URL
default
http://teaching.physiomeproject.org/workspace/17e

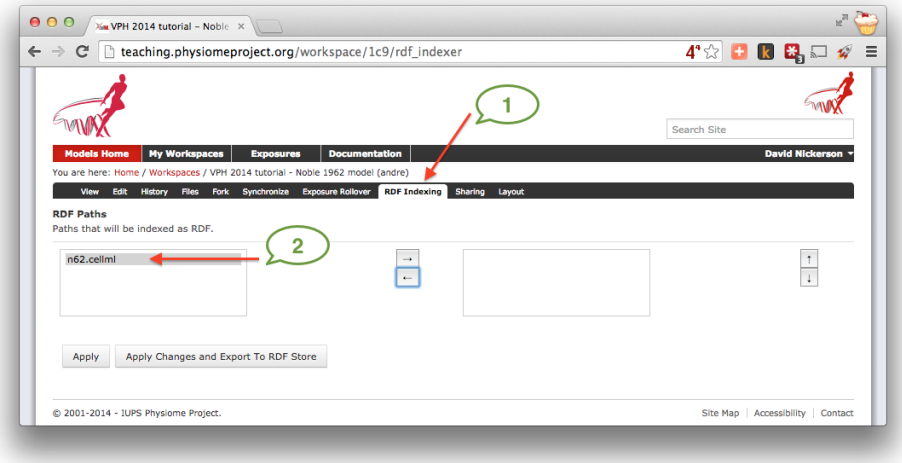

VPH 2014 tutorial - Noble
teaching.physiomeproject.org/workspace/1c9/@@shortlog
Search Site
Models Home
My Workspaces
Exposures
Documentation
David Nickerson
You are here: Home / Workspaces / VPH 2014 tutorial - Noble 1962 model (andre)
View
Edit
History
Files
Fork
Synchronize
Exposure Rollover
RDF Indexing
Sharing
Layout
Shortlog
(0)
tip
Date
Author
Log
Options
Exposure
8 seconds ago
David Nickerson
using OpenCOR to annotate the sodium channel component of my copy of the Noble 1962 model.
[files] [tgz] [zip]
2 minutes ago
David Nickerson
Adding an initial copy of the Noble (1962) electrophysiology model for use in the VPH 2014 ABI software tutorial
[files] [tgz] [zip]
(0)
tip
© 2001-2014 - IUPS Physiome Project.
Site Map
Accessibility
Contact

Now might be a good time to think about *sharing your workspace* with your neighbours. You might also want to have a look at creating an *exposure* for your workspace. To learn how to create exposures, please refer to *Creating CellML exposures*.

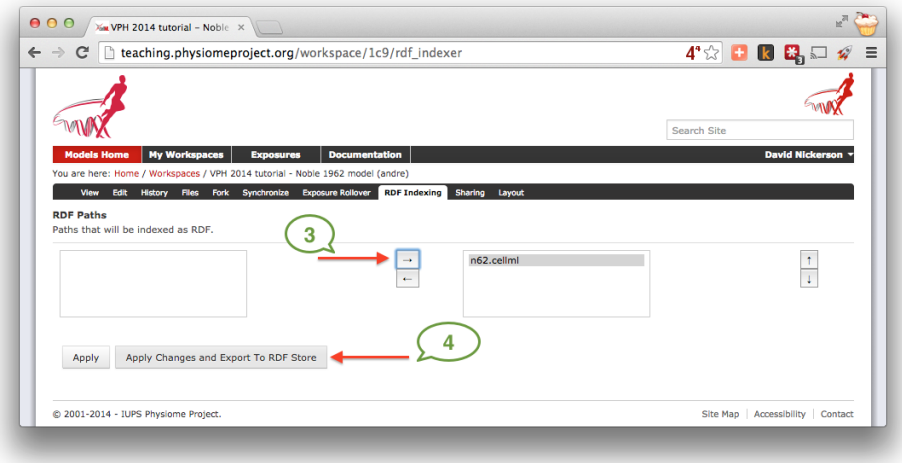
Making use of annotations

Recent additions to *PMR2* have focussed on *working with semantic metadata*. In this part of the tutorial we will demonstrate how to take the annotated Noble (1962) model from the *previous tutorial* and index it in the repository's semantic knowledgebase for later retrival.

In the previous tutorial, you *annotated* your copy of the Noble (1962) model and *pushed* it up to the teaching instance repository. If you now visit your *workspace* URL and navigate to the *RDF Indexing* tab (1), you will see that the `n62.cellml` is the only resource avaiable to be indexed (2).

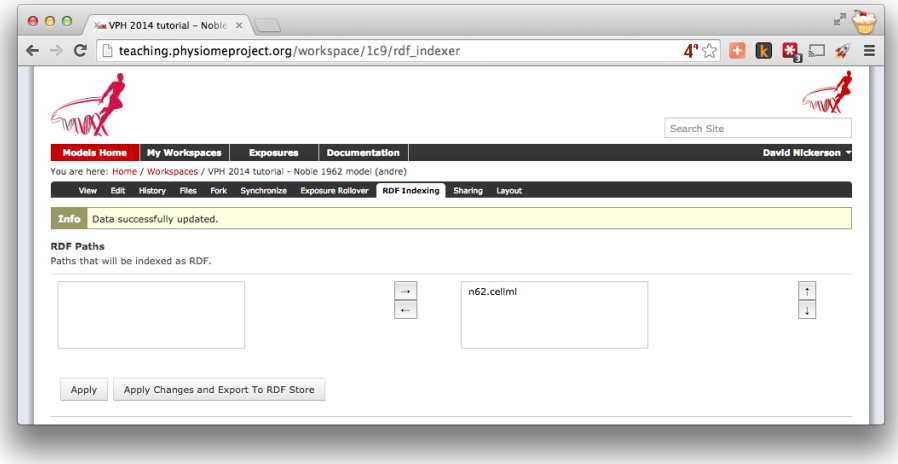


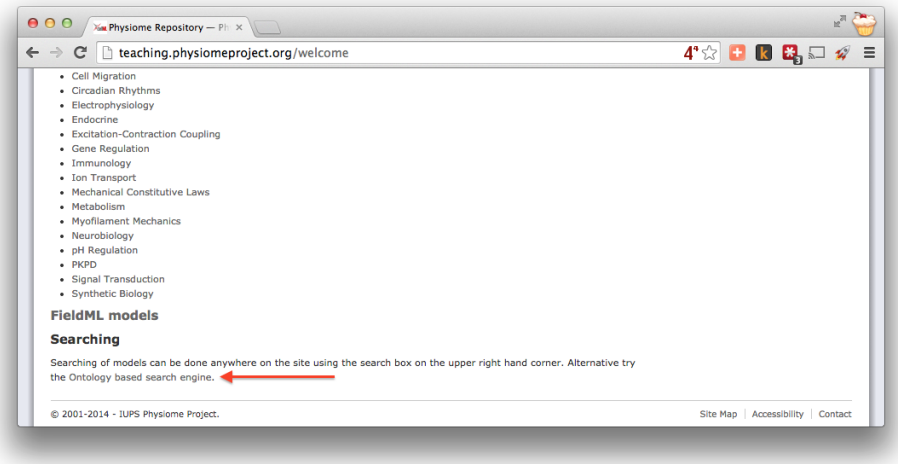
Moving the `n62.cellml` file over to the box on the right indicates that it should be indexed (3) and selecting the *Apply Changes and Export to RDF Store* button (4) will apply the change and index the RDF obtained from the CellML document.



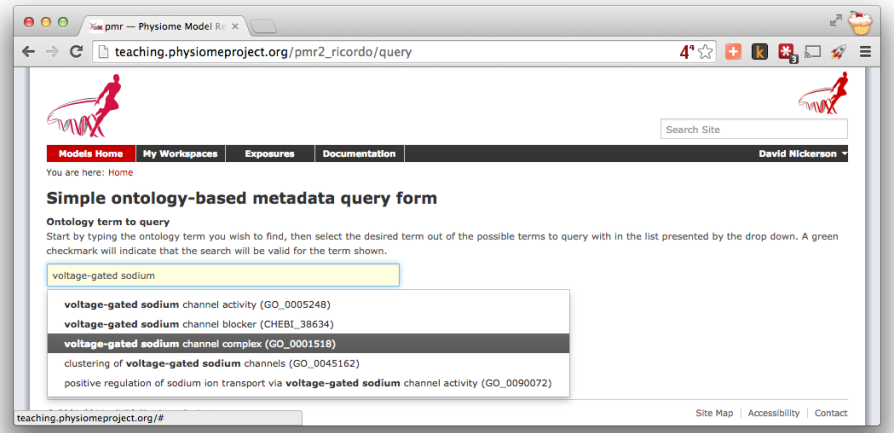
As long as everything is successful, you'll end up with a page similar to that shown below, and future revisions of the `n62.cellml` file will automatically be indexed in the RDF store.

Now that your model is indexed, lets try to find it. You can navigate to the *Ontology based search engine* from the front page of the teaching instance.

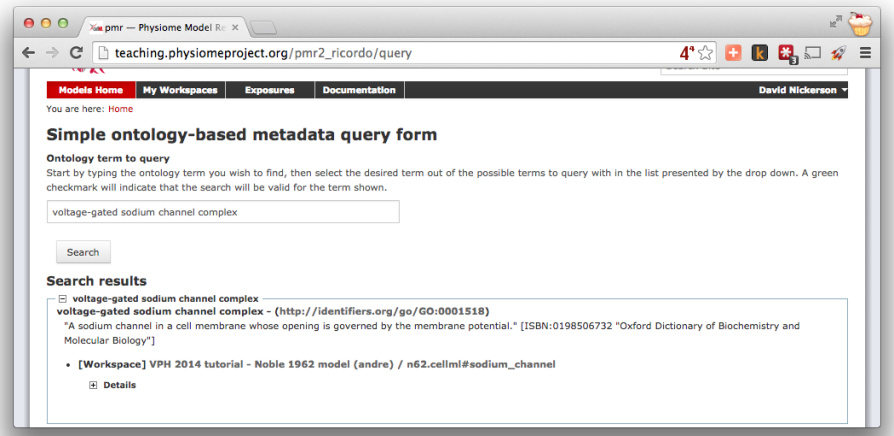

VPH 2014 tutorial - Noble
teaching.physiomeproject.org/workspace/1c9/rdf_indexer
Search Site
Models Home
My Workspaces
Exposures
Documentation
David Nickerson
You are here: Home / Workspaces / VPH 2014 tutorial - Noble 1962 model (andre)
View
Edit
History
Files
Fork
Synchronize
Exposure Rollover
RDF Indexing
Sharing
Layout
Info
Data successfully updated.
RDF Paths
Paths that will be indexed as RDF.
n62.cellml
Apply
Apply Changes and Export To RDF Store


Physiome Repository
teaching.physiomeproject.org/welcome
Cell Migration
Circadian Rhythms
Electrophysiology
Endocrine
Excitation-Contraction Coupling
Gene Regulation
Immunology
Ion Transport
Mechanical Constitutive Laws
Metabolism
Myofilament Mechanics
Neurobiology
pH Regulation
PKPD
Signal Transduction
Synthetic Biology
FieldML models
Searching
Searching of models can be done anywhere on the site using the search box on the upper right hand corner. Alternative try the Ontology based search engine.
© 2001-2014 - IUPS Physiome Project.
Site Map
Accessibility
Contact

In the search text field, you can begin typing the label of one of the terms you added to your copy of the Noble (1962) model. In this *example*, we used the term `voltage-gated sodium channel complex`. As you type the auto-complete will kick in and you'll start to see suggested terms. As you refine your query the list will decrease and you will hopefully see one you remember entering.



Once you choose the desired term, you can click the *Search* button to exectute the search. Assuming you selected an ontology term that you used (or which someone else has used in another workspace) you should see your copy of the Noble (1962) model in the search results.



Note: Because your workspace is still private, only you will see it in the search results even if you use the same annotation terms as others. Once a workspace is published, the associated annotations will become visible and searchable by all. Similarly, if you *share* your workspace with another user they will then see your model show up in their search results for the appropriate ontology terms.

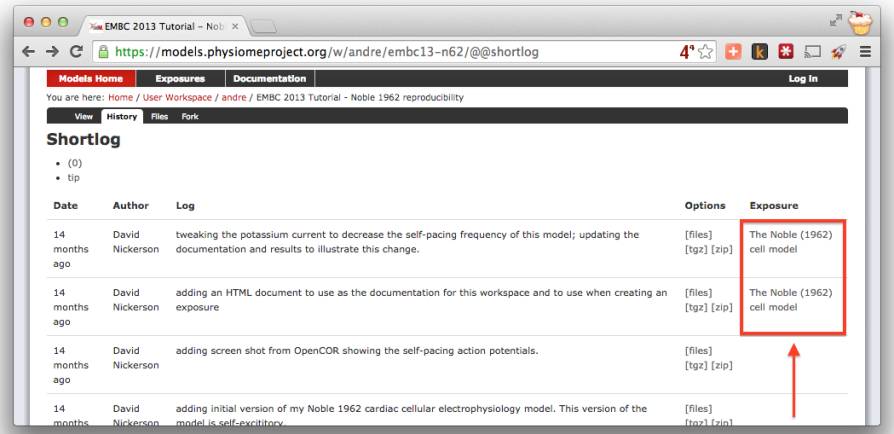
In future, OpenCOR will make use of PMR2 webservices to provide a similar interface as the repository web interface directly in the application. This will allow users to find and reuse existing models all in one place.

Reproducing model behaviour in OpenCOR

In this tutorial, we will be demonstrating how to reproduce the results from a CellML model as they were originally published. Because the repository makes use of *Mercurial*, even if a *workspace* has continued being developed

after a particular revision is published, we are able to step back through the workspace history to reproduce those original published results.

Following on from the *previous tutorial*, we make use of the Noble (1962) cardiac cellular electrophysiology model. In this tutorial, we will use the version of this model published in the repository and available here: https://models.physiomeproject.org/e/174. If you navigate from that *exposure* to the *workspace* you can check the history as shown below.



As you can see highlighted in the *Exposure* column of the history above, there are two exposures for this workspace. For the purposes of this tutorial, we will assume that the earlier exposure corresponds to a study that has been published in a scientific journal. The later exposure is the result of further work on this model following the publication of the journal article. The later exposure illustrates the difference between these two versions of the model. In this tutorial, we aim to reproduce the results as shown in the published journal article - corresponding to the earlier exposure.

Important: It is essential to use a Mercurial client to obtain models from the repository for editing. The Mercurial client is not only able to keep track of all the changes you make (allowing you to back-track if you make any errors), but using a Mercurial client is the only way to add any changes you have made back into the repository.

Cloning an existing workspace

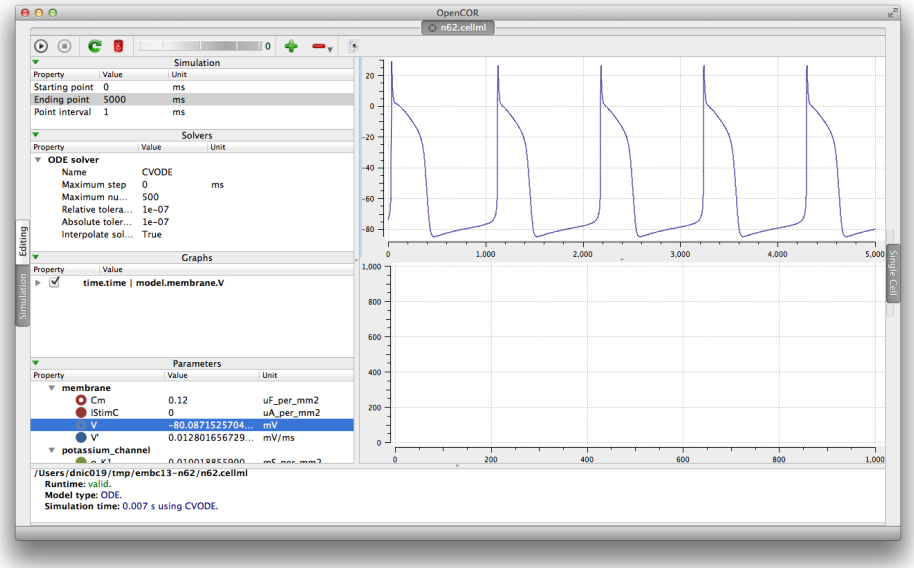
The first step is to *clone* the workspace containing the model we want to work with. The steps to clone a workspace were demonstrated in the *previous tutorial*. In summary:

1. Copy the source URI for Mercurial clone/push/pull (*i.e.*, `https://models.physiomeproject.org/w/andre/embc13-n62`);
2. *Clone* the repository (*TortoiseHG* → *Clone* or `hg clone [uri]`) to a folder on your machine.

Check the model

Now that we have the model, we want to ensure that we are able to produce the current results that it should produce. Load the `n62.cellml` file in the newly cloned folder into OpenCOR and run a simulation for *5000 ms* and plot the membrane potential, `V`. This should result in a similar graph to that shown in the upper figure of the exposure page, reproduced here for convenience.

Notice that in the *5000 ms* simulation there are **five** action potentials.

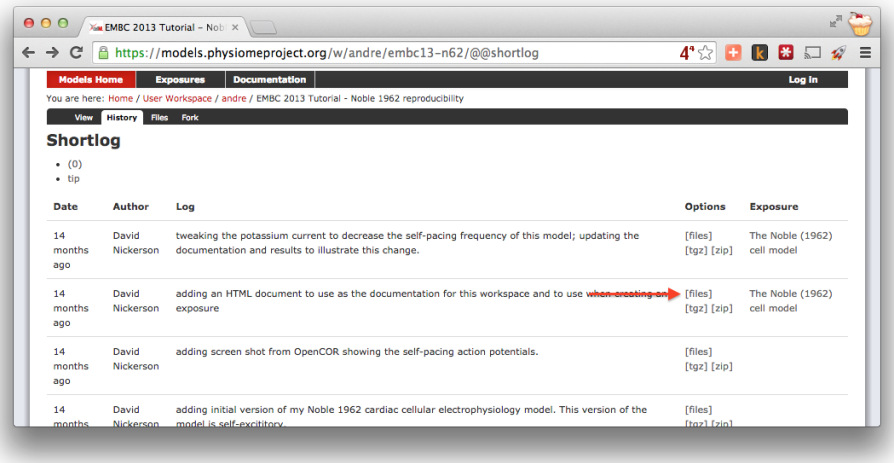


Revert to an earlier version of the model

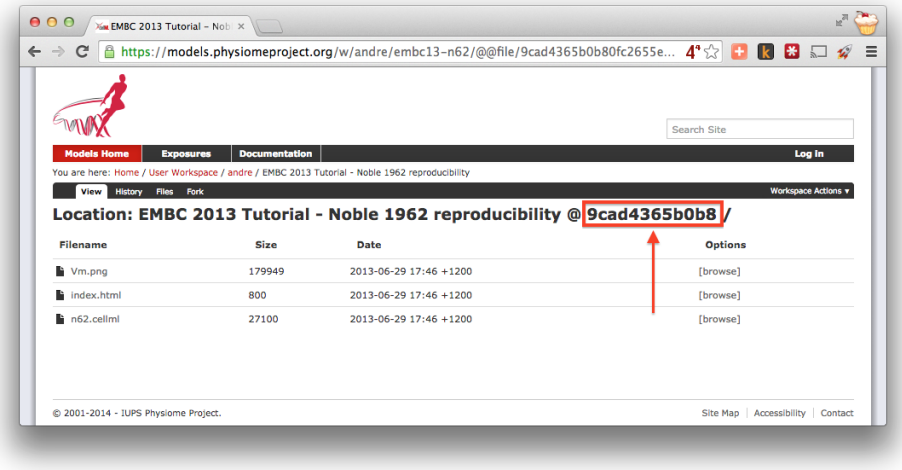
Now that we are happy the current version of the model reproduces the results that it should, we want to go back to the version of the model that was published in a journal article. This is commonly required because the new work you might want to do with the model will be based on the published model, not its latest version which may have deviated from the validated model which was published.

Using *Mercurial*, there are several methods by which you can jump around the history of a *workspace*. The particular method that works best depends a lot on what you want to do with the workspace once you change back to a revision that is not the most recent. Searching the internet for information on the Mercurial (hg) commands: `revert`, `update`, and `branch`; is probably a good place to start working out which is best for your situation. In this case, we have a fairly simple requirement to go back to the revision prior to the current one so that we can reproduce some simulation results. If we were actually going to do further development in this workspace, we would need a more elaborate solution than that described below.

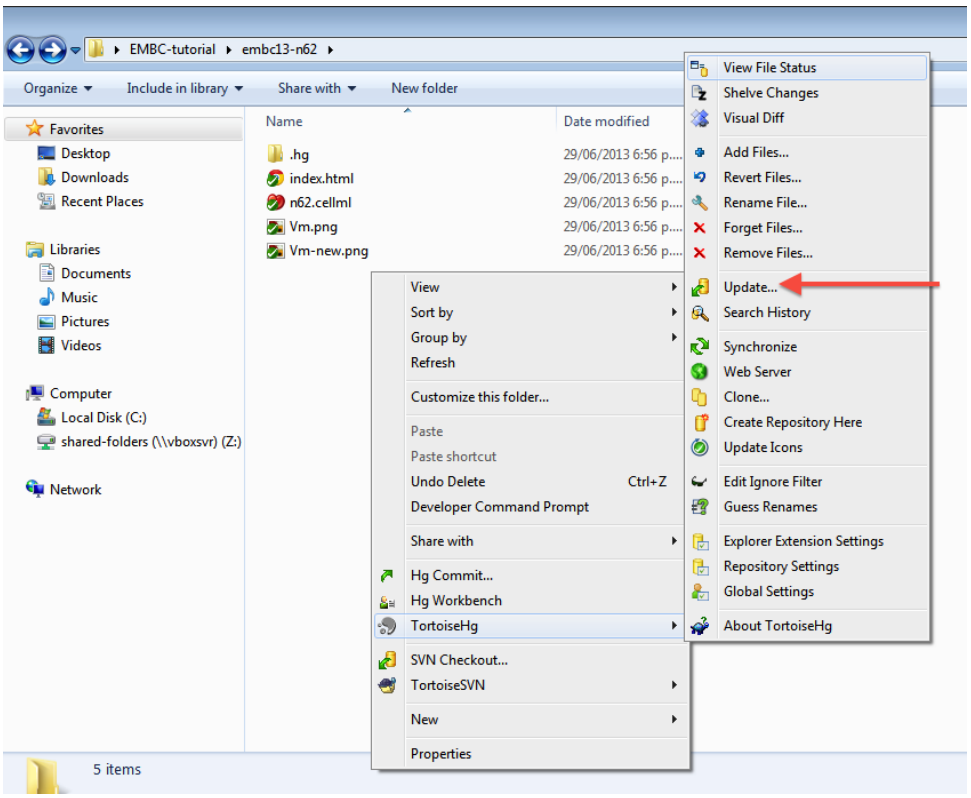
Here, we need to update our local clone of the workspace to a state matching the published journal article. In order to do this, we need to find the appropriate revision identifier to use with our Mercurial client. We can find the revision identifier by navigating to the workspace history tab in the model and choosing the *[files]* link for the revision corresponding to the earlier exposure, shown below.



From the files page, you will see the required revision identifier as highlighted in the image below.



You should copy this identifier to the clipboard ready for use in the next step. In your local clone of the workspace, select *TortoiseHG* → *Update...* from the context menu. This will bring up the *Update* dialog.

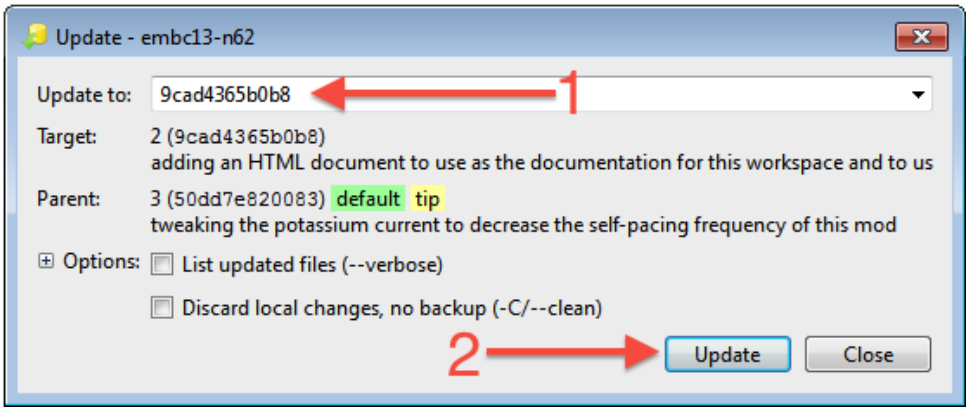


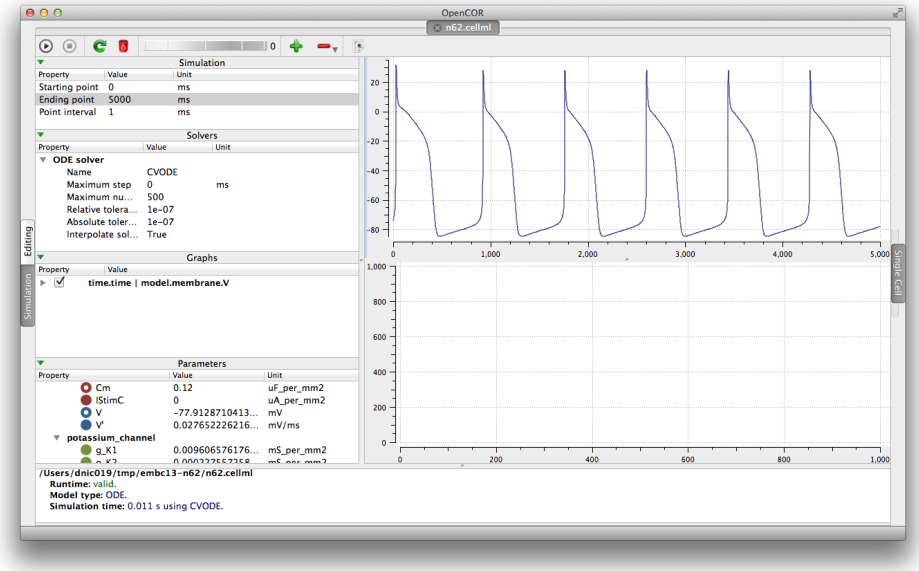
In this dialog, you should paste the revision identifier copied above into the *Update to:* field (1) and then click the *Update* button (2).

Command line equivalent

```
hg update -r 9cad4365b0b8
```

You will now see in your local clone that the files have reverted back to that previous version. Loading this version of `n62.cellml` into OpenCOR and simulating for *5000 ms* should result in the figure matching that presented in the earlier exposure page and reproduced here for convenience.


Update - embc13-n62
Update to:
9cad4365b0b8
1
Target:
2 (9cad4365b0b8)
adding an HTML document to use as the documentation for this workspace and to us
Parent:
3 (50dd7e820083) default tip
tweaking the potassium current to decrease the self-pacing frequency of this mod
Options:
List updated files (--verbose)
Discard local changes, no backup (-C/--clean)
2
Update
Close


OpenCOR
n62.cellml
Simulation
Solvers
Graphs
Parameters
ODE solver
time.time | model.membrane.V
potassium_channel
/Users/dnic019/tmp/embc13-n62/n62.cellml
Runtime: valid.
Model type: ODE.
Simulation time: 0.011 s using CVODE.

Note in particular that there should now be the same **six** action potentials that were present in the published version of the model. In the preceeding tutorials, you have learnt how to create a new piece of work from scratch using the respository and how to reproduce a *"published"* result. In this tutorial, we will demonstrate how to take an exisiting piece of work, stored in a public *workspace*, and develop it further to address a new goal.

Extending an existing CellML model

In this part of the tutorial, we will once again be making use of the Noble (1962) cardiac cellular electrophysiology model. We will be taking the model and making changes to alter its behaviour. For this, we will be using the same version of the model published in the teaching instance of the repository: http://teaching.physiomeproject.org/e/173, but the process described below will also work in the main repository site.

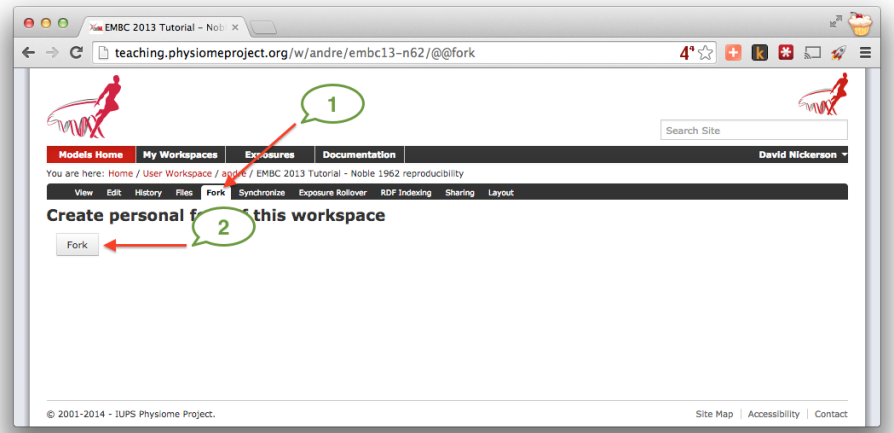
Forking an existing workspace

Important: It is essential to use a Mercurial client to obtain models from the repository for editing. The Mercurial client is not only able to keep track of all the changes you make (allowing you to back-track if you make any errors), but using a Mercurial client is the only way to add any changes you have made back into the repository.

For this tutorial, we will *fork* an existing workspace. This creates a new workspace owned by you, containing a copy of all the files in the workspace you forked including their complete history. This is equivalent to cloning the workspace, creating a new workspace for yourself, and then pushing the contents of the cloned workspace into your new workspace.

Forking a workspace can be done using the repository web interface. The first step is to find the workspace you wish to fork. As before, we will use the *workspace* from the exposure referenced above, which can be found at: http://teaching.physiomeproject.org/w/andre/embc13-n62.

Once you are logged in, click on the *Fork* option in the toolbar, as shown below (1).



You will be asked to confirm the *fork* action by clicking the *Fork* button (2). You will then be shown the page for your forked workspace.

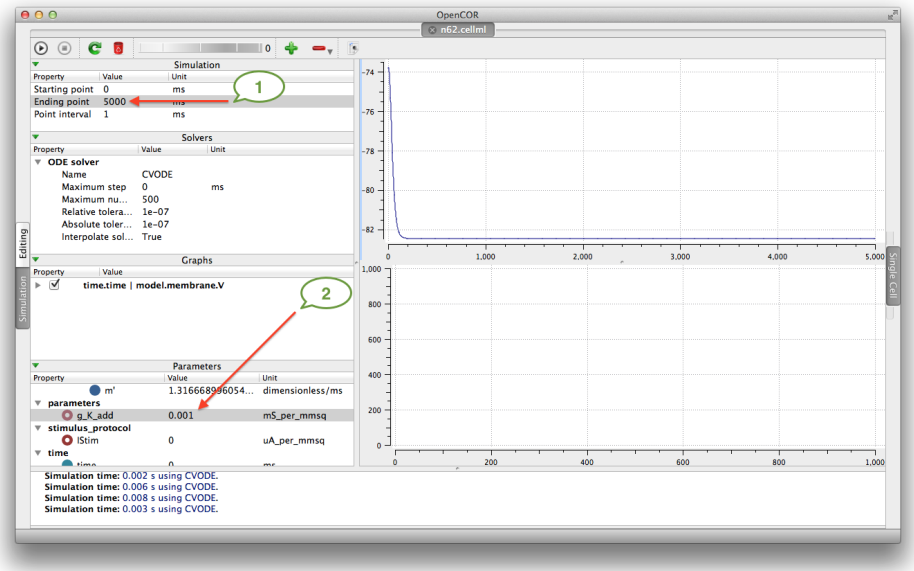
Cloning your forked workspace

In order to make changes to your workspace, you have to *clone* it to your own computer. To do this, follow the procedure as described in the *earlier tutorial*.

Quietening the self excitation

The version of the Noble 1962 model you have just forked and cloned is a model of a Purkinje fibre cell. These cells are capable of acting as pacemaker cells, although usually entrained by the sinoatrial node of the heart. The Noble model reproduces this behavior but is also able to simulate a non-pacing version of the cell model. This is accomplished by decreasing the potassium current which gives rise to the gradual depolarization of the membrane potential seen in the figures from OpenCOR simulations for the model in the previous tutorials. Once the cell is in a quiesent state, we are able to then apply an electrical stimulus to impose our own pacing regime.

If you load the `n62.cellml` file from the workspace you have just cloned into OpenCOR, set the duration of the simulation to *5000 ms* (1), and plot the membrane potential `V`, you will be able to see the effect of altering the value of the variable `g_K_add` in the `parameters` component. As you increase this value you should see the resting potential decrease and the abolution of the self-exciting mechanism. A value of *0.001 mS_per_mmsq* keeps the resting potential in the physiological range and makes the cell quiesent (2).



Changes to the parameter value in the simulation view are not currently saved in the model, so to save the change you will need to switch to an editing view (1, 2), find the `g_K_add` parameter (3), and set the `initial_value` attribute directly (4), as shown below.

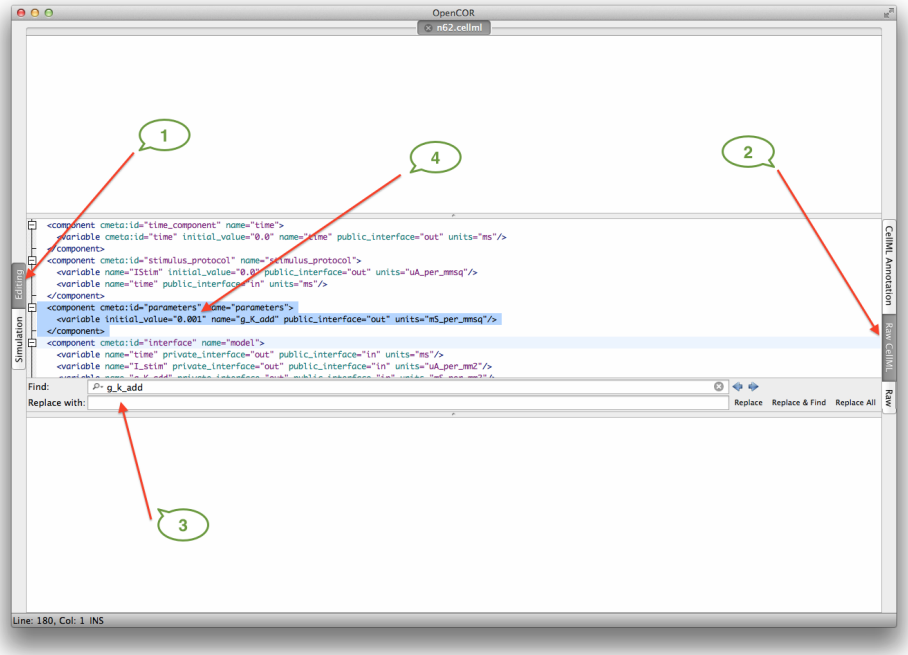
Now would be a good time to *commit your changes* to your clone of the workspace

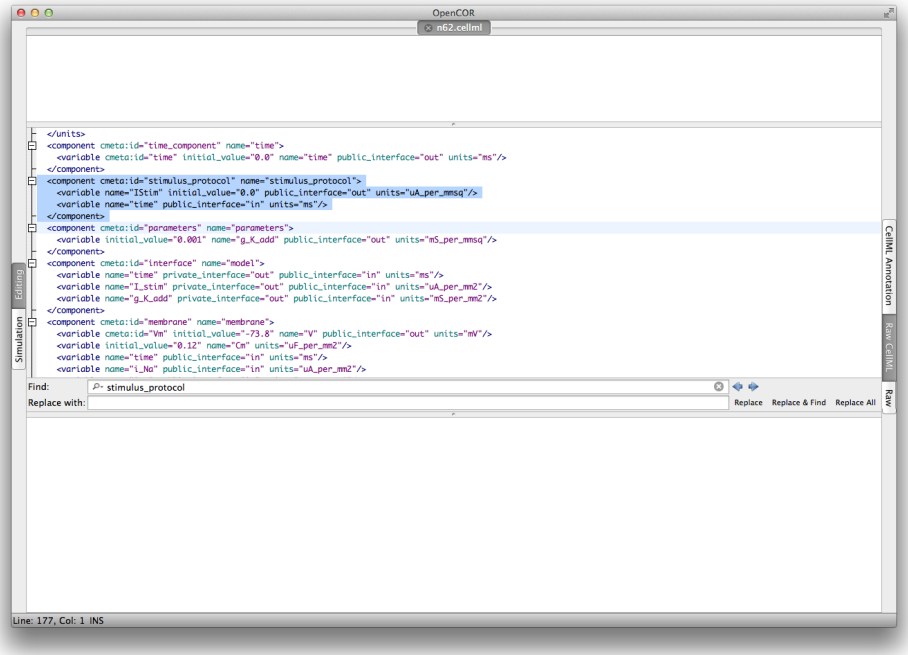
Adding an electrical stimulation protocol

Now that we have a quiesent version of the Noble (1962) model, we are able to consider adding our own electrical stimulation protocol. In the *Raw CellML* view, you will see a component with the name `stimulus_protocol` as shown below.

As you can see in this snippet of the XML source, there is a stimulus current variable, `IStim`, which is given a value of *0.0 uA_per_mm2*. In this extension to the model we will replace this simple assignment of no stimulus current with a definition of a periodic applied stimulus. The code example below shows one way to encode such a periodic stimulus current in CellML.

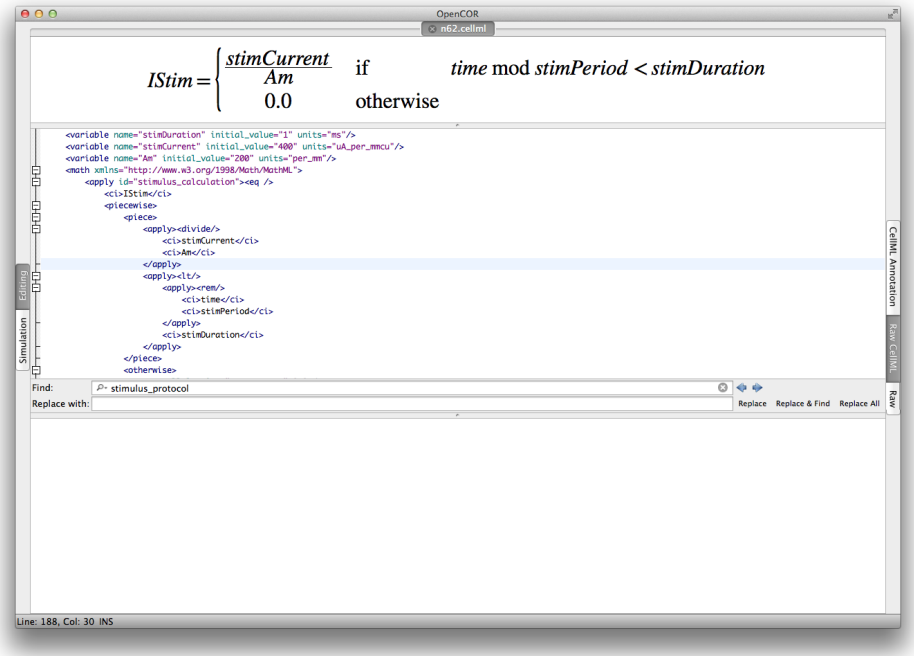
```
<component cmeta:id="stimulus_protocol" name="stimulus_protocol">
  <variable name="IStim" public_interface="out" units="uA_per_mmsq"/>
  <variable name="time" public_interface="in" units="ms"/>
  <variable name="stimPeriod" initial_value="750" units="ms"/>
  <variable name="stimDuration" initial_value="1" units="ms"/>
  <variable name="stimCurrent" initial_value="400" units="uA_per_mmcu"/>
```


OpenCOR
n62.cellml
1
2
3
4
Editing
Simulation
CellML Annotation
Raw CellML
Raw
Find:
g_k_add
Replace with:
Replace
Replace & Find
Replace All
Line: 180, Col: 1 INS


OpenCOR
n62.cellml
Editing
Simulation
CellML Annotation
Raw CellML
Raw
Find:
stimulus_protocol
Replace with:
Replace
Replace & Find
Replace All
Line: 177, Col: 1 INS

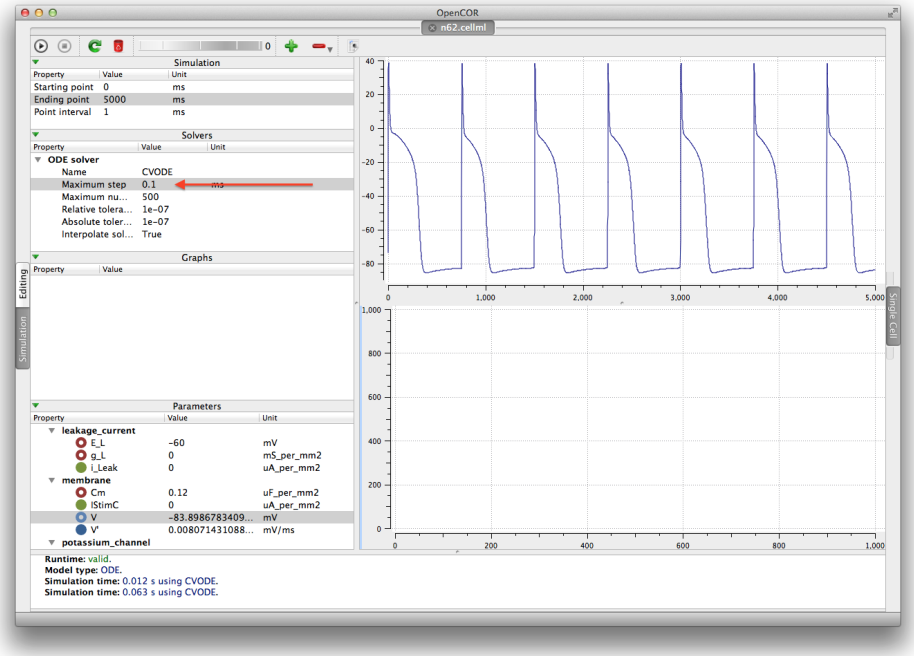
```
  <variable name="Am" initial_value="200" units="per_mm"/>
  <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply id="stimulus_calculation"><eq />
          <ci>IStim</ci>
          <piecewise>
              <piece>
                  <apply><divide/>
                      <ci>stimCurrent</ci>
                      <ci>Am</ci>
                  </apply>
                  <apply><lt/>
                      <apply><rem/>
                          <ci>time</ci>
                          <ci>stimPeriod</ci>
                      </apply>
                      <ci>stimDuration</ci>
                  </apply>
              </piece>
              <otherwise>
                  <cn cellml:units="uA_per_mmsq">0.0</cn>
              </otherwise>
          </piecewise>
      </apply>
  </math>
</component>
```

In the above example, we have introduced some new variables to control the frequency, duration, and magnitude of the applied stimulus current. If you replace the `stimululs_protocol` component in the `n62.cellml` model with the one above, you are able to view a rendering of the mathematics, as below.



Switching back to the *Simulation* view, you are able to have a play with those variables to ensure they are behaving as expected. **Note:** you may need to set the *Maximum step* for CVODE to *0.1* or change to the Forward Euler integrator in OpenCOR to ensure that your specified stimulus in correctly detected by the numerical integration scheme.

Now would be a good time to *commit your changes* to your clone of the workspace and *push* them back to the model repository. You might also want to think about *sharing your workspace* with your neighbors or to have



a look at creating an *exposure* for your workspace. To learn how to create exposures, please refer to *Creating CellML exposures*.

MAP Client

This tutorial is designed to show you the capabilites of MAP Client and how it can make use of the Auckland Physiome Repository through it's webservices. The focus of work on the MAP Client to date has been work for the musculoskeletal system which is not appropriate for this tutorial, the tutorials here are exemplary tutorials designed to exhibit the use and capabilites of MAP Client.

Contents:

Setting Up Pre-requisite Software

MAP Client is dependent on a number of software packages that must be installed before the application can be run. Some of the plugins that MAP Client makes use of have their own dependencies that must also be installed before the plugins can be used. Here we outline the required dependencies and some notes on installation.

Dependencies

Much of what is covered here is relevant for windows users only as OS X and GNU/Linux based operating systems already have the required dependencies for MAP Client. Also GNU/Linux machines have package managers which facilitate the installation of missing dependencies. It is left as an exercise for the user to install dependencies for operating systems that make use of a package manager.

List of Dependencies

- Python - version 2.7.X is preferred, but version 3.X.Y should also work.
- PySide - Python bindings for the Qt libraries.
- requests_oauthlib - Python package for XXXXXXXX

- rdflib - Python package for working with RDF.

Other Dependencies

As MAP Client is focused on scientific processes the Numpy Python package is quite often used. It is not required by MAP Client itself but it is used by a number of plugins that essentially make it a requirement.

- Numpy - Python package for numerical algorithms
- Zinc - OpenCMISS-Zinc visualisation library
- PyZinc - Python bindings for the OpenCMISS-Zinc library

Automatic segmentation of a three-dimensional image stack

The purpose of this first task is to demonstrate some of the capabilities of the MAP Client workflow tool.

Import Workflow from PMR

Start the 'mapclient' application. Use the 'File' menu to select the import action. The dialog that appears connects to webservices on PMR that will enable us to search for MAP Client workflows.

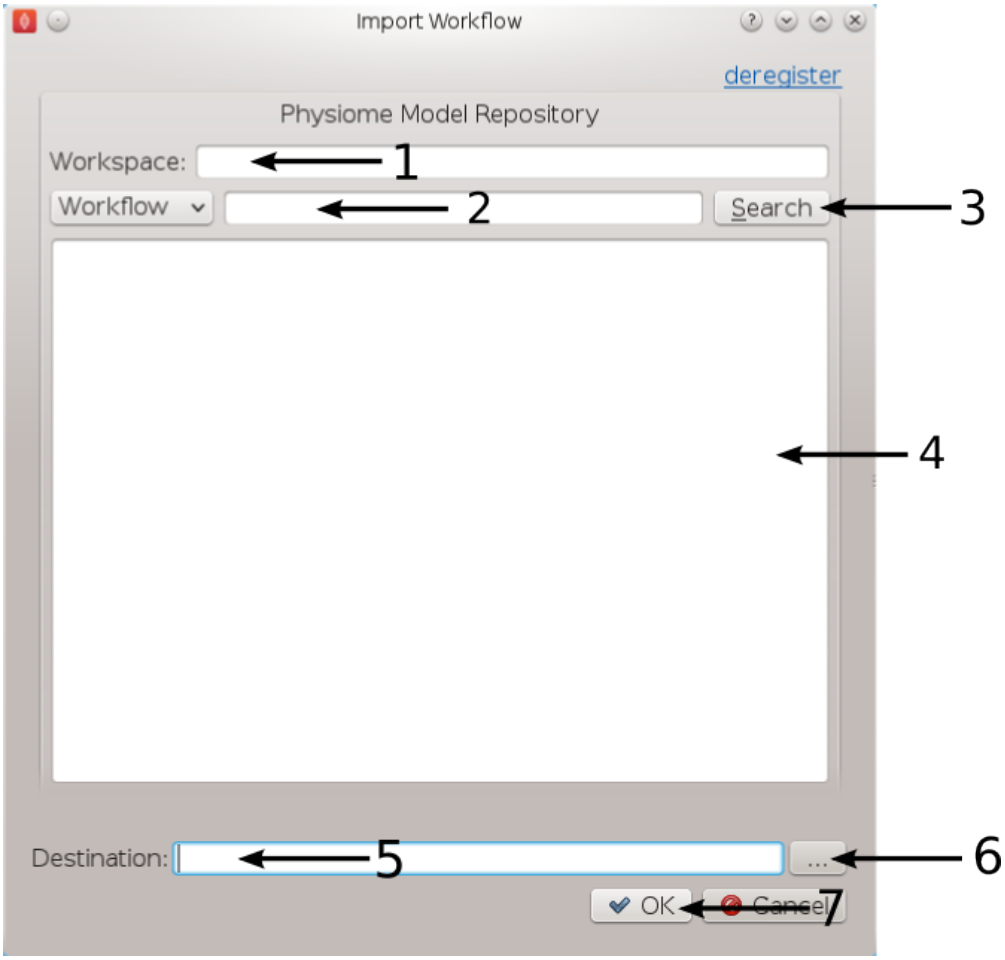


Fig. 1.4: **Figure 1**: Import Dialog [**1**] Workspace url, [**2**] Search text, [**3**] Search button, [**4**] Search results, [**5**] Destination directory, [**6**] Directory chooser button, [**7**] Confirm or cancel import.

We will leave the 'Search text' [**2**] blank and click the search button to search for all Workflows available on PMR. Once the search results are displayed in [**4**] select the entry with the title 'Workflow: BloodVesselAutoSegmentation'. This will put the Workspace url in the 'Workspace url' [**1**] box. Next use the 'Directory chooser button' [**7**] to choose a local directory for importing the 'Workflow' to. The chosen directory will be put in the 'Destination directory' [**5**]. When the 'Workflow url' and 'Directory destination' are correct press the 'Ok' button to complete the import.

Blood Vessel Automatic Segmentation Workflow

The Blood Vessel Automatic Segmentation Workflow consists of three workflow steps as shown in *Figure 2*.

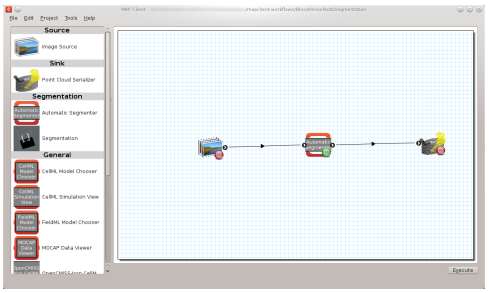


Fig. 1.5: **Figure 2**: Blood Vessel Automatic Segmentation Workflow.

1. An image source step.

   This step is designed to pass the location of an image or images through to another step. This step also has the functionality to download content from PMR.

2. An automatic segmentation step.

   This step takes in the location of the image set and reads in the images using OpenCMISS-Zinc. The images are analyzed, segmented and then discretized into a point cloud. The output from this step is a list of point locations in the image space.

3. A point cloud serialization step.

   This step serializes a list of point locations to disk.

Before the workflow can be executed each step in the workflow must be configured and the workflow saved. A gear icon in the bottom right hand corner of the step icon on the workflow canvas indicates whether the step is configured or not. A red icon represents an unconfigured step whilst a green icon represents a configured step. Clicking on the gear icon will display a configuration dialog for the step (if the step requires manual configuration). When a step has been configured correctly the green icon will be displayed. For our workflow we need to configure the image source step and the point cloud serialization step. A detailed discussion on configuring all the steps in this workflow is given below.

Workflow Configuration

This section describes how each step should be configured.

Image Source Step

The image source step requires a unique identifier for the step to be set. It also requires either a location on the local disk where the image data is located or a PMR workspace url from which the image data may be downloaded.

This step configuration makes use of the PMR search widget which gives us the ability to search available workspaces on PMR. We will make use of this functionality in this example. In the image source step configuration dialog seen in *Figure 3* we can see that there is a place to set a unique identifier for the step and also two

tabs, one tab is for setting the image dataset location on the local disk and the other tab is for searching PMR workspaces for image data. We will leave the local disk edit box on the local file system tab empty and allow the configuration to set the default location.

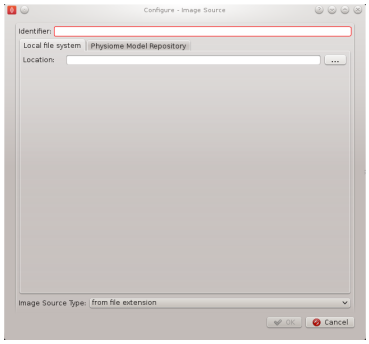


Fig. 1.6: **Figure 3**: Image source step configuration dialog.

Set the identifer edit box to bv_images and select the Physiome Model Repository tab so that we can search PMR for our images. On this tab we see We are going to conduct an ontological term search for our images, we are looking for some images that show an anyeurism in the anterior communicating artery. To do this we can start entering the text anterior communicating artery into the search term edit box [**3**], when we pause in our typing the dialog will query the PMR OWL terms for suitable matches. We will see results similar to what is shown in *Figure 5*, we can click on the matching term in this list and the correct reference will be added to the search term edit box [**3**] for us.

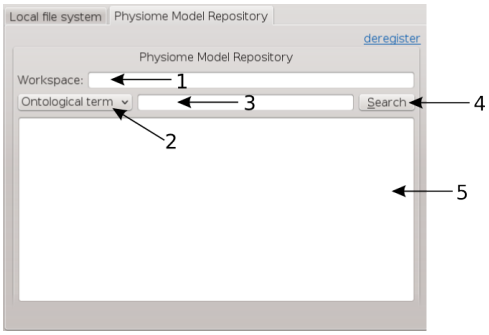


Fig. 1.7: **Figure 4**: PMR search tab, [**1**] Workspace url, [**2**] Search type combobox, [**3**] Search term, [**4**] Search button, [**5**] Search results.

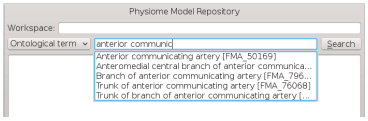


Fig. 1.8: **Figure 5**: PMR OWL terms.

With the correct term in place we can click the search button to return matching results from PMR. We will get back a single result Blood Vessel in MR Images. When we select this result in the search results list [**5**] the url for the workspace will be loaded into the workspace url edit box [**1**]. We should now have the dialog looking similar to *Figure 6*.

This completes the configuration of the image source step. When we click Ok in the dialog the images will be downloaded to the default directory on our local disk.

We can also use the combobox at the bottom of the dialog (*Figure 3*) to set the image type however this is only necessary if the image type cannot be determined through the filename extension. In our case we can leave this as it is.

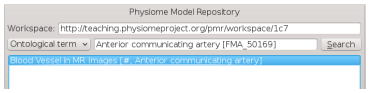


Fig. 1.9: **Figure 6**: Completed Physiome Model Repository search tab.

Alternatively, if PMR is unavailable copy the images from a usb memory stick into a directory on your computer, set the location on the local file system tab to this directory.

Automatic Segmentation Step

The automatic segmentation step does not require any configuration. Whilst this makes the configuration stage unnecessary it limits the usefulness or this step since it is configured to work for a only one set of images. An obvious enhancment to this step would be to expose the configurable properties of the segmentation to the user.

Point Cloud Serialization Step

The point cloud serialization step only requires the identifier for the step to be specified. The identifier will also be used to create an output directory of the same name and the serialization of the input data will be placed into a file under this directory. Set the identifier to 'bv_point_cloud' (as in *Figure 7*).

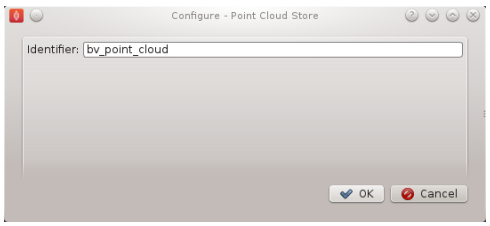


Fig. 1.10: **Figure 7**: Point cloud configuration dialog.

Execute the Workflow

Once all the workflow steps have been correctly configured save the workspace. We can do this through the File menu and selecting the save entry or by using the keyboard shortcut 'ctrl+s'. Because we have a workflow based on a version control system the commit dialog will appear so that we can keep a record of the changes. *Figure 8* shows this dialog, here we want to choose the skip commit option to save our workflow. In this example we are not going to commit our changes back to the workspace on PMR.

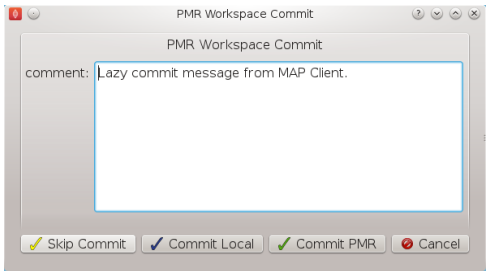


Fig. 1.11: **Figure 8**: PMR workspace commit dialog.

At this point we are ready to begin executing the workflow. To do this we click the execute button in the lower right hand corner of the window.

Execution

Once the execute button has been clicked the workflow will start to traverse the underlying directed graph, in our case starting from the image source step. In this simple workflow the only interactive step is the automatic segmentation step which displays a visualisation of the segmentation.

The automatic segmentation step shows a 3D interactive scene, where we can use the mouse to change the view of the scene. A brief description of some of the possible mouse actions is given here, the left-mouse button will rotate the scene, the right-mouse button will zoom the scene and the middle-mouse button will translate the scene. We also have some controls to show and hide the graphical elements in the scene and a slider that will change the z-component of the image plane. *Figure 9* shows the segmentation step interactive scene.

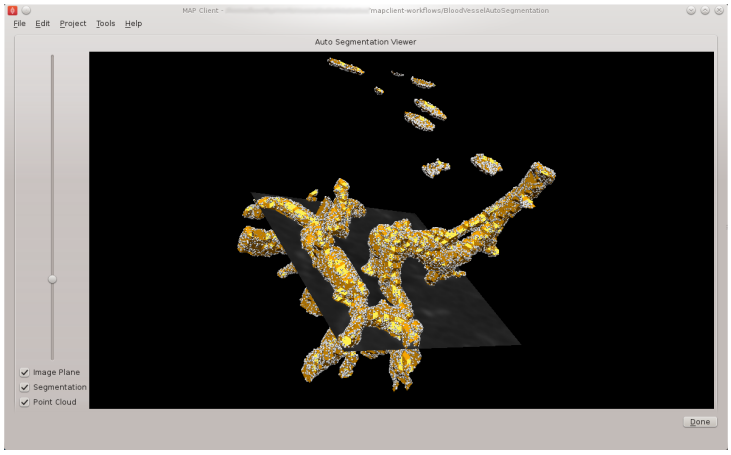


Fig. 1.12: **Figure 9**: Autosegmentation step screen.

To continue with the execution of the workflow click the done button in the lower right hand corner. When the workflow has finished executing all the steps in the workflow the workflow editor screen will be shown.

Check Output

We can now examine the output of the workflow using any text editor. The output is stored in a file called point_cloud.txt in a directory bv_point_cloud which can be found under the workflow directory.

Manually digitising an image stack

The purpose of this task is to demonstrate the coolest step that currently exists for the MAP Client and to highlight the reusable nature of the plugins/steps. Be sure to reference the github collection of steps that is being populated and the idea that people should contribute their steps even if they think they are specific to their own work.

1. Configure the image stack to load
2. Bring up the manual digitisation editor
    - Probably need to provide enough detail that people will be able to use it, possibly link to the relevant documentation in the MAP Client docs?
3. Digitise some points
4. Explain how to load up some points that have already been defined (like wow!)
5. Digitise some more points
6. Handover to the output to text file step.
7. and the visualise in Zinc step

8. Explain how those two output steps are identical to those from the previous task and talk about step-reuse, sharing knowledge, reproducibility, etc.

Preliminary CellML simulation step

What is the state of the CellML-based demo from the CellML workshop? can that be revived and added in here?

Creating your own step

Is it step or plugin?

The purpose of this step is to demonstrate how to define your own step. This will probably be a link to the appropriate section in the MAP Client docs. Using a nice simple process enables the user to intuitively know what output to expect for a given input so it makes it easy for them to play with things, predict the outcome, and check the actual outcome matches their prediction.

As well as linking to the docs for the standard how to create a step, be sure to encourage them to look into the source for the steps used in the tasks above - which are all available somewhere...

CHAPTER 2

Auckland Physiome Repository

The documentation found here is mainly aimed towards providing information to users of the Auckland Physiome Repository. This includes users interested in obtaining and running models from the respository, and those who wish to add models to the repository.

If you wish to deploy an instance of the repository software, *PMR2*, please see the buildout repository on GitHub.

Auckland Physiome Repository - an introduction

The Auckland Physiome Repository, includes the CellML and FieldML repositories, and is powered by software called PMR2. PMR2 relies on the distributed version control system Mercurial (Hg), which allows the repository to maintain a complete history of all changes made to every file contained within repository *workspaces*. In order to use the Physiome Model Repository, you will need to obtain a Mercurial client for your operating system, and become familiar with the basic functions of Mercurial. There are many excellent resources available on the internet, such as Mercurial, the definitive guide. Mercurial clients may be downloaded from the Mercurial website, which also provides documentation on Mercurial usage. A graphical alternative to a command-line client is available for Windows, called TortoiseHg. This provides a Windows explorer integrated system for working with Mercurial repositories.

Downloading and viewing models from the Auckland Physiome Repository

There are several ways of obtaining and using models from the Auckland Physiome Repository, and which you choose will depend on the way you intend to use the models. If you are simply interested in running a particular model and viewing the output, you can use links found on model *exposure* pages to get hold of the model files. There links available for a large number of models that will load the model directly into the OpenCell application, allowing you to explore simulation results with the help of a model diagram.

If you intend to use the model for further work, for example saving changes to the model or creating a new model based on an existing model or parts of an existing model, you should use *Mercurial* to obtain the files. In this way you also obtain the complete revision history of the files, and can add to this history as you make your own changes.

Searching the repository

The Auckland Physiome Repository has a basic search function that can be accessed by typing search terms into the box at the top right hand side of the page. You can use keywords such as `cardiac` or `insulin`, author names, or any other terms relevant to the models you want to find.

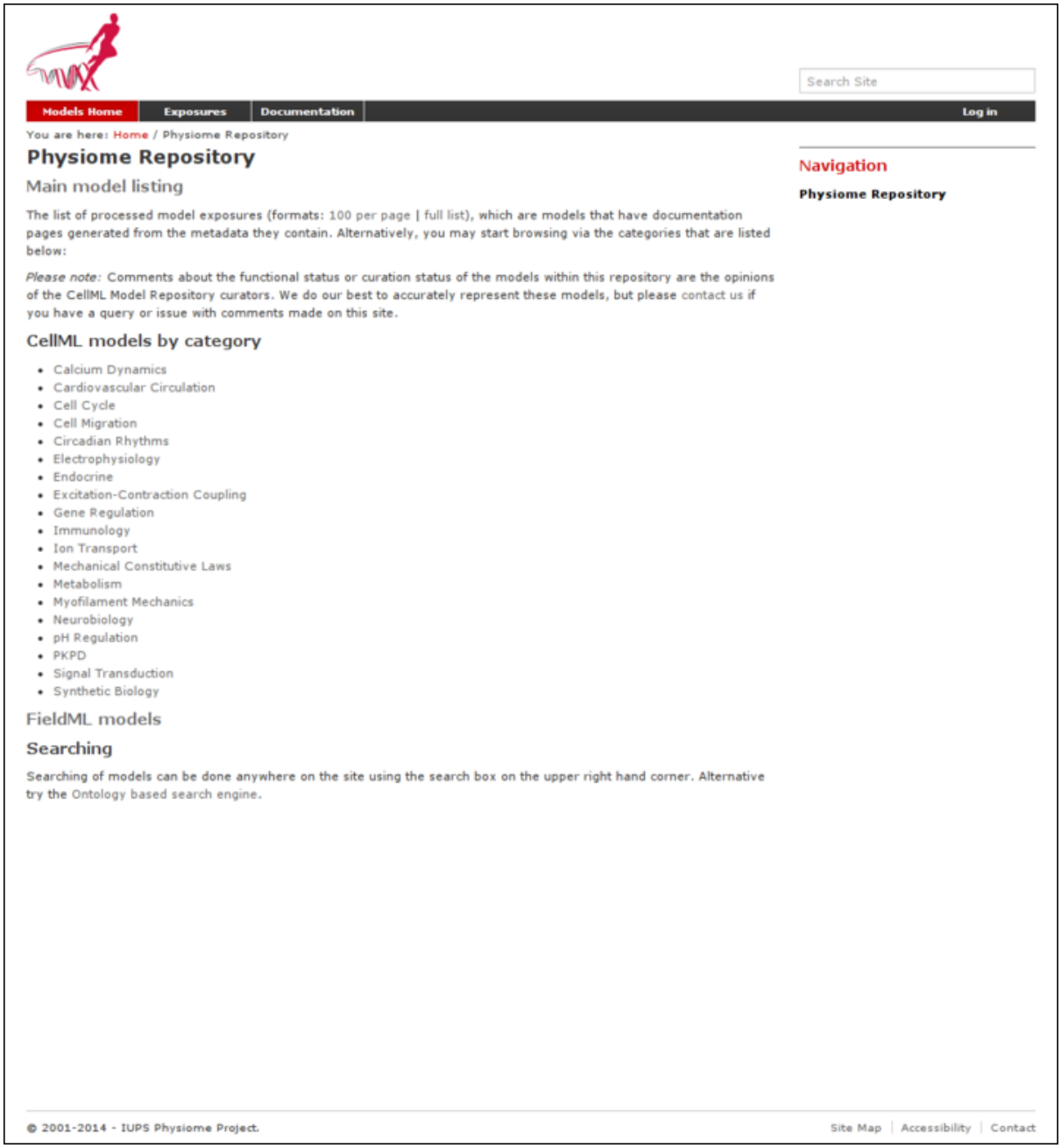


Fig. 2.1: The index page of the model repository provides two methods for finding models. There is a box for entering search terms, or you can click on categories based on model keywords to see all models in those categories.

If your search is yielding too many results, you may either try to narrow it down by choosing more or different keywords (*eg.* `goldbeter 1991` instead of just `goldbeter`), or you can click the *Advanced Search* link just under the search box on the results page. This will take you to a search page where you can select specific item types (*eg.* exposures or workspaces).

Once you have found the model you are interested in, there are several ways you can view or download it.

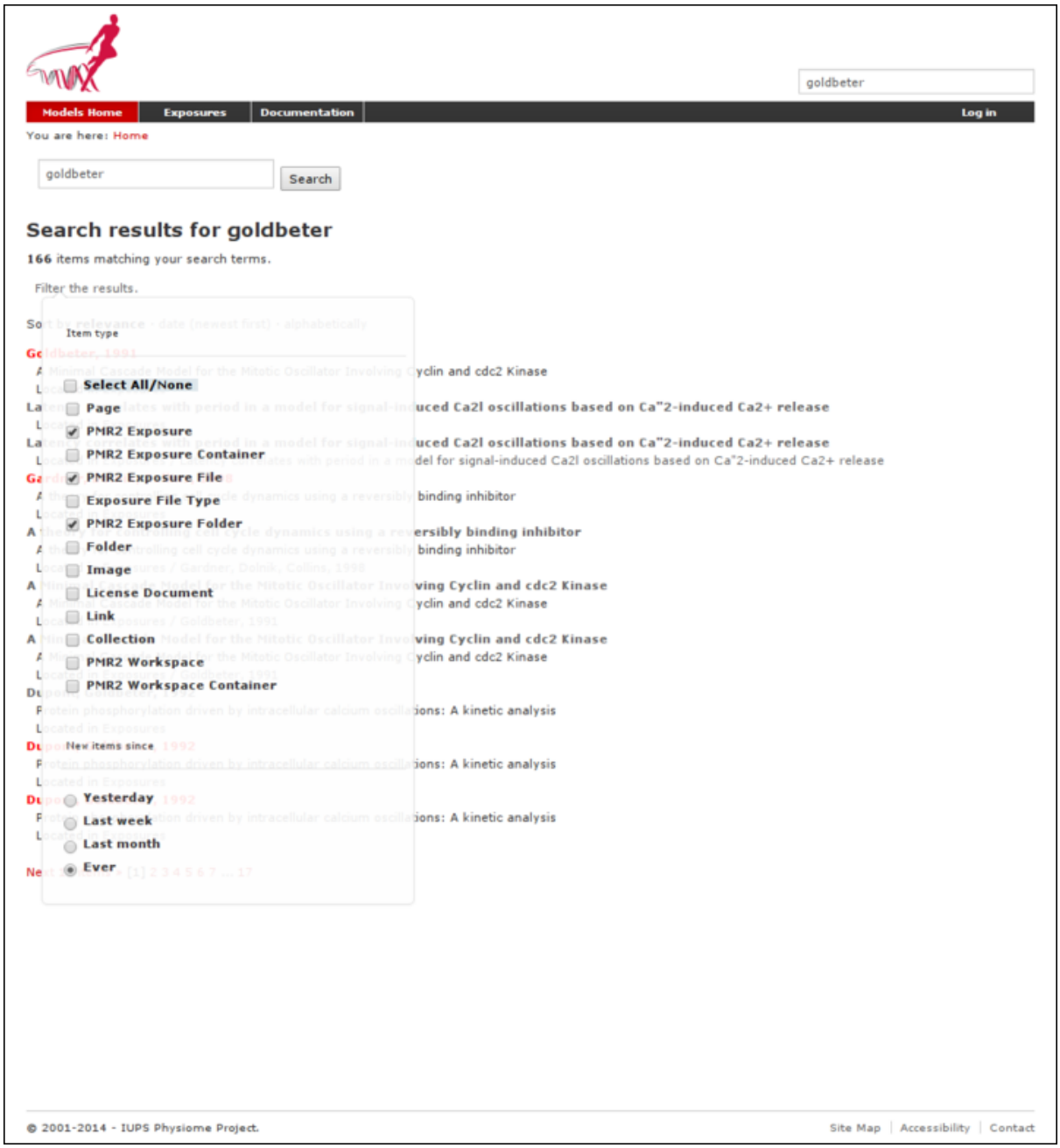


Fig. 2.2: In this search only exposure related items are to be shown in the results.

Viewing models via the respository web interface

The most common use of the Auckland Physiome Repository web interface is probably to view information about models found on exposure pages, and to then download the models from these pages for simulation in a CellML supporting application.

Below is an example of a CellML exposure page. It contains documentation about the model(s), a diagram of the what the model(s) represent, and a navigation pane that allows the user to select between available versions of the model. Many models only have one version, but in this case there are two variants.

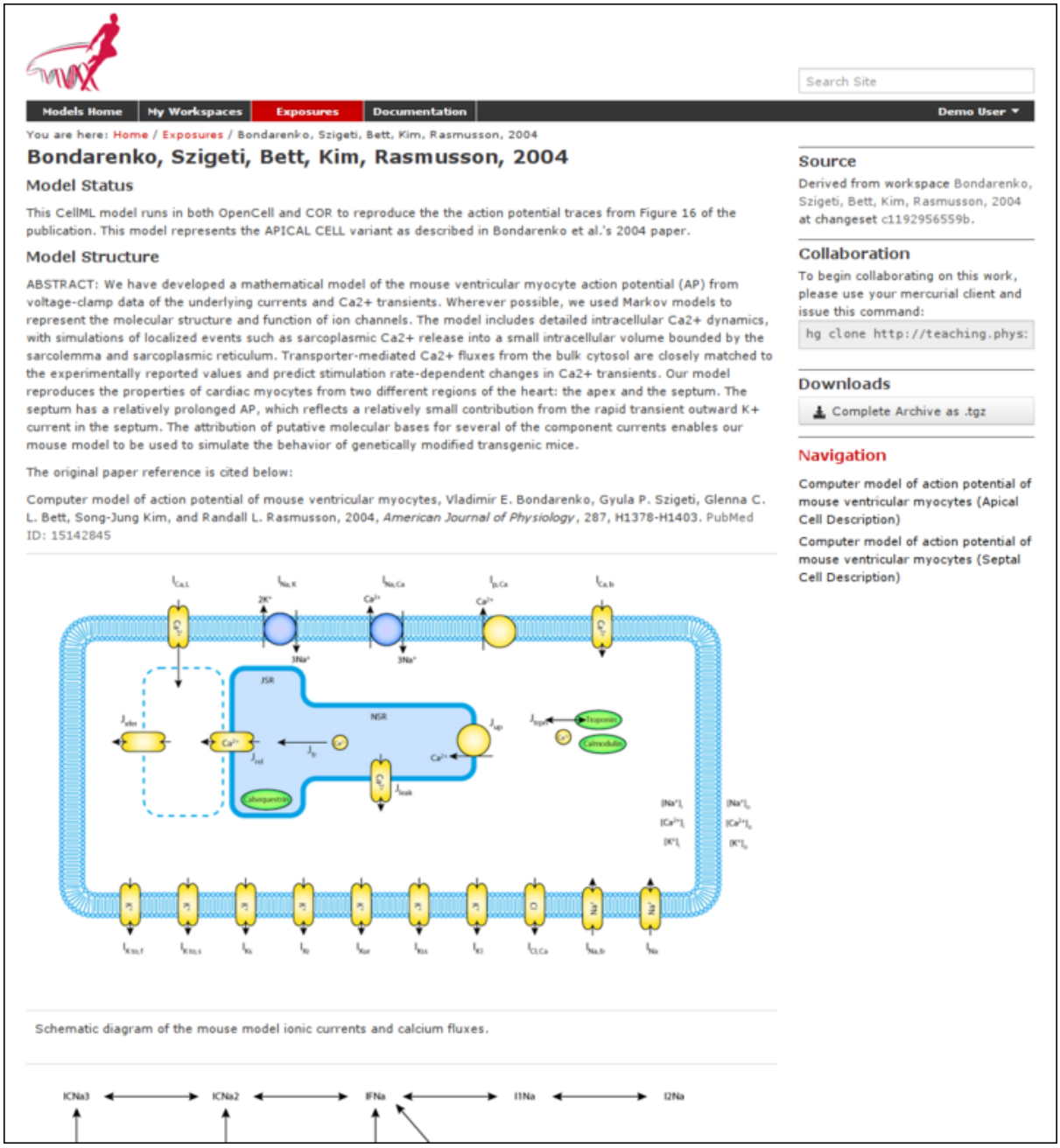


Fig. 2.3: An example of a CellML exposure page.

If you click on one of the model variant navigation links, you will be taken to a sub-page of the exposure which will allow you to view the actual CellML model in a number of ways.

On this page there are a number of options under a *Views available* panel at the right hand side.

- *Documentation* - displays the model documentation, already visible in the main area of the exposure page.

Search Site

Models Home | My Workspaces | Exposures | Documentation | Demo User

You are here: Home / Exposures / Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004 / Computer model of action potential of mouse ventricular myocytes (Apical Cell Description)

Computer model of action potential of mouse ventricular myocytes (Apical Cell Description)

by Dougal Cowan — last modified Jul 15, 2013 01:34 PM — History

Computer model of action potential of mouse ventricular myocytes

Model Status

This CellML model runs in both OpenCell and COR to reproduce the the action potential traces from Figure 16 of the publication. This model represents the APICAL CELL variant as described in Bondarenko et al.'s 2004 paper.

Model Structure

ABSTRACT: We have developed a mathematical model of the mouse ventricular myocyte action potential (AP) from voltage-clamp data of the underlying currents and Ca2+ transients. Wherever possible, we used Markov models to represent the molecular structure and function of ion channels. The model includes detailed intracellular Ca2+ dynamics, with simulations of localized events such as sarcoplasmic Ca2+ release into a small intracellular volume bounded by the sarcolemma and sarcoplasmic reticulum. Transporter-mediated Ca2+ fluxes from the bulk cytosol are closely matched to the experimentally reported values and predict stimulation rate-dependent changes in Ca2+ transients. Our model reproduces the properties of cardiac myocytes from two different regions of the heart: the apex and the septum. The septum has a relatively prolonged AP, which reflects a relatively small contribution from the rapid transient outward K+ current in the septum. The attribution of putative molecular bases for several of the component currents enables our mouse model to be used to simulate the behavior of genetically modified transgenic mice.

The original paper reference is cited below:

Computer model of action potential of mouse ventricular myocytes, Vladimir E. Bondarenko, Gyula P. Szigeti, Glenna C. L. Bett, Song-Jung Kim, and Randall L. Rasmusson, 2004, *American Journal of Physiology*, 287, H1378-H1403. PubMed ID: 15142845

Schematic diagram of the mouse model ionic currents and calcium fluxes.

Model Curation

Curation Status
JSim
COR
OpenCell

Source

Derived from workspace Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004 at changeset c1192956559b.

Collaboration

To begin collaborating on this work, please use your mercurial client and issue this command:

```
hg clone http://teaching.phys:
```

Downloads

Complete Archive as .tgz
Download This File

Views Available

Documentation
Model Metadata
Model Curation
Mathematics
Generated Code
Cite this model
Source View
Simulate using OpenCell

License

This work is licensed under a Creative Commons Attribution 3.0 Unported License.

Navigation

Computer model of action potential of mouse ventricular myocytes (Apical Cell Description)

Computer model of action potential of mouse ventricular myocytes (Septal Cell Description)

Fig. 2.4: An example of a CellML exposure sub-page.

- *Model Metadata* - displays information such as the citation information, model authorship details, and keywords.
- *Model Curation* - displays the curation stars for the model, also visible at the top right of the page. Future additions to the curation system mean that there will be additional information to be displayed on this page.
- *Mathematics* - displays all the equations in the model in graphical form.
- *Generated code* - shows a page where you can view the model in a number of different languages; C, C_IDA, Fortran 77, MATLAB, and Python. You can copy the generated code directly from this page to paste into your code editor.
- *Cite this model* - this page provides generic information about how to cite models in the repository.
- *Source View* - provides a raw view of the CellML (XML) model code.
- *Simulate using OpenCell* - this link will download the model and open it with OpenCell if you have the software installed. If the model has a session file, this will include an interactive diagram which can be clicked on to display traces of the simulation results.

The OpenCell session that is loaded when clicking on the Simulate using OpenCell link looks something like this:

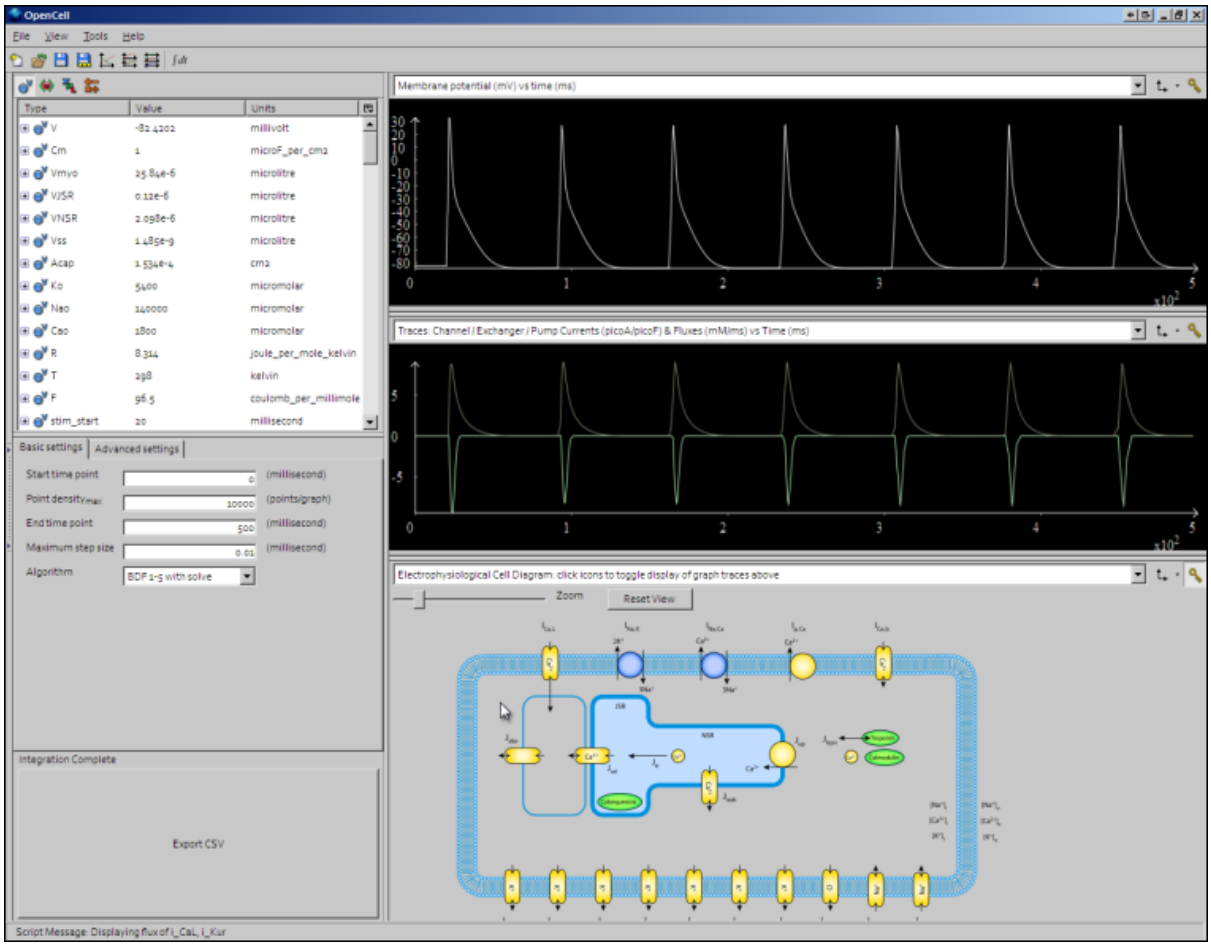


Fig. 2.5: An OpenCell session. Objects such as membrane channels in the diagram can be clicked - this will toggle the graph traces displaying the values for those objects.

Downloading models via Mercurial

All data in the Auckland Physiome Repository are stored in *workspaces* and each *workspace* is a *Mercurial* repository. The most comprehensive method of downloading content from Auckland Physiome Repository is to

clone the workspace containing the desired data. In this manner you will have a local copy of the entire history of that data, including all provenance data, and the ability to step back through the history of the workspace to a state that may not be available via the download links in the exposure pages discussed above. If you would like to modify the contents of workspace, making use of Mercurial will ensure accurate provenance records are maintained as well as all the other benefits of using a version control system.

As software tools like OpenCOR and MAP Client evolve, they will be able to hide a lot of the Mercurial details and present the user with a user interface suitable for their specific application areas. Directly using Mercurial is, however, currently the most powerful way to leverage the full capabilities of Auckland Physiome Repository.

If you are using the command line Mercurial client, you can easily clone the underlying repository for an exposure simply by selecting the text box inside the **Collaboration** portlet and paste that command into a terminal, or right click on the name of the workspace under the **Source** portlet and copy that URL and then paste that into your Mercurial client.

Detailed instructions for working with Mercurial can be found in the *CellML repository tutorial*.

Working with workspaces

Section author: David Nickerson

All models in the Auckland Physiome Repository exist in *workspaces*, which are *Mercurial* repositories that can be used to store any kind of file. Mercurial is a distributed version control system (DVCS).

In order to create your own workspaces, you will first need to create a repository account by registering at models.physiomeproject.org. Near the top right of the repository page there will be links labelled *Log in* and *Register*. Click on the register link, and follow the instructions.

Workspaces in the Auckland Physiome Repository are permanent once they are created. There is a teaching instance of the repository which may be used for *experimenting* with features of the software without worrying about creating permanent workspaces that might have errors in them. Users accounts and data from the Auckland Physiome Repository will be copied to the teaching instance periodically, overwriting all data there in the process, but users may register for an account just on the teaching instance if they prefer. Such accounts will need to be recreated each time the teaching instance is overwritten.

Note: The teaching instance of the repository is a mirror of the main repository site found at http://models.physiomeproject.org/, running the latest development version of *PMR2*.

Any changes you make to the contents of the teaching instance are not permanent, and will be overwritten with the contents of the main repository whenever the teaching instance is upgraded to a new release of PMR2. For this reason, you can feel free to experiment and make mistakes when pushing to the teaching instance. Please subscribe to the cellml-discussion mailing list to receive notifications of when the teaching instance will be refreshed.

See the section *Migrating content to the main repository* for instructions on how to migrate any content from the teaching instance to the main (permanent) Auckland Physiome Repository.

Creating a new workspace

Once a user is logged into Auckland Physiome Repository, they will be presented with a *My Workspaces* link in the top toolbar, as shown below:

The first paragraph includes a link to your dashboard to add a new workspace, shown below:

Currently *Mercurial* is the only avialable option for the storage method for a new workspace, but this may be expanded to include other storage methods in future. A workspace should be given a meaningful title and a brief description to help locate the workspace using the repository search. Both these fields can be edited later, so don't worry if you don't get it perfect the first time.

Clicking the *Add* button with then create the workspace, which will initially be empty, as shown below:

Search Site

Models Home | My Workspaces | Exposures | Documentation | Demo User

You are here: Home / My Workspaces

My Workspaces

by admin — last modified Sep 21, 2013 03:24 AM

A listing of workspaces that are created/owned by you.

The following is a list of workspaces owned by you within the entire model repository. Useful tools:

- **Workspace creation form.**

If you wish, you may view the full list of workspaces that are in this repository that are accessible by you. This list includes workspaces within the global workspace container.

Adrian, Chandler, Hodgkin, 1970 — by Demo User — last modified Aug 22, 2014 02:05 PM
Chay, 1997 — by Demo User — last modified Aug 22, 2014 02:06 PM
Cooling, Hunter, Crampin, 2007 — by Demo User — last modified Aug 22, 2014 02:06 PM
Experiment reproduction model — by Demo User — last modified Aug 22, 2014 02:24 PM
Leloup, Goldbeter, 1998 — by Demo User — last modified Aug 22, 2014 02:25 PM
Leloup, Goldbeter, 2004 — by Demo User — last modified Aug 22, 2014 02:25 PM
Luo, Rudy, 1991 — by Demo User — last modified Aug 22, 2014 02:06 PM
Metadata demonstration — by Demo User — last modified Aug 22, 2014 02:24 PM
Tentusscher, Noble, Noble, Panfilov, 2004 — by Demo User — last modified Aug 22, 2014 02:08 PM

© 2001-2014 - IUPS Physiome Project.

Site Map | Accessibility | Contact


Search Site
Models Home
My Workspaces
Exposures
Documentation
Demo User
You are here: Home / Workspaces
Contents
View
Create a New Workspace
Title
Description
Storage Method
The type of storage backend used for this workspace.
Mercurial
Add
© 2001-2014 - IUPS Physiome Project.
Site Map
Accessibility
Contact

Search Site

Models Home | My Workspaces | Exposures | Documentation | Demo User

You are here: Home / Workspaces / Demonstration workspace for a model

View | Edit | History | Files | Fork | Synchronize | Exposure Rollover | RDF Indexing | Sharing | Layout | Actions | State: Private

Demonstration workspace for a model

Exposure Information

No simplified view available for this workspace as no related exposures were found.

Workspace Summary

Description

This is a workspace to demonstrate features that will be documented in the user documentation.

Owner

Demo User <demouser@example.com>

URI for mercurial clone/pull/push

http://teaching.physiomeproject.org/workspace/1bd

Files

Filename	Size	Date	Options

Site Map | Accessibility | Contact

In the figure above, the URI of the newly created workspace has been highlighted. This is the URI that will be used when operating on the workspace using Mercurial.

Working with collaborators

The repository makes use of *Mercurial* to manage individual workspaces. Mercurial is a Distributed Version Control System (DVCS), and as such encourages collaborative development of your model, dataset, results, *etc*. Using Mercurial, each member of the development team is able to have their own clone of the workspace which can be kept synchronized with the other members of the development team, while ensuring that each team member's contributions are accurately recorded in the workspace history.

Once a *workspace* has been published, any registered users (or members) of the repository is able to access and clone the workspace, including team members and the anonymous public. Only the owner and those with privileges granted by the owner are able to make changes to the workspace, including *pushing* changes into the Mercurial repository. Private workspaces, however, can only be viewed by its owner and those with viewing privileges granted by its owner.

Auckland Physiome Repository provides access controls to manage the ability of its members and anonymous users to interact with workspaces. The access control is managed via the *Sharing* tab for a given workspace, as shown below.

By default, you will initially see that all logged-in user has the **Can add** permission. That is the inherited permission from the global workspace container, and does not imply that they can view your work as that is determined by the **Can view** permission. This also does not mean that they can add data to your workspace. This permission setting is applied to the default workspace container so that you and all other users of the system have the ability to create new workspaces.

PMR2 has the option to provide individual containers per user for their private workspaces, but this option is now disabled in the Auckland Physiome Repository.

You can disable the inherited higher level permissions from your workspace by unchecking the **Inherit permissions from higher levels** checkbox, if you wish, but the administrators of the repository can access your workspace regardless if you wish for them to aid you with your workspace. Using the *Sharing* tab you are able to search for other members, such as the names of people in your development team. These members would then appear in the list of members and you are able to set their access as required.

Using the *Sharing* controls there are currently four possible permissions that can be controlled. The **Can add** and **Can edit** permissions relate to the object that represents the workspace in the website database and are generally left in the default state. When selected for a given member, the **Can view** permission allows that member to view the workspace on the website, even if the workspace is private. Similarly, when the **Can hg push** permission is enabled the selected member is able to *push* into the workspace - this is the most important permission as enabling this allows members to add, modify, and delete the actual content of the workspace. One benefit of using Mercurial means that even if one of the privileged members accidentally modifies the workspace in a detrimental manner, it is possible to revert the workspace back to the correct state.

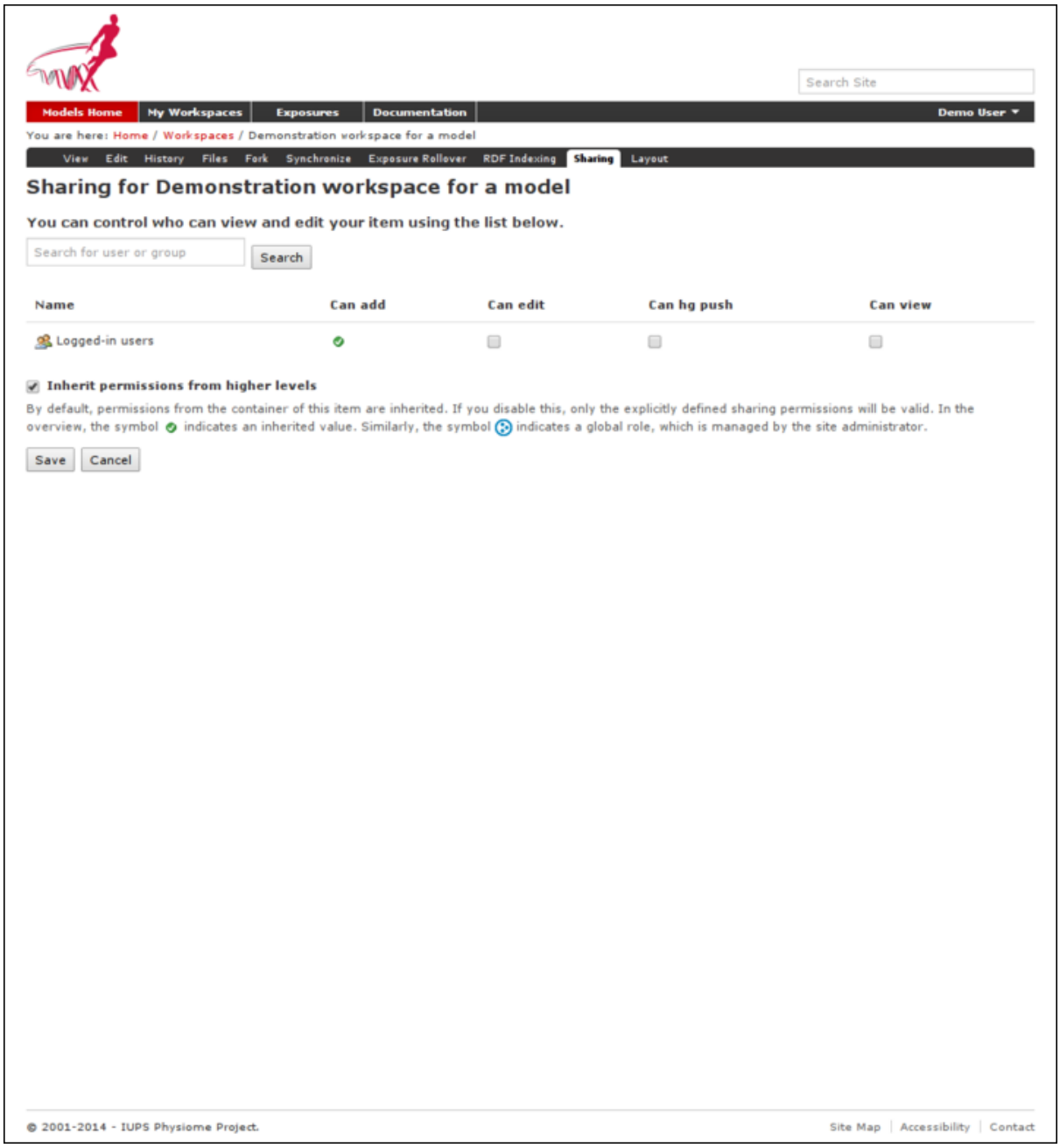
When working in a collaborative team you would generally enable the **Can hg push** and **Can view** permissions for all team members and only enable the **Can add** and **Can edit** permissions for the team members responsible for the workspace presentation in the website.

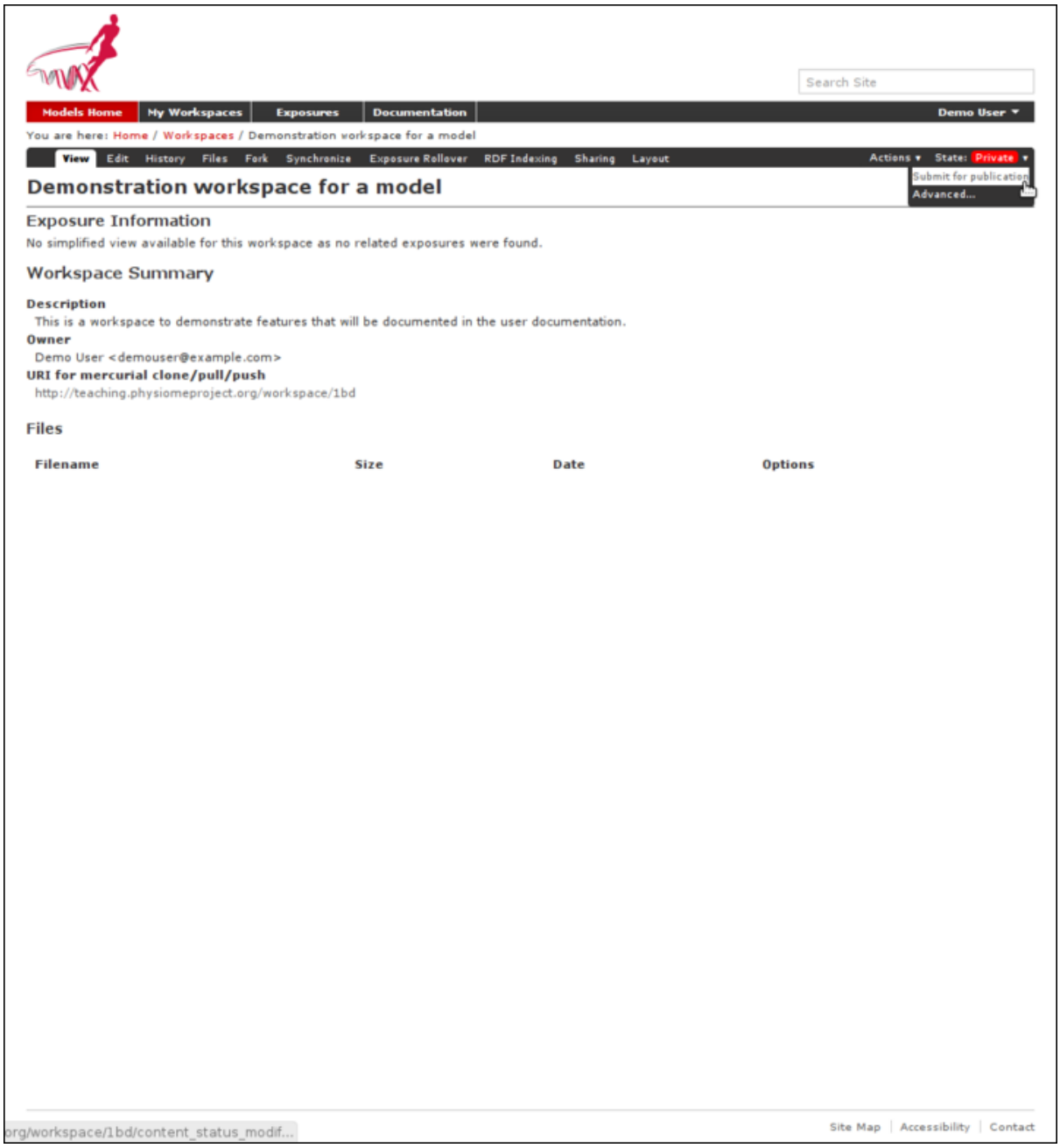
Alternatively, if you wish to make your work available for searching by any users, including the ones who do not have an account with the repository, you may do so by changing the workflow state from "private" to "submit for publication". This will put your workspace into the reviewer queue and they will turn it into the "published" state.

Uploading files to your workspace

The basic process for adding content to a *workspace* consists of the following steps:

1. *Clone* the workspace to your local machine.
2. Add files to cloned workspace.


Search Site
Models Home
My Workspaces
Exposures
Documentation
Demo User
You are here: Home / Workspaces / Demonstration workspace for a model
View
Edit
History
Files
Fork
Synchronize
Exposure Rollover
RDF Indexing
Sharing
Layout
Sharing for Demonstration workspace for a model
You can control who can view and edit your item using the list below.
Search for user or group
Search
Name
Can add
Can edit
Can hg push
Can view
Logged-in users
Inherit permissions from higher levels
By default, permissions from the container of this item are inherited. If you disable this, only the explicitly defined sharing permissions will be valid. In the overview, the symbol indicates an inherited value. Similarly, the symbol indicates a global role, which is managed by the site administrator.
Save
Cancel
© 2001-2014 - IUPS Physiome Project.
Site Map
Accessibility
Contact


Search Site
Models Home
My Workspaces
Exposures
Documentation
Demo User
You are here: Home / Workspaces / Demonstration workspace for a model
View
Edit
History
Files
Fork
Synchronize
Exposure Rollover
RDF Indexing
Sharing
Layout
Actions
State: Private
Submit for publication
Advanced...
Demonstration workspace for a model
Exposure Information
No simplified view available for this workspace as no related exposures were found.
Workspace Summary
Description
This is a workspace to demonstrate features that will be documented in the user documentation.
Owner
Demo User <demouser@example.com>
URI for mercurial clone/pull/push
http://teaching.physiomeproject.org/workspace/1bd
Files
Filename
Size
Date
Options
org/workspace/1bd/content_status_modif...
Site Map
Accessibility
Contact

3. Commit the files using a *Mercurial* client.
4. *Push* the workspace back to the repository.

An example demonstrating these steps can be found in in this tutorial step: Populate with content, or continue on to the *next section of this guide*.

Tutorial on using CellML with Auckland Physiome Repository

Section author: David Nickerson, Randall Britten, Dougal Cowan

About this tutorial

The Auckland Physiome Repository provides extensive support for CellML model and related files. Previously it was called the CellML Model Repository, this has since been merged completely along with the FieldML Model Repository into the unified repository. The underlying software is *PMR2*, which in turn relies on the distributed version control system *Mercurial* (Hg), which allows the repository to maintain a complete history of all changes made to every file it contains. This tutorial demonstrates how to work with the repository using TortoiseHg, which provides a Windows explorer integrated system for working with Mercurial repositories.

```
Brief mention of the equivalent command line versions of the
TortoiseHg actions will also be mentioned, so that these ideas can
also be used without a graphical client, and on Linux and similar
systems. These will be denoted by boxes like this.
```

This tutorial requires you to have:

- A Mercurial client such as TortoiseHg or Mercurial installed
- The OpenCell CellML modelling environment
- A text editor such as Notepad++ or gedit

Basic concepts

The Auckland Physiome Repository use a certain amount of jargon - some is specific to the repository software, and some is related to distributed version control systems (DVCSs). Below are basic explanations of some of these terms as they apply to the repository.

Workspace

A container (much like a folder or directory on your computer) to hold the files that make up a model, as well as any other files such as documentation or metadata, etc. In practical terms, each workspace is a Mercurial repository.

Exposure

An exposure is a publicly viewable presentation of a particular revision of a model. An exposure can present one or many files from your workspace, along with documentation and other information about your model.

The Mercurial DVCS has a range of terms that are useful to know, and definitions of these terms can be found in the Mercurial glossary: http://mercurial.selenic.com/wiki/Glossary.

Working with the repository web interface

This part of the tutorial will teach you how to find models in the Auckland Physiome Repository https://models.physiomeproject.org, how to view a range of information about those models, and how to download models. The first page in the repository consists of basic navigation, a link to the main model listing, a search box at the top right, and a list of model category links as shown below.

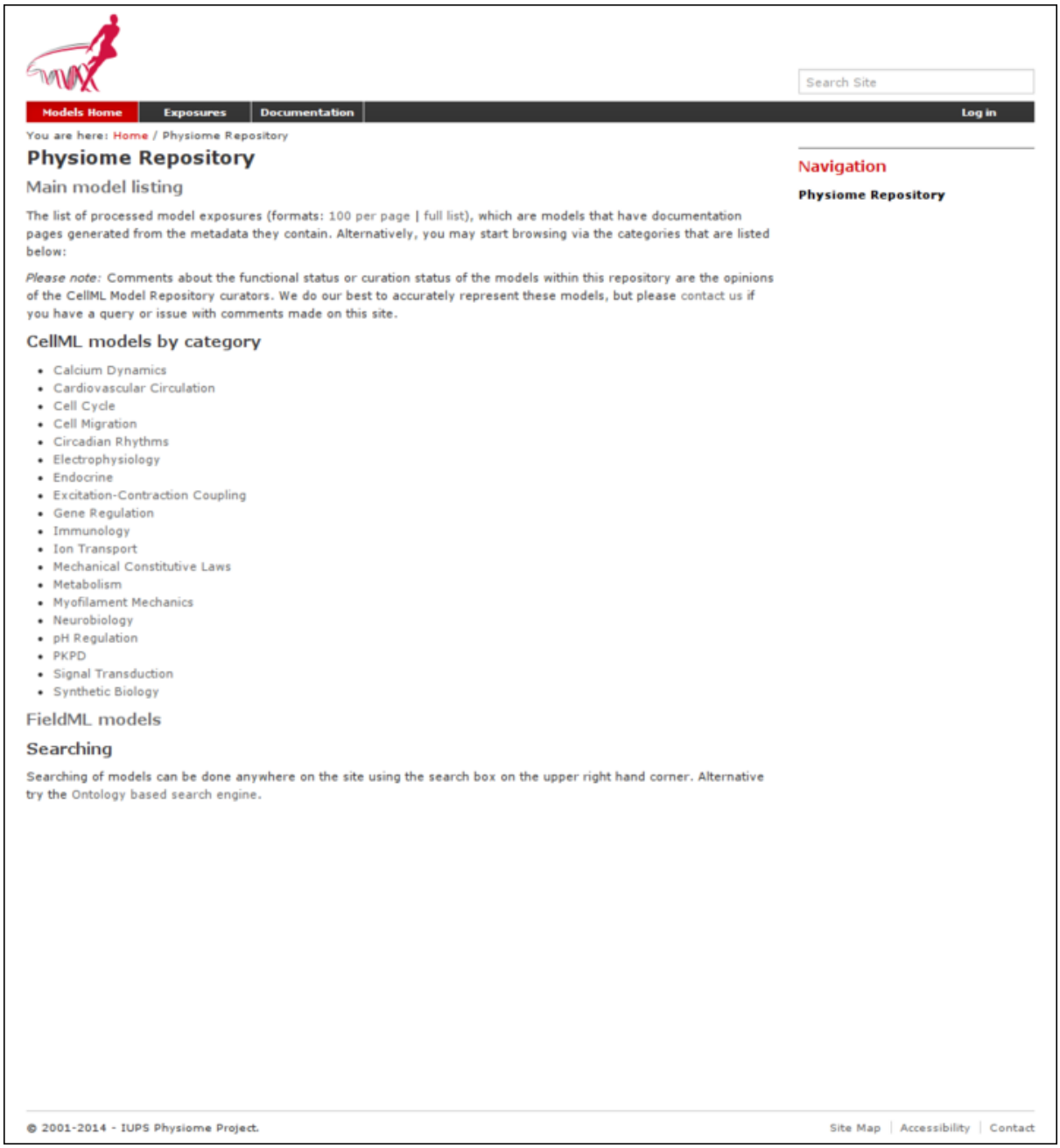


Fig. 2.6: The front page of the Auckland Physiome Repository.

Model listings

Clicking on the main model listing or any of the category listings will take you to a page displaying a list of exposed models in that category. Click on electrophysiology for example, and a list of over 100 exposed models in that category will be displayed, as shown here.

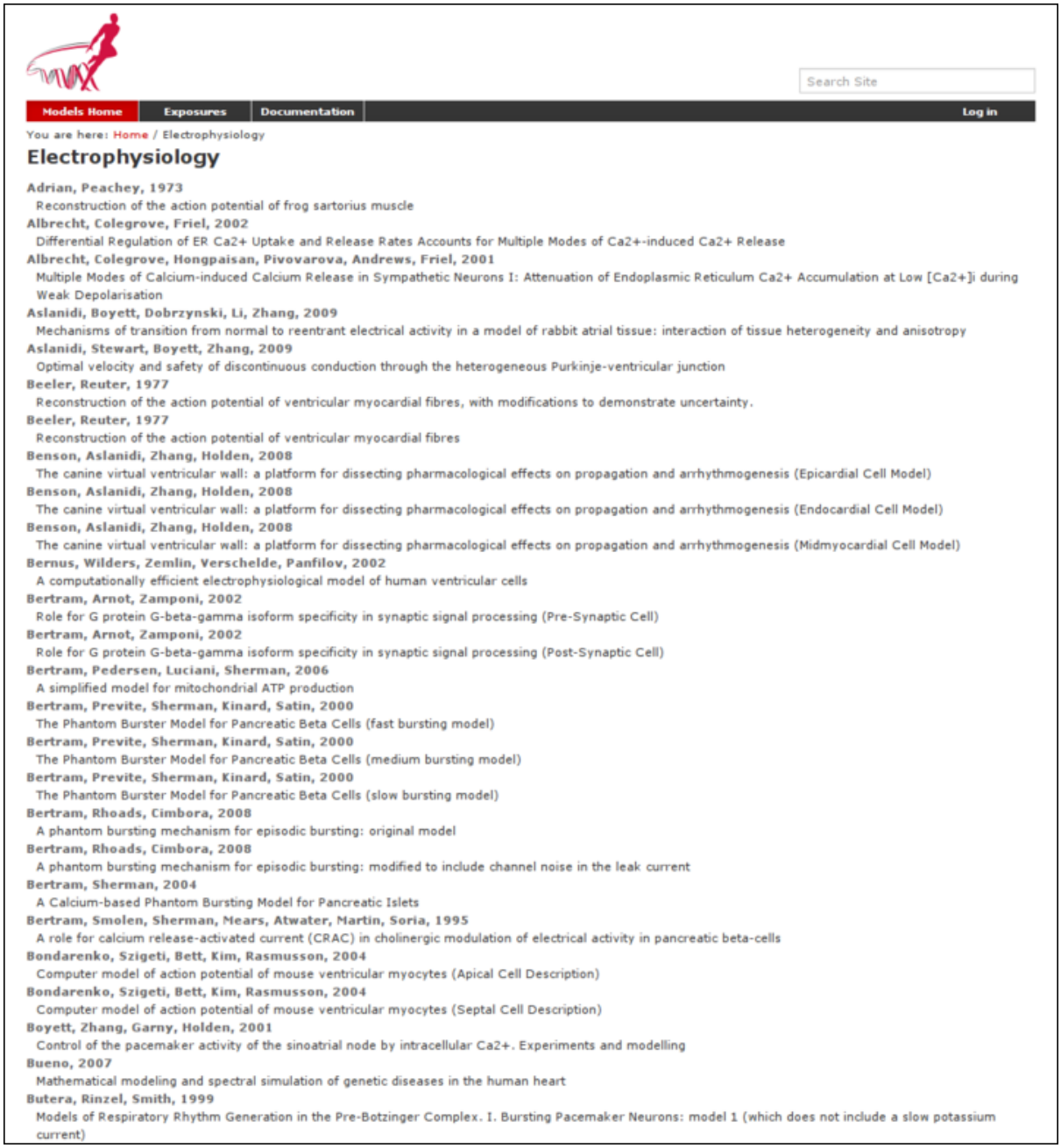


Fig. 2.7: A list of models in the electrophysiology category.

Clicking on an item in the list will take you to the exposure page for that model.

Searching the repository

You can search for the model that you wish to work on by entering a search term in the box at the top right of the page. Many of the models in the repository are named by the first author and publication date of the paper, so a good search query might be something like *goldbeter 1991*. A list of the results of your search will probably

contain both workspaces and exposures - you will need to click on the workspace of the model you wish to work on. Workspaces can be identified by where they are located, as they will be located inside **Workspaces**. In the following screenshot, the first two results are workspaces, and the remainder are exposures. Note that red links are exposures that are marked as expired.

Fig. 2.8: A search results listing on the Auckland Physiome Repository.

Click on an exposure result to view information about the model and to get links for downloading or simulating the model. Click on workspaces to see the contents of the model workspace and the revision history of the model.

Working with the repository using Mercurial

This part of the tutorial will teach you how to *clone* a workspace from the model repository using a Mercurial client, create your own workspace, and then push the cloned workspace into your new workspace in the repository.

We will be using a *fork* of an existing workspace, which provides you with a personal copy of a workspace that you can edit and push changes to.

Registering an account and logging in

First, navigate to the teaching instance of the Auckland Physiome Repository at http://teaching.physiomeproject.org/.

Note: The teaching instance of the repository is a mirror of the main repository site found at http://models.physiomeproject.org/, running the latest development version of *PMR2*.

Any changes you make to the contents of the teaching instance are not permanent, and will be overwritten with the contents of the main repository whenever the teaching instance is upgraded to a new release of PMR2. For this reason, you can feel free to experiment and make mistakes when pushing to the teaching instance. Please subscribe to the cellml-discussion mailing list to receive notifications of when the teaching instance will be refreshed.

See the section *Migrating content to the main repository* for instructions on how to migrate any content from the teaching instance to the main (permanent) Auckland Physiome Repository.

In order to make changes to models in the CellML repository, you must first register for an account. The *Log in* and *Register* links can be found near the top right corner of the page. Your account will have the appropriate access privileges so that you can push any changes you have made to a model back into the repository.

Click on the Register link near the top right, and fill in the registration form. Enter your username and desired password. After completing the email validation step, you can now log in to the repository.

Note: This username and password are also the credentials you use to interact with the repository via Mercurial.

Once logged in to the repository, you will notice that there is a new link in the navigation bar, My Workspaces. This is where all the workspaces you create later on will be listed. The Log in and Register links are also replaced by your username and a Log out link.

Mercurial username configuration

Important: Username setup for Mercurial

Since you are about to make changes, your name needs to be recorded as part of the workspace revision history. When commit your changes using Mercurial, it is initially "offline" and independent of the Auckland Physiome Repository. This means that you have to set-up your username for the Mercurial client software, even though you have registered a username on Auckland Physiome Repository.

You only need to do this once.

Steps for TortoiseHg:

- Right click on any file or folder in Windows Explorer, and select *TortoiseHg → Global Settings*.
- Select *Commit* and then enter your name followed by your e-mail address in "angle brackets" (i.e. less-than "<" and greater-than ">"). Actually, you can enter anything you want here, but this is the accepted best practice. Note that this information becomes visible publicly if the workspace that you push your changes to is public.

Steps for command line:

- **Edit the config text file:**
  - For per repository settings, the file in the repository: `<repo>\.hg\hgrc`
  - System-wide settings for Linux: `%USERPROFILE%\.hgrc`

  - System-wide settings for Windows: `%USERPROFILE%\mercurial.ini`

- Add the following entry:

```
[ui]
username = Firstname Lastname <firstname.lastname@example.net>
```

Forking an existing workspace

Important: It is essential to use a Mercurial client to obtain models from the repository for editing. The Mercurial client is not only able to keep track of all the changes you make (allowing you to back-track if you make any errors), but using a Mercurial client is the only way to add any changes you have made back into the repository.

For this tutorial we will *fork* an existing workspace. This creates new workspace owned by you, containing a copy of all the files in the workspace you forked including their complete history. This is equivalent to cloning the workspace, creating a new workspace for yourself, and then pushing the contents of the cloned workspace into your new workspace.

Forking a workspace can be done using the Physiome Model Repository web interface. The first step is to find the workspace you wish to fork. We will use the Beeler, Reuter 1977 *workspace* which can be found at: http://teaching.physiomeproject.org/workspace/beeler_reuter_1977.

Now click on the *fork* option in the toolbar, as shown below.

You will be asked to confirm the *fork* action by clicking the *Fork* button. You will then be shown the page for your forked workspace.

Cloning your forked workspace

In order to make changes to your workspace, you have to *clone* it to your own computer. In order to do this, copy the URI for mercurial clone/pull/push as shown below:

In Windows explorer, find the folder where you want to create the clone of the workspace. Then right click to bring up the context menu, and select *TortoiseHG* → *Clone* as shown below:

Paste the copied URL into the *Source:* area and then click the *Clone* button. This will create a folder called `beeler_reuter_1977_tut` that contains all the files and history of your forked workspace. The folder will be created inside the folder in which you instigated the clone command.

Command line equivalent

```
hg clone [URI]
```

You will need to enter your username and password to clone the workspace, as the fork will be set to *private* when it is created.

The repository will be cloned within the current directory of your command line window.

Making changes to workspace contents

Your cloned workspace is now ready for you to edit the model file and make a commit each time you want to save the changes you have made. As an example, open the model file in your text editor and remove the paragraph which describes validation errors from the documentation section, as shown below:

Save the file. If you are using TortoiseHg, you will notice that the icon overlay has changed to a red exclamation mark. This indicates that the file now has uncommitted changes.

Search Site

Models Home | My Workspaces | Exposures | Documentation | Demo User

You are here: Home / Workspaces / Beeler, Reuter, 1977

View | History | Files | Fork | Actions | Published

Beeler, Reuter, 1977

Exposure Information

Latest Exposure If you are a new user to the repository, you may wish to view the exposure for this workspace. An exposure will show the summarized information for the content contained here.

Workspace Summary

Owner
admin
URI for mercurial clone/pull/push
http://teaching.physiomeproject.org/workspace/beeler_reuter_1977

Files

Filename	Size	Date	Options
beeler_reuter_1977.ai	140995	2011-12-09 13:38 +1300	[browse]
beeler_reuter_1977.cellml	58593	2011-12-09 13:38 +1300	[browse]
beeler_reuter_1977.png	23661	2011-12-09 13:38 +1300	[browse]
beeler_reuter_1977.session.xml	7584	2011-12-09 13:38 +1300	[browse] [run]
beeler_reuter_1977.svg	133053	2011-12-09 13:38 +1300	[browse]
beeler_reuter_1977.xul	130673	2011-12-09 13:38 +1300	[browse]
beeler_reuter_1977_documentation.html	4469	2011-12-09 13:38 +1300	[browse]
cell_diagram.gif	202954	2011-12-09 13:38 +1300	[browse]
cellml_rendering.gif	204051	2011-12-09 13:38 +1300	[browse]

org/workspace/.../@@fork

Site Map | Accessibility | Contact

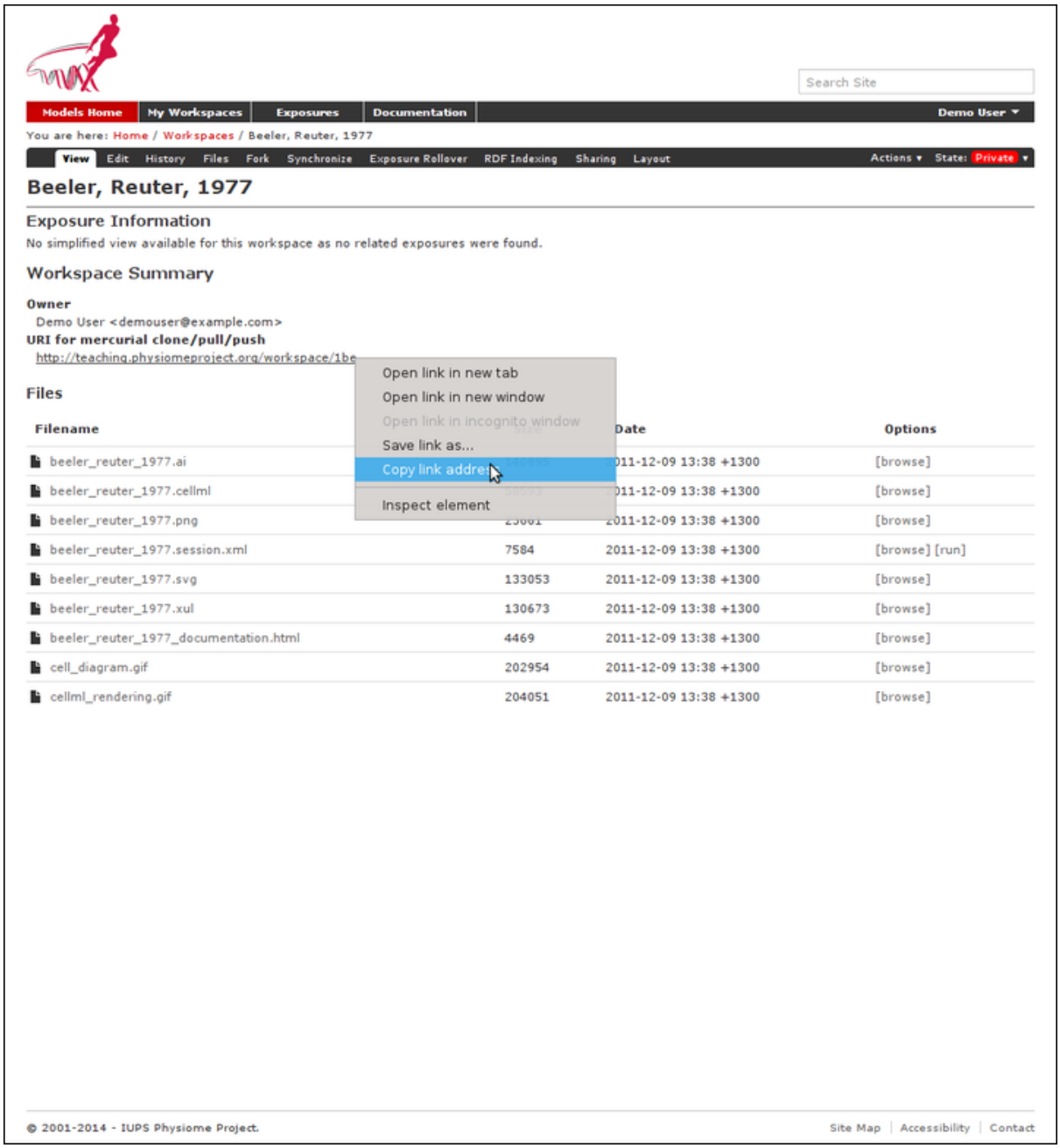
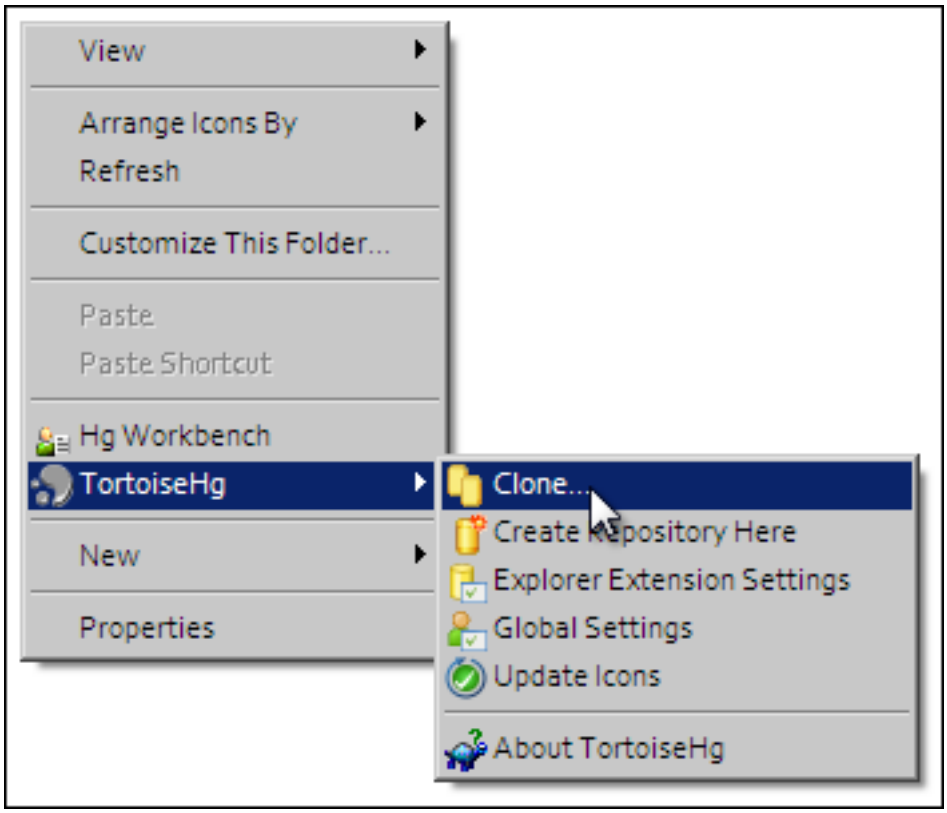
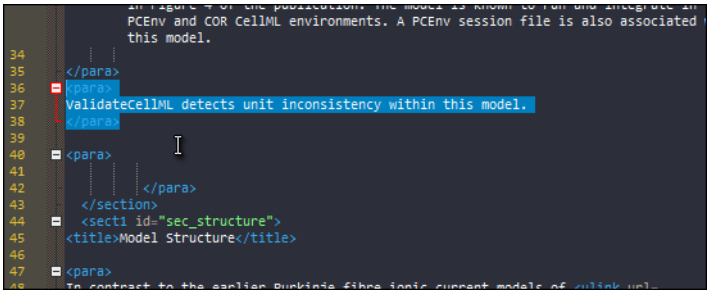


Fig. 2.9: Copying the URI for cloning your workspace.

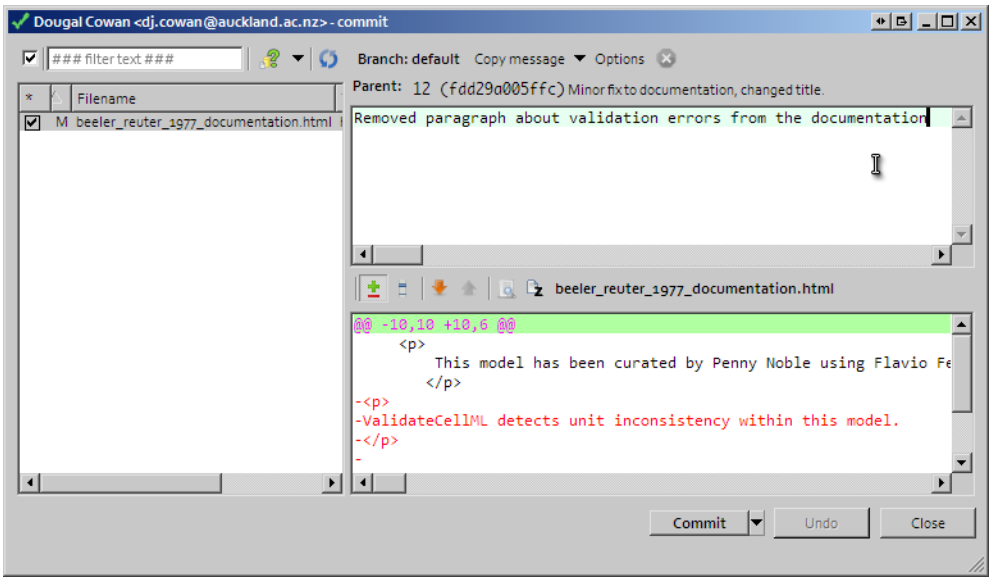

View
Arrange Icons By
Refresh
Customize This Folder...
Paste
Paste Shortcut
Hg Workbench
TortoiseHg
New
Properties
Clone...
Create Repository Here
Explorer Extension Settings
Global Settings
Update Icons
About TortoiseHg


PCEnv and COR CellML environments. A PCEnv session file is also associated
this model.
34
35 </para>
36 <para>
37 ValidateCellML detects unit inconsistency within this model.
38 </para>
39
40 <para>
41
42 </para>
43 </section>
44 <sect1 id="sec_structure">
45 <title>Model Structure</title>
46
47 <para>

Committing changes

If you are using TortoiseHg, bring up the shell menu for the altered file and select *TortoiseHg → Hg Commit*. A window will appear showing details of the changes you are about to commit, and prompting for a commit message. Every time you commit changes, you should enter a useful commit message with information about what changes have been made. In this instance, something like "Removed the paragraph about validation errors from the documentation" is appropriate.

Click on the Commit button at the far left of the toolbar. The icon overlay for the file will now change to a green tick, indicating that changes to the file have been committed.



Command line equivalent

```
hg commit -m "Removed the paragraph about validation errors from the documentation"
```

Pushing changes to the repository

Your cloned workspace on your local machine now has a small history of changes which you wish to *push* into the repository.

Right click on your workspace folder in Windows explorer, and select *TortoiseHg → Hg Synchronize* from the shell menu. This will bring up a window from which you can manage changes to the workspace in the repository. Click on the Push button in the toolbar, and enter your username and password when prompted.

Command line equivalent

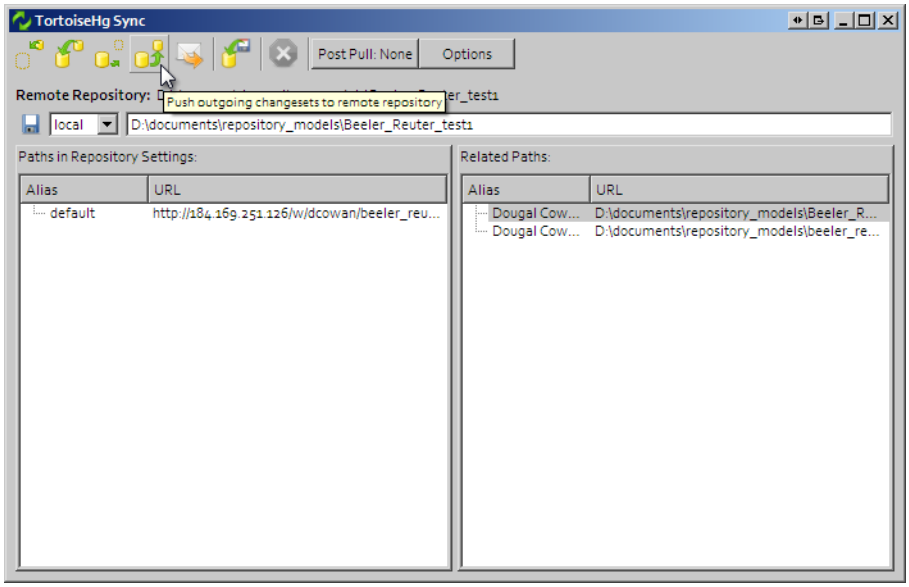
```
hg push
```

Now navigate to your workspace and click on the history toolbar button. This will show entries under the Most recent changes, complete with the commit messages you entered for each commit, as shown below:

Create an exposure

As explained earlier, an *exposure* aims to bring a particular revision to the attention of users who are browsing and searching the repository.

There are two ways of making an exposure - creating a new exposure from scratch, or "Rolling over" an exposure. Rolling over is used when a workspace already has an existing exposure, and the updates to the workspace have not fundamentally changed the structure of the workspace. This means that all the information used in making



the previous exposure is still valid for making a new exposure of a more recent revision of the workspace. Strictly speaking, an exposure can be rolled over to an older revision as well, but this is not the usual usage.

As you are working in a forked repository, you will need to create a new exposure from scratch. To learn how to create exposures, please refer to *Creating CellML exposures*.

Migrating content to the main repository

As noted above, the teaching instance used in this tutorial is not suitable for permanent storage of your work. One of the advantages of using a distributed version control system to manage *workspaces* is that it is straightforward to move the entire workspace, including the full history and provenance record, from one location to another. PMR2 also provides a feature that exports exposures so that they can then be imported into another PMR2 instance.

For example: if you would like to move your work in your workspace on the teaching instance into a *new* workspace on the Auckland Physiome Repository (or from one PMR2 instance to another), you should follow these steps:

1. Ensure that you have pushed all your commits to the source instance;
2. *Create the new workspace* in the destination repository;
3. Navigate to the workspace created and choose the *synchronize* action from the workspace toolbar, as shown below.
4. Fill in the URI of your workspace on the source instance (*e.g.,* http://models.physiomeproject.org/w/andre/cortassa-ECME-2006)
5. Click the *Synchronize* button.

In a similar manner, you are able to copy *exposures* you might have made on the teaching instance over to the main repository, or from the main to the teaching instance if you want to test things out. Follow these steps to migrate an *exposure* from one repository to another.

1. Navigate to the exposure you would like to migrate in the source repository.
2. Choose the *wizard* item from the toolbar as shown below.
3. In the destination repository, navigate to the desired revision of the (published) workspace and choose the *Create exposure* action as described in the directions for *creating an exposure from scratch*
4. Rather than building a new exposure, choose the *Exposure Import via URI* tab in the exposure creation wizard, as shown below.

Search Site

Models Home | My Workspaces | Exposures | Documentation | Demo User

You are here: Home / Workspaces / Beeler, Reuter, 1977

View | Edit | History | Files | Fork | Synchronize | Exposure Rollover | RDF Indexing | Sharing | Layout

Shortlog

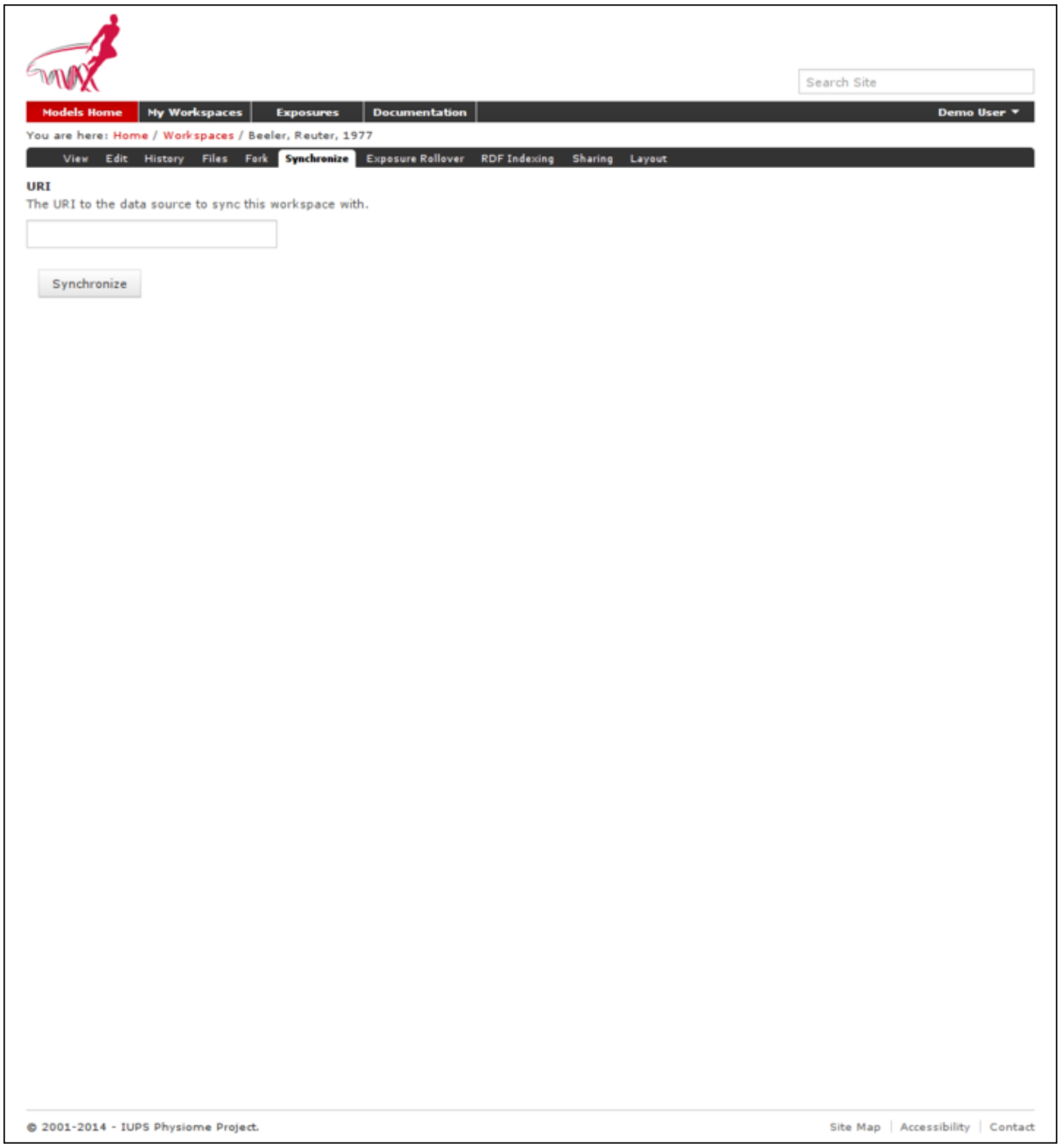
- (0)
- tip

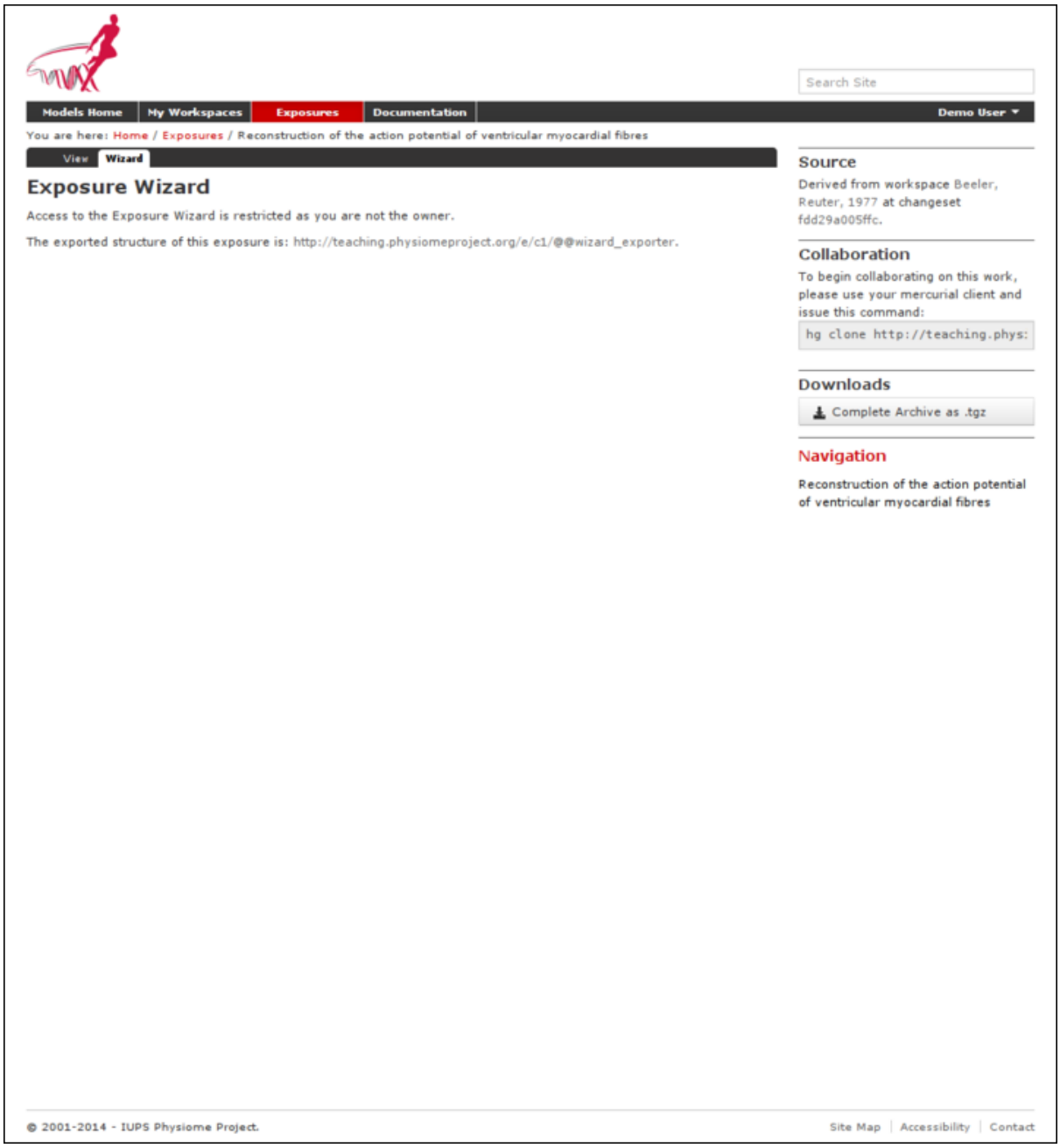
Date	Author	Log	Options	Exposure
13 seconds ago	Demo User	Removed paragraph about validation errors from the documentation	[files] [tgz] [zip]	
2011-12-09	Dougal Cowan	Minor fix to documentation, changed title.	[files] [tgz] [zip]	
2011-12-09	Dougal Cowan	Fixed incorrect figure image.	[files] [tgz] [zip]	
2011-12-09	Dougal Cowan	Adding HTML version of documentation for the model.	[files] [tgz] [zip]	
2010-08-25	Tommy Yu	e-notation fix	[files] [tgz] [zip]	
2009-11-27	Hanne	Added images in ai and svg format	[files] [tgz] [zip]	
2009-06-17	pmr2.import	committing version08 of beeler_reuter_1977	[files] [tgz] [zip]	
2008-05-19	pmr2.import	committing version07 of beeler_reuter_1977	[files] [tgz] [zip]	
2008-05-07	pmr2.import	committing version06 of beeler_reuter_1977	[files] [tgz] [zip]	
2008-01-24	pmr2.import	committing version05 of beeler_reuter_1977	[files] [tgz] [zip]	
2007-03-26	pmr2.import	committing version04 of beeler_reuter_1977	[files] [tgz] [zip]	
2007-03-06	pmr2.import	committing version03 of beeler_reuter_1977	[files] [tgz] [zip]	
2007-03-05	pmr2.import	committing version02 of beeler_reuter_1977	[files] [tgz] [zip]	
2006-09-04	pmr2.import	committing version01 of beeler_reuter_1977	[files] [tgz] [zip]	

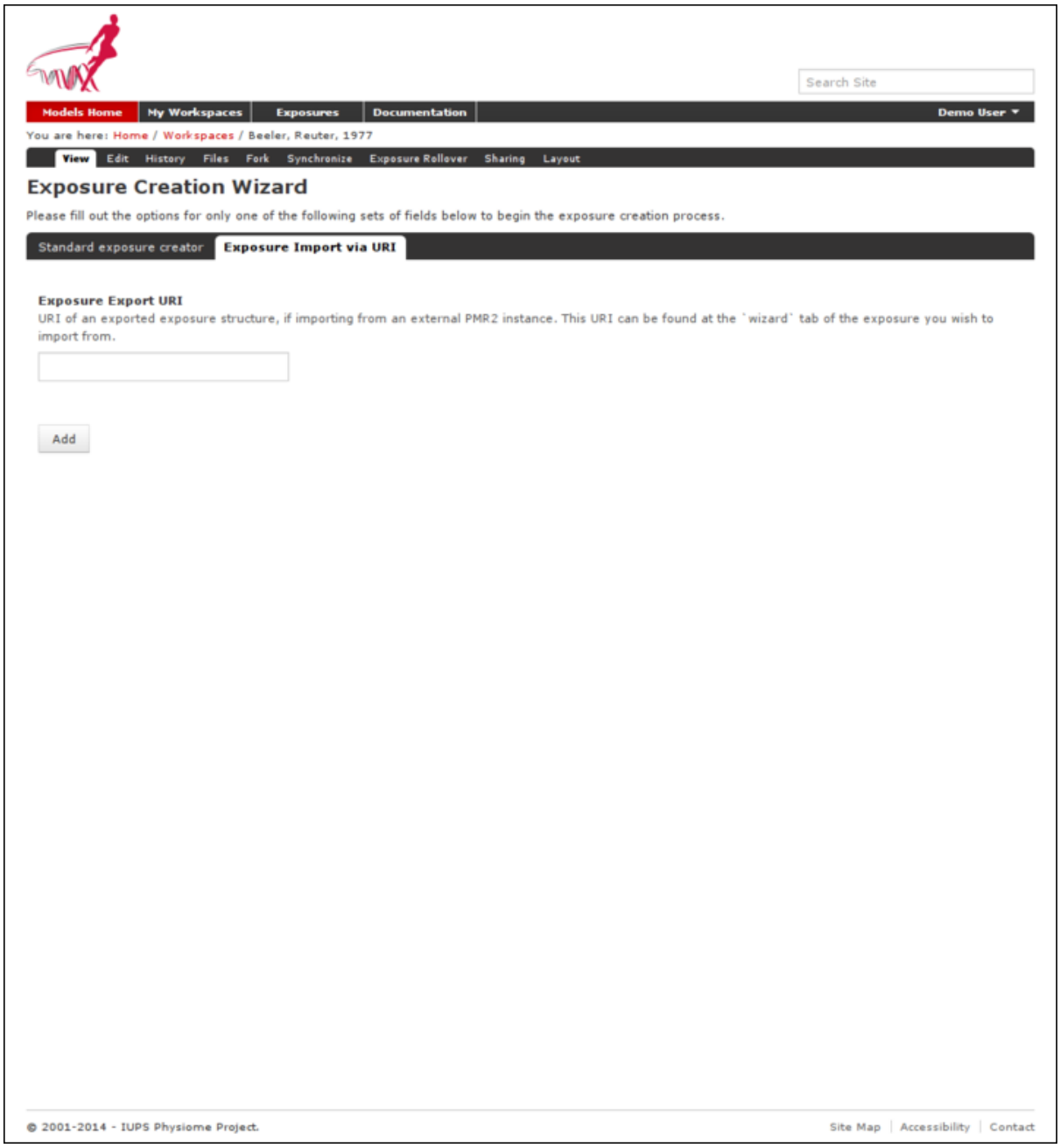
- (0)
- tip

© 2001-2014 - IUPS Physiome Project.

Site Map | Accessibility | Contact


Search Site
Models Home
My Workspaces
Exposures
Documentation
Demo User
You are here: Home / Workspaces / Beeler, Reuter, 1977
View
Edit
History
Files
Fork
Synchronize
Exposure Rollover
RDF Indexing
Sharing
Layout
URI
The URI to the data source to sync this workspace with.
Synchronize
© 2001-2014 - IUPS Physiome Project.
Site Map
Accessibility
Contact


Search Site
Models Home
My Workspaces
Exposures
Documentation
Demo User
You are here: Home / Exposures / Reconstruction of the action potential of ventricular myocardial fibres
View
Wizard
Exposure Wizard
Access to the Exposure Wizard is restricted as you are not the owner.
The exported structure of this exposure is: http://teaching.physiomeproject.org/e/c1/@@wizard_exporter.
Source
Derived from workspace Beeler, Reuter, 1977 at changeset fdd29a005ffc.
Collaboration
To begin collaborating on this work, please use your mercurial client and issue this command:
hg clone http://teaching.phys:
Downloads
Complete Archive as .tgz
Navigation
Reconstruction of the action potential of ventricular myocardial fibres
© 2001-2014 - IUPS Physiome Project.
Site Map
Accessibility
Contact


Search Site
Models Home
My Workspaces
Exposures
Documentation
Demo User
You are here: Home / Workspaces / Beeler, Reuter, 1977
View
Edit
History
Files
Fork
Synchronize
Exposure Rollover
Sharing
Layout
Exposure Creation Wizard
Please fill out the options for only one of the following sets of fields below to begin the exposure creation process.
Standard exposure creator
Exposure Import via URI
Exposure Export URI
URI of an exported exposure structure, if importing from an external PMR2 instance. This URI can be found at the `wizard` tab of the exposure you wish to import from.
Add
© 2001-2014 - IUPS Physiome Project.
Site Map
Accessibility
Contact

5. Copy and paste the URI from the source exposure wizard, highlighted above, into the *Exposure Export URI* field in the exposure creation wizard shown above.
6. Click the *Add* button. This will take you back to the standard *exposure build page*, but now with all the fields pre-populated from the source exposure.
7. Navigate to the bottom of the page and click the *Build* button to actually build the exposure pages. You are free to reconfigure the exposure if desired, some *help is available* for this if needed.

Working with semantic metadata

Section author: Tommy Yu

PMR2 release 8 and 9 brought in the support for semantic metadata, which allows users to add whatever metadata and annotations they might have stored into the repository into the underlying metadata semantic engine, which then allows them to be retrieved using search queries. In this section, we will go over how to use OpenCOR to annotate a model, and how to add the metadata to the underlying metadata engine then query for the results.

Preparation

In this section, we assume that you have already run through the *tutorial on using the repository* for the basic operations of the repository.

For the tutorial, we will use a *fork* of the the Hodgkin, Huxley, 1952 workspace. If you need a quick reminder on how you might do this, please see *this section of the tutorial*.

Once you forked that workspace, you should now clone that workspace onto your system. If you need help on this, please refer to *this help on cloning a workspace*.

Using OpenCOR for model annotation

Use OpenCOR to open your local clone of your model file, specifically the `hodgkin_huxley_1952.cellml` file.

Select the `sodium_channel` component under the list of components, then click on the helpful link to remove the existing metadata for that node.

In the dropdown menu of *Qualifiers*, select bio:isVersionOf

In the textbox *Term*, type in "sodium channel", as the component is named so. Wait for the possible terms to be retrieved and populated by OpenCOR.

Once that is done, hit the green '+' button for the "sodium channel complex" (GO:0034706) to denote that the component is a version of this term.

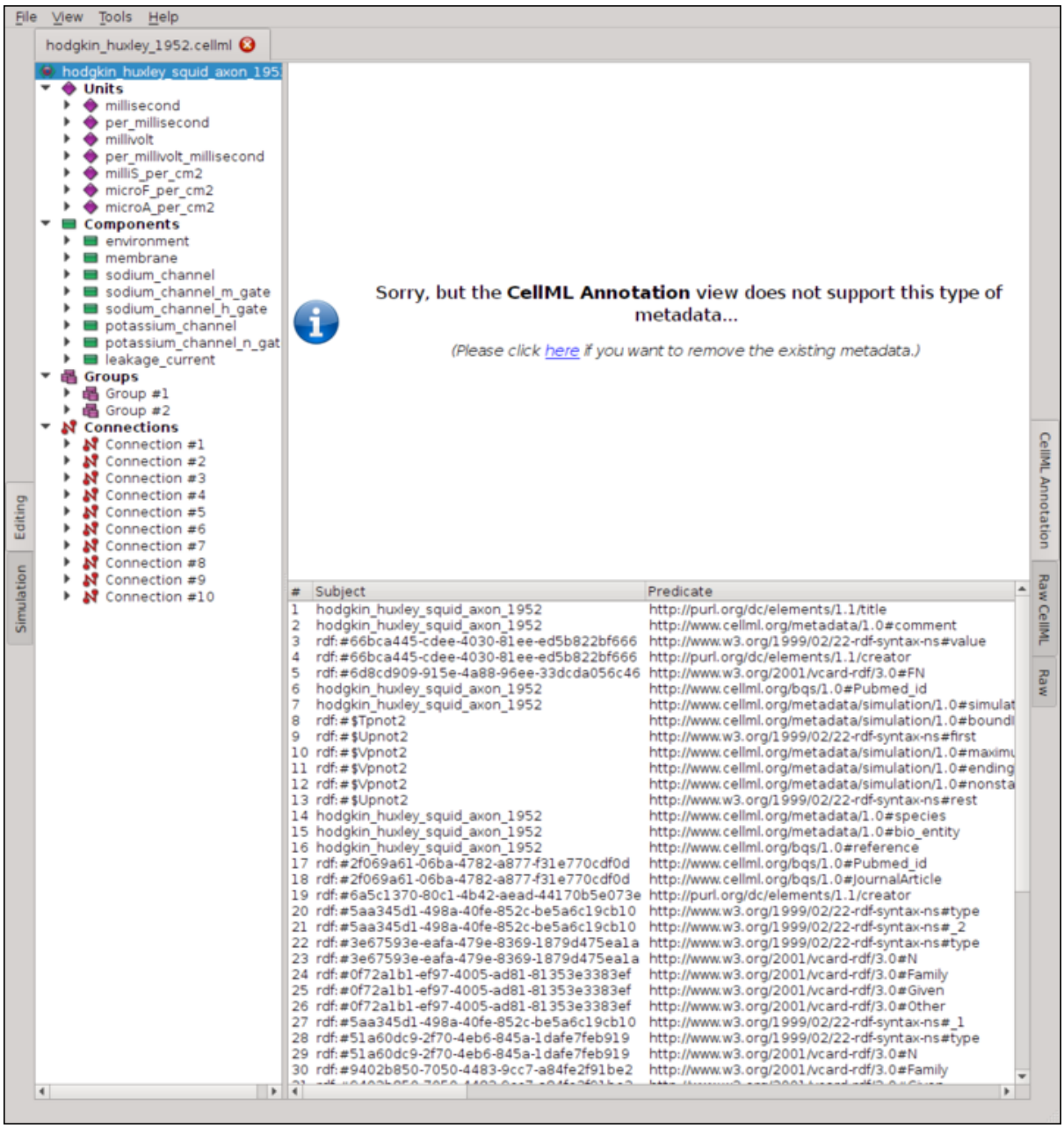
Now select the `potassium_channel` component, and repeat the processs to annotate this with the "potassium channel complex" term.

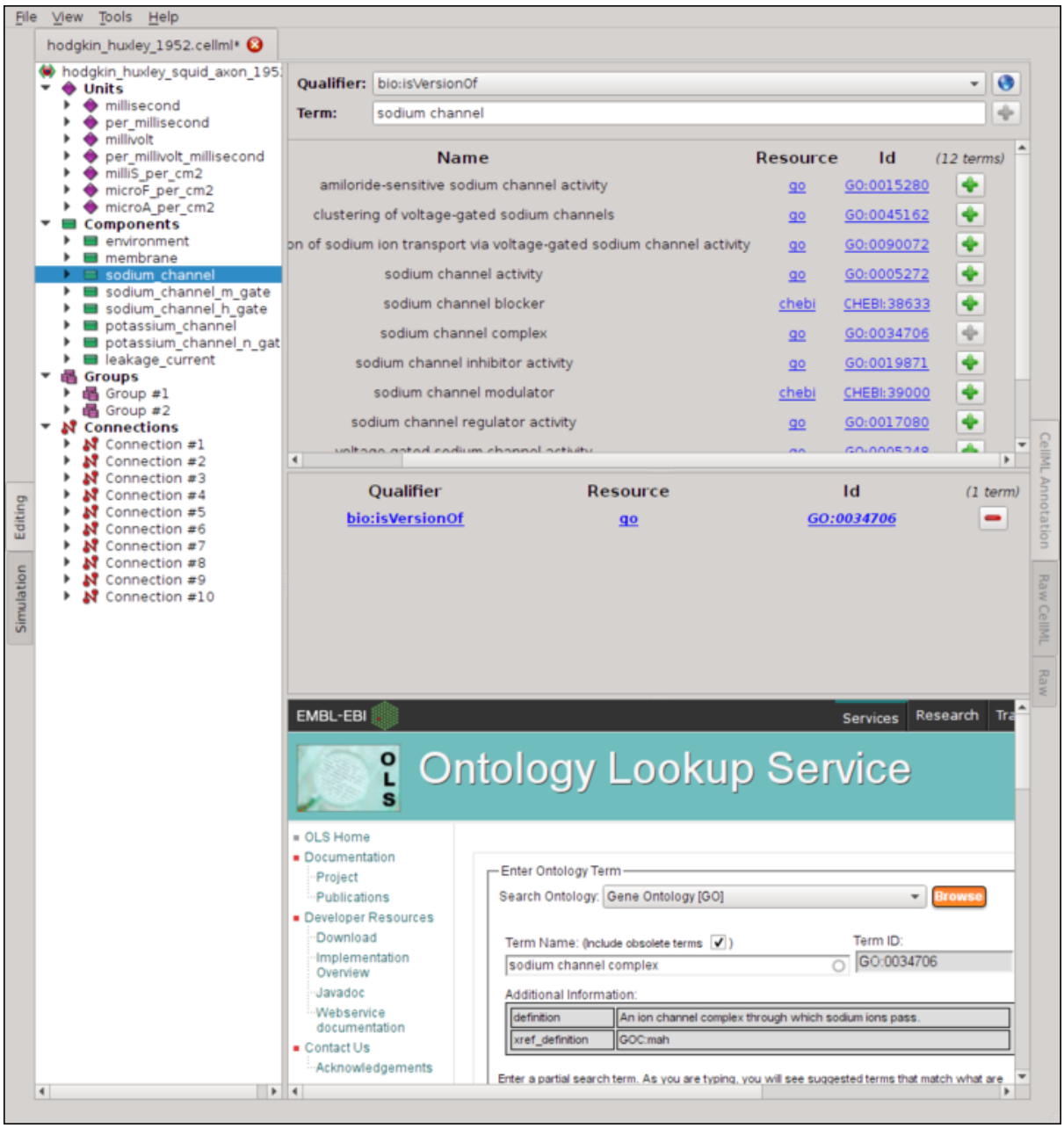
Once you are done, save your changes, commit and push your work back into your private fork. Again, refer to the tutorial linked in the preparation section if you need a primer.

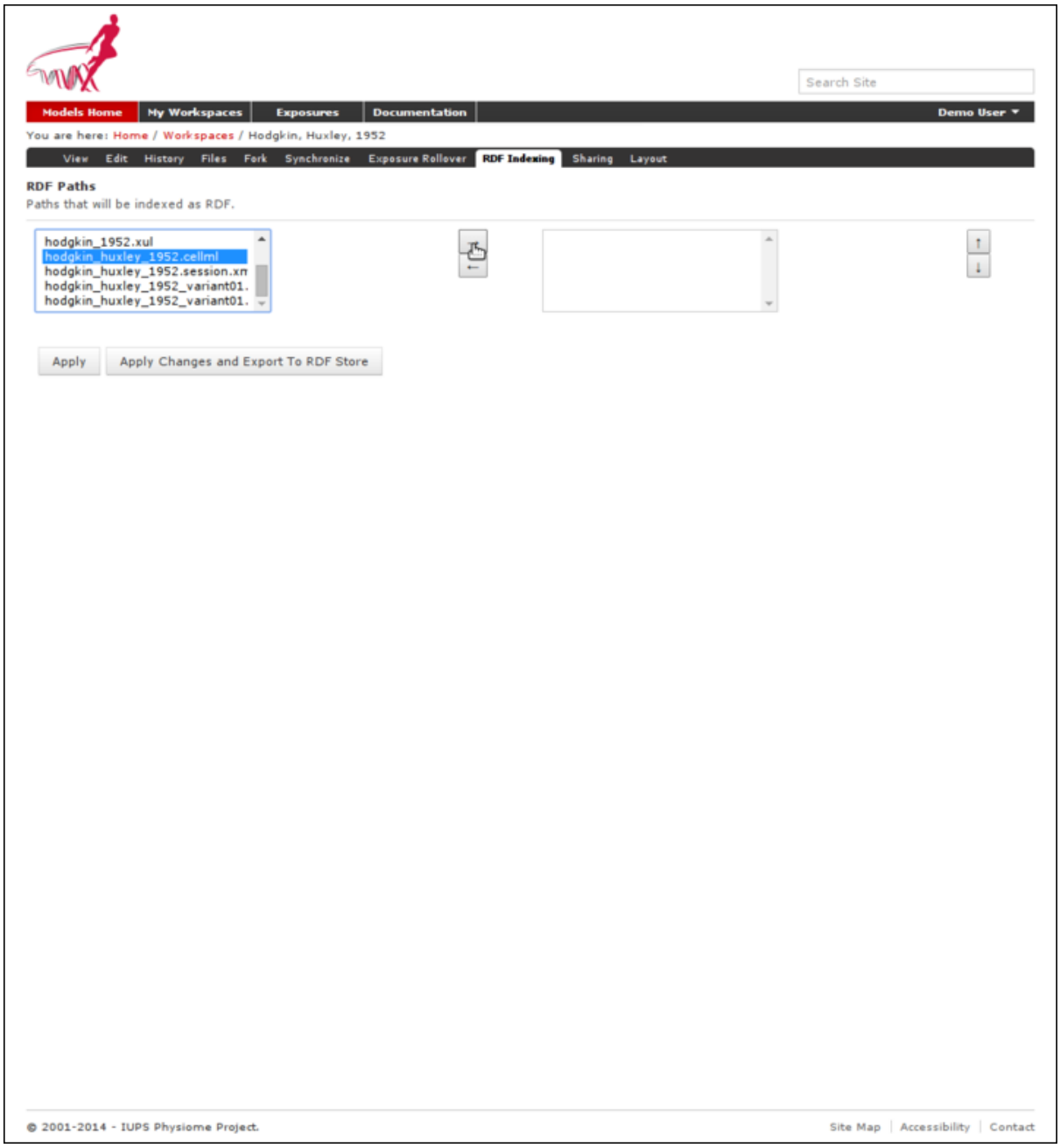
Getting your workspace indexed by the repository

Now that your changes have been pused back, go back to the page for your fork of the model and select the "RDF Indexing" tab.

Scroll down the left-handed list until you see the `hodgkin_huxley_1952.cellml` file, select it, then push the button with the right arrow on it to add it onto the list of paths to be tracked, then select the "Apply Changes and Export To RDF Store" button.


File View Tools Help
hodgkin_huxley_1952.cellml
hodgkin_huxley_squid_axon_195
Units
millisecond
per_millisecond
millivolt
per_millivolt_millisecond
milliS_per_cm2
microF_per_cm2
microA_per_cm2
Components
environment
membrane
sodium_channel
sodium_channel_m_gate
sodium_channel_h_gate
potassium_channel
potassium_channel_n_gat
leakage_current
Groups
Group #1
Group #2
Connections
Connection #1
Connection #2
Connection #3
Connection #4
Connection #5
Connection #6
Connection #7
Connection #8
Connection #9
Connection #10
Editing
Simulation
Sorry, but the CellML Annotation view does not support this type of metadata...
(Please click here if you want to remove the existing metadata.)
CellML Annotation
Raw CellML
Raw
Subject Predicate
1 hodgkin_huxley_squid_axon_1952 http://purl.org/dc/elements/1.1/title
2 hodgkin_huxley_squid_axon_1952 http://www.cellml.org/metadata/1.0#comment
3 rdf:#66bca445-cdee-4030-81ee-ed5b822bf666 http://www.w3.org/1999/02/22-rdf-syntax-ns#value
4 rdf:#66bca445-cdee-4030-81ee-ed5b822bf666 http://purl.org/dc/elements/1.1/creator
5 rdf:#6d8cd909-915e-4a88-96ee-33dcda056c46 http://www.w3.org/2001/vcard-rdf/3.0#FN
6 hodgkin_huxley_squid_axon_1952 http://www.cellml.org/bqs/1.0#Pubmed_id
7 hodgkin_huxley_squid_axon_1952 http://www.cellml.org/metadata/simulation/1.0#simulat
8 rdf:#$Tpnot2 http://www.cellml.org/metadata/simulation/1.0#boundi
9 rdf:#$Upnot2 http://www.w3.org/1999/02/22-rdf-syntax-ns#first
10 rdf:#$Vpnot2 http://www.cellml.org/metadata/simulation/1.0#maximu
11 rdf:#$Vpnot2 http://www.cellml.org/metadata/simulation/1.0#ending
12 rdf:#$Vpnot2 http://www.cellml.org/metadata/simulation/1.0#nonsta
13 rdf:#$Upnot2 http://www.w3.org/1999/02/22-rdf-syntax-ns#rest
14 hodgkin_huxley_squid_axon_1952 http://www.cellml.org/metadata/1.0#species
15 hodgkin_huxley_squid_axon_1952 http://www.cellml.org/metadata/1.0#bio_entity
16 hodgkin_huxley_squid_axon_1952 http://www.cellml.org/bqs/1.0#reference
17 rdf:#2f069a61-06ba-4782-a877-f31e770cdf0d http://www.cellml.org/bqs/1.0#Pubmed_id
18 rdf:#2f069a61-06ba-4782-a877-f31e770cdf0d http://www.cellml.org/bqs/1.0#JournalArticle
19 rdf:#6a5c1370-80c1-4b42-aead-44170b5e073e http://purl.org/dc/elements/1.1/creator
20 rdf:#5aa345d1-498a-40fe-852c-be5a6c19cb10 http://www.w3.org/1999/02/22-rdf-syntax-ns#type
21 rdf:#5aa345d1-498a-40fe-852c-be5a6c19cb10 http://www.w3.org/1999/02/22-rdf-syntax-ns#_2
22 rdf:#3e67593e-eafa-479e-8369-1879d475ea1a http://www.w3.org/1999/02/22-rdf-syntax-ns#type
23 rdf:#3e67593e-eafa-479e-8369-1879d475ea1a http://www.w3.org/2001/vcard-rdf/3.0#N
24 rdf:#0f72a1b1-ef97-4005-ad81-81353e3383ef http://www.w3.org/2001/vcard-rdf/3.0#Family
25 rdf:#0f72a1b1-ef97-4005-ad81-81353e3383ef http://www.w3.org/2001/vcard-rdf/3.0#Given
26 rdf:#0f72a1b1-ef97-4005-ad81-81353e3383ef http://www.w3.org/2001/vcard-rdf/3.0#Other
27 rdf:#5aa345d1-498a-40fe-852c-be5a6c19cb10 http://www.w3.org/1999/02/22-rdf-syntax-ns#_1
28 rdf:#51a60dc9-2f70-4eb6-845a-1dafe7feb919 http://www.w3.org/1999/02/22-rdf-syntax-ns#type
29 rdf:#51a60dc9-2f70-4eb6-845a-1dafe7feb919 http://www.w3.org/2001/vcard-rdf/3.0#N
30 rdf:#9402b850-7050-4483-9cc7-a84fe2f91be2 http://www.w3.org/2001/vcard-rdf/3.0#Family


File View Tools Help
hodgkin_huxley_1952.cellml*
hodgkin_huxley_squid_axon_195
Units
millisecond
per_millisecond
millivolt
per_millivolt_millisecond
milliS_per_cm2
microF_per_cm2
microA_per_cm2
Components
environment
membrane
sodium_channel
sodium_channel_m_gate
sodium_channel_h_gate
potassium_channel
potassium_channel_n_gat
leakage_current
Groups
Group #1
Group #2
Connections
Connection #1
Connection #2
Connection #3
Connection #4
Connection #5
Connection #6
Connection #7
Connection #8
Connection #9
Connection #10
Editing
Simulation
Qualifier: bio:isVersionOf
Term: sodium channel
Name Resource Id (12 terms)
amiloride-sensitive sodium channel activity go GO:0015280
clustering of voltage-gated sodium channels go GO:0045162
on of sodium ion transport via voltage-gated sodium channel activity go GO:0090072
sodium channel activity go GO:0005272
sodium channel blocker chebi CHEBI:38633
sodium channel complex go GO:0034706
sodium channel inhibitor activity go GO:0019871
sodium channel modulator chebi CHEBI:39000
sodium channel regulator activity go GO:0017080
Qualifier Resource Id (1 term)
bio:isVersionOf go GO:0034706
CellML Annotation
Raw CellML
Raw
EMBL-EBI
Services Research
Ontology Lookup Service
OLS Home
Documentation
Project
Publications
Developer Resources
Download
Implementation Overview
Javadoc
Webservice documentation
Contact Us
Acknowledgements
Enter Ontology Term
Search Ontology: Gene Ontology [GO]
Browse
Term Name: (Include obsolete terms)
Term ID:
sodium channel complex
GO:0034706
Additional Information:
definition An ion channel complex through which sodium ions pass.
xref_definition GOC:mah
Enter a partial search term. As you are typing, you will see suggested terms that match what are


Search Site
Models Home
My Workspaces
Exposures
Documentation
Demo User
You are here: Home / Workspaces / Hodgkin, Huxley, 1952
View
Edit
History
Files
Fork
Synchronize
Exposure Rollover
RDF Indexing
Sharing
Layout
RDF Paths
Paths that will be indexed as RDF.
hodgkin_1952.xul
hodgkin_huxley_1952.cellml
hodgkin_huxley_1952.session.xm
hodgkin_huxley_1952_variant01.
hodgkin_huxley_1952_variant01.
Apply
Apply Changes and Export To RDF Store
© 2001-2014 - IUPS Physiome Project.
Site Map
Accessibility
Contact

Go back to the main page, select the "Ontology based search engine" link at the bottom, then enter the relevant search term. As there are limited reasoning capabilities built into the current iteration of the search engine, you may enter a term one level up above the terms we annotated the model with. For our example, please enter "cation channel complex" into the search box, select the term ending with (GO_0034703). The search indicator will give a green checkmark and now you may select the "Search" button. The search result will now list the workspace and the file that contain this annotation.

Creating CellML exposures

Section author: Dougal Cowan

CellML models in the Auckland Physiome Repository are presented through *exposures*. An *exposure* is a view of a particular revision of a workspace, and is quite flexible in terms of what it can present. A workspace may contain one or more models, and any number of models may be presented in a single exposure. Exposures generally take

the form of some documentation about the model(s), a range of ways of looking at the model(s) or their metadata, and links to download the model(s).

The example below shows the main exposure page for the Bondarenko *et al.* 2004 workspace. This workspace contains two models, which can be viewed via the *Navigation* pane on the right hand side of the page.

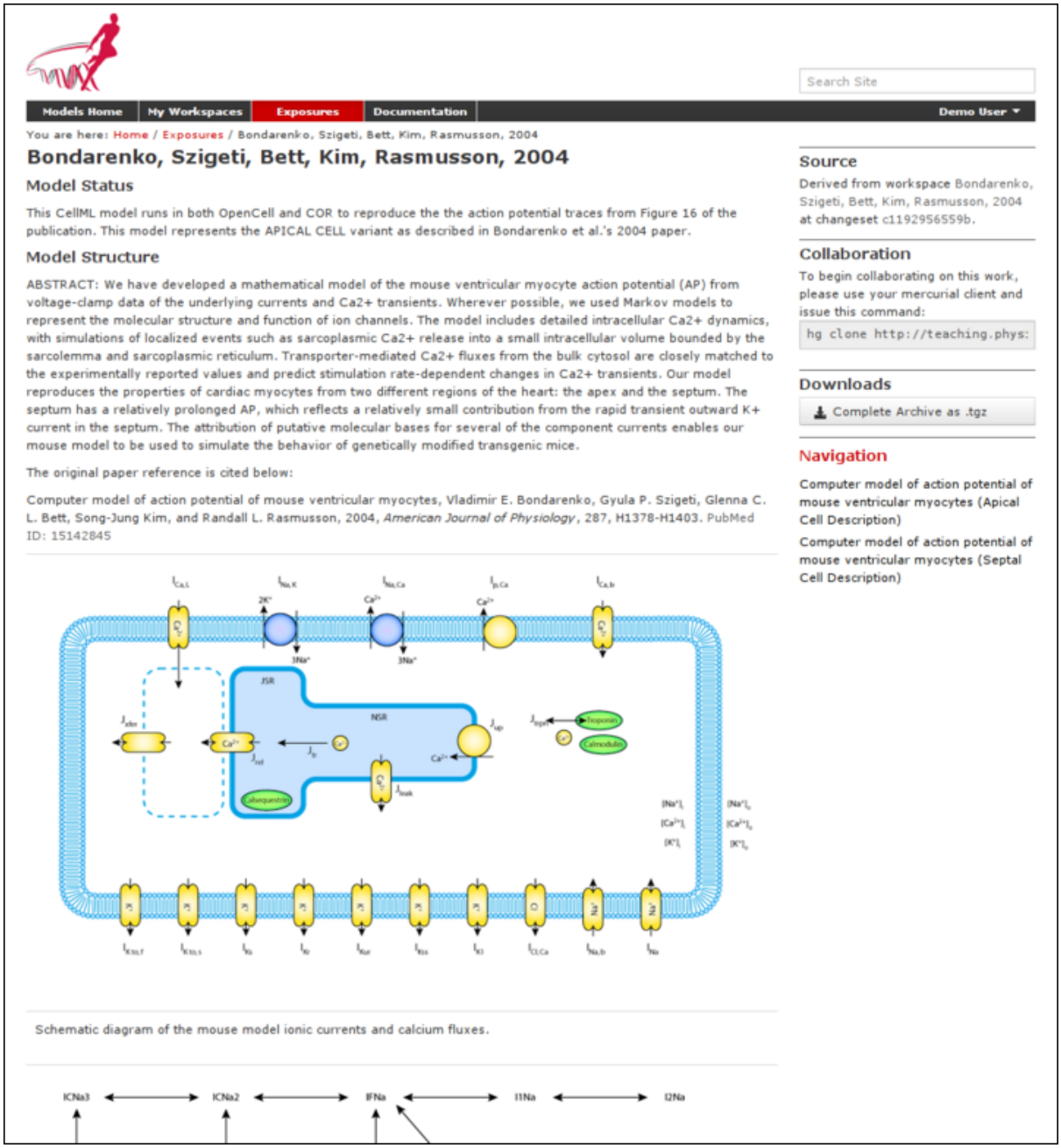


Fig. 2.10: **Example of an exposure page**

If you click on one of the model navigation links, it will take you to the page for that particular model. Exposures most often present a single model, although they can present any number of models, each with its own documentation and views.

Most of the CellML exposures in the repository are currently of this type, with a main documentation page containing navigation links to the model or models themselves.

The model pages have links that enable the user to do things like view the model equations, look at the citation information, or run the model as an interactive session using the OpenCell application. These links are found in the pane titled *Views available* on the right hand side of the page.

Search Site

Models Home | My Workspaces | Exposures | Documentation | Demo User

You are here: Home / Exposures / Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004 / Computer model of action potential of mouse ventricular myocytes (Apical Cell Description)

Computer model of action potential of mouse ventricular myocytes (Apical Cell Description)

by Dougal Cowan — last modified Jul 15, 2013 01:34 PM — History

Computer model of action potential of mouse ventricular myocytes

Model Status

This CellML model runs in both OpenCell and COR to reproduce the the action potential traces from Figure 16 of the publication. This model represents the APICAL CELL variant as described in Bondarenko et al.'s 2004 paper.

Model Structure

ABSTRACT: We have developed a mathematical model of the mouse ventricular myocyte action potential (AP) from voltage-clamp data of the underlying currents and Ca2+ transients. Wherever possible, we used Markov models to represent the molecular structure and function of ion channels. The model includes detailed intracellular Ca2+ dynamics, with simulations of localized events such as sarcoplasmic Ca2+ release into a small intracellular volume bounded by the sarcolemma and sarcoplasmic reticulum. Transporter-mediated Ca2+ fluxes from the bulk cytosol are closely matched to the experimentally reported values and predict stimulation rate-dependent changes in Ca2+ transients. Our model reproduces the properties of cardiac myocytes from two different regions of the heart: the apex and the septum. The septum has a relatively prolonged AP, which reflects a relatively small contribution from the rapid transient outward K+ current in the septum. The attribution of putative molecular bases for several of the component currents enables our mouse model to be used to simulate the behavior of genetically modified transgenic mice.

The original paper reference is cited below:

Computer model of action potential of mouse ventricular myocytes, Vladimir E. Bondarenko, Gyula P. Szigeti, Glenna C. L. Bett, Song-Jung Kim, and Randall L. Rasmusson, 2004, *American Journal of Physiology*, 287, H1378-H1403. PubMed ID: 15142845

Schematic diagram of the mouse model ionic currents and calcium fluxes.

Model Curation

Curation Status
JSim
COR
OpenCell

Source

Derived from workspace Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004 at changeset c1192956559b.

Collaboration

To begin collaborating on this work, please use your mercurial client and issue this command:

```
hg clone http://teaching.phys:
```

Downloads

Complete Archive as .tgz
Download This File

Views Available

Documentation
Model Metadata
Model Curation
Mathematics
Generated Code
Cite this model
Source View
Simulate using OpenCell

License

This work is licensed under a Creative Commons Attribution 3.0 Unported License.

Navigation

Computer model of action potential of mouse ventricular myocytes (Apical Cell Description)

Computer model of action potential of mouse ventricular myocytes (Septal Cell Description)

Fig. 2.11: **Example of a model exposure page**

This tutorial contains instructions on how to create one of these standard CellML exposures, as well as information about how to create other alternative types of exposure.

Creating standard CellML exposures

In this example I will use a *fork* of the the Beeler Reuter 1977 workspace. Creating a *fork* of a workspace creates a *clone* of that workspace that you own, and can push changes to. You can *fork* any publicly available workspace in the AUckland Physiome Repository. For more information on this feature, refer to the information on features or collaboration, or see the *relevant section of the tutorial*.

At this point you are recommended to submit the workspace for publication, using the *state:* menu at the top right of the workspace view page. This is especially important if you decide to make an exposure public, as having a private workspace for a public exposure will impede access of linked data, such as images for the introduction to that particular exposure.

Choose the revision to expose

As an exposure is created to present a particular revision of a workspace, the first thing to do is to navigate to that revision. To do this, first find the workspace - if this is your own workspace, you can click on the *My Workspaces* button in the navigation bar of the repository and find the workspace of interest in the listing displayed. After navigating to your workspace, click on the *history* button in the menu bar.

Now you can select the revision of the workspace you wish to expose by clicking on the *manifest* of that revision. Usually you will want to expose the latest revision, which appears at the top of the list.

After selecting the revision you wish to expose, click on the *workspace actions* menu at the far right end of the menu bar and select *create exposure*.

Building the exposure

Selecting the *create exposure* option in the menu bar will bring you to the first page of the exposure *wizard*. This web interface allows you to select the model files, documentation files, and settings that will be used to create the exposure.

The initial page of the exposure creation wizard allows you to select the main documentation file and the first model file. Select the HTML annotator option and the HTML documentation file for the workspace in the *Exposure main view* section. For the *New Exposure File Entry* section, choose the CellML file you wish to expose, and select CellML as the file type.

Note: Documentation should be written in HTML format. Some previous users of the CellML repository may be familiar with the tmpdoc style documentation, which has be deprecated. For an example of what a fairly standard HTML documentation file might look like, take a look at the documentation for the Beeler Reuter 1977 model.

Once you have selected the documentation and model files and their types, click on the *Add* button. This will take you to the next step of the wizard, where you can select various options for the model you have chosen to expose, and will allow you to add further model files to the exposure if desired.

The wizard shows a *subgroup* for each CellML file to be included in the exposure. For each CellML file, select the following options:

- **Documentation**
  - Documentation file - select the HTML file created to document the model
  - View generator - select HTML annotator option
- **Basic Model Curation**
  - Curation flags - CellML model repository curators may select flags according to the status of the model

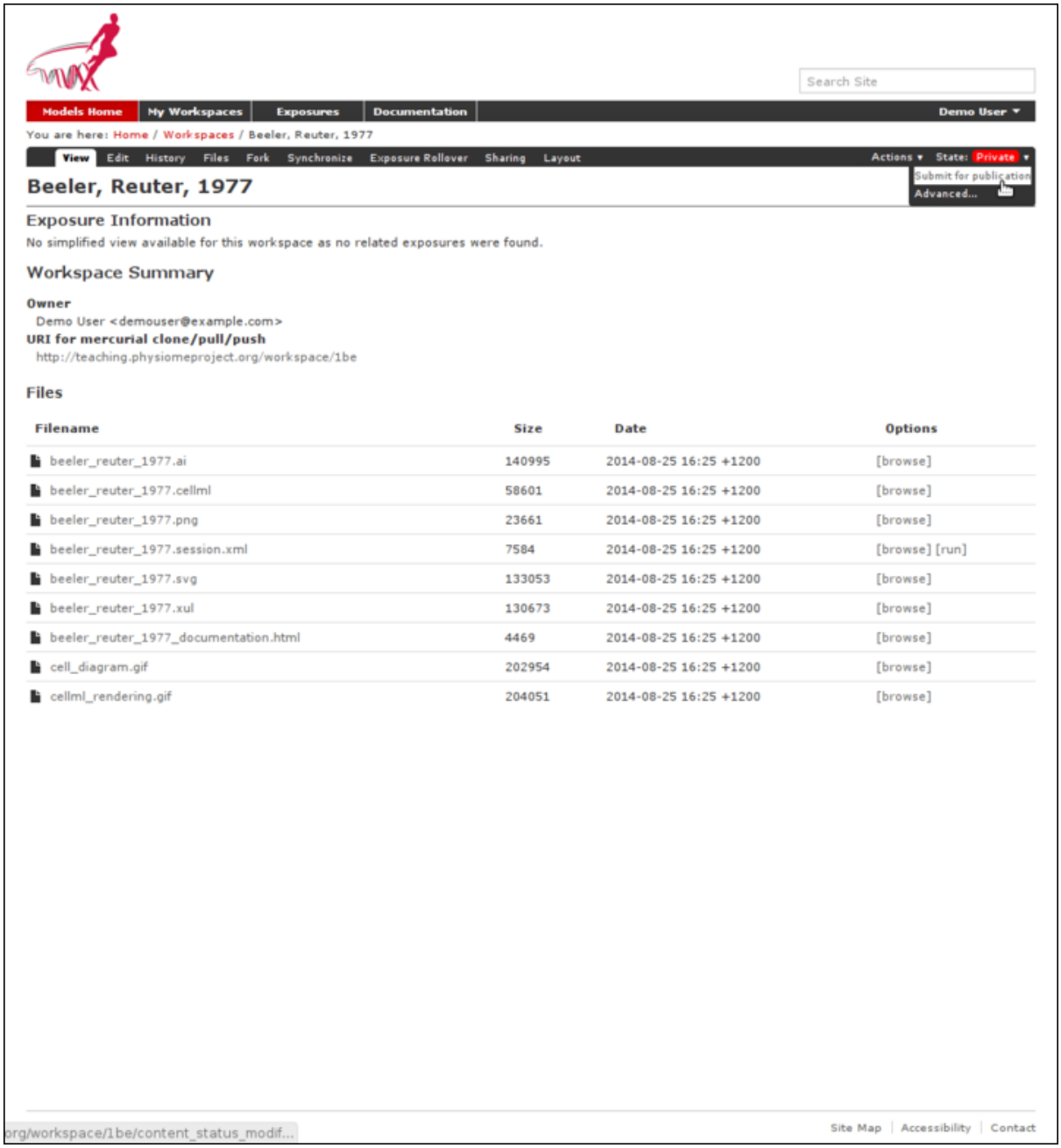


Fig. 2.12: **The state menu is used to submit objects such as workspaces for publication. Submitted items will be reviewed by site administrators and then published.**

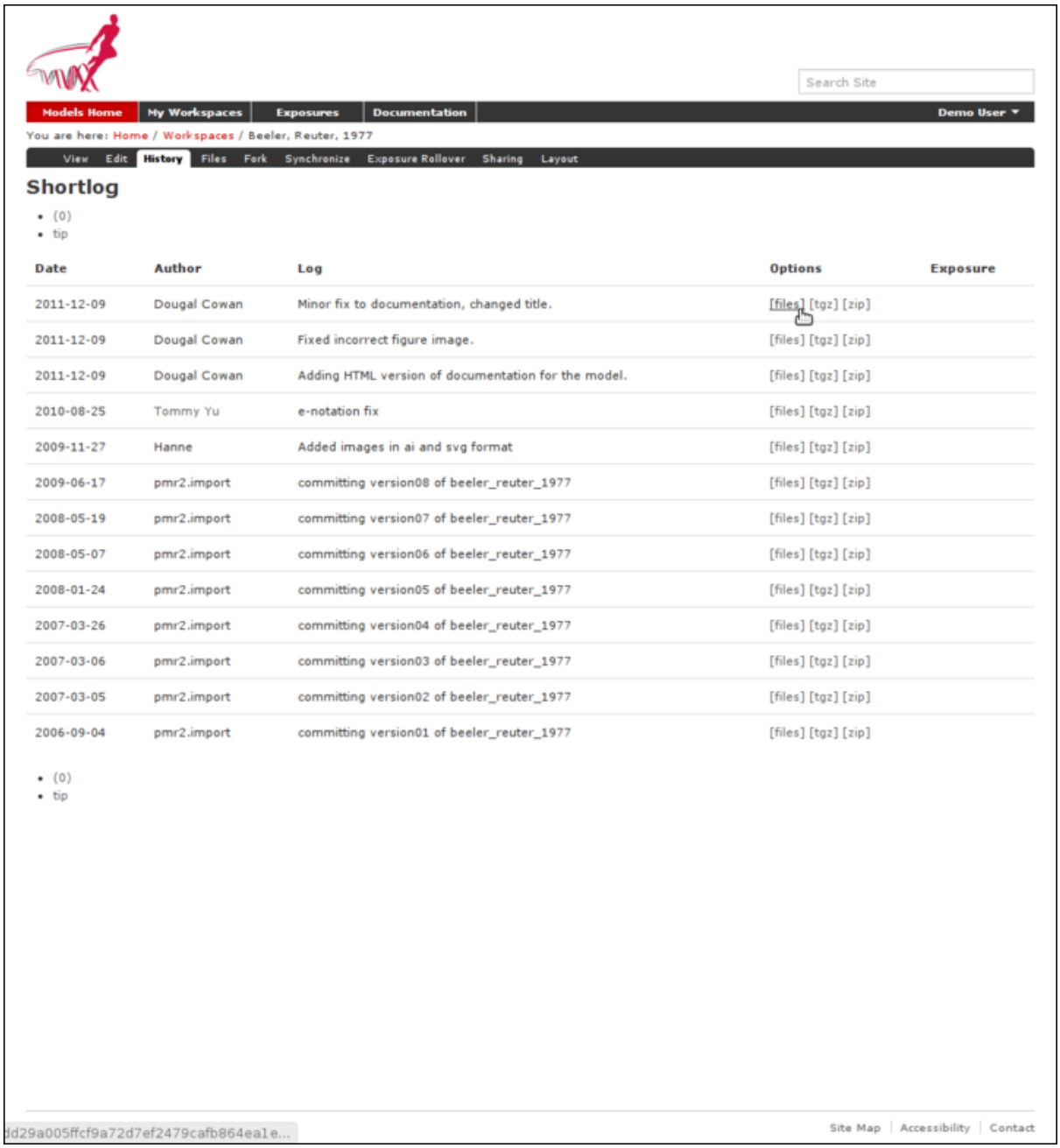


Fig. 2.13: **The revision history of a fork of the Beeler Reuter 1977 workspace**

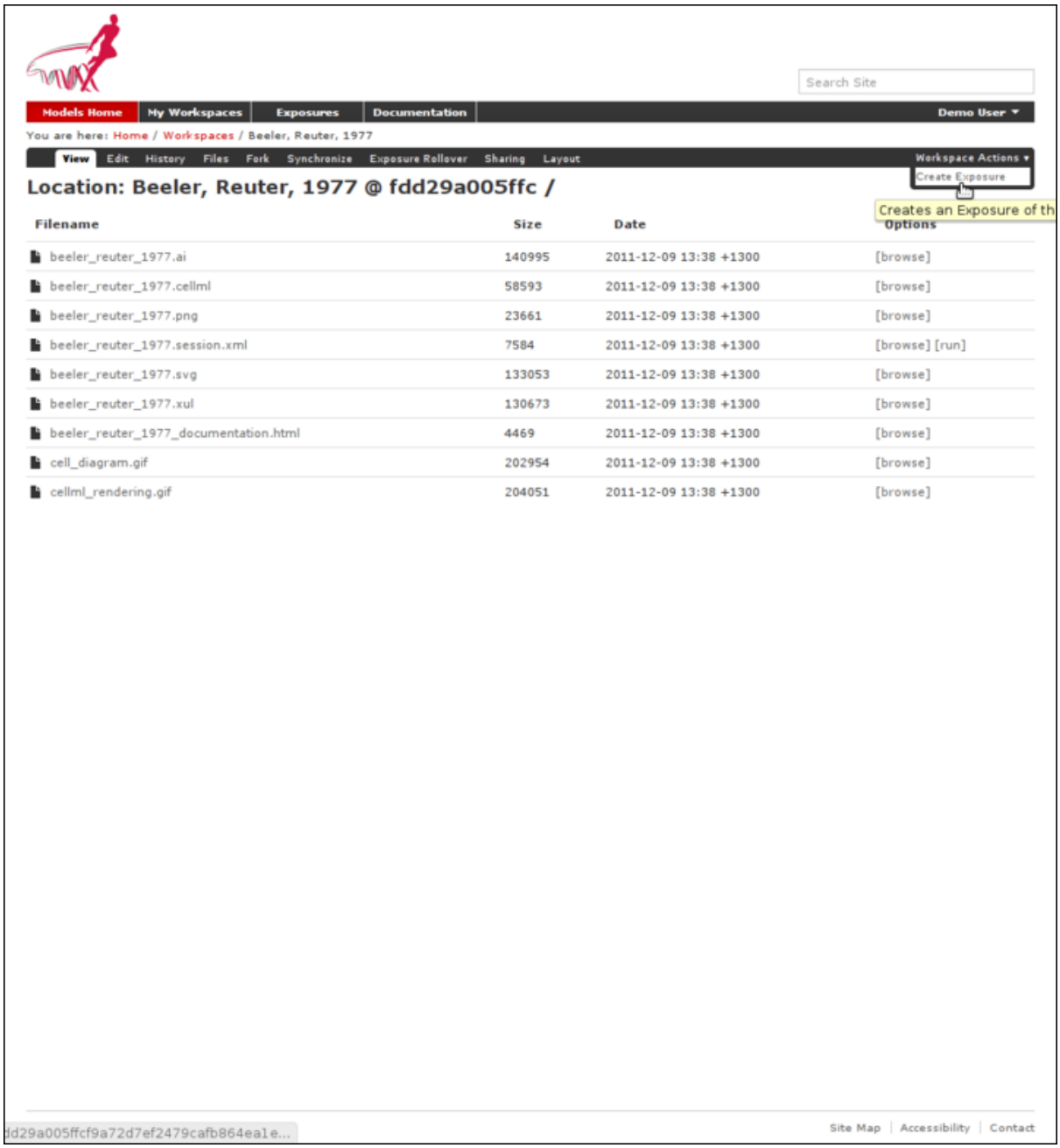


Fig. 2.14: **Selecting the manifest of the revision to expose**

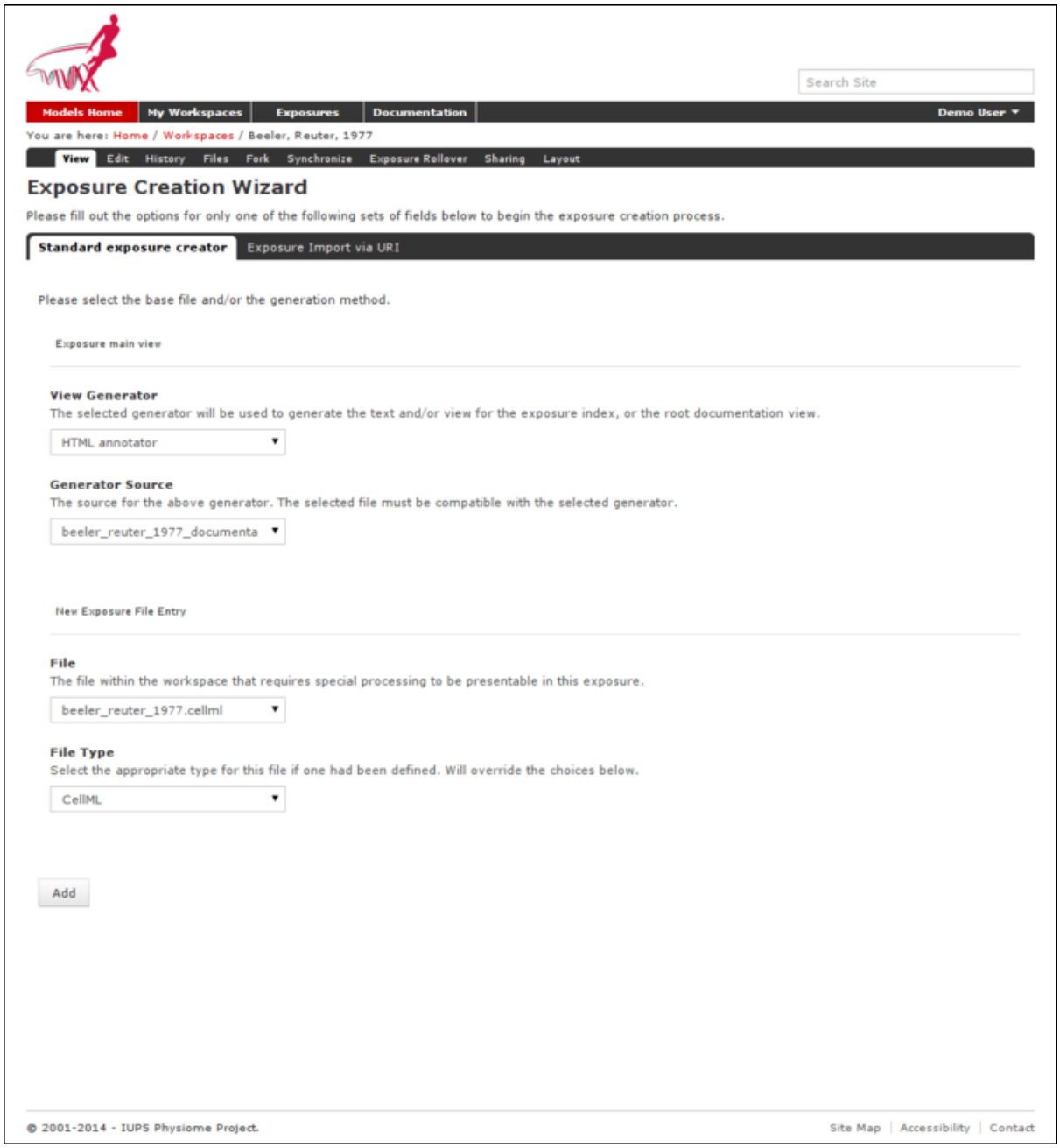


Fig. 2.15: **Selecting the main documentation and the first CellML model file**

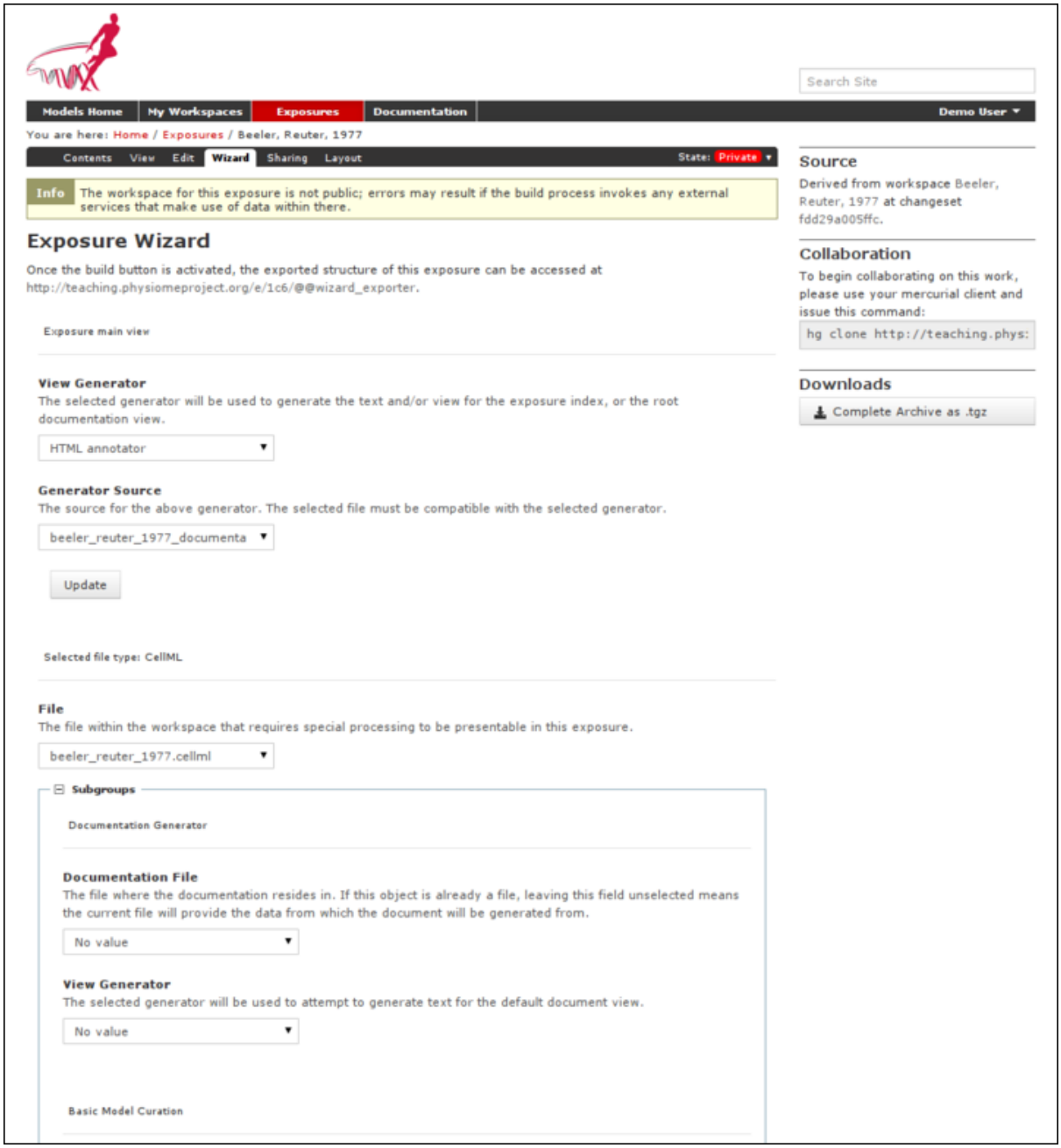


Fig. 2.16: Note that if your workspace is not publicly accessible, there will be an informative note for this which you can safely ignore as there are no process within the generation of the exposure that must require a publicly accessible workspace.

- **License and Citation**
  - File/Citation format - select CellML RDF metadata to automatically generate a citation page using the model RDF
  - License - select Creative Commons Attributions 3.0 Unported, in the cases where the above option is unsuitable.
- **Source Viewer**
  - Language Type - select xml
- **OpenCell Session Link**
  - Session File - select the session.xml if it has been created

After selecting the subgroup options, you need to select the *Update* button to apply the chosen options for the exposure builder, as this is an independent subform to the main form. The options you selected will be ignored if this *Update* button is not selected, and the options will be replaced by the default options when you click *Build* before this was done.

For exposures where you wish to expose multiple models, click on the *Add file* button at this stage to create another subgroup. You can then use this to set up all the same options listed above for the additional model file. Remember to click *Update* when you have completed selecting the options for each subgroup before adding another subgroup.

After setting all the options for the models you wish to expose, click on the *Build* button. The repository software will then create the exposure pages and display the main page of the exposure.

Making your work publicly accessible

In order to make the exposure visible and searchable, you will need to publish it. You can choose to submit your exposure for review, so that the repository administrators or curators will know to publish it for you. Naturally, if you have sufficient privileges you can publish it directly.

Other types of exposure

Because the exposure builder uses HTML documentation, it is possible to create customized types of exposure that differ from the standard type shown above. For example, you might want to create an exposure that simply documents and provides links to models in a workspace that are encoded in languages other than CellML. You can also use the HTML documentation to provide tutorials or other documents, with resources stored in the workspace and linked to from the HTML.

Examples of other exposure types:

- Andre's Hodgkin & Huxley CellML tutorial
- Testing nested SED-ML proposals with CellML
- Aslanidi et al. cardiac models encoded in C

Making an exposure using "rollover"

As explained earlier, an *exposure* aims to bring a particular revision to the attention of users who are browsing and searching the repository.

"Rolling over" an exposure is the method used when a workspace already has an existing exposure, and the updates to the workspace have not fundamentally changed the structure of the workspace. This means that all the information used in making the previous exposure is still valid for making a new exposure of a more recent revision of the workspace. Strictly speaking, an exposure can be rolled over to an older revision as well, but this is not the usual usage.

Selected file type: CellML

File
The file within the workspace that requires special processing to be presentable in this exposure.

beeler_reuter_1977.cellml

Subgroups

Documentation Generator

Documentation File
The file where the documentation resides in. If this object is already a file, leaving this field unselected means the current file will provide the data from which the document will be generated from.

beeler_reuter_1977.cellml

View Generator
The selected generator will be used to attempt to generate text for the default document view.

HTML annotator

Basic Model Curation

Curation Flags
Curation flags assigned to this object.

COR

2 star

JSim

No value

OpenCell

2 star

Curation Status

2 star

License and Citation

File/Citation Format
Select the correct method to generate the citation and license information from the source file. Overwrites the values below.

CellML RDF Metadata

License
The license this work is licensed under. Will be overwritten if Citation For is specified.

No value

dcterms:license
The link to the license this model is licensed under. It is automatically assigned if one of the assignment methods above is set.

Source Viewer

Search Site

Models Home | My Workspaces | Exposures | Documentation | Demo User

You are here: Home / Exposures / Reconstruction of the action potential of ventricular myocardial fibres

Contents | View | Edit | Wizard | Sharing | Layout | State: Private | Submit for publication | Advanced...

Reconstruction of the action potential of ventricular myocardial fibres

Encoded in CellML by Catherine Lloyd

Bioengineering Institute, University of Auckland

Model Status

This model has been curated by Penny Noble using Flavio Fenton's Java code as a reference (See http://thevirtualheart.org/ for Java applet rendering of model - Java code is available from Dr Fenton.) An artificial stimulus component has been added this model to allow it to reproduce the action potential simulation shown in Figure 4 of the publication. The model is known to run and integrate in the PCEnv and COR CellML environments. A PCEnv session file is also associated with this model.

ValidateCellML detects unit inconsistency within this model.

Model Structure

In contrast to the earlier Purkinje fibre ionic current models of D. Noble (1962) and R.E. McAllister, D. Noble and R.W. Tsien (1975), the G.W. Beeler and H. Reuter 1977 model was developed to describe the mammalian ventricular action potential. Not all the ionic currents of the Purkinje fibre model are present in ventricular tissue; therefore, this model is simpler than the MNT model. The total ionic flux is divided into only four discrete, individual ionic currents (see the figure below). The main additional feature of the Beeler-Reuter ionic current model is a representation of the intracellular calcium ion concentration.

The complete original paper reference is cited below:

Reconstruction of the action potential of ventricular myocardial fibres, Beeler, G.W. and Reuter, H. 1977, *Journal of Physiology*, 268, 177-210. PubMed ID: 874889

A schematic diagram describing the current flows across the cell membrane that are captured in the BR model.

Membrane

Source

Derived from workspace Beeler, Reuter, 1977 at changeset fdd29a005ffc.

Collaboration

To begin collaborating on this work, please use your mercurial client and issue this command:

```
hg clone http://teaching.phys:
```

Downloads

Complete Archive as .tgz

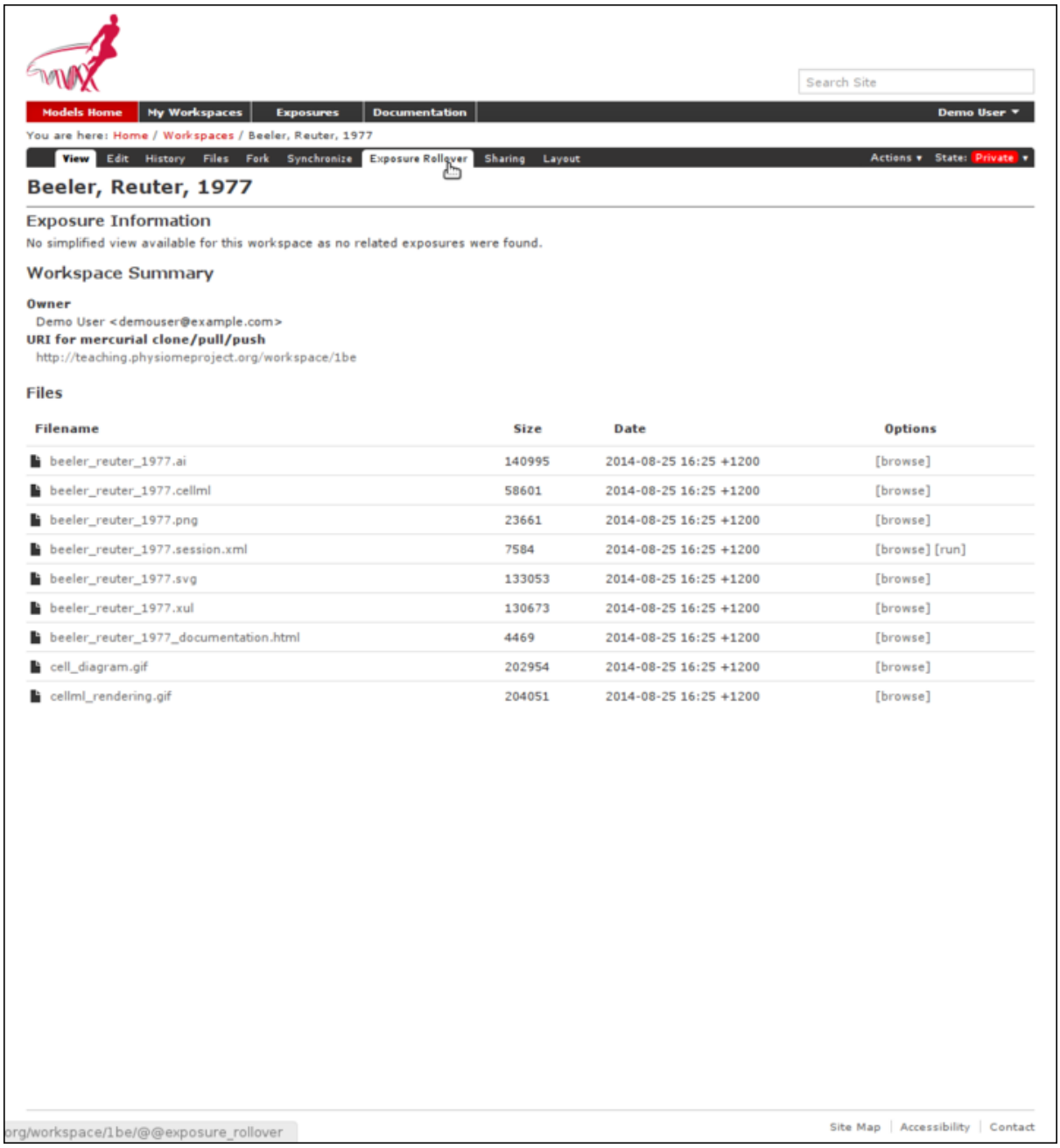
Navigation

Reconstruction of the action potential of ventricular myocardial fibres

Fig. 2.18: **Publish your exposure to make it visible to others.**

Note: A forked workspace contains all of the revision history of the workspace it was created from, but has no linkages to any of the exposures that existed for the original workspace. However, you may navigate to the history of the original workspace and select any exposure, then select the wizard tab to the link to its exported structure, from which the exposure can be migrated over. Please see *the section on migrating exposure* for more details.

From the view page of your workspace, select "exposure rollover".



The exposure rollover button takes you to a list of revisions of the workspace, with existing exposures on the right hand side, and revision ids on the left. Each revision id has a radio button, used to select the revision you wish to create a new rolled over exposure for. Each existing exposure also has a radio button, used to select the exposure you wish to base your new one on. The most common use case is to select the latest exposure and the latest revision, and then click the *Migrate* button at the bottom of the list.

The new exposure will be created and displayed. When a new exposure is created, it is initially put in the *private* state. This means that only the user who created it or other users with appropriate permissions can see it, and it

Search Site

Models Home | My Workspaces | Exposures | Documentation | Demo User

You are here: Home / Workspaces / Beeler, Reuter, 1977

View | Edit | History | Files | Fork | Synchronize | Exposure Rollover | Sharing | Layout

Exposure Rollover

Changeset	Date	Author	Log	Options	Exposure
3037c7cbc7b1	2 minutes ago	Demo User	Removed paragraph about validation errors from the documentation	[files] [tgz] [zip]	(none)
fdd29a005ffc	2011-12-09	Dougal Cowan	Minor fix to documentation, changed title.	[files] [tgz] [zip]	Reconstruction of the action potential of ventricular myocardial fibres
afb6088af651	2011-12-09	Dougal Cowan	Fixed incorrect figure image.	[files] [tgz] [zip]	(none)
9c113c174a7c	2011-12-09	Dougal Cowan	Adding HTML version of documentation for the model.	[files] [tgz] [zip]	(none)
266541f690f7	2010-08-25	Tommy Yu	e-notation fix	[files] [tgz] [zip]	(none)
a5dfb07efdd3	2009-11-27	Hanne	Added images in ai and svg format	[files] [tgz] [zip]	(none)
d4ac7e982034	2009-06-17	pmr2.import	committing version08 of beeler_reuter_1977	[files] [tgz] [zip]	(none)
fbca003b1306	2008-05-19	pmr2.import	committing version07 of beeler_reuter_1977	[files] [tgz] [zip]	(none)
0972d9d006d0	2008-05-07	pmr2.import	committing version06 of beeler_reuter_1977	[files] [tgz] [zip]	(none)
3a3e1da600f9	2008-01-24	pmr2.import	committing version05 of beeler_reuter_1977	[files] [tgz] [zip]	(none)
54956de520c0	2007-03-26	pmr2.import	committing version04 of beeler_reuter_1977	[files] [tgz] [zip]	(none)
f34048e5adfa	2007-03-06	pmr2.import	committing version03 of beeler_reuter_1977	[files] [tgz] [zip]	(none)
ff7b583032b5	2007-03-05	pmr2.import	committing version02 of beeler_reuter_1977	[files] [tgz] [zip]	(none)
968f2d560b77	2006-09-04	pmr2.import	committing version01 of beeler_reuter_1977	[files] [tgz] [zip]	(none)

Migrate

will not appear in search results or model listings. In order to publish the exposure, you will need to select *submit for publication* from the *state* menu.

The state will change to "pending review". The administrator or curators of the repository will then review and publish the exposure, as well as expiring the old exposure.

Creating FieldML exposures

Section author: Dougal Cowan

FieldML models in the Auckland Physiome Repository are presented through *exposures*. A FieldML exposure has some similarities to a CellML exposure - usually consisting of a main documentation page with some information about the model, accompanied by a range of different views of the model data and or metadata. FieldML exposures also allow the real-time three-dimensional display of model meshes within the browser through the use of the *Zinc plugin*.

The example screenshots below show the main documentation page view and the 3D visualization provided by the Zinc viewer.

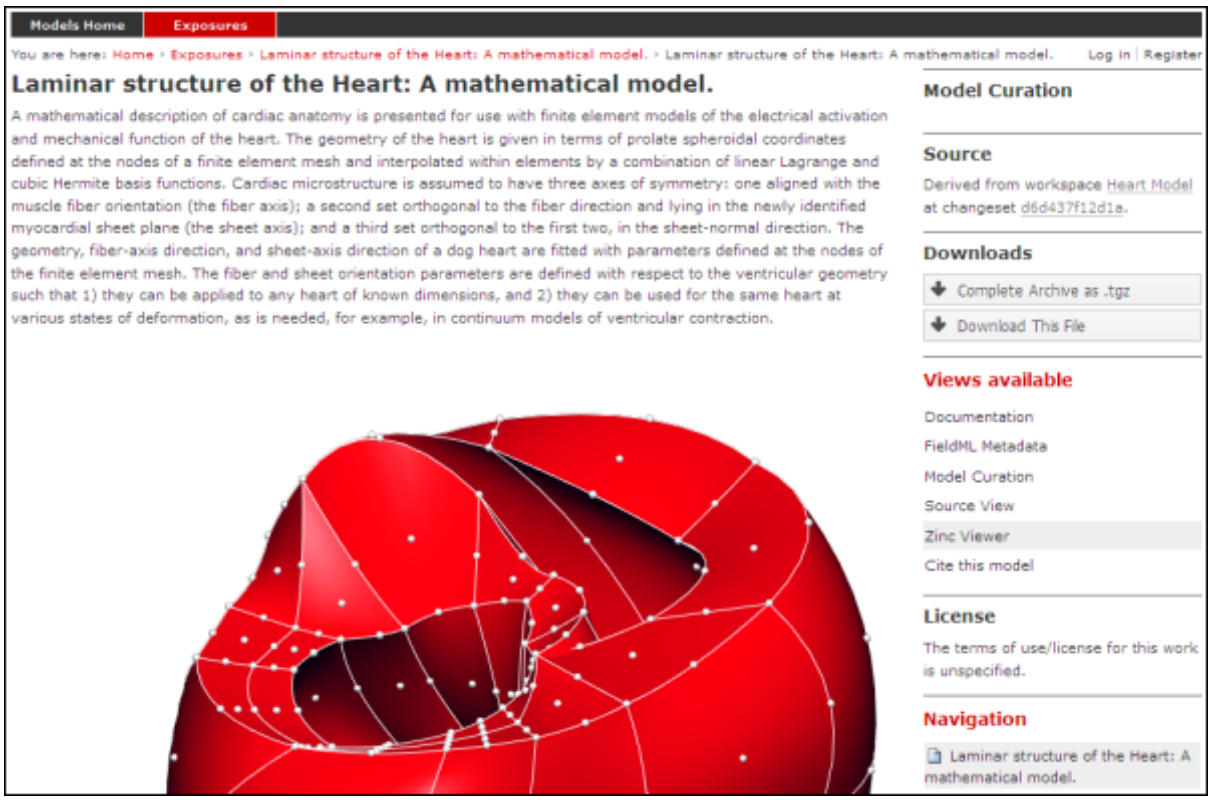


Fig. 2.19: The main documentation view of a FieldML exposure

Creating the exposure files

To create a FieldML exposure, the following files will need to be stored in a workspace in the repository:

- The FieldML model file(s)
- An RDF file containing metadata about the model, and specifying the JSON file to be used to specify the visualization.
- The JSON file that specifies the Zinc viewer visualization.

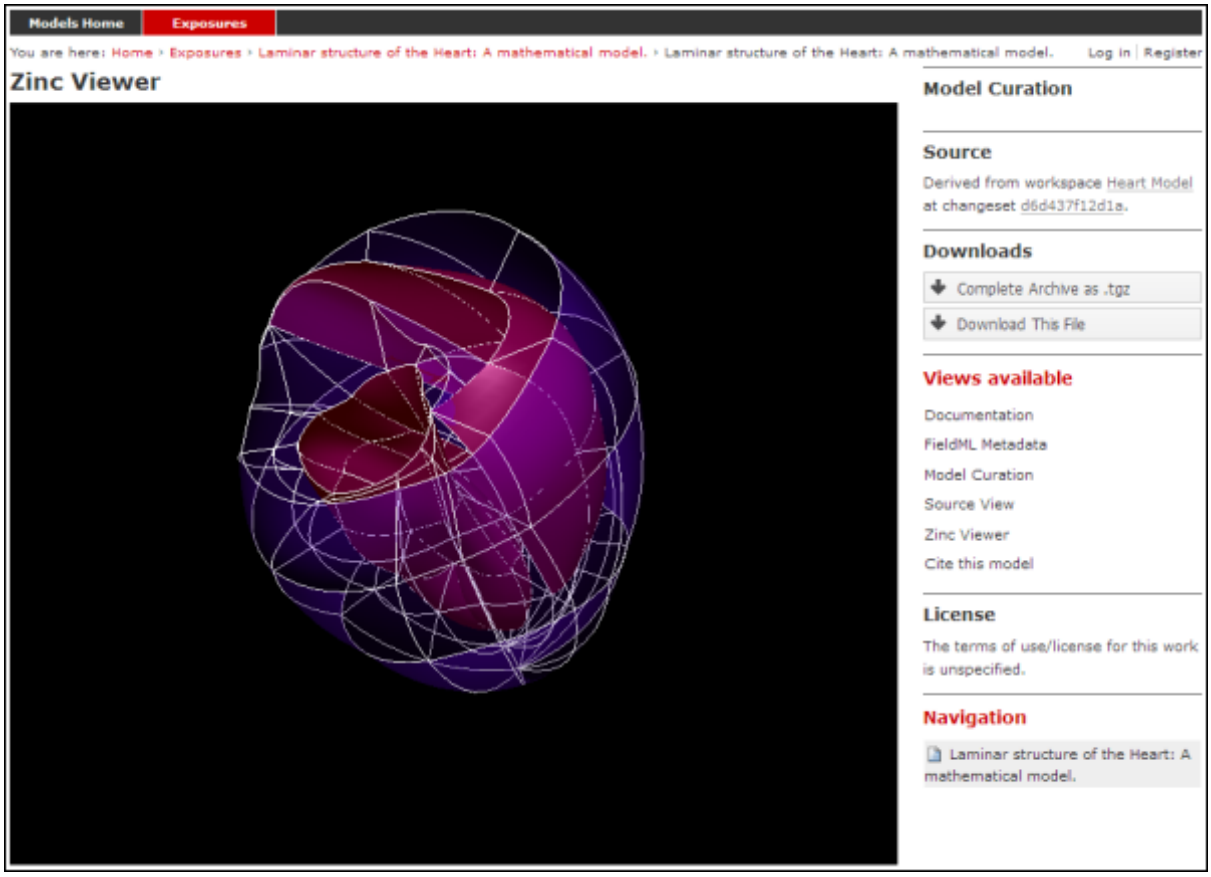


Fig. 2.20: The main Zinc viewer view of the same FieldML exposure

- Optionally, documentation (HTML) and images (PNG, JPG etc).

The following example RDF file from comes from the Laminar Structure of the Heart workspace in the repository:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <rdf:RDF
3        xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5        xmlns:dc="http://purl.org/dc/elements/1.1/"
6        xmlns:dcterms="http://purl.org/dc/terms/"
7        xmlns:vCard="http://www.w3.org/2001/vcard-rdf/3.0#"
8        xmlns:pmr2="http://namespace.physiomeproject.org/pmr2#">
9     <rdf:Description rdf:about="">
10       <dc:title>
11             Laminar structure of the Heart: A mathematical model.
12       </dc:title>
13       <dc:creator>
14          <rdf:Seq>
15             <rdf:li>LeGrice, I.J.</rdf:li>
16             <rdf:li>Hunter, P.J.</rdf:li>
17             <rdf:li>Smaill, B.H.</rdf:li>
18          </rdf:Seq>
19       </dc:creator>
20       <dcterms:bibliographicCitation>
21             American Journal of Physiology 272: H2466-H2476, 1997.
22       </dcterms:bibliographicCitation>
23       <dcterms:isPartOf rdf:resource="info:pmid/9176318"/>
24       <pmr2:annotation rdf:parseType="Resource">
25          <pmr2:type
26                rdf:resource="http://namespace.physiomeproject.org/pmr2/note#json_
   ↪zinc_viewer"/>
27          <pmr2:fields>
28             <rdf:Bag>
29                <rdf:li rdf:parseType="Resource">
30                   <pmr2:field rdf:parseType="Resource">
31                      <pmr2:key>json</pmr2:key>
32                      <pmr2:value>heart.json</pmr2:value>
33                   </pmr2:field>
34                </rdf:li>
35             </rdf:Bag>
36          </pmr2:fields>
37       </pmr2:annotation>
38    </rdf:Description>
39 </rdf:RDF>
```

This file provides citation metadata and a reference to the resource that specifies the Zinc viewer JSON file which will be used to describe the 3D visualisation of the FieldML model. The file breaks down into three main sections:

- Lines 3-8, namespaces used.
- Lines 10-23, citation metadata.
- Lines 24-37, resource description. Used to specify the JSON file that specifies the visualisation.

Example of the JSON file from the same (Laminar Structure of the Heart) workspace:

```
1  {
2      "View" : [
3        {
4        "camera" : [9.70448, -288.334, -4.43035],
5        "target" : [9.70448, 6.40667, -4.43035],
6        "up"     : [-1, 0, 0],
7        "angle" : 40
8        }
9      ],
```

```
10      "Models": [
11          {
12              "files": [
13                  "heart.xml"
14              ],
15              "externalresources": [
16                  "heart_mesh.connectivity",
17                  "heart_mesh.node.coordinates"
18              ],
19              "graphics": [
20                  {
21                      "type": "surfaces",
22                      "ambient" : [0.4, 0, 0.9],
23                      "diffuse" : [0.4, 0,0.9],
24                      "alpha" : 0.3,
25                      "xiFace" : "xi3_1",
26                      "coordinatesField": "heart.coordinates"
27                  },
28                  {
29                      "type": "surfaces",
30                      "ambient" : [0.3, 0, 0.3],
31                      "diffuse" : [1, 0, 0],
32                        "specular" : [0.5, 0.5, 0.5],
33                      "shininess" : 0.5,
34                      "xiFace" : "xi3_0",
35                      "coordinatesField" : "heart.coordinates"
36                  },
37                  {
38                      "type": "lines",
39                      "coordinatesField" : "heart.coordinates"
40                  }
41              ],
42              "elementDiscretization" : 8,
43              "region_name" : "heart",
44              "group": "Structures",
45              "label": "heart",
46              "load": true
47          }
48      ]
49  }
```

- Lines 2-8, sets up the camera or viewpoint for the initial Zinc viewer display.
- Lines 12-18, specifies the FieldML model files
- Lines 19-41, set up the actual visualisations of the mesh - in this case, two different surfaces and a set of lines.
- Lines 42-46, specify global visualisation settings.

For more information on these settings, please see the cmgui documentation.

Note: The specifics of these RDF and JSON files are a work in progress, and may change with each new version of the Zinc viewer plugin or *PMR2*.

Creating the exposure in the Auckland Physiome Repository

First you will need to create a workspace to put your model in, following the process outlined in the document on working with workspaces.

- Upload your FieldML model files and Zinc viewer specification files.

- Find revision of workspace you wish to expose and create exposure

Exposure wizard procedure

View generator as per CellML; select HTML annotator and HTML doc file

New exposure file entry: select .rdf file and select FieldML (JSON) type. Click *Add*.

- Documentation file - same as above
- Curation flags - none (should be removed?)
- No other settings

Click *Update*.

Click *Build*.

To see the 3D visualisation, you will need to have the latest Zinc plugin installed.

Embedded workspaces and their uses

Section author: David Nickerson

Todo

This section needs more work.

Workspaces in PMR are currently implemented as *Mercurial* repositories. One Mercurial feature that is quite useful in the context of the Auckland Physiome Repository is nested repositories. Using the more general *PMR2* concepts, we term such nesting as *embedded workspaces*.

Embedded workspaces:

- are intended to manage the separation of modules which are integrated to create a model;
- facilitate the sharing and reuse of model components independently from the source model;
- enable the development of the modules to proceed independently, thus the version of the workspaces embedded is also tracked; and
- allow authors to make use of relative URIs when linking between data resources providing a file system agnostic method to describe complex module relationships in a portable manner.

Workspaces can be embedded at a specific revision or set to track the most recent revision of the source workspace. Changes made to the source workspace will not affect any embedding workspace until the author explicitly chooses to update the embedded workspace. This provides the author with the opportunity to review the changesets and make an informed decision regarding alterations to embedded revisions. Any alterations in the specific revision of an embedded workspace is data captured in a changeset in the embedding workspace – thus providing a clear provenance record of the entire dataset in the workspace.

Uses

Best practice

See also the recommendations from the Mercurial project.

CellML Curation in the legacy Physiome Model Repository

As the Auckland Physiome Repository contains much of the data ported over from the legacy software products that powered what was called the CellML Model Repository, the curation system from that system was ported to Auckland Physiome Repository verbatim. This document describing the curation aspect of the repository is derived from documentation on the CellML site.

CellML Model Curation: the Theory

The basic measure of curation in a CellML model is described by the curation level of the model document. We have defined four levels of curation:

- Level 0: not curated.
- Level 1: the CellML model is consistent with the mathematics in the original published paper.
- Level 2: the CellML models has been checked for (i) typographical errors, (ii) consistency of units, (iii) that all parameters and initial conditions are defined, (iv) that the model is not over-constrained, in the sense that it contains equations or initial values which are either redundant or inconsistent, and (v) that running the model in an appropriate simulation environment reproduces the results published in the original paper.
- Level 3: the model is checked for the extent to which it satisfies physical constraints such as conservation of mass, momentum, charge, etc. This level of curation needs to be conducted by specialised domain experts.

CellML Model Curation: the Practice

Our ultimate aim is to complete the curation of all the models in the repository, ideally to the level that they replicate the results in the published paper (level 2 curation status). However, we acknowledge that for some models this will not be possible. Missing parameters and equations are just one limitation; at this point it should also be emphasised that the process of curation is not just about "fixing the CellML model" so that it runs in currently available tools. Occasionally it is possible for a model to be expressed in valid CellML, but not yet able to be solved by CellML tools. An example is the seminal Saucerman et al. 2003 model, which contains ODEs as well as a set of non-linear algebraic equations which need to be solved simultaneously. The developers of the CellML editing and simulation environment OpenCell are currently working on addressing these requirements.

The following steps describe the process of curating a CellML model:

- **Step 1:** the model is run through OpenCell and COR. COR in particular is a useful validation tool. It renders the MathML in a human readable format making it much easier to identify any typographical errors in the model equations. COR also provides a comprehensive error messaging system which identifies typographical errors, missing equations and parameters, and any redundancy in the model such as duplicated variables or connections. Once these errors are fixed, and assuming the model is now complete, we compare the CellML model equations with those in the published paper, and if they match, the CellML model is awarded a single star - or level 1 curation status.
- **Step 2:** Assuming the model is able to run in OpenCell and COR, we then go onto compare the CellML model simulation output from COR and OpenCell with the published results. This is often a case of comparing the graphical outputs of the model with the figures in the published paper, and is currently a qualitative process. If the simulation results from the CellML model and the original model match, the CellML model is awarded a second star - or level 2 curation status.
- **Step 3:** if, at the end of this process, the CellML model is still missing parameters or equations, or we are unable to match the simulation results with the published paper, we seek help from the original model author. Where possible, we try to obtain the original model code, and this often plays an invaluable role in fixing the CellML model.
- **Step 4:** Sometimes we have been able to engage the original model author further, such that they take over the responsibility of curating the CellML model themselves. Such models include those published by Mike Cooling and Franc Sachse. In these instances the CellML model is awarded a third star - or level 3 curation status. While this is laudable, ideally we would like to take the curation process one step further, such that

level 3 curation should be performed by a domain expert who is not the author of the original publication (i.e., peer review). This expert would then check the CellML model meets the appropriate constraints and expectations for a particular type of model.

A point to note is that levels 1 and 2 of the CellML model curation status may be mutually exclusive - in our experience, it is rare for a paper describing a model to contain no typographical errors or omissions. In this situation, Version 1 of a CellML model usually satisfies curation level 1 in that it reflects the model as it is described in the publication - errors included, while subsequent versions of the CellML model break the requirements for meeting level 1 curation in order to meet the standards of level 2. Taking this idea further, this means that a model with 2 yellow stars doesn't necessarily meet the requirements of level 1 curation but it does meet the requirements of level 2. Hopefully this conflict will be resolved when we replace the current star system with a more meaningful set of curation annotations.

Ultimately, we would like to encourage the scientific modeling community - including model authors, journals and publishing houses - to publish their models in CellML code in the Auckland Physiome Repository concurrent with the publication of the printed article. This will eliminate the need for code-to-text-to-code translations and thus avoid many of the errors which are introduced during the translation process.

CellML Model Simulation: the Theory and Practice

As part of the process of model curation, it is important to know what tools were used to simulate (run) the model and how well the model runs in a specific simulation environment. In this case, the theory and the practice are essentially the same thing, and carry out a series of simulation steps which then translate into a confidence level as part of a simulator's metadata for each model. The four confidence levels are defined as:

- Level 0: not curated (no stars);
- Level 1: the model loads and runs in the specified simulation environment (1 star);
- Level 2: the model produces results that are qualitatively similar to those previously published for the model (2 stars);
- Level 3: the model has been quantitatively and rigorously verified as producing identical results to the original published model (3 stars).

Glossary

Clone Clone is a Mercurial term that means to make a complete copy of a Mercurial repository. This is done in order to have a local copy of a repository to work in.

Embedded workspace

Embedded workspaces A Mercurial concept that allows workspaces to be nested within other workspaces.

Exposure

Exposures A publicly available page that provides access to and information about a specific revision of a workspace. Exposures are used to publish the contents of workspaces at points in time where the model(s) contained are considered to be useful.

Exposures are created by the PMR software, and offer views appropriate to the type of model being exposed. CellML files for example are presented with options such as code generation and mathematics display, whereas FieldML models might offer a 3D view of the mesh.

Fork A copy of the workspace which includes all the original version history, but is owned by the user who created the fork.

Mercurial Mercurial is a distributed version control system, used by the Physiome Model Repository software to maintain a history of changes to files in *workspaces*. See a tour of the Mercurial basics for some good introductory material.

PMR2 The software that powers the Auckland Physiome Repository.

Pull

Pulling The term used with distributed version control systems for the action of pulling changes from one clone of the repository into another. With PMR, this usually implies pulling from a workspace in the model repository into a clone of the workspace on your local machine.

Push

Pushing The term used with distibuted version control systems for the action of pushing changes from one clone of the repository into another. With PMR, this usually implies pushing from a workspace clone on your local machine back to the workspace in the model repository, but could be into any other clone of the workspace. See a tour of the Mercurial basics for some good introductory material.

Python Python is a programming language that lets you work more quickly and integrate your systems more effectively. See http://python.org for all the details.

Synchronize Used to pull the contents or changes from other *Mercurial* repositories into a workspace via a URI.

Workspace

Workspaces A *Mercurial* repository hosted on the Physiome Model Repository. This is essentially a folder or directory in which files are stored, with the added feature of being version controlled by the distributed version control system called Mercurial.

Note: The teaching instance of the repository is a mirror of the main repository site found at http://models.physiomeproject.org/, running the latest development version of *PMR2*.

Any changes you make to the contents of the teaching instance are not permanent, and will be overwritten with the contents of the main repository whenever the teaching instance is upgraded to a new release of PMR2. For this reason, you can feel free to experiment and make mistakes when pushing to the teaching instance. Please subscribe to the cellml-discussion mailing list to receive notifications of when the teaching instance will be refreshed.

See the section *Migrating content to the main repository* for instructions on how to migrate any content from the teaching instance to the main (permanent) Auckland Physiome Repository.

Using SED-ML to specify simulations

Section author: Dougal Cowan

Hopefully PMR will support SED-ML simulations as part of the CellML views.

Todo

- Update all documentation to reflect workspace ID changes and user workspace changes, if they go ahead.
- Get embedded workspaces doc written.
- Get some best practice docs written.

CHAPTER 3

OpenCOR

OpenCOR is an open source, cross-platform and CellML-based modelling environment. The following documentation refers to the `0.3` version of OpenCOR, for which supported platforms can be found *here*. This version of OpenCOR can be downloaded from the OpenCOR download page.

User Interfaces

OpenCOR provides two types of user interfaces:

Command Line Interface (CLI)

Help

```
$ ./OpenCOR -h
Usage: OpenCOR [-a|--about] [-c|--command [<plugin>::]<command> <options>] [-h|--
↪help] [-p|--plugins] [-s|--status] [-v|--version] [<files>]
-a, --about     Display some information about OpenCOR
-c, --command   Send a command to one or all the CLI plugins
-h, --help      Display this help information
-p, --plugins   Display all the CLI plugins
-s, --status    Display the status of all the plugins
-v, --version   Display the version of OpenCOR
```

Version

```
$ ./OpenCOR -v
OpenCOR 0.3 (64-bit)
```

About

```
$ ./OpenCOR -a
OpenCOR 0.3 (64-bit)
OS X 10.9 (Mavericks)
Copyright 2011-2014

OpenCOR is a cross-platform CellML-based modelling environment, which can be used␣
↪to organise, edit, simulate and analyse CellML files.
```

Plugins

```
$ ./OpenCOR -p
The following plugin is loaded:
 - CellMLTools: a plugin to access various CellML-related tools.
```

Status

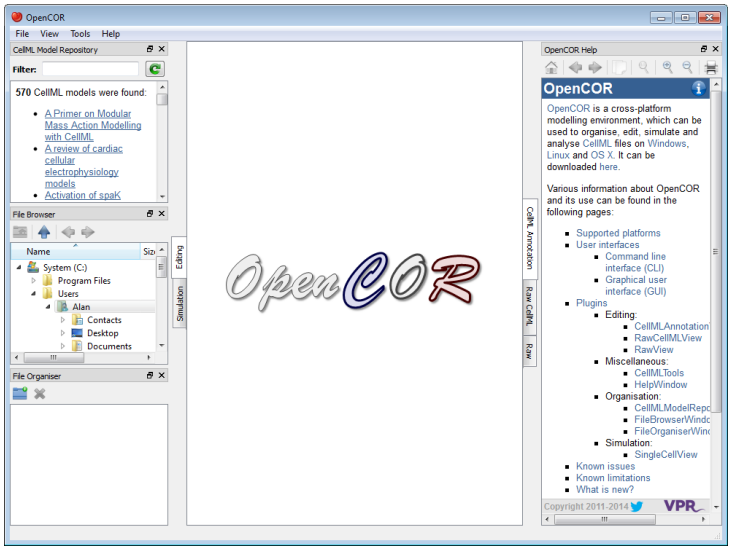
```
$ ./OpenCOR -s
The following plugins are available:
 - CellMLAPI: the plugin is loaded and fully functional.
 - CellMLSupport: the plugin is loaded and fully functional.
 - CellMLTools: the plugin is loaded and fully functional.
 - Compiler: the plugin is loaded and fully functional.
 - Core: the plugin is loaded and fully functional.
 - CoreSolver: the plugin is loaded and fully functional.
 - LLVM: the plugin is loaded and fully functional.
```

Command

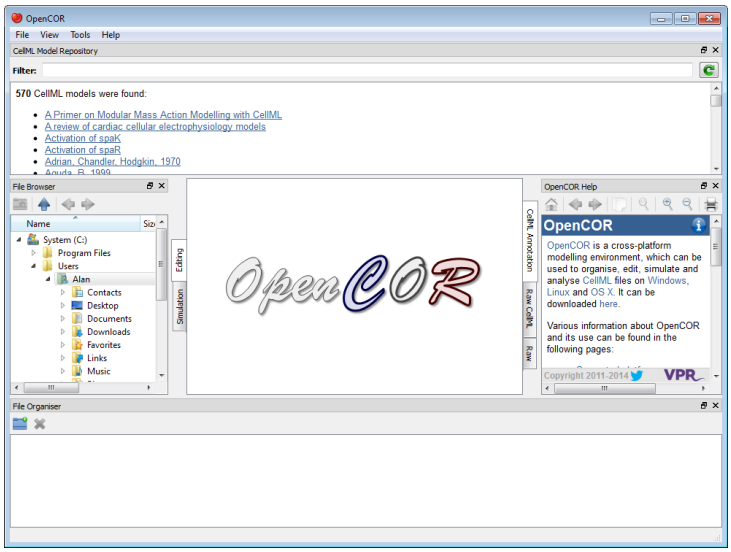
```
$ ./OpenCOR -c help
Commands supported by CellMLTools:
 * Display the commands supported by CellMLTools:
      help
 * Export <in_file> to <out_file> using <predefined_format> as the destination␣
↪format or <user_defined_format_file> as the file describing the destination␣
↪format:
      export <in_file> <out_file> [<predefined_format>|<user_defined_format_file>]
   <predefined_format> can take one of the following values:
      cellml_1_0: to export a CellML 1.1 file to CellML 1.0
$ ./OpenCOR -c CellMLTools::export in.cellml out.cellml cellml_1_0
$ ./OpenCOR -c CellMLTools::export http://mydomain.com/in.cellml out.txt format.xml
```

Graphical User Interface (GUI)

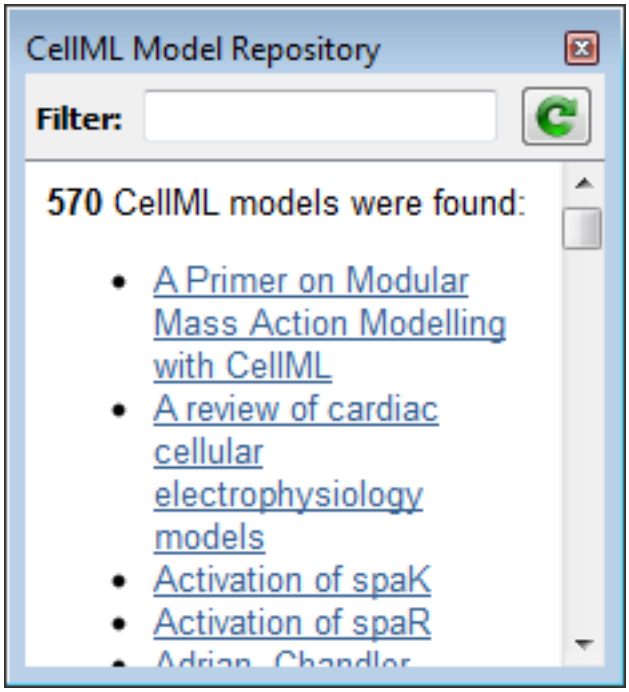
OpenCOR offers a consistent GUI across the *different platforms* it supports. The look and feel of the interface is determined by the *plugins* which are selected. The first time you run OpenCOR, it will look something like this:



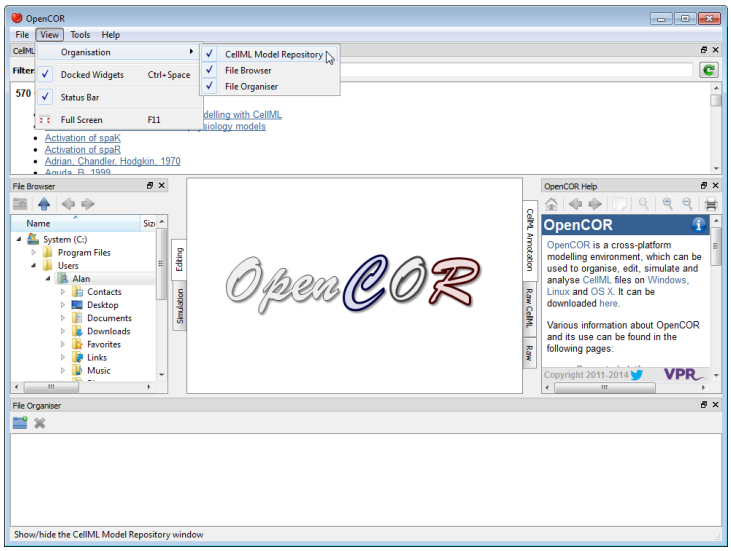
The central area is used to interact with files. By default, no files are open, hence the OpenCOR logo is shown instead. To the sides, there are dockable windows, which provide additional features. Those windows can be dragged and dropped to the top or bottom of the central area:



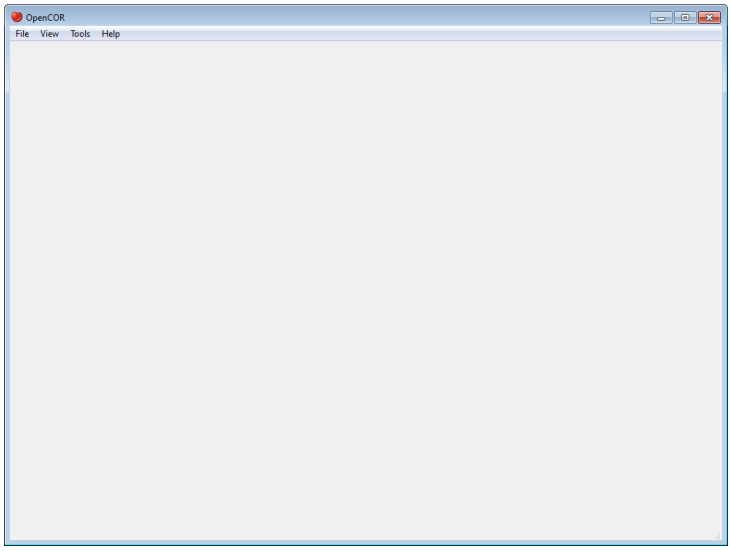
Alternatively, they can be undocked:



Or even closed, either by directly closing the window itself or by unticking the corresponding menu item (under the *View* menu, or the *Help* menu for the Help window):



To unselect all the *plugins* will result in OpenCOR looking 'empty':



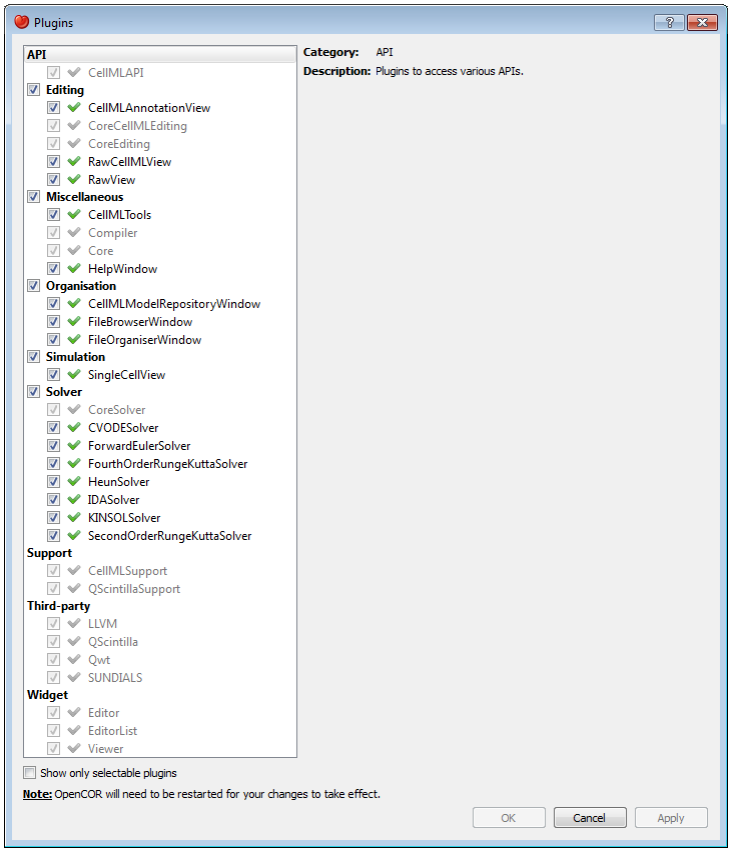
Menu

- **File:**
  - **Exit ~ Alt+F4:** exit OpenCOR.
- **View:**
  - **Status Bar:** show/hide the status bar.
  - **Full Screen ~ F11:** switch to / back from full screen mode.
- **Tools:**
  - **Language:** select the language to be used by OpenCOR.
  - **Plugins...:** un/select plugins.
  - **Reset All:** reset all your settings.
- **Help:**
  - **Home Page:** open the OpenCOR home page.
  - **About...:** some general information about OpenCOR.

Plugins

OpenCOR is a plugin-based application. This means that if no plugins are selected, then OpenCOR can do *next to nothing*.

As can be seen by opening the Plugins dialog box (by selecting the *Tools* → *Plugins* menu) and by unselecting *Show only selectable plugins* (if necessary), OpenCOR supports different types of plugins:



You can select which plugins you want to use. However, plugins which are needed by other plugins (e.g. the Core plugin is needed by the *CellMLModelRepositoryWindow* plugin) cannot be directly selected. Instead, they will be automatically selected if and only if they are needed by at least one other plugin.

Most of the selectable plugins come with some kind of a *GUI*, which is of one of two types:

- **Window:** such a plugin (e.g. the *CellMLModelRepositoryWindow* and *HelpWindow* plugins) can be docked around the central area, undocked or hidden, as illustrated *here*.
- **View:** such a plugin (e.g. the *CellMLAnnotationView* and *SingleCellView* plugins) is used to interact with a file, be it to edit it, simulate it or analyse it.

API

(Non-selectable) API plugins are used to provide access to various APIs:

- **CellMLAPI:** a plugin to access the CellML API.

Data Store

Data Store plugins are used to store and manipulate simulation data:

- **CSVDataStore:** a CSV specific data store plugin.

There is also one non-selectable Data Store plugin:

- **CoreDataStore:** the core data store plugin.

Editing

Editing plugins are used to edit files:

- *CellMLAnnotationView*: a plugin to annotate CellML files.
- *RawCellMLView*: a plugin to edit CellML files using the raw CellML format.
- *RawView*: a plugin to edit any file.

There are also some non-selectable Editing plugins:

- **CoreCellMLEditing:** the core CellML editing plugin.
- **CoreEditing:** the core editing plugin.

Miscellaneous

Miscellaneous plugins are used for various purposes:

- *CellMLTools*: a plugin to access various CellML-related tools.
- *HelpWindow*: a plugin to provide help.

There are also some non-selectable Miscellaneous plugins:

- **Compiler:** a plugin to support code compilation.
- **Core:** the core plugin.

Organisation

Organisation plugins are used to organise files:

- *CellMLModelRepositoryWindow*: a plugin to access the CellML Model Repository.
- *FileBrowserWindow*: a plugin to access your local files.
- *FileOrganiserWindow*: a plugin to virtually organise files.

Simulation

Simulation plugins are used to simulate files:

- *SingleCellView*: a plugin to run single cell simulations.

Solver

Solver plugins are used to provide access to various solvers:

- CVODESolver: a plugin that uses CVODE to solve ODEs.
- ForwardEulerSolver: a plugin that implements the Forward Euler method to solve ODEs.
- FourthOrderRungeKuttaSolver: a plugin that implements the fourth-order Runge-Kutta method to solve ODEs.
- HeunSolver: a plugin that implements the Heun method to solve ODEs.
- IDASolver: a plugin that uses IDA to solve DAEs.
- KINSOLSolver: a plugin that uses KINSOL to solve non-linear algebraic systems.
- SecondOrderRungeKuttaSolver: a plugin that implements the second-order Runge-Kutta method to solve ODEs.

There is also a non-selectable Solver plugin:

- CoreSolver: the core solver plugin.

Support

(Non-selectable) support plugins are used to provide support for various third-party libraries and APIs:

- CellMLSupport: a plugin to support CellML.
- QScintillaSupport: a plugin to support QScintilla.

Third-party

(Non-selectable) third-party plugins are used to provide access to various third-party libraries:

- LLVM: a plugin to access LLVM (as well as Clang).
- QScintilla: a plugin to access QScintilla.
- Qwt: a plugin to access Qwt.
- SUNDIALS: a plugin to access CVODE, IDA and KINSOL solvers from the SUNDIALS library.

Widget

(Non-selectable) widget plugins are used to provide access to various ad hoc widgets:

- Editor: a plugin to edit and display text.
- EditorList: a plugin to handle issues in a text editor.
- Viewer: a plugin to visualise mathematical equations.

Glossary

Clone Clone is a Mercurial term that means to make a complete copy of a Mercurial repository. This is done in order to have a local copy of a repository to work in.

Embedded workspace

Embedded workspaces A Mercurial concept that allows workspaces to be nested within other workspaces.

Exposure

Exposures A publicly available page that provides access to and information about a specific revision of a workspace. Exposures are used to publish the contents of workspaces at points in time where the model(s) contained are considered to be useful.

Exposures are created by the PMR software, and offer views appropriate to the type of model being exposed. CellML files for example are presented with options such as code generation and mathematics display, whereas FieldML models might offer a 3D view of the mesh.

Fork A copy of the workspace which includes all the original version history, but is owned by the user who created the fork.

Mercurial Mercurial is a distributed version control system, used by the Physiome Model Repository software to maintain a history of changes to files in *workspaces*. See a tour of the Mercurial basics for some good introductory material.

Pull

Pulling The term used with distributed version control systems for the action of pulling changes from one clone of the repository into another. With PMR, this usually implies pulling from a workspace in the model repository into a clone of the workspace on your local machine.

Push

Pushing The term used with distibuted version control systems for the action of pushing changes from one clone of the repository into another. With PMR, this usually implies pushing from a workspace clone on your local machine back to the workspace in the model repository, but could be into any other clone of the workspace. See a tour of the Mercurial basics for some good introductory material.

Python Python is a programming language that lets you work more quickly and integrate your systems more effectively. See http://python.org for all the details.

Synchronize Used to pull the contents or changes from other *Mercurial* repositories into a workspace via a URI.

Workspace

Workspaces A *Mercurial* repository hosted on the Physiome Model Repository. This is essentially a folder or directory in which files are stored, with the added feature of being version controlled by the distributed version control system called Mercurial.

Supported platforms

OpenCOR can be used on the following versions of Windows, Linux and OS X.

Windows

OpenCOR is supported on the 32-bit and 64-bit versions of Windows XP and later.

Linux

- OpenCOR 0.1.x and 0.2: supported on both the 32-bit and 64-bit versions of Ubuntu 12.04 LTS (Precise Pangolin) and later.
- OpenCOR 0.3: supported on both the 32-bit and 64-bit versions of Ubuntu 14.04 LTS (Trusty Tahr) and later.

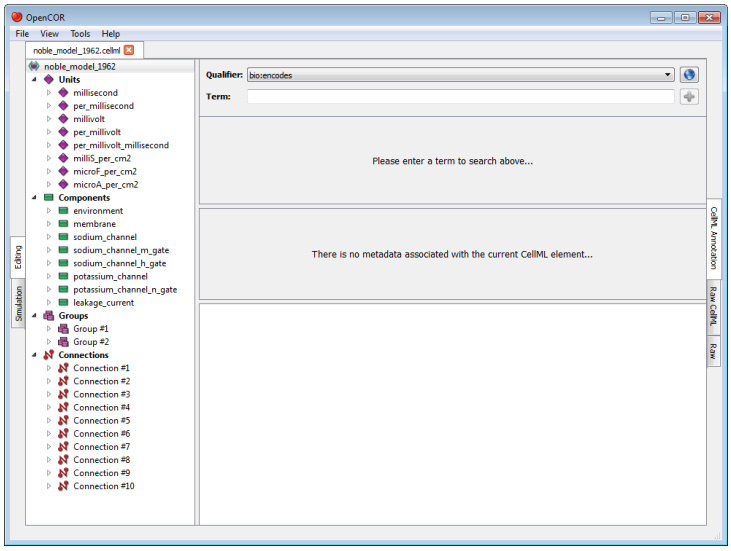
Note: in all cases, OpenCOR may also work with earlier versions of Ubuntu, as well as with other Linux distributions, but additional system libraries may be needed in the latter case.

OS X

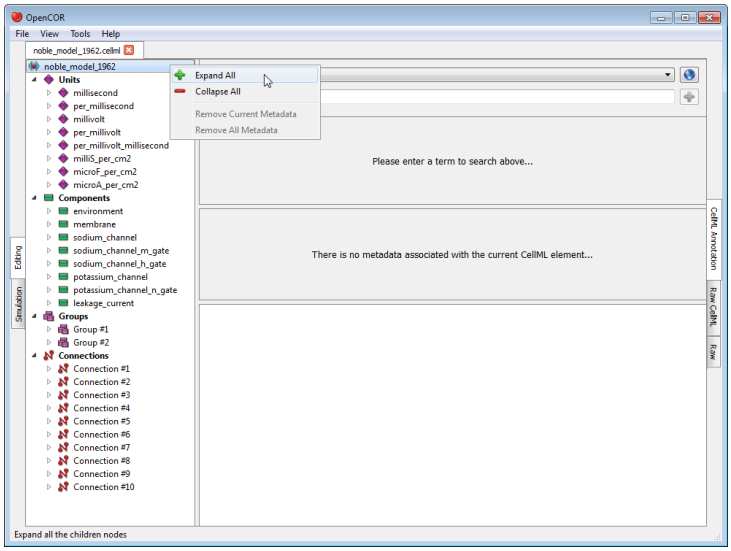
- OpenCOR 0.1.x: supported on OS X 10.8 (Mountain Lion) and later.
- OpenCOR 0.2 and later: supported on Mac OS X 10.7 (Lion) and later.

CellMLAnnotationView Plugin

The CellMLAnnotationView plugin can be used to annotate CellML files. If you open a CellML file which does not contain any annotation, then it will look something like this:

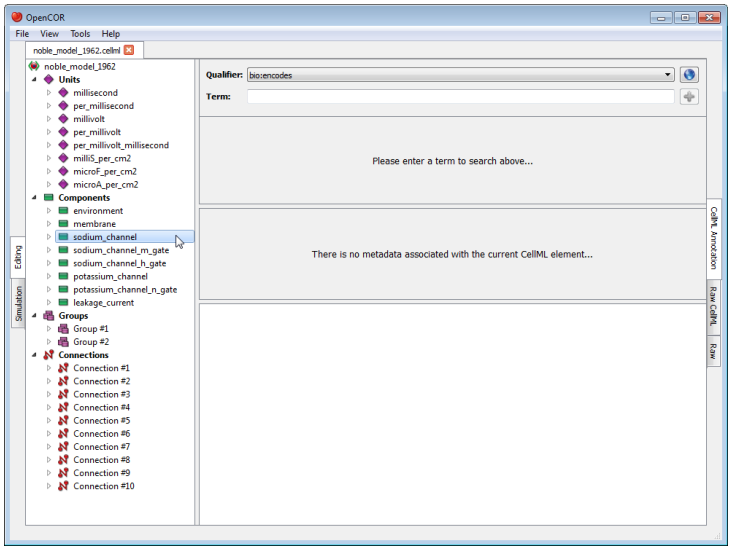


All the CellML elements which can be annotated are listed to the left of the view. If you right click on any of them, you will get a popup menu which you can use to expand/collapse all the child nodes, as well as remove the metadata associated with the current CellML element or the whole CellML file:

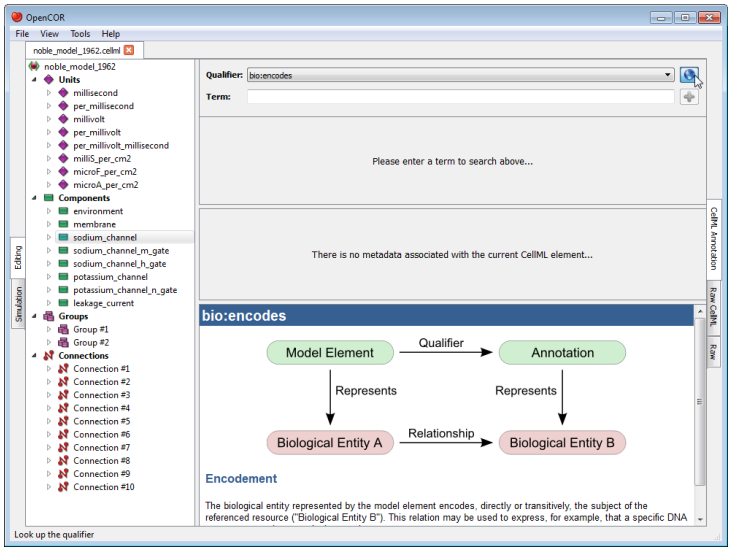


Annotate a CellML element

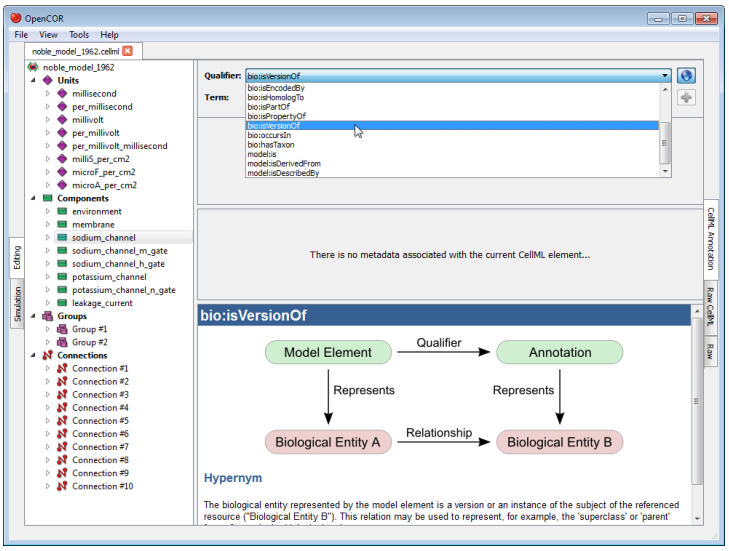
Say that you want to annotate the `sodium_channel` component. First, you need to select it:



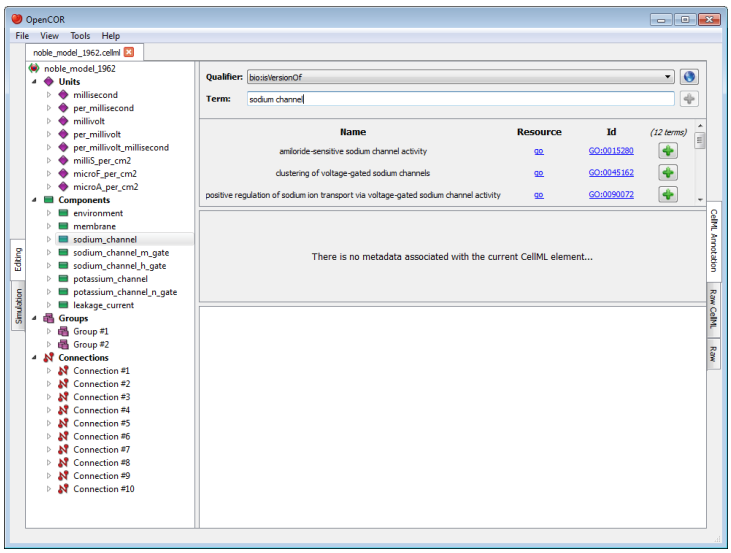
Next, you need to specify a BioModels.net qualifier. If you do not know which one to use, click on the button to get some information about the current BioModels.net qualifier:



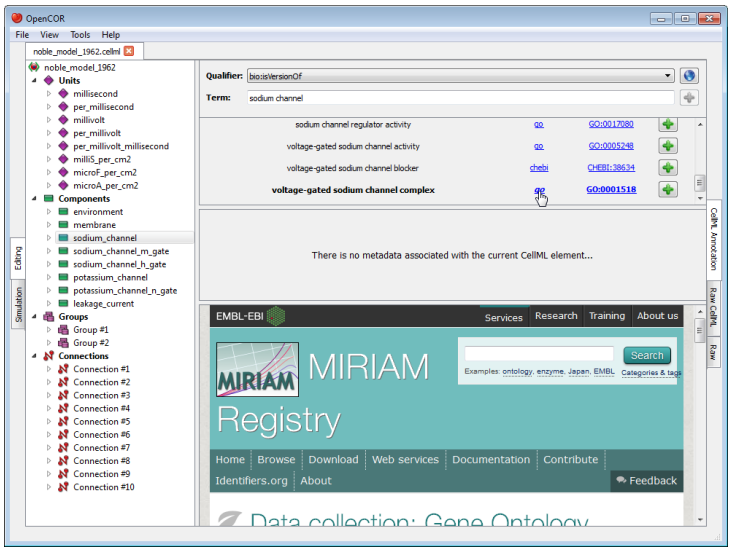
From there, go through the list of BioModels.net qualifiers until you find the one you are happy with. Here, we will use `bio:isVersionOf`:



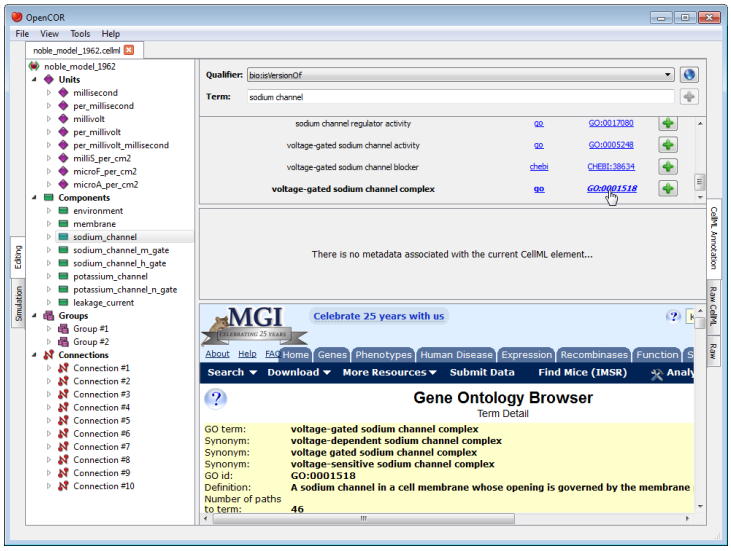
Now, you need to retrieve some possible ontological terms to describe the `sodium_channel` component. For this, you must enter a search term which in our case is going to be `sodium channel` (note: regular expressions are supported). As can be seen, OpenCOR returns 12 possible ontological terms:



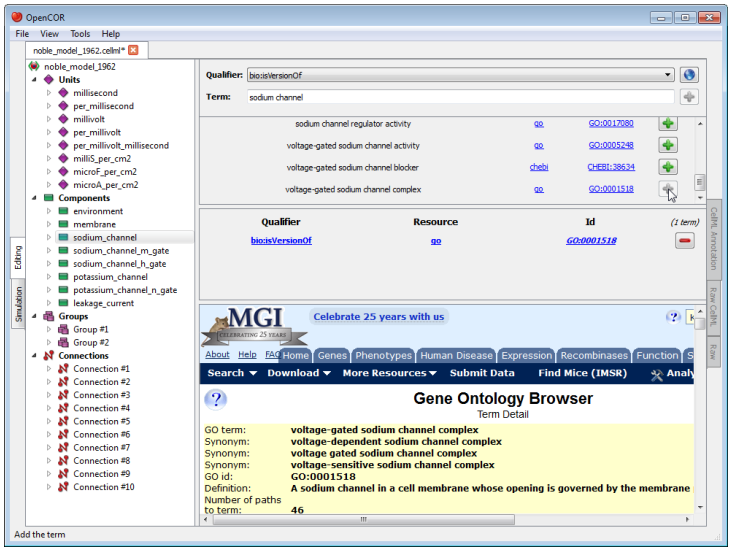
A quick look through the list tells us that you might want to use the one for `voltage-gated sodium channel complex`. If you want to know more about the GO resource, you can click on its corresponding link:



Similarly, if you want to know more about the GO identifier:

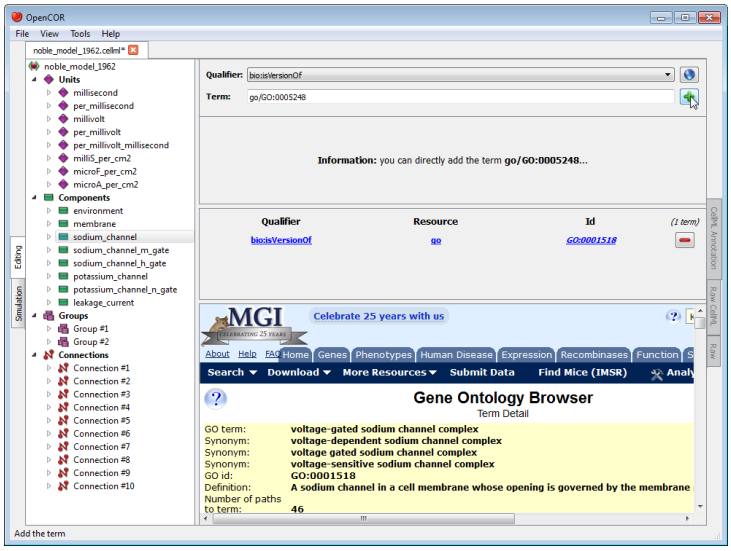


Now that you are happy with your choice of ontological term, you can associate it with the `sodium_channel` component by clicking on its corresponding button:



As you will have seen, the ontological term you have just added cannot be added anymore, but it can be removed by clicking on its corresponding button or by using the context menu (see above).

Now, say that you also want to add the next ontological term. You can obviously do so by clicking on the corresponding button, but you could also enter its resource-id duple, e.g. `go/GO:0005248` (i.e. `<resource>/<id>`) in the term field. Indeed, OpenCOR will recognise this 'term' as being a a resource-id duple and will offer you to add its corresponding ontological term directly:

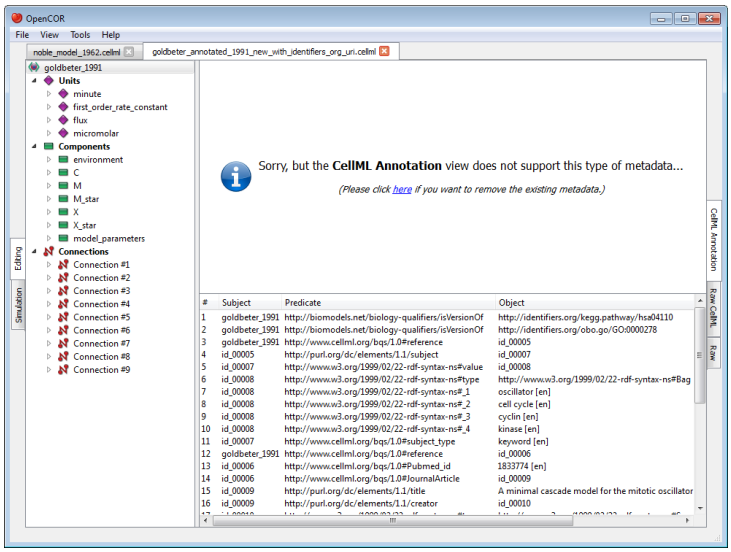


Unrecognised annotations

Annotations consist of RDF triples which are made of a subject, a predicate and an object. OpenCOR recognises RDF triples which subject identifies a CellML element while it expects the predicate to be a BioModels.net qualifier and the object an ontological term.

Ontological terms used to be identified using MIRIAM URNs, but these have now been deprecated in favour of identifiers.org URIs. OpenCOR recognises both, but it will only serialise annotations using identifiers.org URIs.

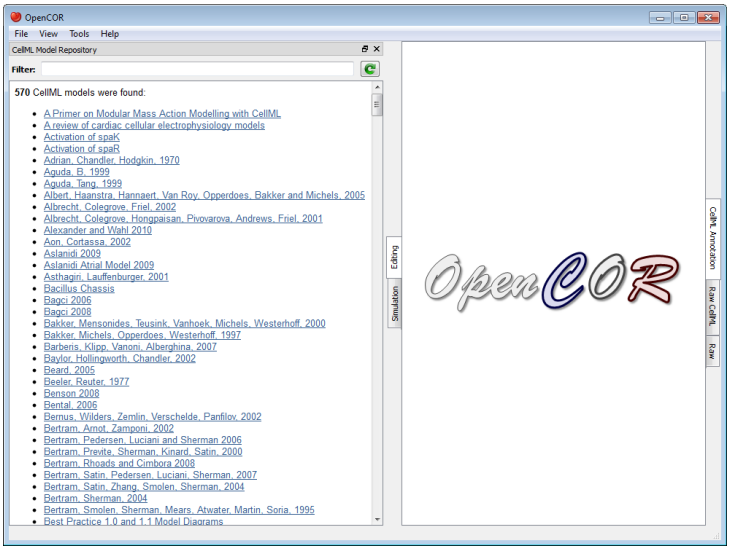
Now, it may happen that a file contains annotations that are not recognised by OpenCOR. In this case, OpenCOR will display the annotations as a simple list of RDF triples:



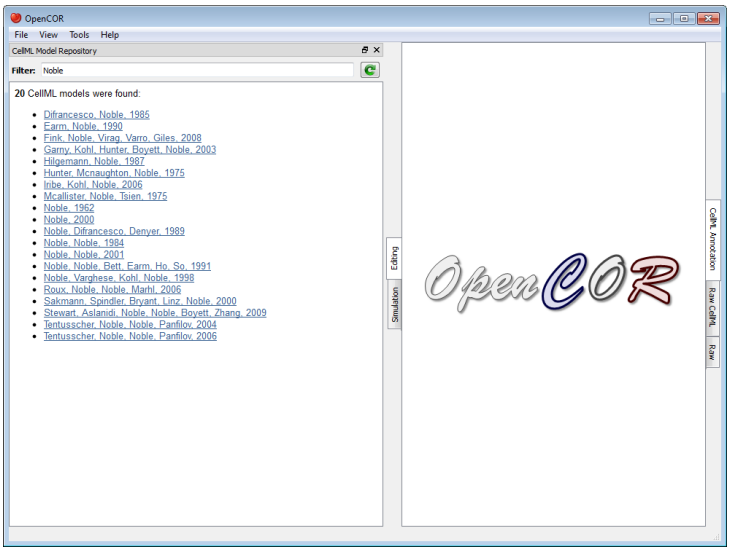
If you ever come across a type of annotations which you think OpenCOR ought to recognise, but does not, then please do contact us.

CellMLModelRepositoryWindow Plugin

The CellMLModelRepositoryWindow plugin offers an interface to the CellML Model Repository. By default, it lists all the CellML models found in the repository:



The list can then be filtered. For example, if you enter `Noble` as a filter, you will get:



To click on any of the listed links will open the *workspace* for that model in your (default) web browser. From there, you can retrieve the latest *exposure* for that model.

CellMLTools Plugin

The CellMLTools plugin consists of various CellML-related tools, which can be accessed through the *Tools* menu.

CellML File Export To...

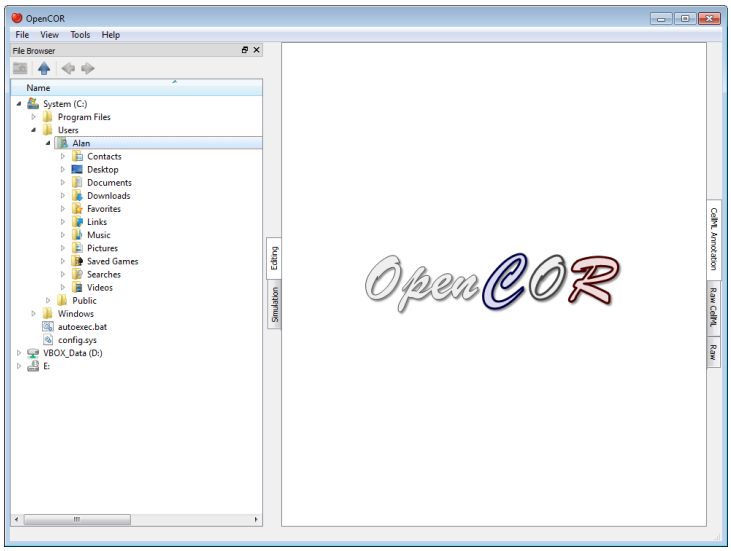
These tools can be used to export a CellML model to various formats:

- **CellML 1.0:** to flatten a CellML 1.1 model.
- **User-defined format:** to export a CellML model to some user-defined format.

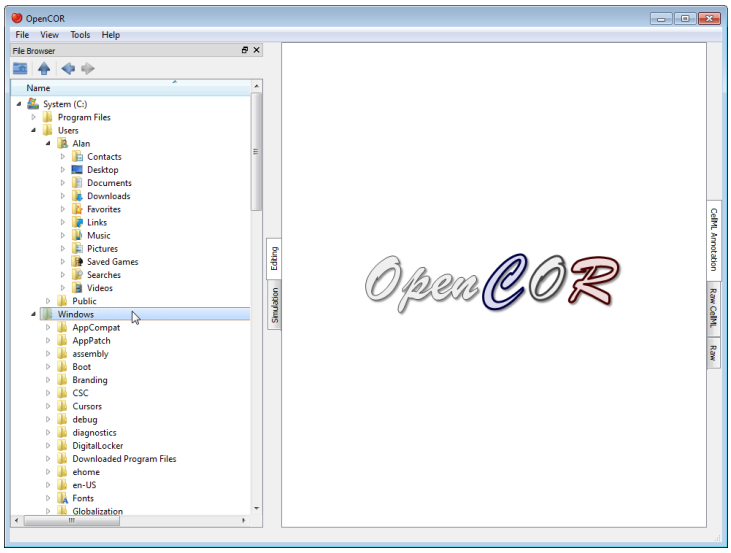
Note: The CellML 1.0 export is adapted from Jonathan Cooper's CellML 1.1 to 1.0 converter and therefore has the same limitations.

FileBrowserWindow Plugin

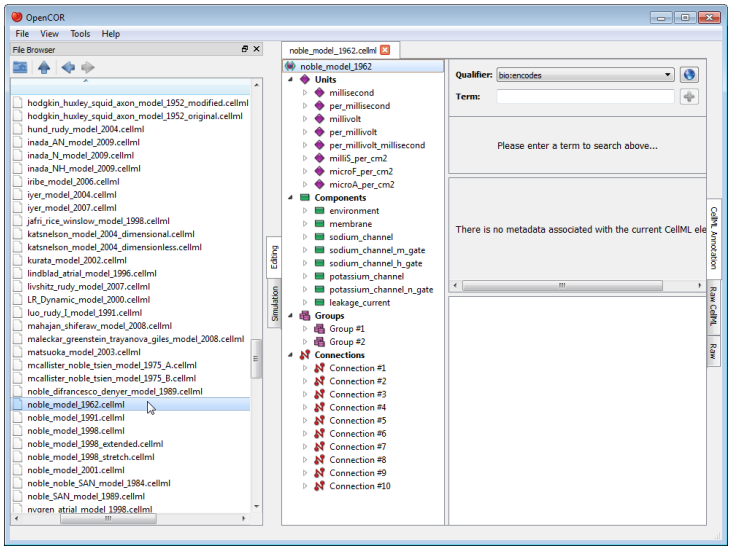
The FileBrowserWindow plugin offers a convenient way to access your physical files, remembering the folder or file that was selected when you last ran OpenCOR. By default, it will select your home directory:



As you would expect, to double click on a folder will expand its contents, as can be seen by double clicking on the Windows directory:



On the other hand, to double click on a file will result in it being opened in OpenCOR. The rendering of the file will depend on the current view being selected. In the case of the *CellML Annotation* view, it will look something like this:



Folders and files can also be dragged from the File Browser window and dropped onto the *File Organiser* window.

Tool bar

Go to the home folder

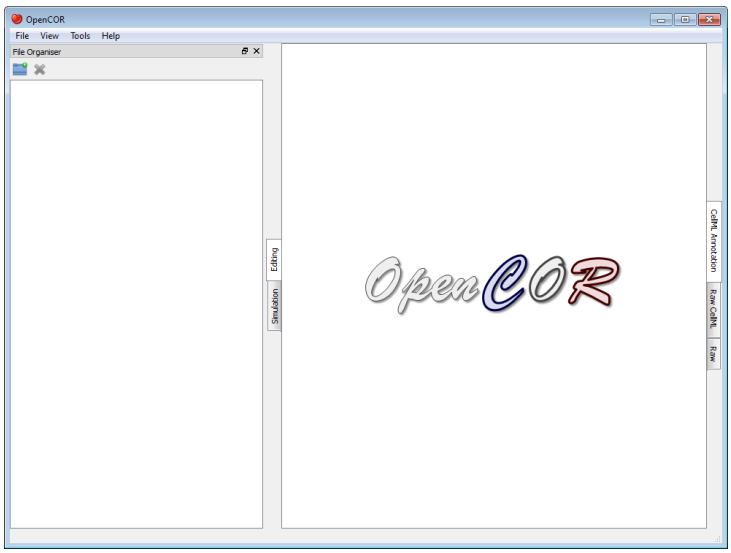
Go to the parent folder

Go to the previous folder or file

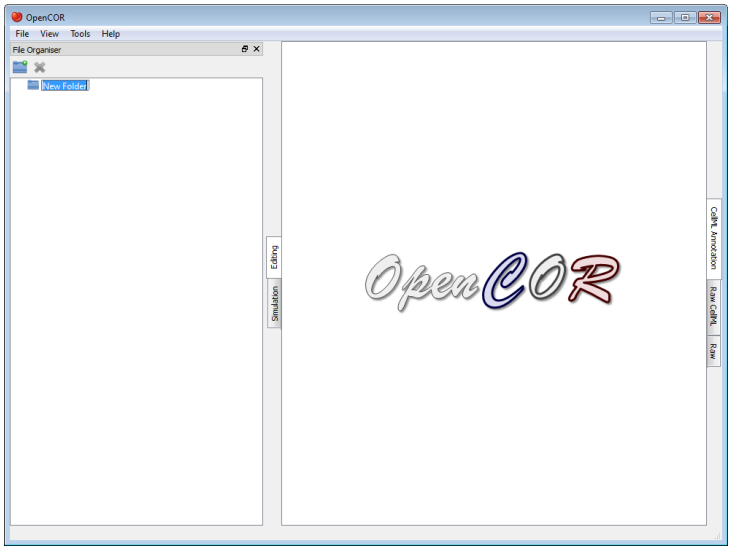
Go to the next folder or file

FileOrganiserWindow Plugin

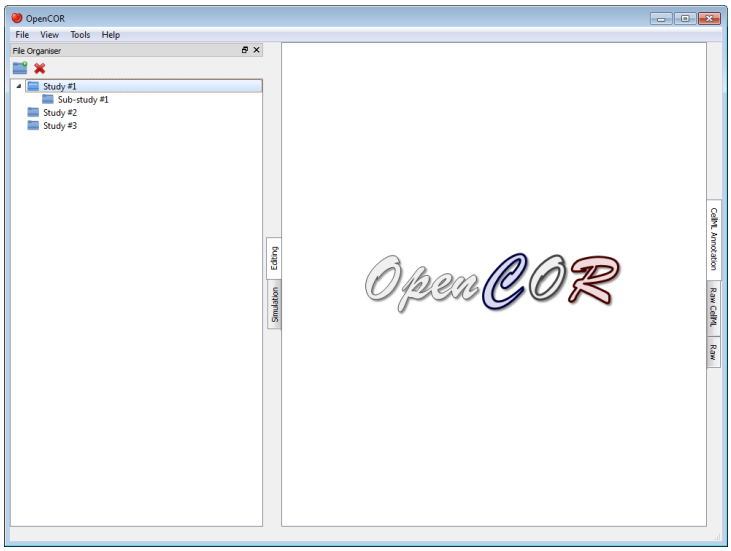
The FileOrganiserWindow plugin allows you to organise your files in a virtual manner, i.e. independently of where they are physically located. Your virtual environment is remembered from one session to another and is originally empty:



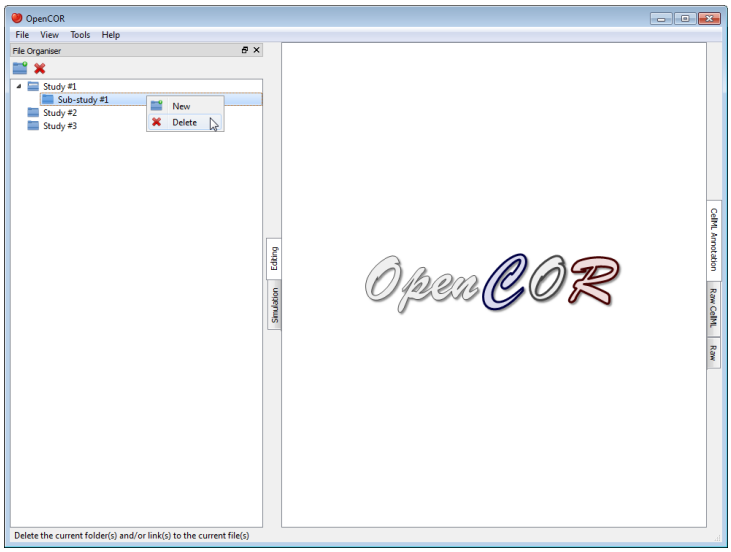
Now, say that you are working on a specific project. You might then want to create a (virtual) folder, which contains (a virtual link to) all the files you need for your project. For this, you first need to click on the button in the toolbar (or use the context menu). This will add a folder to your virtual environment:



You can rename the folder as you wish, as well as create other (sub-)folders, if needed:

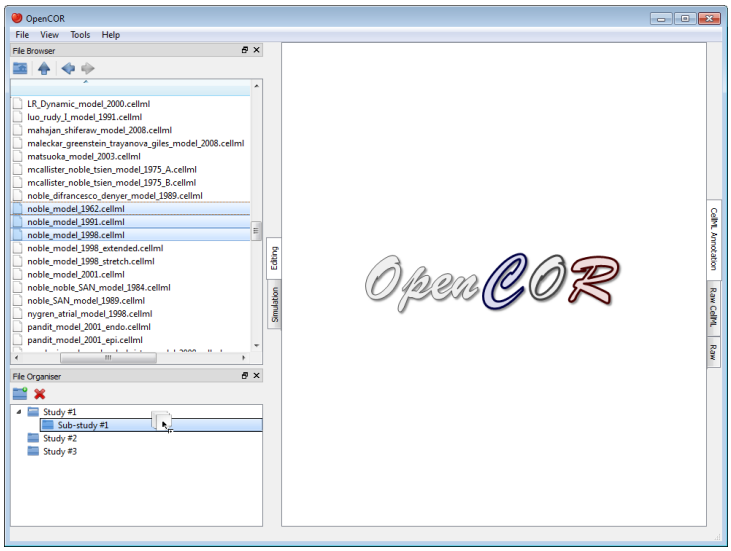


You can also move the (sub-)folders around by dragging and dropping them within your virtual environment, or delete an existing (sub-)folder by clicking on the ✖ button in the toolbar (or by using the context menu):

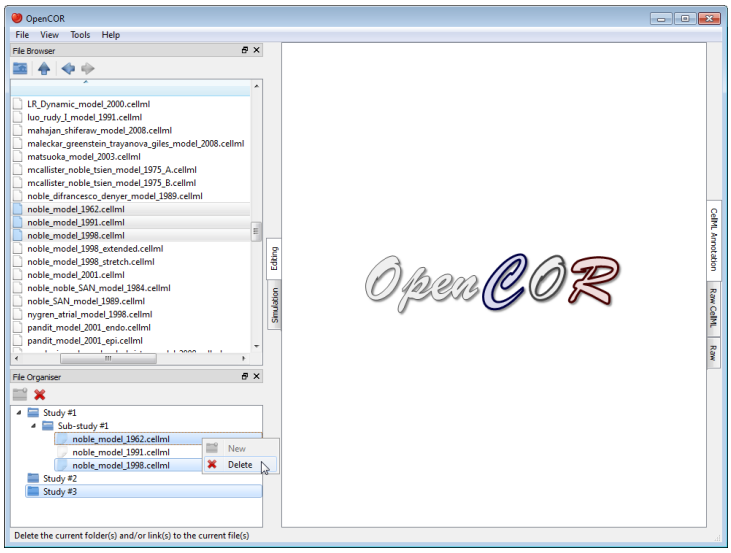


Next, you might want to open the *File Browser* window, so you can start dragging and dropping files into your

virtual environment (alternatively, you can use your system's file manager):



As for folders, you can move and delete your (virtual) files:



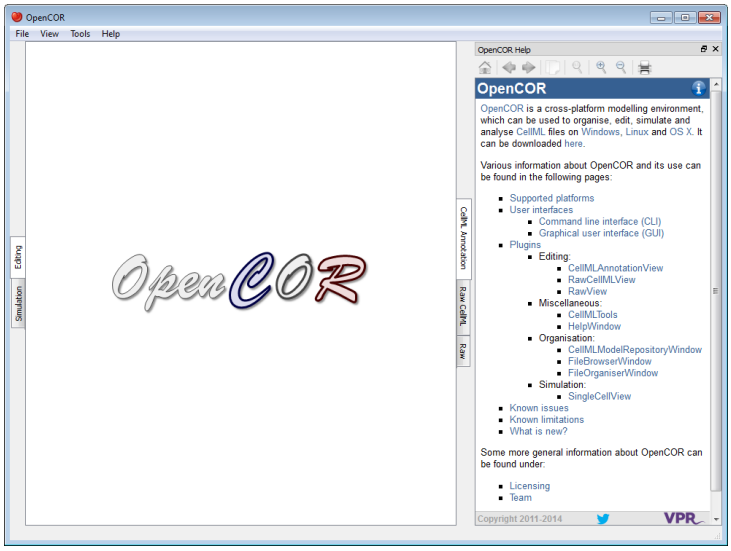
Tool bar

Create a new folder

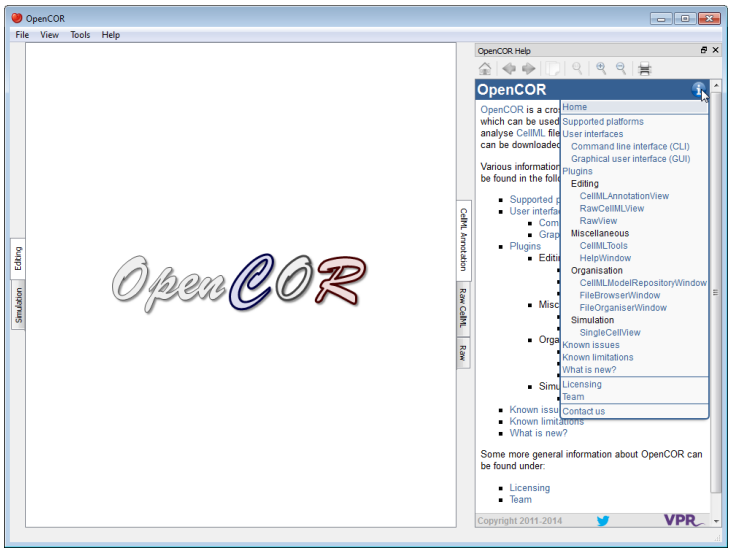
Delete the current folder(s) and/or link(s) to the current file(s)

HelpWindow Plugin

The HelpWindow plugin provides some user documentation that looks as follows:



The contents of the documentation is the same as the one that can be found in the user documentation section of the OpenCOR website. This includes a menu that gets shown whenever you move your mouse pointer over the information icon (top right):



In addition to what is shown on the website, the HelpWindow plugin also displays special links, which when clicked send a command to OpenCOR. For example, open the current page both in OpenCOR and on the OpenCOR website. Now, if you check the bold text below, you will see that its contents is slightly different, depending on whether you are reading this in OpenCOR or from the OpenCOR website:

To open the About box, select the *Help* → *About*... **menu...**

Tool bar

Go to the home page

Go back

Go forward

Copy the selection to the clipboard

Reset the size of the help page contents

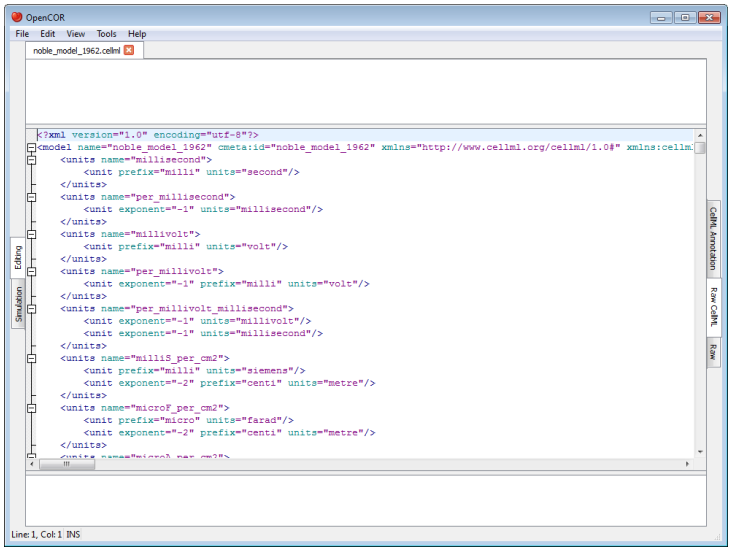
Zoom in the help page contents

Zoom out the help page contents

Print the help page contents

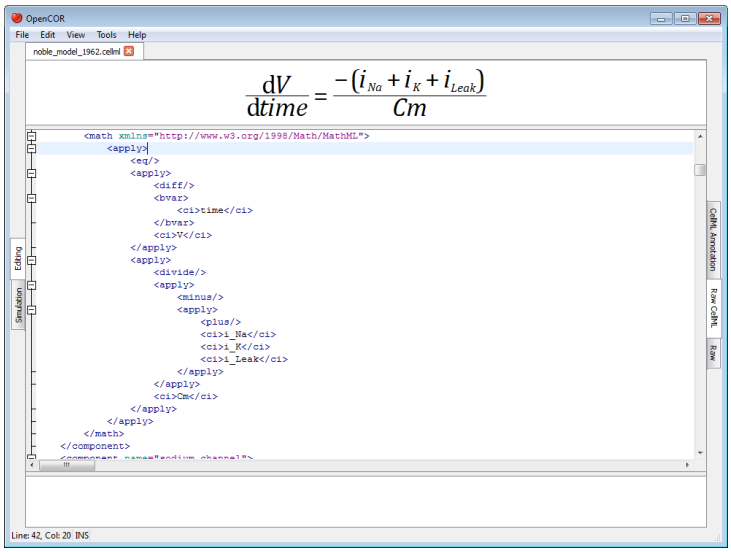
RawCellMLView Plugin

The RawCellMLView plugin can be used to edit CellML files in their raw format using a text editor. If you open a file, it will look something like:

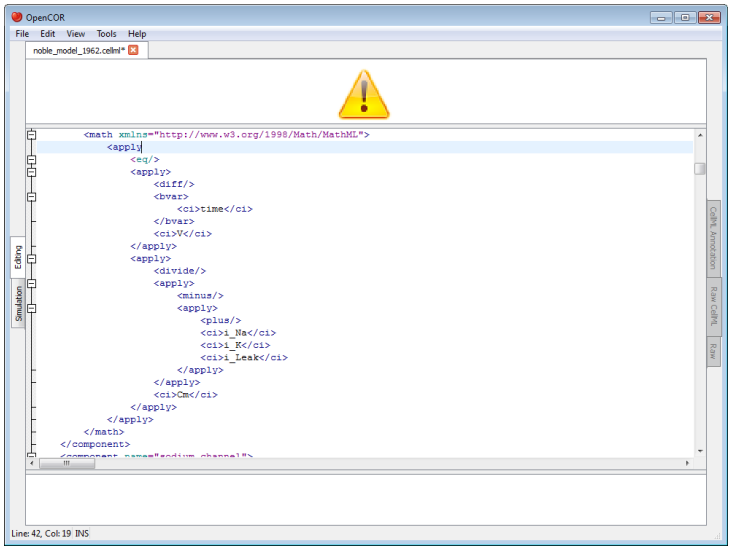


Besides using syntax highlighting, the text editor behaves in exactly the same way as the text editor in the *Raw* view.

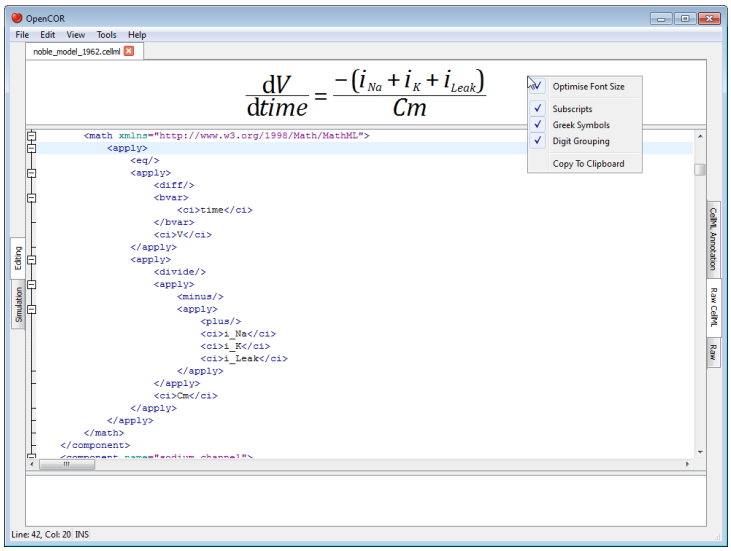
The panel above the text editor is used to visualise mathematical equations in real-time. You just need the caret to be within a valid `apply` MathML block:



If the equation is not valid, a warning sign gets displayed:

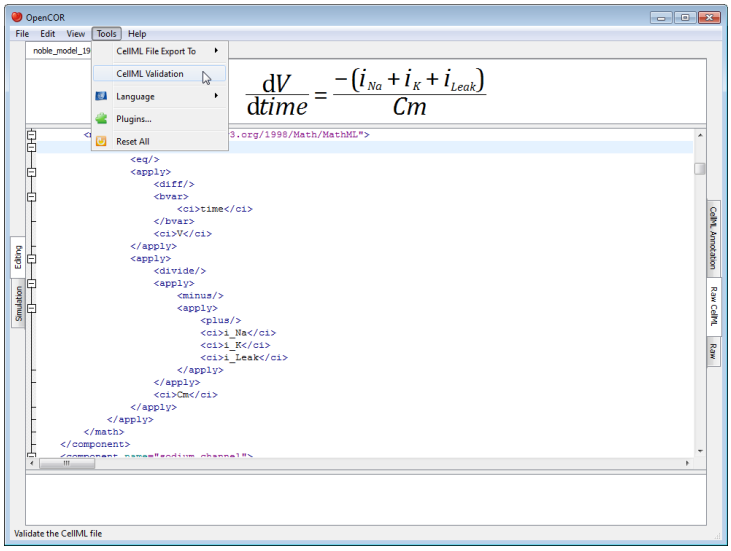


The equation viewer can be customised using its context menu:

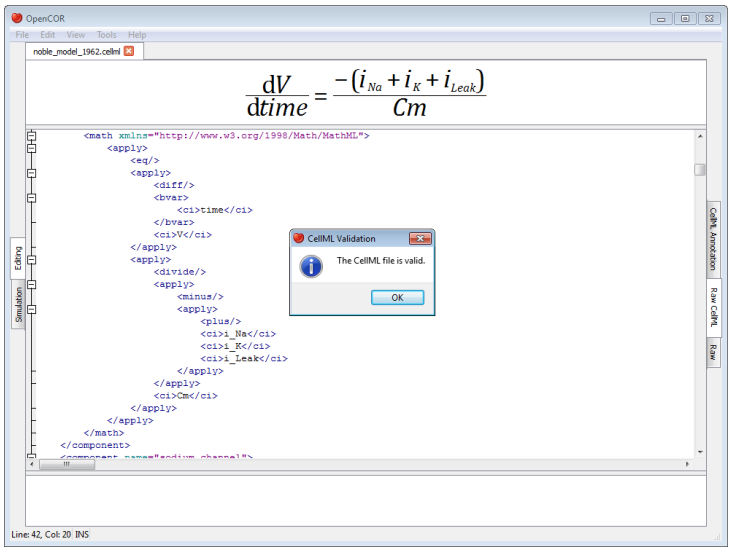


The font size can thus be optimised, so that an equation can take as much space as possible when rendered. Subscripts are also supported (e.g. `a_b` will be rendered as a_b), as are Greek symbols (i.e. `alpha`, `beta`, etc. are replaced with α, β, etc.) and digit grouping (e.g. `1000` will be rendered as 1,000). A rendered equation can also be copied to the clipboard for use in another program.

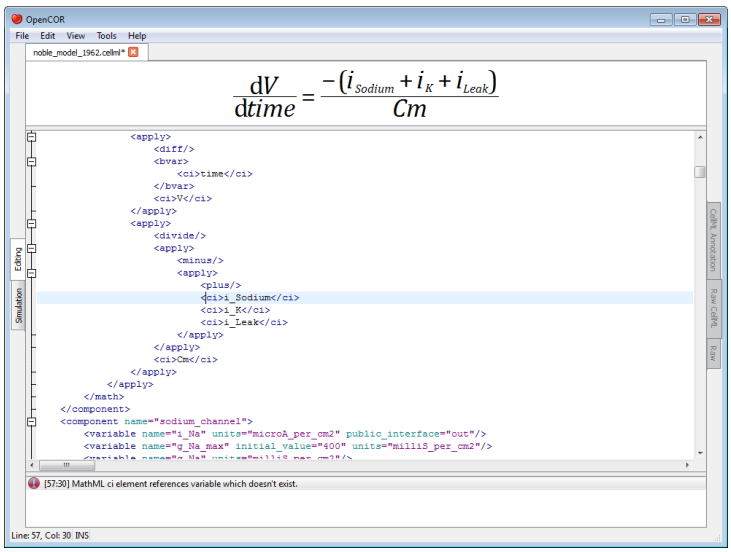
The panel below the text editor is used to list any CellML issue that results from trying to validate a CellML file:



If the CellML file is valid, then a dialog box confirming its validity is displayed:



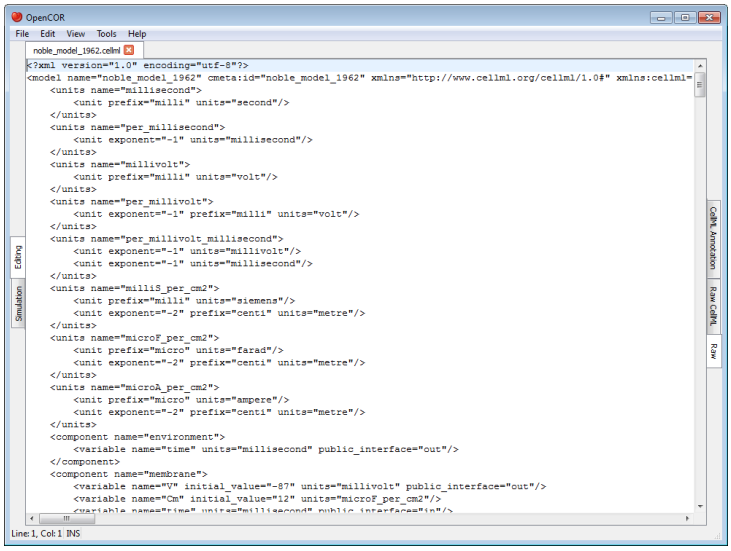
Otherwise, the bottom panel lists all the issues with the CellML file:



To double click on an issue will get the text editor to navigate to the corresponding line.

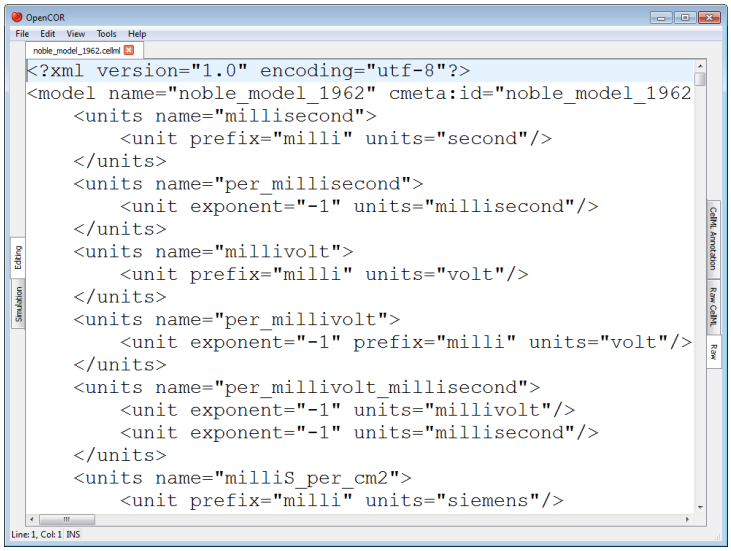
RawView Plugin

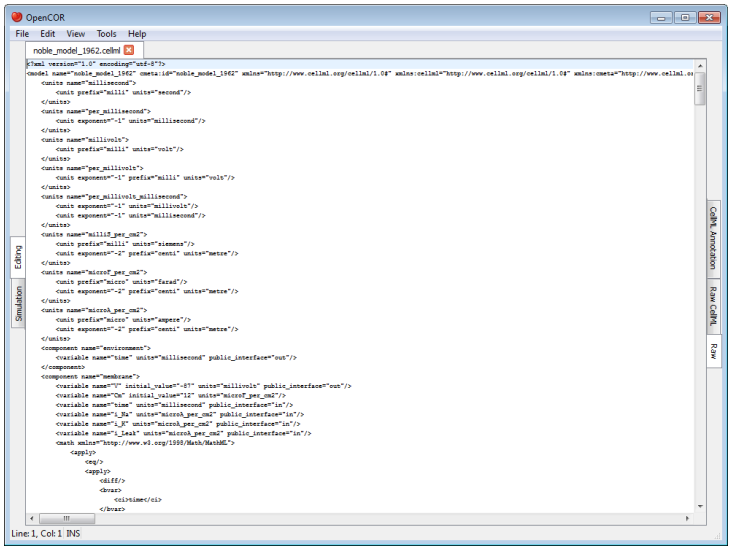
The RawView plugin can be used to edit text-based files. If you open a file, it will look something like:



The bluish line at the top is used to highlight the line that contains the caret, which line and column numbers can be found at the bottom left of the screen, together with the current editing mode (`INS`: insert, `OVR`: overwrite).

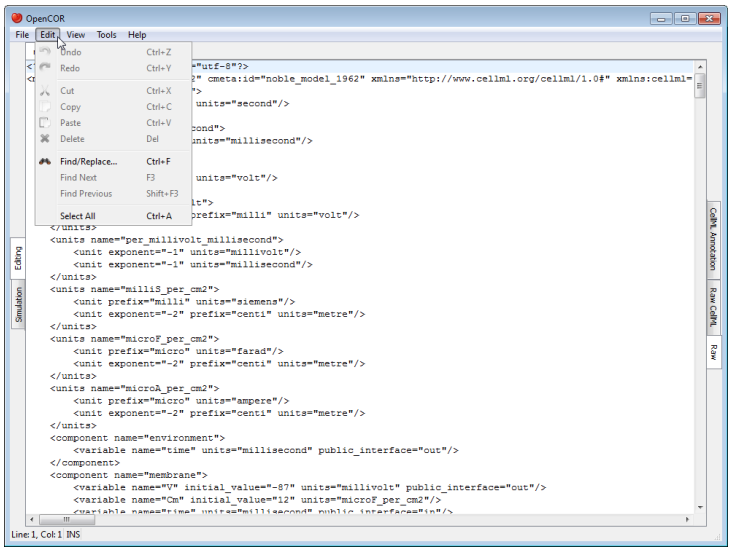
The size of the text can be increased and decreased by pressing `Control-+` (or `Control-=`) and `Control--`, respectively. You can also change the size of the text by pressing `Control` and moving the mouse wheel up or down. To reset the font size, press `Control-0`.

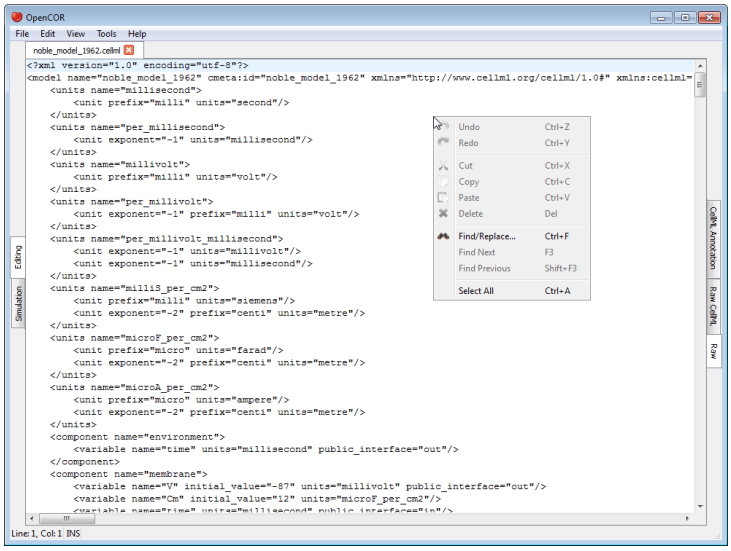




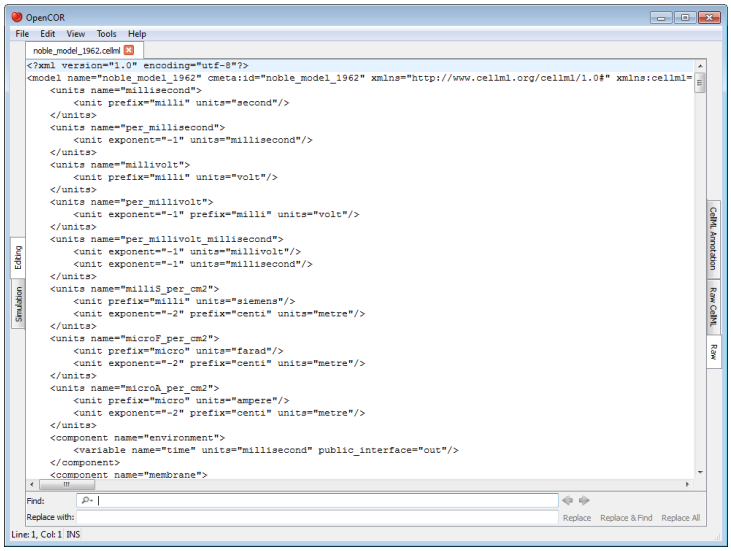
To change the size of the text will do so for all the files that use this view and will be remembered from one session to another.

Traditional editing features can be accessed through the *Edit* menu, various keyboard shortcuts and the context menu of the editor:

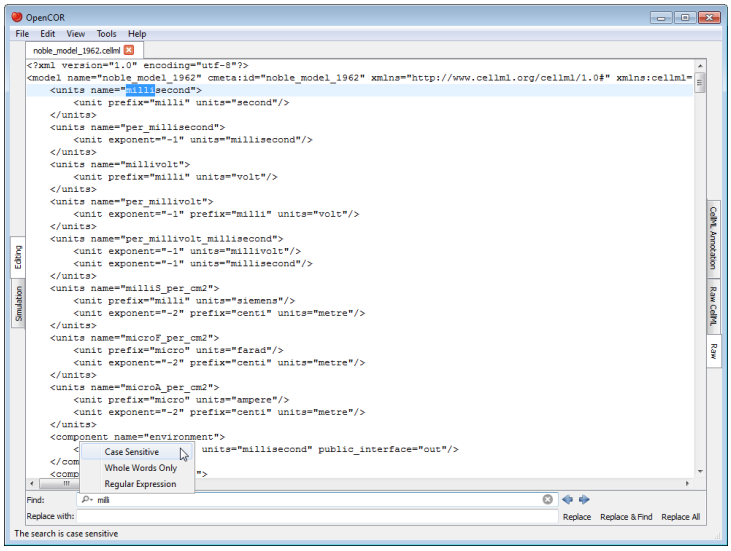




The find/replace feature can be activated by, for example, pressing `Control-F` (and hidden by pressing `ESC`), as can be seen at the bottom of the screen:



As soon as you enter some text in the *Find* field, the view will jump to the first occurrence of that text. You can then search for the next or previous occurrence of that text by pressing `F3` (or `Control-G`, depending on your operating system) and `Shift-F3` (or `Control-Shift-G`), respectively. You can make the search case sensitive, look for whole words only and/or use a regular expression by selecting the requested option(s) from the drop-down menu to the left of the *Find* field:



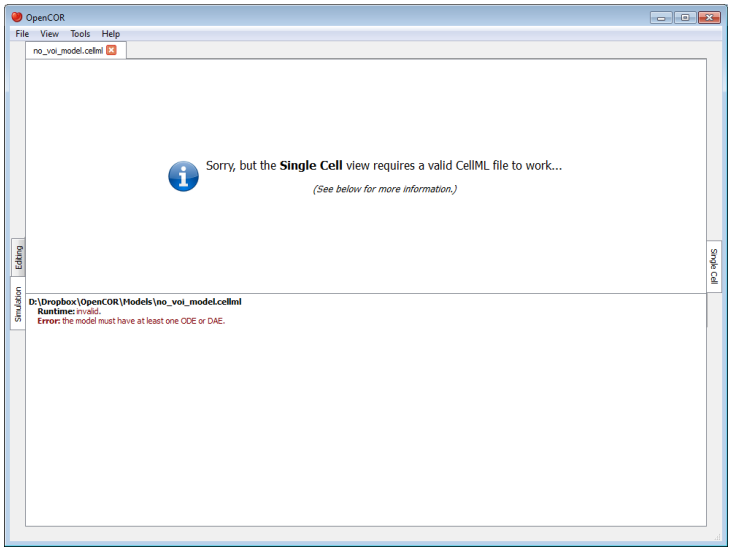
To replace some text, you can use the *Replace with* field. From there, use one of the *Replace*, *Replace & Find* and *Replace All* buttons at the bottom right of the screen.

SingleCellView plugin

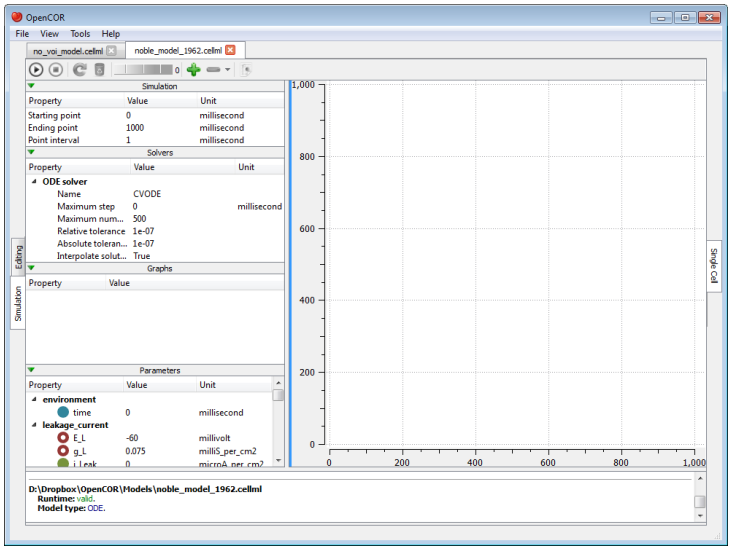
The SingleCellView plugin can be used to run CellML models which consists of either a system of ordinary differential equations (ODEs) or differential algebraic equations (DAEs). The system may be non-linear.

Open a CellML file

Upon opening a CellML file, OpenCOR will check that it can be used for simulation. If it cannot, then a message will describe the issue:



Alternatively, if the CellML file is valid, then the view will look as follows:

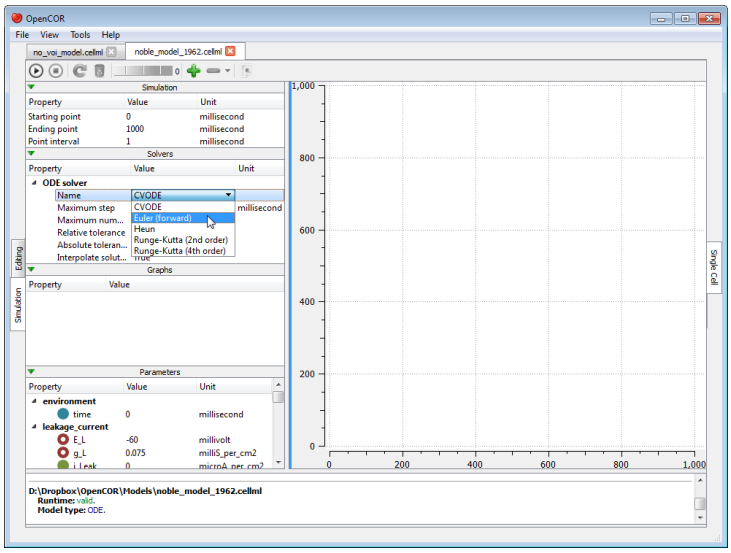


The view consists of two main parts, the first of which allows you to customise the simulation, the solver and the model parameters. The second part is used to plot simulation data. In the *Parameters* section, each model parameter has an icon associated with it to highlight its type:

	Variable of integration
	(Editable) constant
	Computed constant
	(Editable) state
	Rate
	Algebraic

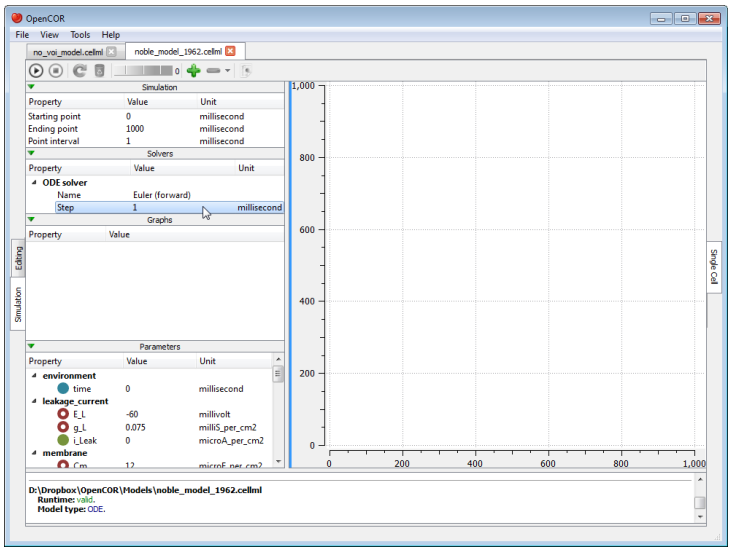
Simulate an ODE model

To simulate a model, you need to provide some information about the simulation itself, i.e. its starting point, ending point and point interval. Then, you need to specify the solver that you want to use. The solvers available to you will depend on which solver *plugins* you selected, as well as on the type of your model (i.e. ODE or DAE). In the present case, we are dealing with an ODE model and all the solver plugins are selected, so OpenCOR offers CVODE, forward Euler, Heun, Midpoint, and second- and fourth-order Runge-Kutta as possible solvers for our model.

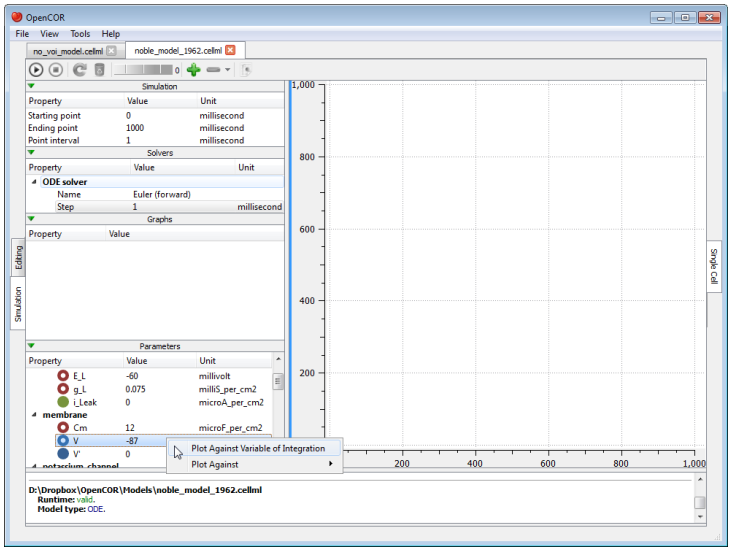


Each solver comes with its own set of properties which you can customise. For example, if we select Euler

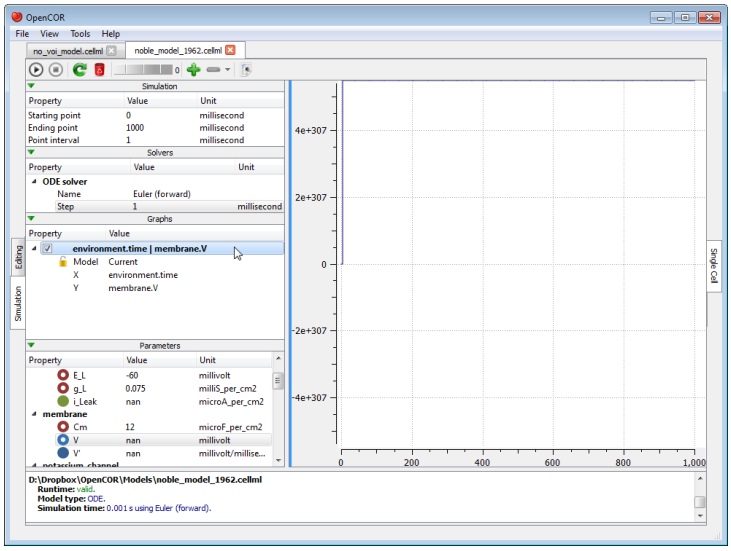
(forward) as our solver, then we can customise its `Step` property:



At this stage, we can run our model by pressing the F9 key or by clicking on the button. Then, or before, you can add a graph. All the model parameters are listed to the bottom-left of the view, grouped by components in which they were originally defined. To add a graph, right click on a model parameter and select against which other model parameter you want it to be plotted. For example, to create a graph for `V` (from the `membrane` component) against the variable of integration (i.e. time since the simulation properties are expressed in milliseconds):

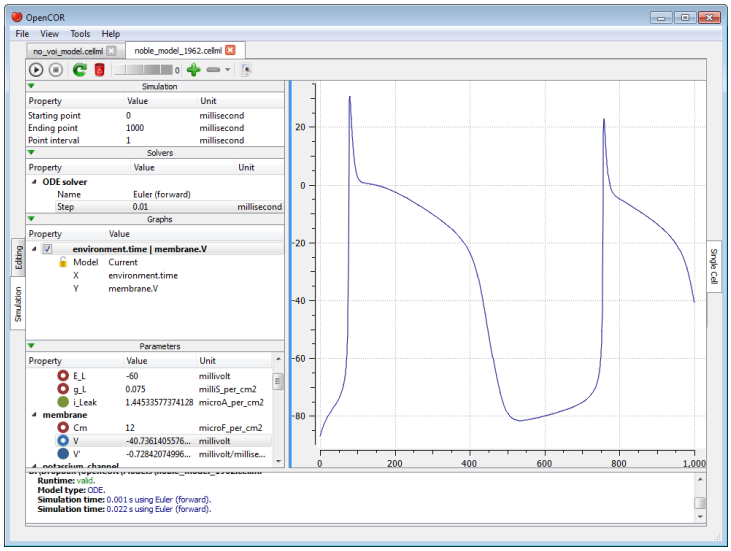


You can get the information associated with a graph by double clicking on it:

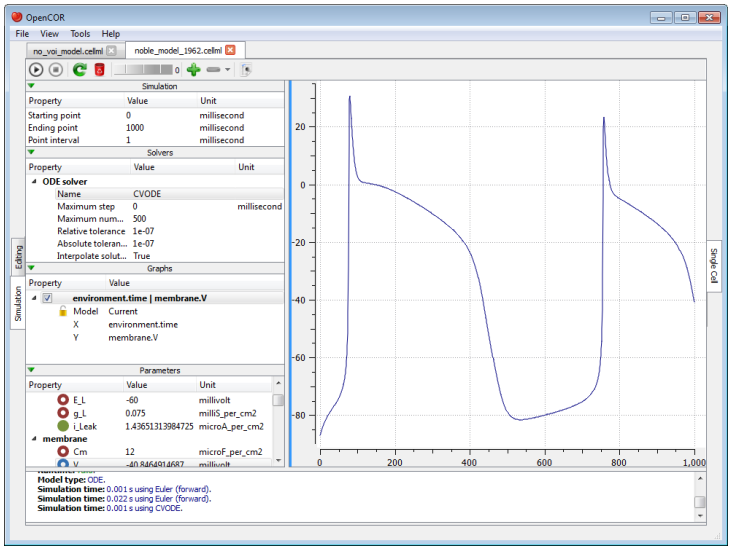


The *Model* property is used to associate the graph with a particular CellML file. By default, it has a value of *Current*, which means that if you select another CellML file, then OpenCOR will try to associate the graph with it (the icon will be shown next to the check box, if it cannot, as well as next to the X and/or Y properties to highlight which model parameter(s) could not be found in the other CellML file). The icon indicates that the graph is not locked, i.e. its Model property has a value of Current, while the icon is used when a graph is specifically associated with a CellML file (resulting in a red trace rather than a blue one). The X and Y properties can be modified either by editing their value or by right clicking on them and selecting another model parameter from the context menu, which can also be used to add or remove a graph.

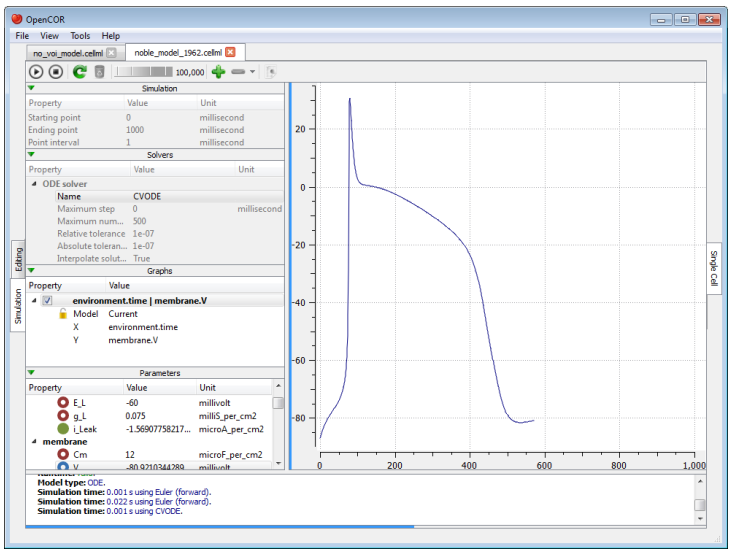
Back to the simulation, you can see that it failed with several model parameters having a value of nan (i.e. not a number). This is because the solver was not properly set up: its Step property is too big. If you set it to 0.01 milliseconds, reset all the model parameters (by clicking on the button) and clear the simulation data (by clicking on the button), and restart the simulation, then you get the following trace:



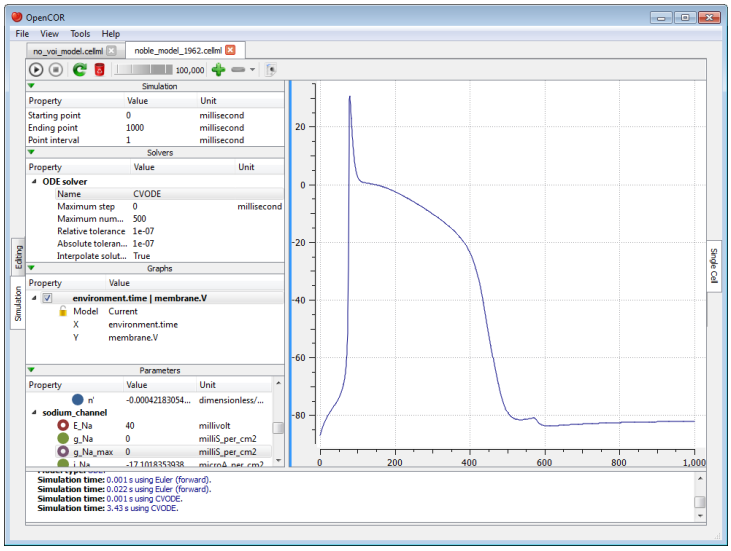
The (roughly) same trace can also be obtained using the CVODE solver:



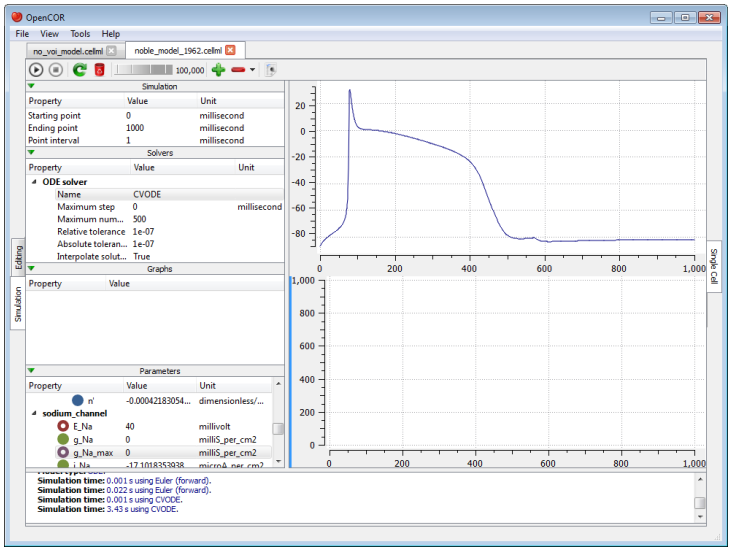
However, the simulation is so quick to run that we do not get a chance to see the progress of the simulation. Between the and buttons, there is a wheel which we can use to add a short delay between the output of two data points. Here, we set the delay to `13 ms`. This allows us to rerun the simulation, after having reset the model parameters, and pause it at a point of interest:



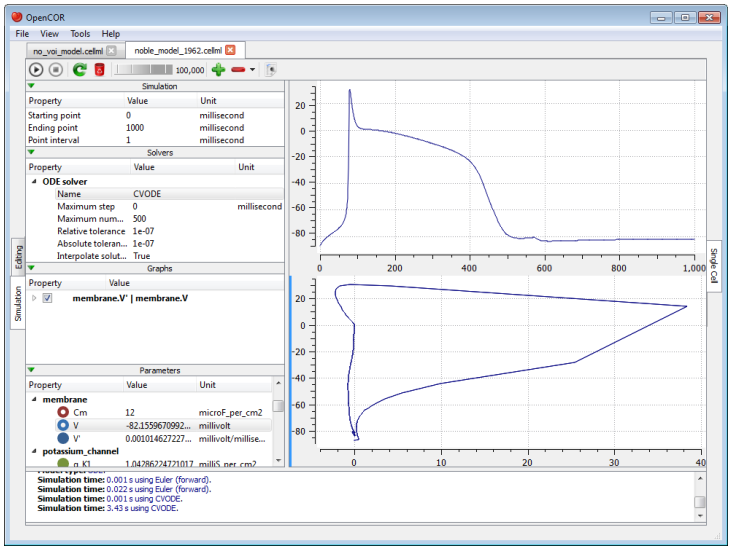
Now, we can modify any of the model parameters identified by either the or icon, but let us just modify `g_Na_max` (under the `sodium_channel` component) by setting its value to `0 milliS_per_cm2`. Then, we resume the simulation and we can see the effect on the model:



If you want, you can export all the simulation data to a comma-separated values (CSV) file. To do so, you need to click on the  button. Alternatively, if you want to create other graphs, but do not want them on the same graph panel as the existing one, you can click on the  button to create a new graph panel:



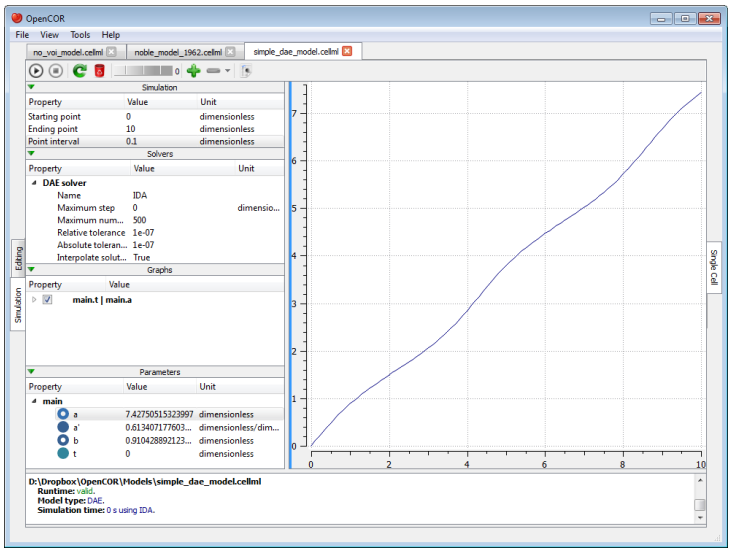
You might have noticed that the bottom graph panel has a blue vertical line to its left. This is to indicate that it is the currently selected graph panel (a graph panel can be selected by clicking on it). Something else you might have noticed is that the graphs area is now empty. This is because there are currently no graphs associated with the graph panel. Just for illustration, you can create a graph to plot V (from the membrane component) against V' (also from the membrane component):



You can create as many graph panels (and graphs) as you want. The current graph panel or all the graph panels (but the top one) can be removed by clicking on the button.

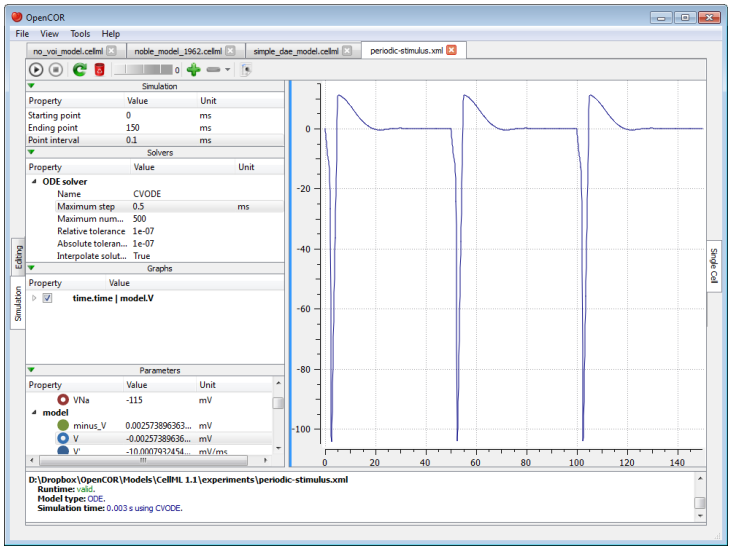
Simulate a DAE model

To simulate a DAE model is similar to simulating an ODE model, except that OpenCOR only offers one DAE solver (IDA) at this stage:



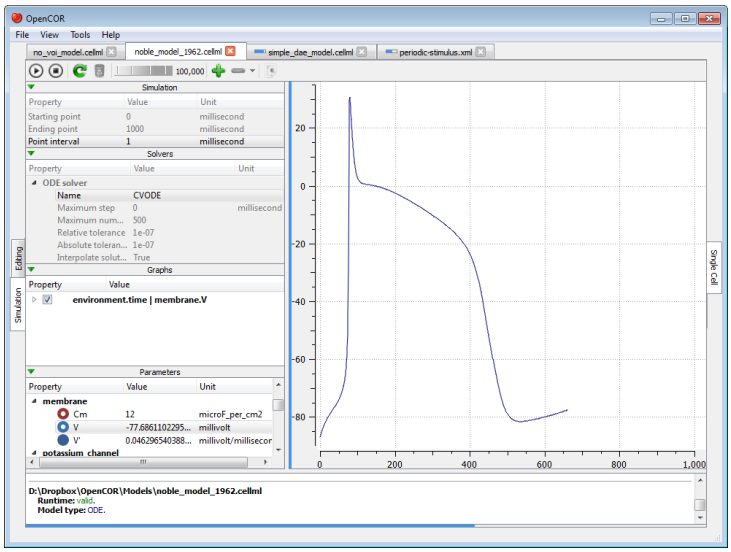
Simulate a CellML 1.1 model

So far, we have only simulated CellML 1.0 models, but we can also simulate CellML 1.1 models, i.e. models which import units and/or components from other models:



Simulate several models at the same time

Each simulation is run in its own thread which means that several simulations can be run at the same time. Simulations running in the 'background' display a small progress bar in the top tab bar while the 'foreground' simulation uses the main progress bar at the bottom of the view:



Plotting area

The plotting area offers several features that can be activated by:

- **Zooming in/out:**
  - Holding the right mouse button down, and moving the mouse to the bottom-right/top-left to zoom in/out; or
  - Moving the mouse wheel up/down; or
  - Using the context menu.
- **Resetting the zoom level:**
  - Double-clicking on the left mouse button; or
  - Using the context menu.

- **Zooming into a region of interest:**
  - Pressing Ctrl and holding the right mouse button down, and moving the mouse around.
- **Panning:**
  - Holding the left mouse button down, and moving the mouse around.
- **Showing the coordinates of any point:**
  - Pressing Shift and holding the left mouse button down, and moving the mouse around.
- **Copying the contents of the plotting area to the clipboard:**
  - Using the context menu.

Tool bar

	Run the simulation
	Pause the simulation
	Stop the simulation
	Reset all the model parameters
	Clear the simulation data
	Add a graph panel
	Remove the current graph panel or all the graph panels
	Export the simulation data to CSV

CHAPTER 4

Musculoskeletal Atlas Project (MAP) Client

The Musculoskeletal Atlas Project Client (MAP Client) is a cross-platform framework for managing workflows. A workflow consists of a number of connected workflow steps. The MAP Client framework is a plugin-based application where the plugins are workflow steps.

The MAP Client framework has a number of tools for creating, managing and sharing workflows, workflow steps and the outputs generated from the workflow steps. It is an application written in Python and based on Qt, the cross-platform application and UI framework.

One of the central ideas for the MAP Client is to allow users to easily develop and share their own plug-ins/workflow steps. The requirements for developing a workflow step have been kept as low as practicable allowing creators to concentrate on the practical implementation of the workflow step rather than concerning themselves with conforming to the plugin API. The Plugin Wizard tool greatly simplifies the first stage in creating a workflow step and generates a considerable amount of the skeleton code required.

Another of the central ideas for the MAP Client is making the output from the workflow steps available and searchable to others. To achieve this the MAP Client uses the Physiome Model Repository (PMR). PMR has been designed to provide data upload, storage and distribution capabilities, despite the name PMR is not just for models but for any data that users wish to track development changes of. The MAP Client has the PMR Tool to make use of this facility. Using the PMR Tool we can make sure the important data that a workflow produces is secure and available into the future.

A feature of having a plugin based framework is that it is possible for groups to share their workflows and workflow steps without requiring a lot of extraneous software. Also having users create and share their plugins increases the flexibility of the MAP Client and distances users from relying on an external team of developers. To further the reach of workflow steps if they are made to be as general as possible we can increase the re-usability and shareability for other users to use in their own work or alternatively extend to fit their purposes.

Further details on the MAP Client are available in the documents listed below.

MAP Client Installation and Setup Guide

This document describes how to install and setup the MAP Client software for use on your machine. The MAP Client software is a Python application that uses the PySide Qt library bindings.

The *Installation* section details getting the MAP Client and it's dependencies installed on your system. There are two main ways of getting the MAP Client installed on your operating system. This document will cover both of those methods. For users and plugin developers the suggested method is to *Install Using Pip*, for developers of the MAP Client framework the suggested method is to *Install Using Bazaar*.

The *Install Using Pip* method is covered first followed by the instructions on how to *Install Using Bazaar*. For most operating systems Python is already installed but for some, most notably Windows based operating systems, it is not. For instructions on installing Python for Windows based operating systems see the *Installing Python on Windows* section.

The *Setup* section details getting the MAP Client setup with external plugins.

Installation

Install Using Pip

Pip is a tool for installing and managing Python packages. It is particularly suited for the installation and management of source distributions of Python software, of which the MAP Client is one. The downside to using pip is that it is not great for installing binary packages, and there is one such binary package that the MAP Client requires, namely PySide. This creates something of a problem for the installation of the MAP Client. To make the installation via pip as easy as possible we must do some of the installation manually.

The manual part of the installation concerns installing PySide. For PySide we need to first install it, for Ubuntu:

```
sudo apt-get install python-pyside pyside-tools
```

For OSX download the appropriate PySide binaries from the qt-project and follow the instructions in the dmg. For Windows download the PySide installer binaries from the qt-project, make sure the binaries for PySide match the installed Python you have and follow the instructions in the installer.

Then we need to let pip know that PySide is installed, this takes the form of creating an empty file called 'PySide-X.Y.Z.egg-info' in the site-packages or dist-packages directory, depending on where your PySide libraries were installed. The X.Y.Z are given values that match the actual version of PySide you have. For example on Ubuntu I would create the file:

```
sudo touch /usr/lib/python2.7/dist-packages/PySide-1.1.2.egg-info
```

and on Windows I would create the file:

```
echo pyside > c:\Python27\Lib\site-packages\PySide-1.1.2.egg-info
```

and on Mac OSX I would create the file:

```
sudo touch /somewhere/PySide-1.1.2.egg-info
```

At this point we can hand over to pip to finish the installation for us, if you don't have pip installed then read the section on *Installing Pip*. The command for installing the MAP Client is:

```
pip install mapclient
```

The MAP Client application should now be installed on your system. It can be launched from the command line with this command:

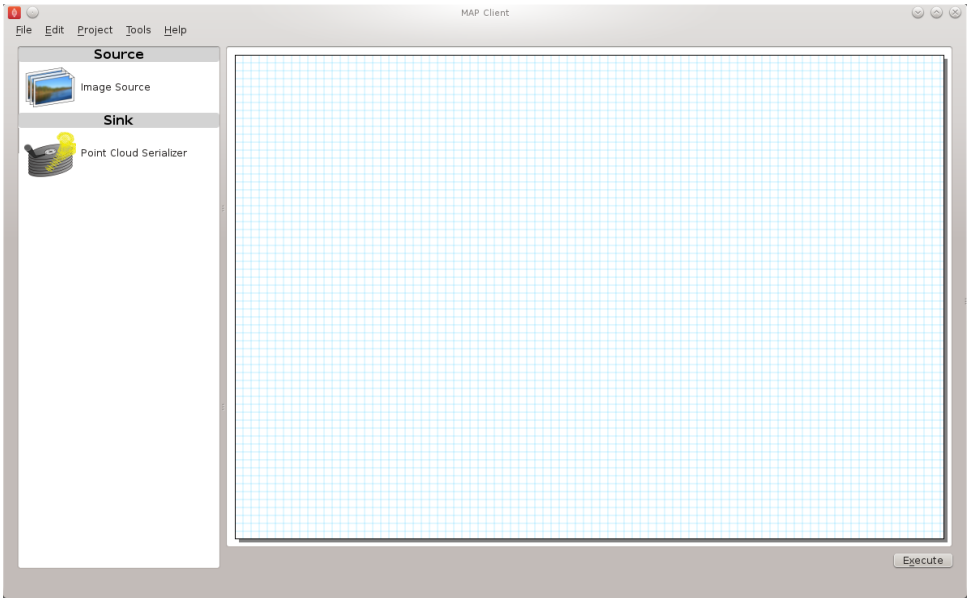
```
mapclient
```

which should result in an application window similar to that shown below.

The MAP Client relies heavily on plugins to do anything interesting, you can either create these yourself or add already available ones to your application by downloading them and using the Plugin Manager Tool in the MAP Client, read the documents *MAP Features Demonstration* and *MAP Plugin Creator Wizard* to learn more.

Install Using Bazaar

Bazaar is a distributed revision control tool. It is used by Launchpad for open source project hosting where the MAP Client source code is situated. To get bazaar use you systems package management system to install it. If you are on windows then download and install it from:



http://wiki.bazaar.canonical.com/Download

and checkout the source code and manually setup the required software

Installing Pip

Pip is a tool for installing and managing Python packages. It relies on setuptools to work, first you must install setuptools which has very good instructions available here

https://pypi.python.org/pypi/setuptools#installation-instructions

Next test to make sure that easy_install is available, open a command window and issue the command:

```
easy_install --version
```

If this command prints out the version of setuptools you have installed then you can install pip with the command:

```
easy_install pip
```

otherwise you will probably need to adjust the PATH system variable so that the easy_install application is available.

Installing Python on Windows

This section is for setting up Python on Windows as other operating systems supported by the MAP Client already have Python available. The MAP Client framework is written in *Python* and is designed to work with Python 2 and Python 3. The MAP Client framework is tested against Python 2.6, Python 2.7 and Python 3.3 and should work with any of these Python libraries.

With a Python installation for windows there are a number of choices to make:

1. Which version?
2. 32-bit or 64-bit?

The choices made here must be the same for PySide. The current recommendation is to choose the 64 bit version of the latest Python 2.7 binary release. Current versions of Python are available from:

http://www.python.org/download/

Downloaod an msi installer that matches your choices and follow the onscreen prompts. Make sure to add the `Python` and `Python\Scripts` folders to your system `PATH`.

Setup

External Plugins

The installation of external MAP Client plugins is a two step process. The first step is to download the plugins onto the local file system and the second step is to use the *MAP plugin manager* to get the MAP Client to load them.

There is a github orginisation which has a collection of MAP Client plugins. Some of the plugins here are more advanced and have a dependency on the Zinc and PyZinc libraries. To use these plugins please read the *Zinc and PyZinc* section on how to setup them up.

Zinc and PyZinc

Zinc is an advanced field manipulation and visualisation library and PyZinc provides *Python* bindings to the Zinc library. The MAP client is able to make use of Zinc for advanced visualisation and image processing steps through PyZinc. Binaries for Zinc and PyZinc are available from here and here for Linux, Windows, and OS X.

First install Zinc, for Ubuntu download the debian package and install it with the folowing command:

```
sudo dpkg -i zinc-X.Y.Z-x86_64-Ubuntu-10.04.4-LTS.deb
```

for Windows download the executable installer and follow the onscreen instructions. For Mac OSX download the dmg and follow the onscreen instructions. Archived versions exist for installing the Zinc library manually if you prefer.

To get PyZinc installed, follow these steps:

1. Download the PyZinc archive that matches the Zinc library already downloaded.
2. Extract the downloaded PyZinc archive (unzip on Windows, tar for Ubuntu and Mac OSX).
3. In a command window, make the current directory the directory where PyZinc was extracted.
4. Execute the following command: `python setup.py install`.

note:

```
The Zinc and PyZinc packages must have the same version
number.
```

MAP Features Demonstration

Section author: Hugh Sorby

Note: MAP is currently under active development, and this document will be updated to reflect any changes to the software or new features that are added. You can follow the development of MAP at the launchpad project.

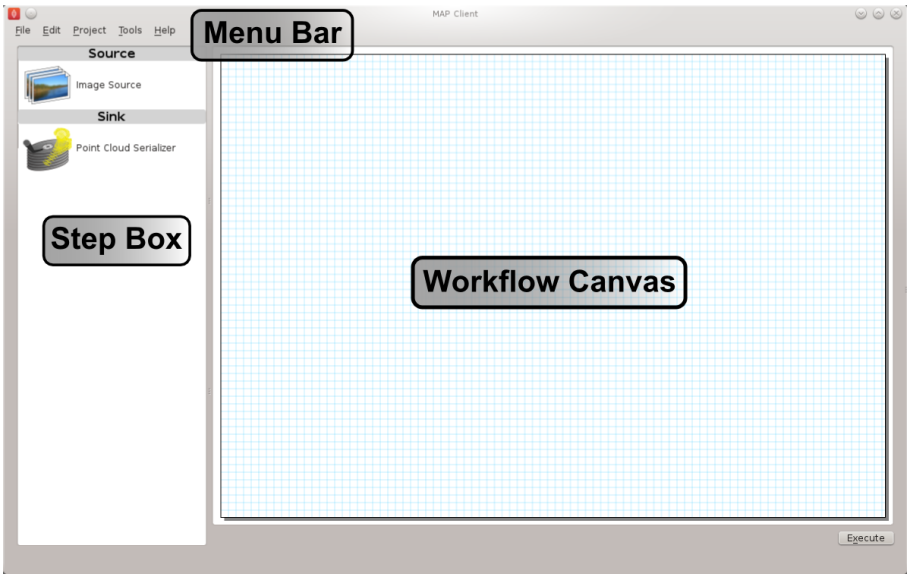
This document details the features of MAP, a cross-platform framework for managing workflows. MAP is a plugin-based application that can be used to create workflows from a collection of workflow steps.

In this demonstration is based on version 0.9.0 of MAP, available from the project downloads. Directions for installing MAP and getting the MAP plugins are available in the *MAP Client Installation and Setup Guide*.

In this demonstration we will cover the features of MAP. We will start with a quick tour and then create a new workflow that will help us segment a region of interest from a stack of images.

Quick Tour

When you first load MAP, it will look something like this:



In the main window we can see three distinct areas that make up the workflow management side of the software. These three areas are the menu bar (at the top), the step box (on the left) that contains the steps that you can use to create your workflow and the workflow canvas (on the right) an area for constructing a workflow.

In the Step box we will only see two steps, this is because we have only loaded the default Steps and not loaded any of the external plugins that MAP can use.

Menu Bar

The Menu bar provides a selection of drop down menus for accessing the applications functions. The File menu provides access to opening, importing, closing workspaces as well as quitting the application. The Edit menu provides access to the undo/redo functionality. The Tools menu provides access to the Plugin Manager tool, Physiome Model Repository (PMR) tool and the Annotation tool. The Help menu provides access to the about box which contains information on contributors and the license that the MAP application is released under.

The 'New' menu has two sub-menus: 'New/PMR Workflow' and 'New/Workflow'. The PMR Workflow menu command will create a new workflow in the chosen directory and use Mercurial to track changes to your project. When saving the workflow the contents of the project will be transferred to PMR via Mercurial, this transfer is managed by the application. For more information on the benefits and use of PMR please read the documentation available at read-the-docs .

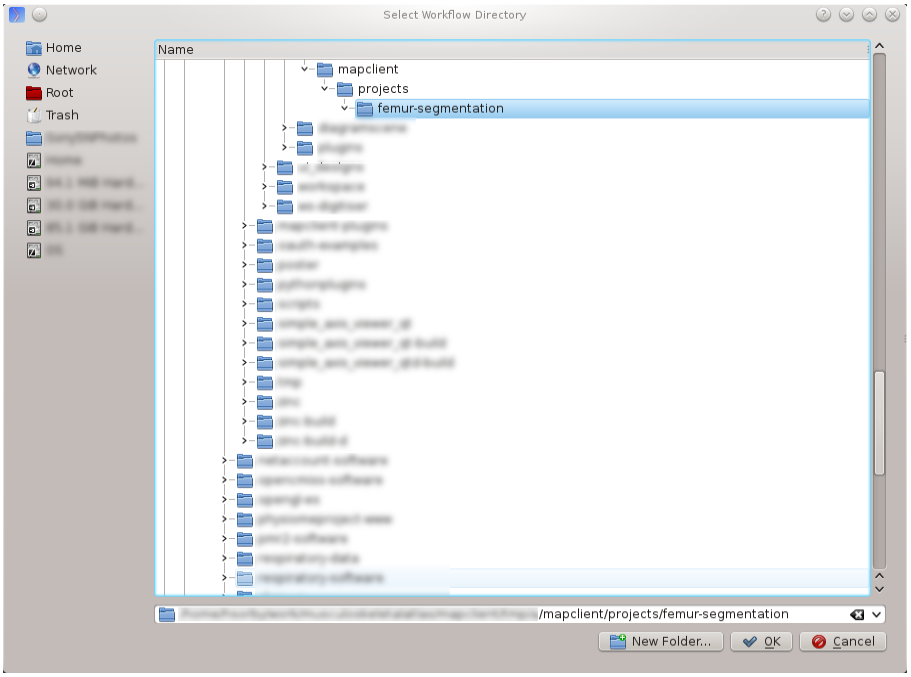
The Workflow menu command will create a new workflow on your local disk in the selected directory.

Step Box

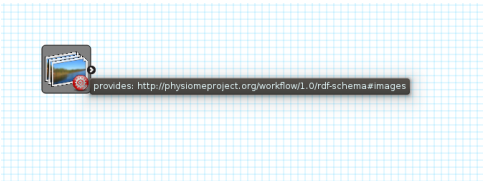
The Step box provides a selection of steps that are available to construct a workflow from. The first time we start the program only the default plugins are available. To add more steps we can use the Plugin Manager tool. To use a step in our workflow we drag the desired step from the step box onto the workflow canvas.

Workflow Canvas

The workflow canvas is where we construct and edit our workflow. We do this by adding the steps to the workflow canvas from the step box that make up our workflow. We then make connections between the workflow steps to define the complete workflow.



When a step is added to the workflow the icon which is visible in the Step box is augmented with visualisations of the Steps ports and the steps configured status. The annotation of the steps ports will show when the mouse is hovered over a port. The image below shows the Image Source step with the annotation for the port displayed.



Tools

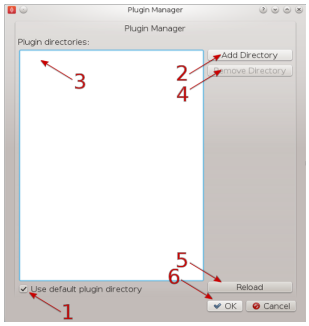
MAP currently has three tools that may be used to aide the management of the workflow. They are the Plugin Manager tool, the Physiome Model Repository (PMR) tool and the Annotation tool. For a description of each tool see the relevant sections.

Plugin Manager Tool

The plugin tool is a simple tool that enables the user to add or remove additional plugin directories. MAP comes with some default plugins which the user can decide to load or not by checking or unchecking the check box (1) at the bottom of the dialog. External directories are added with the add directory button (2). Directories are removed by selecting the required directory in the Plugin directories list (3) and clicking the remove directory button (4). To reload plugins from the current plugin directories use the reload button (5).

Note: The reload will only reload the plugins from the current plugin directories, this will not include any changes to the directories in the current dialog. To confirm changes and load plugins from the plugin directories listed in the plugin manager click the OK button (6).

Whilst additions to the plugin path will be visible immediately in the Step box deletions will not be apparent until the next time the MAP Client is started. This behaviour is a side-effect of the Python programming language.



Physiome Model Repository (PMR) Tool

The PMR tool uses webservices and OAuth to communicate between itself (the consumer) and the PMR website (the server). Using this tool we can search for and find suitable resources on PMR.

The PMR website uses OAuth to authenticate a consumer and determine consumer access privileges. Here we will discuss the parts of OAuth that are relevant to getting you (the user) able to access resources on PMR. Please read the section *Simplified OAuth Primer* for a quick overview of OAuth authentication.

If you want the PMR tool to have access to privileged information (your non-public workspaces stored on PMR) you will need to register the PMR tool with the PMR website. We do this by clicking on the *register* link as shown in the figure below. This does two things: it shows the Application Authorisation dialog; opens a webbrowser at the PMR website. [If you are not logged on at the PMR website you will need to do so now to continue, instructions on obtaining a PMR account are available here XXXXX]. On the PMR website you are asked to either accept or deny access to the PMR tool. If you allow access then the website will display a temporary access token that you will need to copy and paste into the Application Authorisation dialog so that the PMR tool can get the permanent access token.

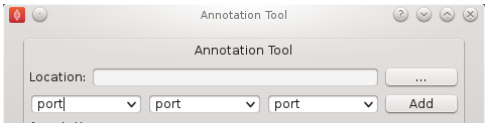
Simplified OAuth Primer

In OAuth we have three players the server, the consumer and the user. The server is providing a service that the consumer wishes to use. It is up to the user to allow the consumer access to the servers resources and set the level of access to the resource. For the the consumer to access privileged information of the user stored on the server the user must register the consumer with the server, this is done by the user giving the consumer a temporary access token. This temporary access token is then used by the consumer to finalise the transaction and acquire a permanent access token. The user can deny the consumer access at anytime by logging into the server and revoking the permanent access token.

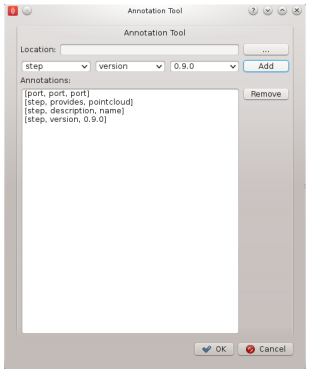
Annotation Tool

Note: Please note that the annotation tool is currently under development, this tool in it's current form as documented here does not integrate well with the latest version of PMR. This version of the annotation tool has been marked as deprecated

The Annotation tool is a very simple tool to help a user annotate the Workflow itself and the Step data directories that are linked to PMR. At this stage there is a limited vocabulary that the Annotation tool knows about, but this is intended to be extended in coming releases. The vocabulary that the annotation is aware of is available in the three combo-boxes near the top of the dialog.



The main part of the Annotation tool shows the current annotation from the current target.



In the above image we can see the list of annotations that have been added to the current target. This is a simplified view of the annotation with the prefix of the terms removed for clarity.

MAP Plugins

Section author: Hugh Sorby

The Plugin lies at the heart of the MAP framework. The key idea behind the plugins is to make them as simple as possible to implement. The interface is defined in documentation and the plugin developer is expected to adhere to it. The framework leaves the responsibility of conforming to the plugin interface up to the plugin developer. The plugin framework is based on Marty Alchin's [1] article on a plugin framework for Django. The plugin framework is very lightweight and requires no external libraries and can be made to work with Python 2 and Python 3 simultaneously.

Workflow Step

The Workflow Step is the basic item that a plugin developers need to place their software within. A workflow step can be of any size and complexity. Although it must adhere to the plugin design to work properly with the application. Every step that wishes to act like a Workflow Step must derive itself from the Workflow step mountpoint. The Workflow step mountpoint is the interface between the application and the plugin. The Workflow step mountpoint can be imported like so:

```
from mapclient.mountpoints.workflowstep import WorkflowStepMountPoint
```

A skeleton step is provided as a starting point for the developer to create their own workflow steps. The skeleton step is actually a valid step in its own right and it will show up in the Step box if enabled. However the skeleton

step has no use other than as an item to drag around on the workflow area. The skeleton step is discussed below, before that the plugin interface itself is discussed.

Plugin Interface

The plugin interface is the layer between the application and the developers plugin. The plugin interface is not defined by contract as we so often see in Java. But rather the plugin interface is defined by documentation. This puts the burden of the specification on the documentation and the conformity of the specification on the developer. The underlying theory is that the developer is able to follow the specification without the application having to do rigourous checks to make sure this is the case. The phrase 'If it walks like a duck' is often used.

In this section the specification of the Workflow step plugin interface is given. It is then upto the developer to make sure their plugin behaves like one.

The details of the plugin interface are provided in the documentation of the source code in the relevant source file and additionally here for easy reference. The documentation provided with the source code is very direct with little explanation the following documentation provides a bit more explanation and discussion on the various aspects of the plugin interface. The documentation provided here should be considered the slave documentation and the documentation provided with the source code as the master documentation.

There are essentially, what may be considered, three different levels of the plugin design.

1. The Musts
2. The Shoulds
3. The Coulds

Creating a workflow step that satisifies the musts will create an actual workflow step that can be added to the workflow area and interacted with. But it won't be very useful. Satisfying the shoulds will usually be sufficient for the very simplest of steps. Simple steps are for instance ones that provide images, or location information for data. Doing some of the coulds will create a much more interesting step.

The requirements for creating a step have been kept as simple as possible, this is to allow the developer a quick route into the development of the step content.

The following three sections discuss these three levels in more detail.

A Step Must

- The plugin must be derived from the WorkflowStepMountPoint class defined in the package map-client.mountpoints.workflowstep
- Accept a single parameter in it's __init__ method.
- Define a name for itself, this must be passed into the initialisation of the base class.
- Define the methods

```
def configure(self):
    pass

def getIdentifier(self):
    pass

def setIdentifier(self, identifier):
    pass

def serialize(self, location):
    pass

def deserialize(self, location):
    pass
```

A Step Should

- Implement the configure method to configure the step. This is typically in the form of a dialog. When implementing this function the class method self._configuredObserver() should be called to inform the application that the step configuration has finished.
- Implement the getIdentifier/setIdentifier methods to return the identifier of the step.
- Implement the serialize/deserialize methods. The steps should serialize and deserialize from a file on disk located at the given location.
- Define a class attribute _icon. That is of the type QtGui.QImage.
- Information about what the step uses and/or what it provides. This is achieved through defining ports on the step.

A Step Could

- Implement the method 'setPortData(self, index, dataIn)' if it uses some information from another step.
- Implement the method 'getPortData(self, index)' if it was providing some information to another step.
- Implement the method 'execute(self)' If a step implements the 'execute(self)' method then it must call '_doneExecution()' when the step is finished.
- Define a category using the '_category' attribute. This attribute will add the step to the named category in the step box, or it will create the named category if it is not present.
- Set a widget as the main widget for the MAP Client application. Calling '_setCurrentWidget(step_widget)' with a widget passed as a parameter will set that widget to the main widget for the MAP Client application. The widget will be removed when '_doneExecution()' is called.

Pre-defined Step Attributes

A step has a number of pre-defined attributes with default values, they are:

- self._name = name
- self._location = location
- self._category = 'General'
- self._ports = []
- self._icon = None
- self._configured = False

The '_name' and '_location' attributes are passed in to the '__init__' method of the mount point. The '_category' attribute can be used to group steps in the step box. By default a step has no ports and at least one port must be defined before it can be used in a workflow. If the '_icon' attribute is not defined then a default icon is supplied. The '_configured' property is set to False initially as most steps will not be configured in their initial state.

Pre-defined Step Methods

A step has a number of pre-defined methods, they are:

- **execute(self)** A method that gets called when execution passes to this step.
- **getPortData(self, index)** A method that returns the object that is defined by the port for the given index of the step
- **setPortData(self, index, dataIn)** A method that sets the ports data for the given index.

- **configure(self)** A method called by the framework to inform the step that it needs to follow it's configuration procedure.
- **isConfigured(self)** A method to return the value of '_configued'. In most cases this method will not need to be overridden.
- **_configuredObserver** A method to call to let the framework know that the step configuration has finished.
- **_identifierOccursCount** A method to call to determine the number of identifiers with the given value. This method can be used to decide whether the current identifier is unique across the workflow.
- **addPort** Adds a port to the step, the port is defined using an RDF triple. See the Ports section for more information.
- **getName(self)** Returns the '_name' attribute if it is set otherwise returns the class name. In most cases this method will not need to be overridden.
- **deserialize(self, location)** Must be implemented in the plugin otherwise an exception is raised.
- **serialize(self, location)** Must be implemented in the plugin otherwise an exception is raised.
- **_setCurrentWidget(step_widget)** Set widget 'step_widget' to the main widget for the framework.
- **_doneExecution()** Inform the framework that the step has finished it's task.
- **registerDoneExecution(self, observer)** A method used by the framework to set the callable when execution is done. This method should not be overwritten.
- **registerOnExecuteEntry(self, observer, undoRedoObserver)** A method used by the framework to set a callable to set up the step for execution. This method should not be overwritten.
- **registerConfiguredObserver(self, observer)** A method used by the framework to set a callable for notifying when the step has been configured. This method should not be overwritten.
- **registerIdentifierOccursCount** A method used by the framework to set a callable for determining the number of times the given identifier occurs in the current workflow. This method should not be overwritten.

Ports

A port is a device to specify what a workflow step provides or uses. A port is described using Resource Description Framework (RDF) triples. The port description is used to determine whether or not two ports may be connected together. One port can either use or provide one thing. A single port must not both provide a thing and use a thing. Ports are ordered by entry position.

A port is defined with the subject of *http://physiomeproject.org/workflow/1.0/rdf-schema#port* and it can be defined with a property or characteristic as either providing (*http://physiomeproject.org/workflow/1.0/rdf-schema#provides*) or using (*http://physiomeproject.org/workflow/1.0/rdf-schema#uses*) an object. What that object is is defined by the step, for example the image source step defines the following port:

> (http://physiomeproject.org/workflow/1.0/rdf-schema#port, http://physiomeproject.org/workflow/1.0/rdf-schema#provides, http://physiomeproject.org/workflow/1.0/rdf-schema#images)

Any step that understands the *http://physiomeproject.org/workflow/1.0/rdf-schema#images* object can define it's own port that uses this object. Ports are added to a step by using the 'addPort(self, triple)' method from the base class.

Skeleton Step

The skeleton step satisfies the musts of the plugin interface. It is a minimal step and it is set out as follows.

A Python package with the step name is created, in this case 'skeletonstep', in the module file we add the code that needs to be read when the plugins are loaded.

The module file performs four functions. It contains the version information and the authors name of the module. For instance the skeleton step has a version of '0.1.0' and authors name of 'Xxxx Yyyyy'. It adds the current directory into the Python path, this is done so that the steps python files know where they are in relation to the python path. It also (optionally) prints out a message showing that the plugin has been loaded successfully. But the most important function it performs is to call the python file that contains the class that derives from the workflow step mountpoint.

The 'SkeletonStep' class in the skeletonstep.step package is a very simple class. It derives from the 'WorkflowStepMountPoint', calls the base class with the name of the step, accepts a single parameter in it's init method and defines the five required functions to satisfy the plugin interface.

When enabled the skeleton step will be a fully functioning step in the MAP Client.

References

[1] http://martyalchin.com/2008/jan/10/simple-plugin-framework/ Marty Alchin on January 10, 2008

MAP Plugin Creator Wizard

Section author: Hugh Sorby

Note: MAP is currently under active development, and this document will be updated to reflect any changes to the software or new features that are added. You can follow the development of MAP at the launchpad project.

The plugin lies at the heart of the MAP framework and the Plugin Creator Wizard creates skeleton plugins conforming to the MAP framework plugin interface. The Plugin Creator Wizard assists with the initial plugin creation allowing the plugin developer to concentrate on implementing the plugins task. For basic familiararity with the MAP Client please read the feature demonstration document *MAP Features Demonstration*.

For more detailed information on the plugin interface read the *MAP Plugins* document, this document defines the plugin interface that the new plugin must adhere to.

The Plugin Creator Wizard takes the user through a series of pages/dialogs that user fills out as suits their needs. The pages and a description about the elements in each page is given below. To move from one page to the next use the 'next' button at the bottom of the page, for some pages the 'next' button is only available once the page is complete. If the 'next' button is not available for a page it will be because at least one of the pages required fields is incomplete. Required fields that are incomplete will be marked with a small cross icon (✖). Once all the required fields are complete the 'next' button will become available, or the 'finish' button in the case of the last page/dialog.

Introduction Page

The introduction page contains a short welcome message and a paragraph on the Plugin Creator Wizards purpose.

Identification Page

The identification page sets the name for the Workflow step, the Python package name and optionally the step icon. The Workflow step name can be set in the text box (1). As a recommendation Workflow step names should be defined in camel case as this name will be given to a class, spaces between words are acceptable however. The Workflow step name is visible in the Step box when active in the application so a descriptive name will aide users. The 'cross' icon (6) indicates that the entry for the step name is not valid. When a valid step name has been entered in the text box the 'cross' icon will be removed. Examples of valid step names are: 'Image Source', 'Point Cloud Serializer' and 'Segmentation'.

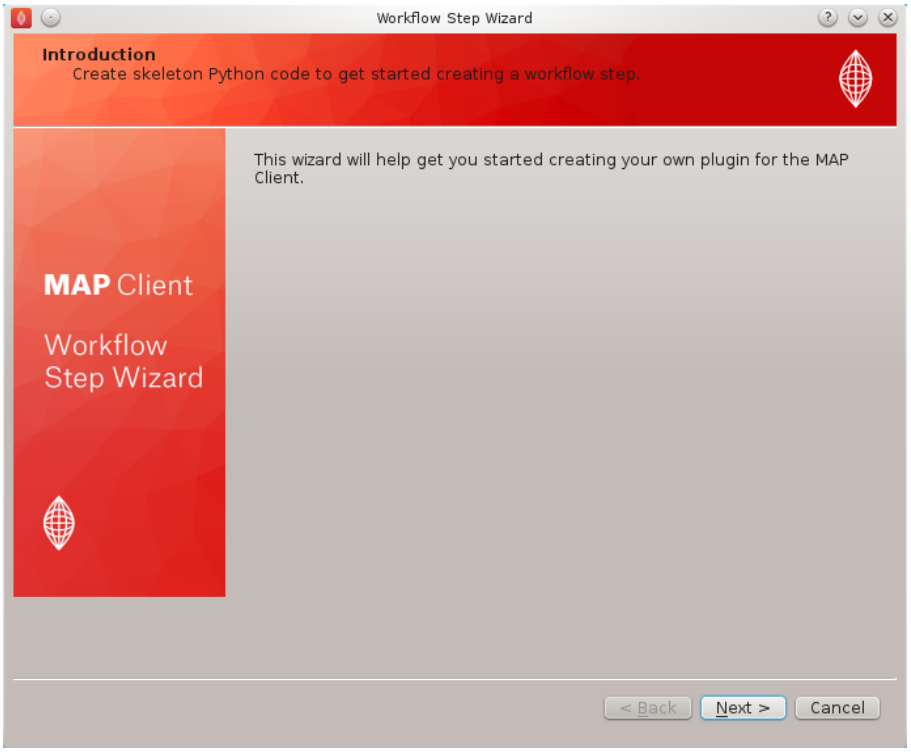


Fig. 4.1: **Figure:** The introduction page.

The package name for the step will be automatically derived from the step name and set into the package name text box (2). The wizard will make changes so that the package name conforms to the PEP8 guidelines for Python. The wizard will also append the text 'step' to the package name. However if the default name is unsatisfactory the package name can be edited directly and given an alternative name. The matching package names for the examples given above would be: 'imagesourcestep', 'pointcloudserializerstep' and 'segementationstep'.

An icon may be specified using the icon text box (3), the icon file may be chosen from the file system using the file chooser button (4). When an icon is specified it will be copied into the created skeleton step and be made available as a Qt resource. The suggested size of the icon is that it should be around 128px by 128px.

The step icon is an important part of the Workflow step as it is used to identify it graphically on the Workflow canvas. The default icon displays the step name across the icon to help differentiate it from other steps with no icon specified. A preview of the step icon (5) is shown so that you can see how it will look in the application.

Note: The PySide resource compiler application 'pyside-rcc' is required when choosing an icon image from the file system

Note: When a 'cross' icon appears on any page of the wizard it is used to indicate that the current field is not valid. When a field on a page is not valid the wizard cannot be progressed or finished. Therefore the 'cross' icon also indicates which fields require modification before the wizard can be continued.

Ports Page

The ports page sets up the ports for the step. To add a port use the 'Add' button (1). This will create an entry in the port list (2) with a default type of 'provides' and an empty object. A port can either provide or use a given object. The object should be uniquely identified using a namespace prefix, for example 'http://my.example.org/1.0/workflowstep'.

To remove a port, select an entry in the port list (2) and click the 'Remove' button (3).

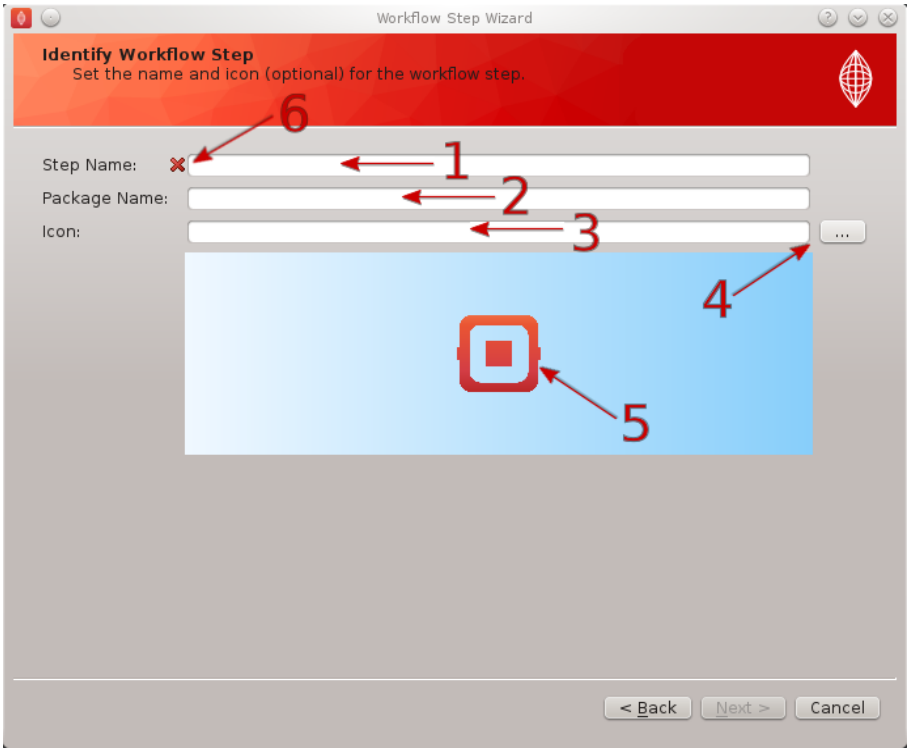


Fig. 4.2: **Figure:** The identification page.

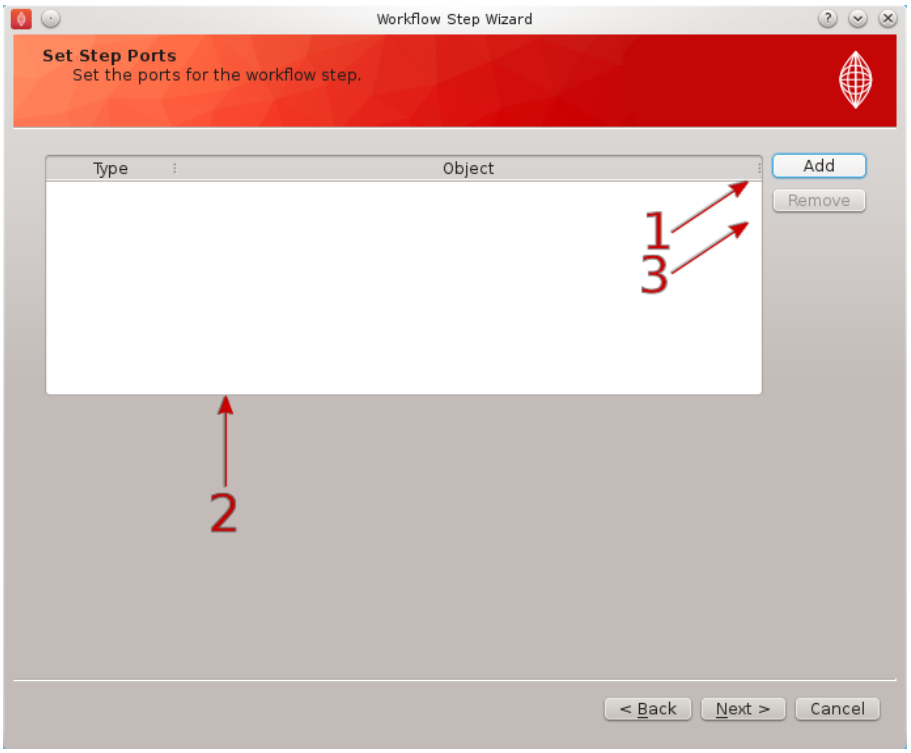


Fig. 4.3: **Figure:** The ports page.

For one port to be connected to any other the objects of both ports must match. The match is a determination of object compatibility (currently this is just a simple string matching test). Additionally to this one port must be the provider and the other the user (the order that the connection is made in when using the MAP Client is important). In summary the second port must use the object that the first port provides.

Example

As an example imagine that I wish to define a port that uses images. The images object that my step uses is particular class that I have defined. To create my port I would add a port using the 'Add' button [*plugin wizard ports* (1)]. Then select the 'uses' type from the drop down combo box in the *type column*.

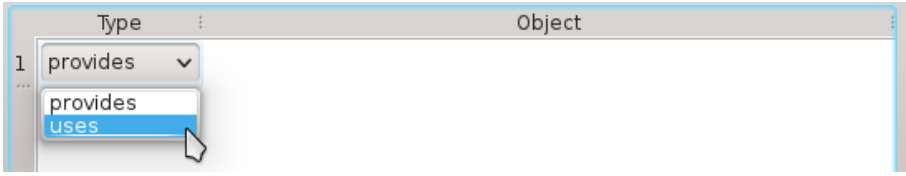


Fig. 4.4: **Figure:** Select the type of port using the drop down combo box.

Because my images class is of my own design I give it a unique name by prefixing it with a namespace. The namespace I use is 'http://my.example.org/1.0/workflowstep'. So to finish defining my port for using images, in the object column I enter the following text 'http://my.example.org/1.0/workflowstep#images'. The finished port definition should look *like this*.

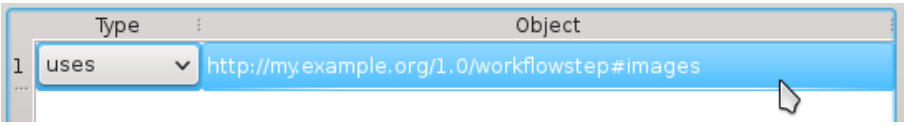


Fig. 4.5: **Figure:** An example port definition for using a users proprietary images object.

Configuration Page

The configuration page can help setup the configuration dialog for the step. The 'Identifier' check box (1) will add standard code to the step to set up the getIdentifier/setIdentifier methods in the step, it will also add an entry to the 'ConfigurationDialog' and validate the identifier. It is highly recommended that the 'Identifier' check box is checked. Use the 'Add' button (2) to add a configuration parameter to the configuration list (3). The configuration list has a 'Label' column (4), the value entered here will become a label on the configuration dialog. The 'Default Value' column (5) will be used to set the default value for the corresponding label. Edit the values in this list as appropriate. The 'Remove' button (6) can be used to delete the selected rows. The configuration parameters entered will be used in generating a configuration dialog.

Note: The PySide ui compiler application 'pyside-uic' is required when using the wizard to generate a step which has at least one configuration parameter.

Miscellaneous Page

The miscellaneous page sets a number of properties that are not important to the function of the step. The author name(s) for the step can be set in the text box (1). The author's name appears when the step plugin is loaded and is not seen or used anywhere else. The category for the step can be set in the text box (2). The category determines the group that the step appears in in the Step Box of the application.

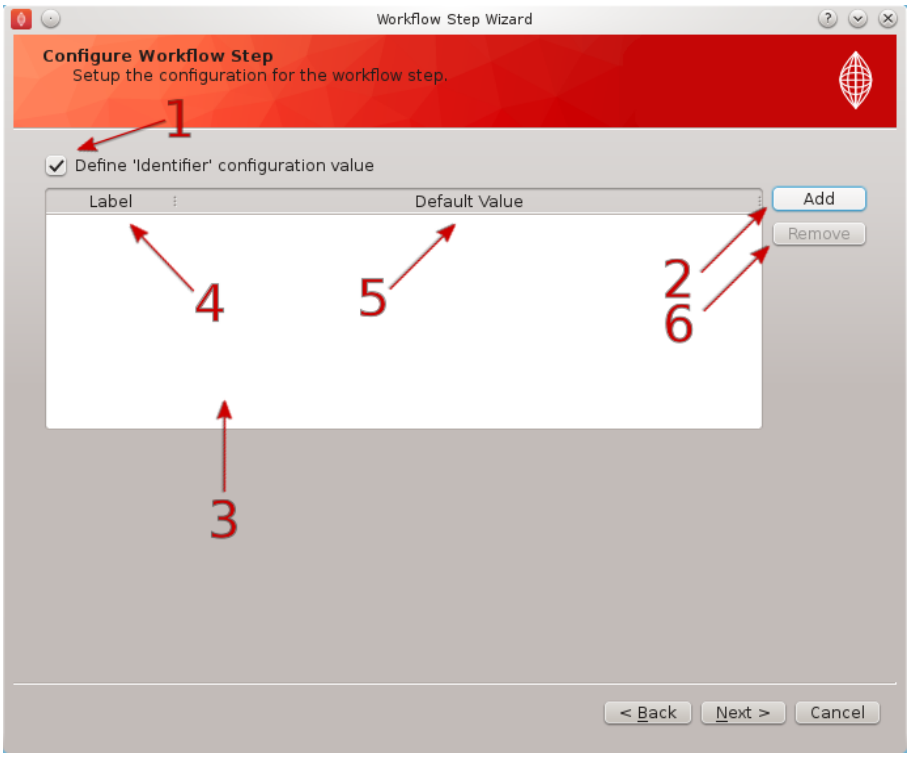


Fig. 4.6: **Figure:** The configuration page.

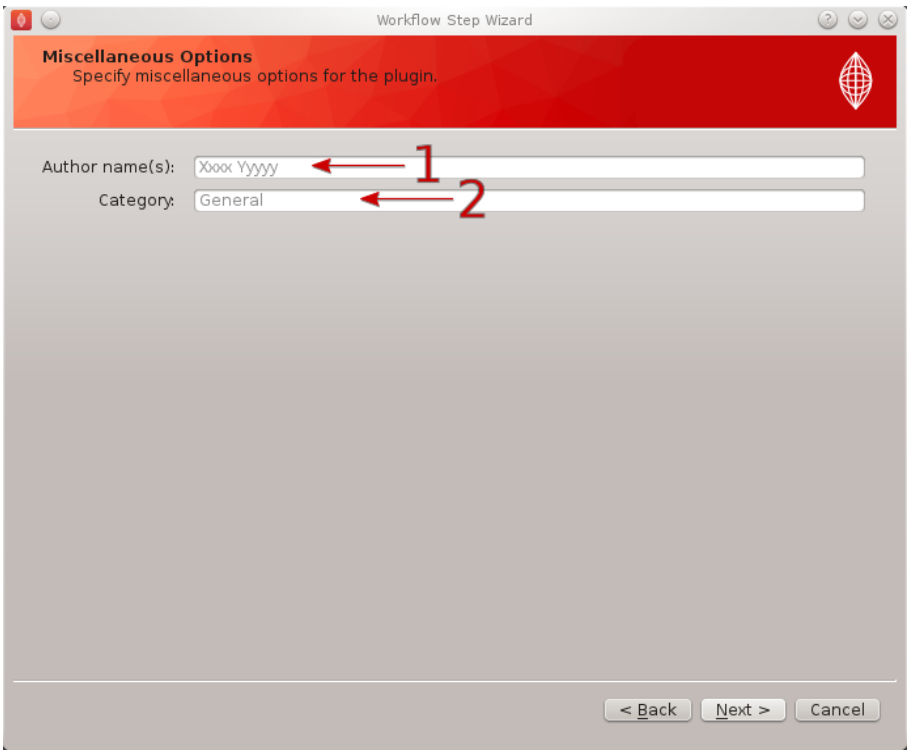


Fig. 4.7: **Figure:** The miscellaneous page.

Output Page

The output page sets the directory where the skeleton step will be generated. The output directory can be set in the text box (1), or selected from the file system using the directory chooser button (2). The 'cross' icon (3) indicates that the current directory entry is not a directory that can be written into. The output directory specified in (1) must be an existing directory that you have the ability/permission to write to before the wizard can be successfully finished.

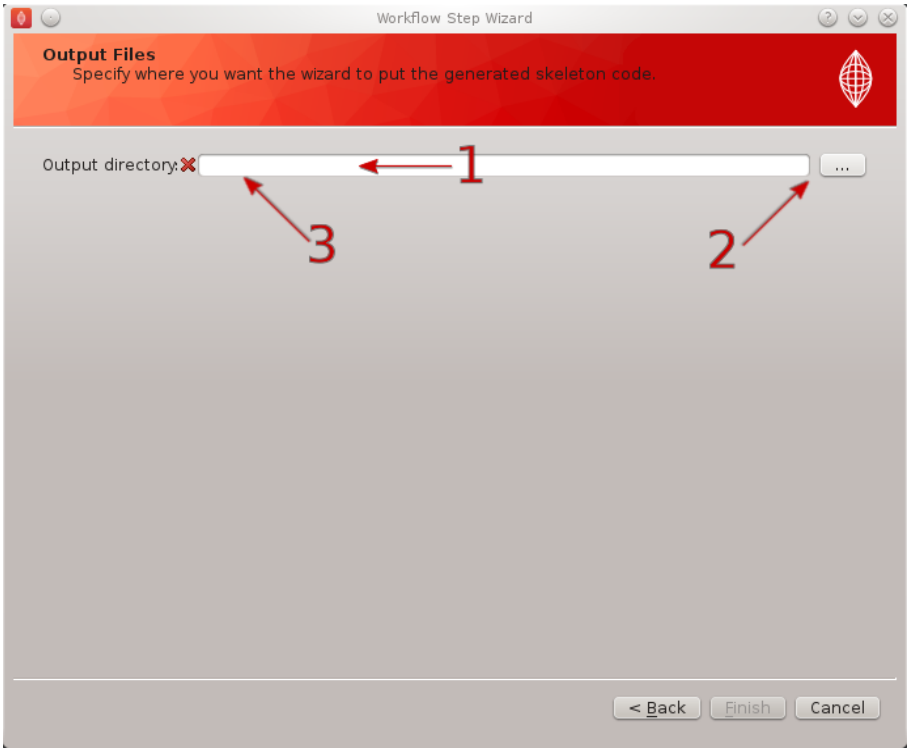


Fig. 4.8: **Figure:** The output page.

Generation

When the wizard has been completed, the skeleton step will be generated in the chosen directory. To load the skeleton step use the Plugin Manager to add the chosen directory into the list of plugin directories or use the Reload button if the new skeleton step is in an existing plugin directory.

MAP Tutorial - Create Workflow

Section author: Hugh Sorby

Note: MAP is currently under active development, and this document will be updated to reflect any changes to the software or new features that are added. You can follow the development of MAP at the launchpad project.

This document details takes the reader through the process of creating a workflow from existing MAP plugins. Having a read through the *MAP Features Demonstration* is a good way to become familiar with the features of the MAP application.

Getting Started

To get started with MAP we need to create a new workflow. To do this we use File -> New -> Workflow menu option (Ctrl-N shortcut). This option will present the user with a directory selection dialog. Use the dialog to select a directory where the workflow can be saved. Once we have chosen a directory the step box and workflow canvas will become enabled.

To create a meaningful workflow we will need to use some external plugins. To load these plugins we will use the Plugin Manager tool. The Plugin Manager tool can be found under the Tools menu. Use the Plugin Manager to add the directory location of the MAP plugins. After confirming the changes to the Plugin Manager you should see a few new additions to the Step box.

Creating the Workflow

To create a workflow we use Drag 'n' Drop to drag steps from the Step box and drop the step onto the workflow canvas. When steps are first dropped onto the canvas they show a red gear icon to indicate that the step is not configured. At a minimum a step requires an identifier to be set before it can be used.

Drag the steps *Image Source*, *Data Store* and *Automatic Segmenter* onto the workflow canvas. All the steps will show a red gear, except the 'Automatic Segmenter' step, this red gear icon indicates that the step needs to be configured. To configure a step we can either right click on the step to bring up a context menu from which the configure action can be chosen or simply click the red gear directly. See the relevant section for the configuration of a particular step.

Note: When configuring a step you are asked to set an identifier. The identifier you set must unique within the workflow and it must not start with a '.'.

Configuring the Image Source Step

The image source step requires a unique identifier for the step to be set. It also requires either a location on the local disk where the image data is located or a PMR workspace url from which the image data may be downloaded. Here we will show how to configure the Image Source step with images that have been stored in a workspace on PMR.

This step requires a unique id to be manually set. The id is used to create a file containing the step configuration information. This id for the Image Source step is also used to create a default directory under the workflow project directory if required. Once a valid identifier is entered the red highlight around the edit box will disappear.

This step configuration makes use of the PMR search widget which gives us the ability to search available workspaces on PMR. In the image source step configuration dialog seen in *Figure 1* we can see that there is a place to set a unique identifier for the step and also two tabs, one tab is for setting the image dataset location on the local disk and the other tab is for searching PMR workspaces for image data. We will leave the local disk edit box on the local file system tab empty and allow the configuration to set the default location for us.

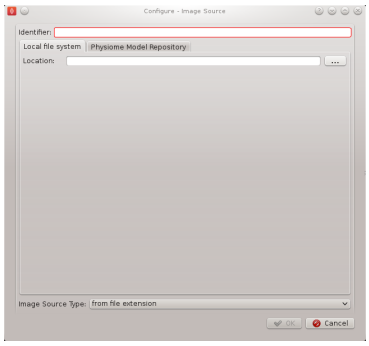


Fig. 4.9: **Figure 1**: Image source step configuration dialog.

Set the identifer edit box to bv_images and select the Physiome Model Repository tab so that we can search PMR for our images. On this tab we see We are going to conduct an ontological term [**2**] search for our images, we are looking for some images that show an anyeurism in the anterior communicating artery. To do this we can start entering the text anterior communicating artery into the search term edit box [**3**], when we pause in our typing the dialog will query the PMR OWL terms for suitable matches. We will see results similar to what is shown in *Figure 3*, we can click on the matching term in this list and the correct reference will be added to the search term edit box [**3**] for us.

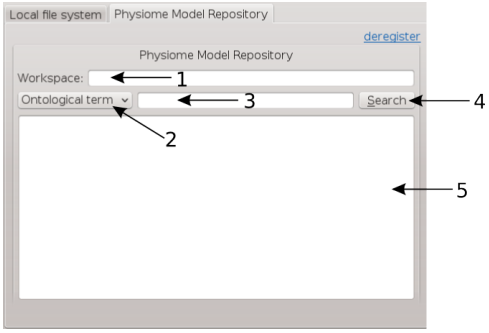


Fig. 4.10: **Figure 2**: PMR search tab, [**1**] Workspace url, [**2**] Search type combobox, [**3**] Search term, [**4**] Search button, [**5**] Search results.

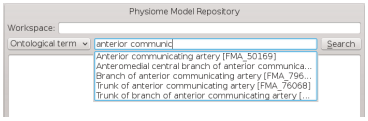


Fig. 4.11: **Figure 3**: PMR OWL terms.

With the correct term in place we can click the search button to return matching results from PMR. We will get back a single result Blood Vessel in MR Images. When we select this result in the search results list [**5**] the url for the workspace will be loaded into the workspace url edit box [**1**]. We should now have the dialog looking similar to *Figure 4*.

Fig. 4.12: **Figure 4**: Completed Physiome Model Repository search tab.

This completes the configuration of the image source step. When we click Ok in the dialog the images will be downloaded to the default directory on our local disk.

We can also use the combobox at the bottom of the dialog (*Figure 1*) to set the image type however this is only necessary if the image type cannot be determined through the filename extension. In our case we can leave this as it is.

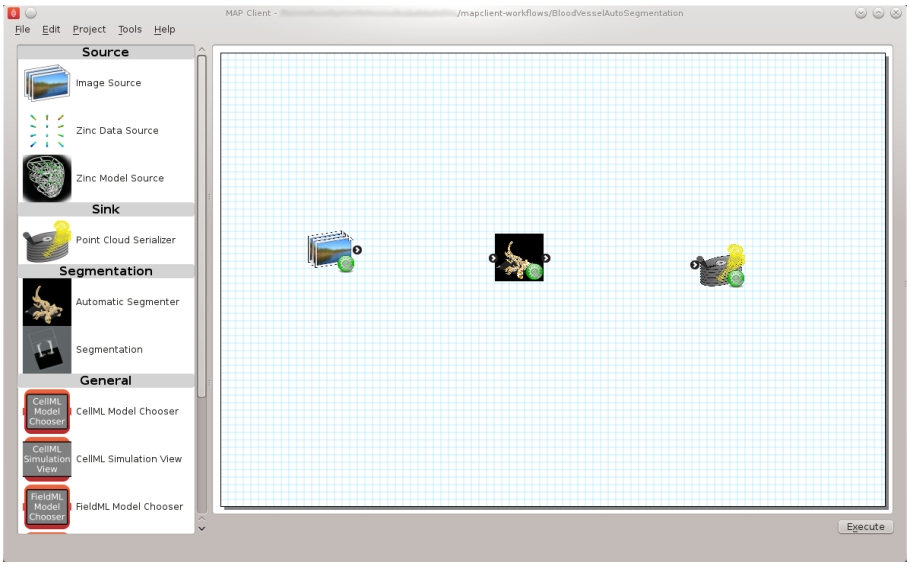
MAP is not setup to work with streamed resources so we must download the workspace from PMR to our local disk.

Configuring the Point Cloud Step

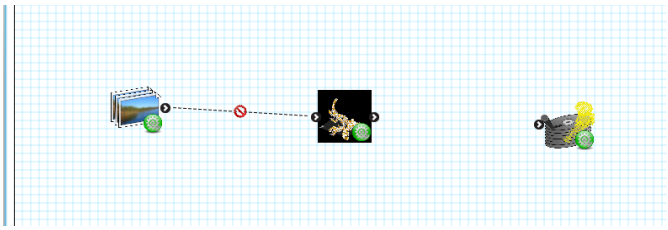
Configuring the Point Cloud step is trivial at this time. This is because the step only requires an identifier to be set. The identifier will be used to create a directory where the received point cloud will be serialized.

Executing the Workflow

At this point you should have a workflow area looking like this:



Once the All the steps in the workflow are configured (i.e. no more red gear icons) we can make connections between the steps. To make a connection between two steps the first step must provide what the second step uses. When trying to connect two steps that cannot be connected you will see a no entry icon over the connection for a short period of time and then the connection will be removed. The following image shows an incorrect connection trying to be made.



If the mouse is hovered over a port you will see a description of what the port provides or uses. To make a connection click on a port and drag the mouse to the port to be connected.

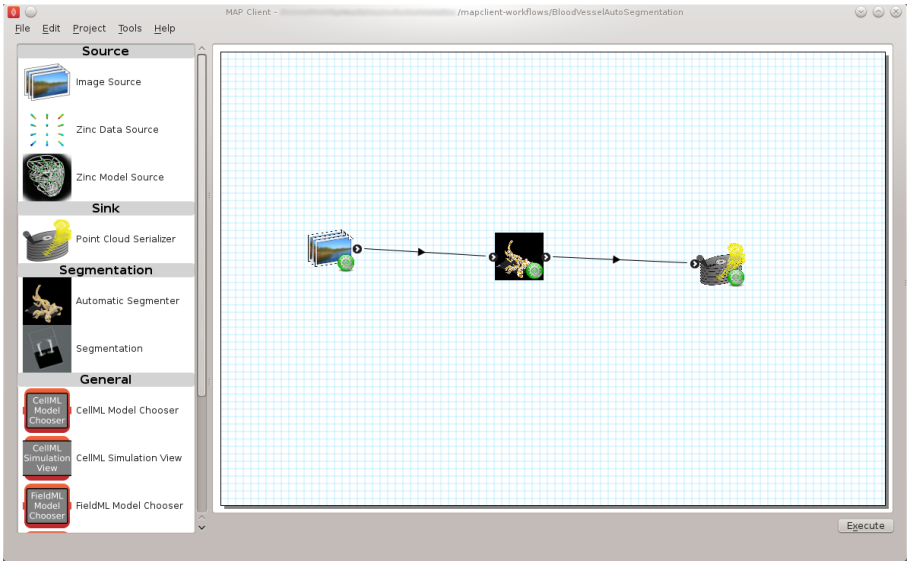
To execute the workflow we need to connect up the steps in the correct manner and save the workflow. The workflow should be connected up as can be seen in the following image.

Once the workflow has been saved the execute button in the lower left corner should become enabled. Clicking the execute button will, naturally enough, execute the workflow step by step.

Note: We can make connections between steps at anytime not just when all steps have been properly configured.

Automatic Segmenter Step

The 'Automatic Segmenter' actually allows us to interact with executing workflow. With this step we can move the image plane up and down and change the visibility of the graphical items in the scene. The image plane is moved through the use of the slider on the left hand side. The visibility of the graphical items is controlled by checking or unchecking the relevant check boxes. To continue execution of the workflow click the Done button in the lower right hand corner.



MAP Tutorial - Create Plugin

Section author: Hugh Sorby

Note: MAP is currently under active development, and this document will be updated to reflect any changes to the software or new features that are added. You can follow the development of MAP at the launchpad project.

This document details takes the reader through the process of creating a new plugin for the MAP Client. The *MAP Plugins* document defines the plugin interface that the new plugin must adhere to.

A Simple Source Step Example

In this example we will create a source step for supplying Zinc model files. There are six steps we will need to complete, and they are:

1. *Modifying the Skeleton Step*
2. *Creating an Icon*
3. *Defining the Port*
4. *Identification*
5. *Serialization*
6. *Configuration*

Modifying the Skeleton Step

We could start from scratch and create everything but our task is made a little easier by the presence of the skeleton step. The skeleton step is a step in it's own right, but it is not useful. Our task here is to take the skeleton step and rename it to provide the starting point for our new step.

To start with copy the skeletonstep directory to another directory. To make this step our own we change all occurrences of the skeletonstep name to zincmodelsourcestep. The places we have to change are:

1. The topmost directory name
2. The inner directory name, this directory is used to namespace our new step.

3. In __init__.py file in the topmost directory, we also need to uncomment the lines:

```
from zincmodelsourcestep import step
print("Plugin '{0}' version {1} by {2} loaded".format(tail, __version__, __
↪author__))
```

4. In __init__.py file in the inner directory. We have to change the name of the class to 'ZincModelSourceStep' and change the name of the step to 'Zinc Model Source'.

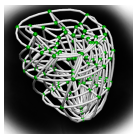
I will refer to the topmost directory as the 'step root' directory for the remainder of this example.

Creating an Icon

Creating an icon is optional as a default icon is provided, however it is nice to see your own icona and visually differentiate it from the other steps in the framework.

The are an all many of ways to create an icon to represent our step in the MAP application. So I will leave this as an exercise for the reader.

For the purposes of this example I created the icon below using the Gimp image manipulation program and it is available `here`.



We can use Qt's designer application to create a resource file from which we can generate a Python resource file. The creation of a resource file in Qt designer is out of the scope of this example but there are numerous demonstrations of how to do this available on the web.

Once the resources file has been created we can generate the Python version of this file like so:

```
pyside-rcc -py3 -o resources_rc.py qt/resources.qrc
```

A few things to note:

1. the current working directory is assumed to be '<step root>/zincmodelsourcestep/widgets' (and it exists).
2. the qt specific files are saved in a directory called 'qt', which is a subdirectory of the current working directory.
3. 'resources_rc' is the default resource file name used by the Python ui compiler, in this particular situation it is not important but just easier to name the generated resource file as the Python ui compiler expects for situations when the resources are needed by the user interface.
4. the use of the -py3 flag, when creating image resources the presence or lack thereof doesn't make much difference at the end of the day but maintaining compatibility with both Python 2 and Python 3 is desirable.

Defining the Port

To make our step useful we need to make it provide/use information for/from another step. To do this we define a port for the step. The port is described using Resource Description Framework (RDF) triples. The MAP application defines the terms 'http://physiomeproject.org/workflow/1.0/rdf-schema#port', ' http://physiomeproject.org/workflow/1.0/rdf-schema#provides' and 'http://physiomeproject.org/workflow/1.0/rdf-schema#uses' among others. We can use to these terms to interchange information about the port we will create, for this example we are interchanging information between the plugin and the application. Further we can add this information into the semantic web so that others may search for and utilise it. While adding information about our step and it's ports into the semantic web is outside of the scope of the current example, it is important to understand the other ways in which we might inform other users and developers of our work.

If we define the term 'http://physiomeproject.org/workflow/1.0/rdf-schema#zincmodeldata' to define our Zinc model data object. The tacit knowledge we take from this definition is that it is a class derived from a Python object class with three attributes:

1. _identifier
2. _elementLocation
3. _nodeLocation

Furthermore the _elementLocation will identify a file resource that defines the elements (and the nodes if _nodeLocation is empty) for the model and the _nodeLocation will identify a file resource that defines the nodes for the model. The class also has access methods 'elementFile()' and 'nodeFile()' which return a Python string holding the values of the respective attributes. The Python representation of this definition is given by the ZincModelData class:

```
class ZincModelData(object):

    def __init__(self):
        self._identifier = ''
        self._elementLocation = ''
        self._nodeLocation = ''

    def elementFile(self):
        return self._elementLocation

    def nodeFile(self):
        return self._nodeLocation
```

Identification

The step needs to be identified, among other things it determines where we deserialise and serialise to as well as being helpful for annotations. For this example we could simply supply a randomly generated identifier but we will allow the user to define one. The identifier can be used by the serialization/deserialization methods to store the step state in a file. Using the step identifier assures the developer that no-one else will write to that file. This enforces a requirement onto the identifier to be unique within a workflow.

Serialization

Serialization is the process of translating the object state into a format that can be stored (for example in a file) and later used to reinstate the object to how it was when the serialization took place. The exact how of the step serialization is up to the step author to decide, the following is just one way to approach this issue. The state of our step is stored within the ZincModelData object so we need to be able to serialize and deserialize this class. We will use the QSettings class from the Qt framework to do the serialization and deserialization for us. In the Step class we add the following two methods:

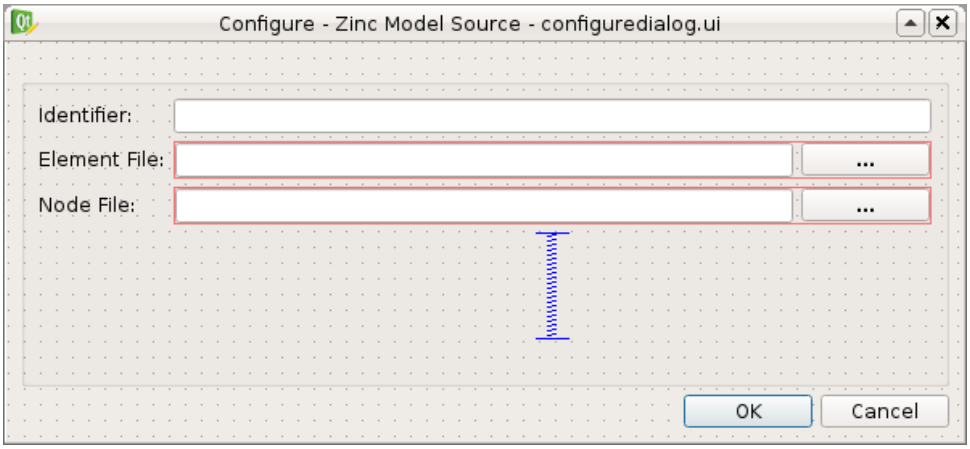
```
def serialize(self, location):
    configuration_file = os.path.join(location, getConfigFilename(self._state._
↪identifier))
    s = QtCore.QSettings(configuration_file, QtCore.QSettings.IniFormat)
    s.beginGroup('state')
    s.setValue('identifier', self._state._identifier)
    s.setValue('element', self._state._elementLocation)
    s.setValue('node', self._state._nodeLocation)
    s.endGroup()

def deserialize(self, location):
    configuration_file = os.path.join(location, getConfigFilename(self._state._
↪identifier))
    s = QtCore.QSettings(configuration_file, QtCore.QSettings.IniFormat)
    s.beginGroup('state')
```

```
    self._state._identifier = s.value('identifier', '')
    self._state._elementLocation = s.value('element', '')
    self._state._nodeLocation = s.value('node', '')
    s.endGroup()
```

The 'location' parameter that is passed into these two methods is the location of the project directory. The serialization and deserialization write to a file in this directory using the step identifier as the part of the filename. In this way with the step identifier being unique within the workflow the serialization process won't overwrite (or get overwritten by) another serialization process.

Configuration

Next we need to enable the user to be able to configure the step. To do this we can use qt-designer to create a 'configuredialog.ui' file that we can convert into Python code using 'pyside-uic'. We want the configuredialog.ui to look like this:



The Qt designer .ui file for this dialog can be found `here`. As it can be seen in the figure above we allow the user to set an identifier for the step and define the location of the element and node file that define the Zinc model. To generate the Python code from the .ui file execute the following command:

```
pyside-uic --from-imports -o ui_configuredialog.py qt/configuredialog.ui
```

Similarly for creating the resources there a couple of things to note:

1. the current working directory is assumed to be '<step root>/zincmodelsourcestep/widgets' (and it exists).
2. the .ui file is saved in a directory called 'qt', which is a subdirectory of the current working directory.
3. the use of the –from-imports flag for Python 3 compatibility.

Having created the user interface part of the configuration dialog we need to add the Python code to handle the user interaction. We will use composition of the user interface code rather than multiple-inheritance to combine the user interface code with the user interaction code. Create a Python module 'configuredialog' in the 'zincmodelsourcestep/widgets' package. In this module create a class that derives from QtGui.QDialog and sets up the user interface in the __init__ method. The code should look like this:

```
from PySide.QtGui import QDialog

from zincmodelsourcestep.widgets.ui_configuredialog import Ui_ConfigureDialog

class ConfigureDialog(QDialog):
    '''
    Configure dialog to present the user with the options to configure this step.
    '''
```

```
    def __init__(self, state, parent=None):
        '''
        Constructor
        '''
        QDialog.__init__(self, parent)
        self._ui = Ui_ConfigureDialog()
        self._ui.setupUi(self)
```

It can be seen in this code snippet that I am passing in an object using the label 'state' into the constructor of my ConfigureDialog class. This object is used to represent the state of the ConfigureDialog object for the purposes of serialization and validation. This object is defined in another Python module called 'zincmodeldata' and contains a class named 'ZincModelData' that has three attributes:

1. _identifier
2. _elementLocation
3. _nodeLocation

This class is used by and returned from two public methods of the ConfigureDialog class setState and getState. These two methods set the state and get the state of the corresponding user interface elements accordingly. The implementation of these two methods look like this:

```
def setState(self, state):
    self._ui.identifierLineEdit.setText(state._identifier)
    self._ui.elementLineEdit.setText(state._elementLocation)
    self._ui.nodeLineEdit.setText(state._nodeLocation)

def getState(self):
    state = ZincModelData()
    state._identifier = self._ui.identifierLineEdit.text()
    state._elementLocation = self._ui.elementLineEdit.text()
    state._nodeLocation = self._ui.nodeLineEdit.text()

    return state
```

The ConfigureDialog class is also going to help us validate the step configuration. When we have a valid step we can execute the workflow that uses the step. So when validating our step we need to ensure that it has everything required for successful execution. In this case, the requirements are an existing element file. A node file isn't strictly necessary as it may be incorporated into the element file.

With this in mind we define the 'validate' method of the ConfigureDialog class to return True when we have the location of an existing exelem file and False otherwise. It is also important to document the condition(s) under which the step is considered valid so that other uses understand the expected behaviour. The 'validate' method should look like this:

```
def validate(self):
    element_filename = self._ui.elementLineEdit.text()
    element_valid = len(element_filename) > 0 and os.path.exists(element_filename)

    self._ui.buttonBox.button(QDialogButtonBox.Ok).setEnabled(element_valid)

    return element_valid
```

By manipulating the state of the 'Ok' button we know that the step is valid when returning from the dialog when the 'Ok' button has been activated.

As far as the ConfigureDialog is concerned all it requires is for the connections between the widget signals and class methods to be defined. To make the required connections we can create a method called '_makeConnections' which we can call from the constructor and add three supporting methods for handling the responses to user actions. Here is the code we need to add:

```
def _makeConnections(self):
    self._ui.elementButton.clicked.connect(self._elementButtonClicked)
    self._ui.nodeButton.clicked.connect(self._nodeButtonClicked)
    self._ui.elementLineEdit.textChanged.connect(self.validate)

def _lineEditFile(self, line_edit):
    (fileName, _) = QFileDialog.getOpenFileName(self, 'Select Zinc File')

    if fileName:
        location = os.path.basename(fileName)
        line_edit.setText(fileName)

    self.validate()

def _elementButtonClicked(self):
    self._lineEditFile(self._ui.elementLineEdit)

def _nodeButtonClicked(self):
    self._lineEditFile(self._ui.nodeLineEdit)
```

There are a number of niceties that we have not added into this example code that we could have. We have also not added any checks to make sure the file selected is an exelem file. But this fits in with the approach where we consider that TUINAI.

Glossary

Python The Python interpreter

Mercurial Distributed version control system.

Appendix A - Generating html documentation

This appendix covers how to generate html files from the ReStructured text documentation source files. The documentation is generated using the Sphinx documentation tool. Sphinx is a tool that makes it easy to create intelligent and beautiful documentation.

Generating the documentation is very easy. First you need to download and install Sphinx if you don't already have it. Then you use the command line to run the sphinx build tool, which will generate the documentation in the target format.

There are two ways of generating the documentation. You can either use the supplied Makefile in the resources directory or you can use 'sphinx-build' directly. The Makefile is setup to use specific locations, but these location can be overridden when invoking the make command. The 'sphinx-build' application requires the source directory, the build directory, the configuration directory and the documentation target format to be supplied on the command line.

The commands for these two methods of generating the documentation are given here:

```
# Method 1.
make -f docs/resources/Sphinx.Makefile html

# Method 2.
sphinx-build -t html -c docs/resources docs build
```

note:

- This assumes your current working directory is the parent of the 'docs' directory
- If a directory 'build' doesn't exist in the current directory it will be created

That's it! Now you can use your favourite webbrowser to read the documentation. The 'index.html' file for method 1. is located in 'build/html' and for method 2. it is available in 'build'.

CHAPTER 5

Glossary

Clone Clone is a Mercurial term that means to make a complete copy of a Mercurial repository. This is done in order to have a local copy of a repository to work in.

Embedded workspace

Embedded workspaces A Mercurial concept that allows workspaces to be nested within other workspaces.

Exposure

Exposures A publicly available page that provides access to and information about a specific revision of a workspace. Exposures are used to publish the contents of workspaces at points in time where the model(s) contained are considered to be useful.

Exposures are created by the PMR software, and offer views appropriate to the type of model being exposed. CellML files for example are presented with options such as code generation and mathematics display, whereas FieldML models might offer a 3D view of the mesh.

Fork A copy of the workspace which includes all the original version history, but is owned by the user who created the fork.

Mercurial Mercurial is a distributed version control system, used by the Physiome Model Repository software to maintain a history of changes to files in *workspaces*. See a tour of the Mercurial basics for some good introductory material.

Pull

Pulling The term used with distributed version control systems for the action of pulling changes from one clone of the repository into another. With PMR, this usually implies pulling from a workspace in the model repository into a clone of the workspace on your local machine.

Push

Pushing The term used with distibuted version control systems for the action of pushing changes from one clone of the repository into another. With PMR, this usually implies pushing from a workspace clone on your local machine back to the workspace in the model repository, but could be into any other clone of the workspace. See a tour of the Mercurial basics for some good introductory material.

Python Python is a programming language that lets you work more quickly and integrate your systems more effectively. See http://python.org for all the details.

Synchronize Used to pull the contents or changes from other *Mercurial* repositories into a workspace via a URI.

Workspace

Workspaces A *Mercurial* repository hosted on the Physiome Model Repository. This is essentially a folder or directory in which files are stored, with the added feature of being version controlled by the distributed version control system called Mercurial.

CHAPTER 6

Tutorial to do list

General

Todo

- Add many more references (`.. _like-this:`) to docs for cross-referencing.
- make sure all references to the staging instance are updated to teaching.physiomeproject.org

Within sections

Todo

This section needs more work.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/andre-test-xy2345/checkouts/latest/PMR2/embeddedworkspaces.rst, line 9.)

Todo

- Update all documentation to reflect workspace ID changes and user workspace changes, if they go ahead.
- Get embedded workspaces doc written.
- Get some best practice docs written.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/andre-test-xy2345/checkouts/latest/PMR2/index.rst, line 36.)

Todo

These images need to be updated if there is time.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/andre-test-xy2345/checkouts/latest/opencor-tutorials/newWork.rst, line 66.)

Todo

- Add many more references (`.. _like-this:`) to docs for cross-referencing.
- make sure all references to the staging instance are updated to teaching.physiomeproject.org

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/andre-test-xy2345/checkouts/latest/todoList.rst, line 10.)

CHAPTER 7

MAP Client Documentation

The documentation for MAP Client.

CHAPTER 8

The MAP Client

This section of the tutorial covers the MAP Client.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Index