Developer document
Multiscale modelling in Moose

Dilawar Singh *
dilawars@ncbs.res.in

November 1, 2013

Abstract

This documentation is a literate program. This describes and ongoing attempt to make moose more capable of doing multi-scale modelling. There are various XML based models available, each describing a particular aspect of neural activities. Some describes the chemical activities inside neuron while some other describe their electrical properties etc. We wish to write a super-XML model which can make use of all these models and map them onto moose. Currently, we call it `adaptorML`. Later we should find a cool name for it such as `mooseML` or `moooml`.

Dependencies You need `noweb` tool to generate documentation from this file. This file contains some macros which are not understood by `noweb`. These macros are used by a python script `./pynoweb.py` to generate `noweb` file. You need not know how to use it. Just run `./generate_code_and_docs.sh --doc` to generate documentation after installing `noweb`. Run the same command without `--doc` switch and you have your working application in `src` directory.

Contact the very friendly Homo Sapience Sapience named Dilawar Singh if you need any help. He can be reached at `dilawars@ncbs.res.in`.

Wr
duc
mu
mo

1 Initialize store-house

Dependencies and import We need `lxml.etree` for XML parsing. To print error and warning messages, I wrote a small module `DebugModule` 7. This module is implemented in file `debug.nw`.

Imports This chunk `Imports` keeps the essentials modules which we'll need in almost all files. We might occasionally also need a logger from python `logging` library. Let's create a standard logger too.

1 ⟨*Import* 1⟩≡ (2a 4a 23b)

```
# Basic imports
import os
import sys
import logging
import debug

logger = logging.getLogger('multiscale')
from lxml import etree
```

*Junior Research Fellow at National Center for Biological Sciences. Ph.D. candidate at EE IIT Bombay (currently on leave)

Entry point This is entry point of this program. Let's write down the sturcture of program. This is what we want to do in this application.

2a ⟨*main.py* 2a⟩≡
⟨*Import* 1⟩
⟨*functions in main* 3a⟩
⟨*argument parser* 2b⟩
⟨*parse xml models and handover control to main class* 3b⟩

Argument parser This application accepts paths of XML based models from command line. More than one XML model can be passed from command line. Python comes with a standard library `argparse` well suited to do this job. If more than two XML models are to be passed, each should be passed with its own `--xml` switch e.g. to pass `modelA.xml` and `modelB.xml` from command line, use the string `--xml modelA.xml --xml modelB.xml`. We must also provide the path of `adaptorML` file.

2b ⟨*argument parser* 2b⟩≡ (2a)

```
# standard module for building a command line parser.
import argparse

# This section build the command line parser
argParser = argparse.ArgumentParser(description= 'Mutiscale modelling of neurons')
argParser.add_argument('--nml', metavar='nmlpath'
    , required = True
    , nargs = '+'
    , help = 'nueroml model'
    )
argParser.add_argument('--sbml', metavar='nmlpath'
    , nargs = '*'
    , help = 'sbml model'
    )
argParser.add_argument('--mechml', metavar='mechml'
    , nargs = '*'
    , help = 'mechml model'
    )
argParser.add_argument('--chml', metavar='channelml'
    , nargs = '*'
    , help = 'Channelml model'
    )
argParser.add_argument('--3dml', metavar='3dml'
    , nargs = '*'
    , help = '3DMCML model'
    )
argParser.add_argument('--meshml', metavar='meshml'
    , nargs = '*'
    , help = 'MeshML model'
    )
argParser.add_argument('--adaptor', metavar='adaptor'
    , required = True
    , nargs = '+'
    , help = 'AdaptorML for moose'
    )
args = argParser.parse_args()
```

Once we have verified paths of XML models, we need a module to parse them. For the purpose of modularity, we wrote this module in its on literate file `parser.nw` and you can see its documentation in section 6.

Parse XML models But before we parse, we need a helper function to check if given paths exists and are readable.

3a ⟨*functions in main* 3a⟩≡ (2a)

```
def ifPathsAreValid(paths) :
  ''' Verify if path exists and are readable. '''
  if paths :
    paths = vars(paths)
    for p in paths :
      if not paths[p] : continue
      for path in paths[p] :
        if not path : continue
        if os.path.isfile(path) : pass
        else :
          debug.printDebug("ERROR"
            , "Filepath {0} does not exists".format(path))
          return False
      # check if file is readable
      if not os.access(path, os.R_OK) :
        debug.printDebug("ERROR", "File {0} is not readable".format(path))
  return True
```

Parse XML files At least one model must be provided by the user. Validation is not enable in this version.

Download new neuroML2 models and turn validate=True in `parseModels` function call.

3b ⟨*parse xml models and handover control to main class* 3b⟩≡ (2a)

```
import parser
if args :
  if ifPathsAreValid(args) :
    logger.info("Started parsing XML models")
    debug.printDebug("INFO", "Started parsing XML models")
    etreeDict = parser.parseModels(args, validate=False)
    debug.printDebug("INFO", "Parsing of models is done")
    ⟨hand over control to class in multiscale module 3c⟩
    print("Done!")
  else :
    debug.printDebug("FATAL", "One or more model file does not exists.")
    sys.exit()
else :
  debug.printDebug("FATAL", "Please provide at least one model. None given.")
  sys.exit()
```

Create multi-scale models in Moose We are done initializing our application. Lets define a class `Multiscale` and hand over the control to its object. This class is defined in `multiscale.nw`. We need to import module `multiscale` to be able to initialize and object of this class.

3c ⟨*hand over control to class in multiscale module* 3c⟩≡ (3b)

```
import multiscale
multiScaleObj = multiscale.Multiscale(etreeDict)
multiScaleObj.buildMultiscaleModel()
```

Over to 2.

2 Construct a Database/AST from XML models

This is our defining module and it contains the basic functionality of this application. All XML modules are parsed into a dictionary and this dictionary is passed to the object of class `Multiscale` which is the main class in this module. We also have `adaptorML` file parsed in dictionary.

Remark 1. *Why AST/Database?*

Should I directly map XML models to moose objects or a intermediate representation would be better. I am leaning towards haivng a intermediate sqlite3 based data-structure. The benefit of using sqlite is that we can simply insert and query the database without having to keep all XML models open. We'll simply populate the database for each given model. Details are bit hazy in my mind right now and I should add the them to this document as they become clearer to me.

Structure Structure of this module is following.

4a ⟨*multiscale* 4a⟩≡
⟨*Import* 1⟩
⟨*Local imports* 14c⟩
⟨*Definition of class* `Multiscale` 4b⟩

A skeleton of class Now we have parsed XML. We are passing the parsed XML in dictionary to a method `buildMultiscaleModel` of this class. We know, what this class must have at this point. Let's write it down and we'll wonder later how to add more functionality.

4b ⟨*Definition of class* `Multiscale` 4b⟩≡ (4a)

```
class Multiscale :

  def __init__(self, xmlDict) :
    self.xmlDict = xmlDict
    ⟨initialize members 14e⟩
    debug.printDebug("INFO", "Object of class Multiscale intialized ...")

  ⟨methods 14d⟩

  # This is the entry point of this class.
  def buildMultiscaleModel(self) :
      debug.printDebug("INFO", "Starting to build multiscale model")
      ⟨flow of executation 15a⟩

  def exit(self) :
    # Clean up before you leave
    ⟨clean up the mess 14f⟩

  # Write down the tests, whenever needed.
  ⟨tests (never defined)⟩
```

Before we move on, lets discuss some models developed by others we need to import into moose.

3 Traub models and their translation into Python

This file is based on file `proto8.py`. Following is based on the header of this file.

Model information This implementation is mostly based on on the graub91proto.g by Dave Beeman. Main difference is addition of Glu and NMDA [1] channels. The 1991 Traub set of voltage and concentration dependent channels implemented as tabchannels by Dave Beeman R.D.Traub, R. K. S. Wong, R. Miles, and H. Michelson Journal of Neurophysiology, Vol. 66, p. 635 (1991)

This file depends on functions and constants defined in defaults.g As it is also intended as an example of the use of the tabchannel object to implement concentration dependent channels, it has extensive comments. Note that the original units used in the paper have been converted to SI (MKS) units. Also, we define the ionic equilibrium potentials relative to the resting potential, `EREST_ACT`. In the paper, this was defined to be zero. Here, we use -0.060 volts, the measured value relative to the outside of the cell.

November 1999 update for GENESIS 2.2: Previous versions of this file used a combination of a table, `tabgate`, and `vdep\_channel` to implement the Ca-dependent K Channel - $K(C)$. This new version uses the new tabchannel "instant" field, introduced in GENESIS 2.2, to implement an "instantaneous" gate for the multiplicative Ca-dependent factor in the conductance. This allows these channels to be used with the fast `hsolve` chanmodes ¿ 1.

This Traub model is now converted to an equivalent python model described here. It is used in `pymoose`.

5a ⟨*proto.py* 5a⟩≡

```
import moose
import numpy
import math

⟨Define constants 5b⟩
⟨Functions to create channels 6a⟩
⟨Functions to maniputate property of channels 8⟩
⟨Glu receptor 12c⟩
⟨NMDA receptor 13a⟩
⟨Spike detector 14b⟩
```

EREST_ACT	Hippocampal cell resting potentional	$Volt$
ENA	Equilibrium potential of Sodium	$Volt$
EK	Equilibrium potential of Potassium	$Volt$
ECA	Equilibrium potential of Calcium	$Volt$
SOMA_A	Area of soma	m^2

Table 1: Constants in this model

Constants in model

5b ⟨*Define constants* 5b⟩≡ (5a)

```
EREST_ACT = -0.060      # /* hippocampal cell resting potl */
ENA = 0.115 + EREST_ACT # // 0.055
EK = -0.015 + EREST_ACT # // -0.075
ECA = 0.140 + EREST_ACT # // 0.080
SOMA_A = 3.320e-9       # // soma area in square meters
```

[1] A predominant molecular device for controlling synaptic plasticity and memory function

Channels in model For these channels, the maximum channel conductance (*Gbar*) has been calculated using the `CA3` [2] soma channel conductance densities and soma area. Typically, the functions which create these channels will be used to create a library of prototype channels. When the cell reader creates copies of these channels in various compartments, it will set the actual value of Gbar by calculating it from the cell parameter file.

6a ⟨*Functions to create channels* 6a⟩≡ (5a)
⟨*Ordinary Ca channel* 6b⟩
⟨*Ca-dependent K-AHP channel* 9⟩
⟨*Ca-dependent K-C channel* 10⟩
⟨*Tabchannel Na Hippocampal cell channel* 11⟩
⟨*Tabchannel K-DR Hippocampal cell channel* 12a⟩
⟨*Tabchannel K-A Hippocampal cell channell* 12b⟩

Tabulated Calcium channel

Note 1. *This note is from Traub's model written in Fortran*

Often, the alpha and beta rate parameters can be expressed in the functional form $y = \frac{A+Bx}{C+\exp\frac{x+D}{F}}$. *When this is the case, the command* `setupalpha chan gate AA AB AC AD AF BA BB BC BD BF` *can be used to simplify the process of initializing the A and B tables for the X, Y and Z gates. Although* `setupalpha` *has been implemented as a compiled GENESIS command, it is also defined as a script function in the* `neurokit/prototypes/defaults.g` *file. Although this command can be used as a "black box", its definition shows some nice features of the* `tabchannel` *object, and some tricks we will need when the rate parameters do not fit this form.*

We can give a short summary of the variables used in this function and relationship among them.

$$Gbar = GkX^{Xpower}Y^{Ypower}Z^{Zpower} \tag{1}$$

Ek	Reversal potential of channel	constant
Gk	Channel conductance	variable
Gbar	Maximum channel conductance	constant
Xpower [3]	Power of X-gate	constant
Ypower	Power of Y gate	constant
Zpower	Power of Z gate	constant

Table 2: A short summary of variables used in function `make_Ca`

6b ⟨*Ordinary Ca channel* 6b⟩≡ (6a)
```
# Traub tabulated calcium channel
def make_Ca():
  if moose.exists( 'Ca' ):
      return
  Ca = moose.HHChannel( 'Ca' )
  Ca.Ek = ECA
  Ca.Gbar = 40 * SOMA_A
  Ca.Gk = 0
  Ca.Xpower = 2
  Ca.Ypower = 1
  Ca.Zpower = 0

  xgate = moose.element( 'Ca/gateX' )
  ⟨setup X-gate using Traub model 7a⟩

  ygate = moose.element( 'Ca/gateY' )
  ⟨setup Y-gate using Traub model 7b⟩
  ⟨put information into cell-reader 7c⟩
```

[2] hippocampal pyramidal cell

Set-up X-gate Use `setupAlpha` which is similar to Traub `setupalpaha`. We pass a numpy array of 13 elements.

7a ⟨*setup X-gate using Traub model* 7a⟩≡ (6b)

```
xA = numpy.array( [ 1.6e3, 0, 1.0, -1.0 * (0.065 + EREST_ACT), -0.01389
        , -20e3 * (0.0511 + EREST_ACT), 20e3, -1.0, -1.0 * (0.0511 + EREST_ACT)
        , 5.0e-3, 3000, -0.1, 0.05 ] )
xgate.setupAlpha( xA )
```

Set-up Y-gate The Y gate gCa/r is not quite of this form. For $V > EREST\_ACT$, $alpha = 5\exp(-50(V -$ EREST_ACT)). Otherwise, $alpha = 5$. Over the entire range, $alpha + beta = 5$. To create the `Y_A` and `Y_B` tables, we use some of the pieces of the `setupalpha` function.

7b ⟨*setup Y-gate using Traub model* 7b⟩≡ (6b)

```
ygate.min = -0.1
ygate.max = 0.05
ygate.divs = 3000
yA = numpy.zeros( (ygate.divs + 1), dtype=float)
yB = numpy.zeros( (ygate.divs + 1), dtype=float)

#Fill the Y_A table with alpha values and the Y_B table with (alpha+beta)
dx = (ygate.max - ygate.min)/ygate.divs
x = ygate.min
for i in range( ygate.divs + 1 ):
  if ( x > EREST_ACT):
    yA[i] = 5.0 * math.exp( -50 * (x - EREST_ACT) )
  else:
    yA[i] = 5.0
  yB[i] = 5.0
  x += dx
ygate.tableA = yA
ygate.tableB = yB
```

Setup cell-reader As we typically use the cell reader to create copies of these prototype elements in one or more compartments, we need some way to be sure that the needed messages are established. Although the cell reader has enough information to create the messages which link compartments to their channels and to other adjacent compartments, it must be provided with the information needed to establish additional messages. This is done by placing the message string in a user-defined field of one of the elements which is involved in the message. The cell reader recognizes the added object names "addmsg1", "addmsg2", etc. as indicating that they are to be evaluated and used to set up messages. The paths are relative to the element which contains the message string in its added field. Thus, "../Ca_conc" refers to the sibling element `Ca_conc` and "." refers to the Ca element itself.

7c ⟨*put information into cell-reader* 7c⟩≡ (6b)

```
# Tell the cell reader that the current from this channel must be fed into
# the Ca_conc pool of calcium.
addmsg1 = moose.Mstring( '/library/Ca/addmsg1' )
addmsg1.value = '.  IkOut  ../Ca_conc  current'

# in some compartments, whe have an NMDA_Ca_conc object to put the current
# into.
addmsg2 = moose.Mstring( '/library/Ca/addmsg2' )
addmsg2.value = '.  IkOut  ../NMDA_Ca_conc  current'
```

Convert Ca current to Ca concentration Next, we need an element to take the Calcium current calculated by the Ca channel and convert it to the Ca concentration. The `Ca_concen` object solves the equation $\frac{dC}{dt} = BI_{Ca} - \frac{C}{\tau}$, and sets $Ca = Ca_{base} + C$. As it is easy to make mistakes in units when using this Calcium diffusion equation, the units used here merit some discussion.

Note 2. *Upi's notes on Traub*

With $Ca_{base} = 0$, this corresponds to Traub's diffusion equation for concentration, except that the sign of the current term here is positive, as GENESIS uses the convention that `I_Ca` *is the current flowing INTO the compartment through the channel. In SI units, the concentration is usually expressed in* `moles`$/m^3$ *(which equals* `millimoles/liter`*), and the units of B are chosen so that* $B = \frac{1}{ion\_charge \times Faraday \times volume}$*. Current is expressed in* `Amperes` *and one* `Faraday` *= 96487* `Coulombs`*. However, in this case, Traub expresses the concentration in arbitrary units, current in* `micro-Amps` *and uses τ = 13.33 msec. If we use the same concentration units, but express current in amperes and tau in seconds, our B constant is then 10^{12} times the constant (called "phi") used in the paper. The actual value used will be typically be determined by the cell reader from the cell parameter file. However, for the prototype channel we wlll use Traub's corrected value for the soma. (An error in the paper gives it as 17,402 rather than 17.402.) In our units, this will be 17.402e12.*

8 ⟨*Functions to maniputate property of channels* 8⟩≡ (5a) 14a▷

```
def make_Ca_conc():
  if moose.exists( 'Ca_conc' ):
    return
  conc = moose.CaConc( 'Ca_conc' )
  conc.tau = 0.013333  # sec
  conc.B  = 17.402e12 # Curr to conc conversion for soma
  conc.Ca_base = 0.0
```

This `Ca_concen` element should receive a message from any calcium channels with the current going through the channel. Here we have this specified in the Ca channel, with the idea that more than one channel might contribute Ca ions to this calcium pool. In the original GENESIS file this was specified here in `make_Ca_conc`.

Tabulated Ca-dependent Potassium AHP Channel This is a `tabchannel` [4] which gets the calcium concentration from `Ca_conc` in order to calculate the activation of its Z gate. It is set up much like the Ca channel, except that the A and B tables have values which are functions of concentration, instead of voltage.

9 ⟨*Ca-dependent K-AHP channel* 9⟩≡ (6a)

```
def make_K_AHP():
    if moose.exists( 'K_AHP' ):
        return

    K_AHP = moose.HHChannel( 'K_AHP' )
    K_AHP.Ek = EK                       # V
    K_AHP.Gbar = 8 * SOMA_A             # S
    K_AHP.Gk = 0                        # S
    K_AHP.Xpower = 0
    K_AHP.Ypower = 0
    K_AHP.Zpower = 1

    zgate = moose.element( 'K_AHP/gateZ' )
    xmax = 500.0
    zgate.min = 0
    zgate.max = xmax
    zgate.divs = 3000
    zA = numpy.zeros( (zgate.divs + 1), dtype=float)
    zB = numpy.zeros( (zgate.divs + 1), dtype=float)
    dx = (zgate.max - zgate.min)/zgate.divs
    x = zgate.min
    for i in range( zgate.divs + 1 ):
        if (x < (xmax / 2.0 )):
            zA[i] = 0.02*x
        else:
            zA[i] = 10.0
        zB[i] = zA[i] + 1.0
        x = x + dx

    zgate.tableA = zA
    zgate.tableB = zB

    # Use an added field to tell the cell reader to set up a message from the
    # Ca_Conc with concentration info, to the current K_AHP object.
    addmsg1 = moose.Mstring( '/library/K_AHP/addmsg1' )
    addmsg1.value = '../Ca_conc  concOut  . concen'
```

[4] Tabulated channel. Instead of computing the parameter directly, one used look-up table approach

Ca-dependent Pottasium Channel - K(C) - `vdep_channel` **with table and tabgate** The expression for the conductance of the potassium C-current channel has a typical voltage and time dependent activation gate, where the time dependence arises from the solution of a differential equation containing the rate parameters alpha and beta. It is multiplied by a function of calcium concentration that is given explicitly rather than being obtained from a differential equation. Therefore, we need a way to multiply the activation by a concentration dependent value which is determined from a lookup table. This is accomplished by using the Z gate with the new tabchannel "instant" field, introduced in GENESIS 2.2, to implement an "instantaneous" gate for the multiplicative Ca-dependent factor in the conductance.

10 ⟨*Ca-dependent K-C channel* 10⟩≡ (6a)

```
def make_K_C():
  if moose.exists( 'K_C'):
    return

  K_C = moose.HHChannel( 'K_C' )
  K_C.Ek = EK          #  V
  K_C.Gbar = 100.0 * SOMA_A   #  S
  K_C.Gk = 0          #  S
  K_C.Xpower = 1
  K_C.Zpower = 1
  K_C.instant = 4        # Flag: 0x100 means Z gate is instant.

  # Now make a X-table for the voltage-dependent activation parameter.
  xgate = moose.element( 'K_C/gateX' )
  xgate.min = -0.1
  xgate.max = 0.05
  xgate.divs = 3000
  xA = numpy.zeros( (xgate.divs + 1), dtype=float)
  xB = numpy.zeros( (xgate.divs + 1), dtype=float)
  dx = (xgate.max - xgate.min)/xgate.divs
  x = xgate.min
  for i in range( xgate.divs + 1 ):
    alpha = 0.0
    beta = 0.0
    if (x < EREST_ACT + 0.05):
      alpha = math.exp( 53.872 * (x - EREST_ACT) - 0.66835 ) / 0.018975
      beta = 2000* (math.exp ( (EREST_ACT + 0.0065 - x)/0.027)) - alpha
    else:
      alpha = 2000 * math.exp( ( EREST_ACT + 0.0065 - x)/0.027 )
      beta = 0.0
    xA[i] = alpha
    xB[i] = alpha + beta
    x = x + dx
  xgate.tableA = xA
  xgate.tableB = xB

    # Create a table for the function of concentration, allowing a
    # concentration range of 0 to 1000, with 50 divisions.  This is done
    # using the Z gate, which can receive a CONCEN message.  By using
    # the "instant" flag, the A and B tables are evaluated as lookup tables,
    #  rather than being used in a differential equation.
  zgate = moose.element( 'K_C/gateZ' )
  zgate.min = 0.0
  xmax = 500.0
  zgate.max = xmax
  zgate.divs = 3000
  zA = numpy.zeros( (zgate.divs + 1), dtype=float)
  zB = numpy.zeros( (zgate.divs + 1), dtype=float)
  dx = ( zgate.max -  zgate.min)/ zgate.divs
  x = zgate.min
  for i in range( xgate.divs + 1 ):
    if ( x < ( xmax / 4.0 ) ):
      zA[i] = x * 4.0 / xmax
```

```
    else:
      zA[i] = 1.0
    zB[i] = 1.0
    x += dx
  zgate.tableA = zA
  zgate.tableB = zB

  # Now we need to provide for messages that link to external elements.
  # The message that sends the Ca concentration to the Z gate tables is stored
  # in an added field of the channel, so that it may be found by the cell
  # reader.
  addmsg1 = moose.Mstring( '/library/K_C/addmsg1' )
  addmsg1.value = '../Ca_conc  concOut  . concen'
```

Tabchannel Na Hippocampal cell channel

11 ⟨*Tabchannel Na Hippocampal cell channel* 11⟩≡ (6a)

```
def make_Na():
  if moose.exists( 'Na' ):
    return
  Na = moose.HHChannel( 'Na' )
  Na.Ek = ENA        #  V
  Na.Gbar = 300 * SOMA_A  #  S
  Na.Gk = 0        #  S
  Na.Xpower = 2
  Na.Ypower = 1
  Na.Zpower = 0

  xgate = moose.element( 'Na/gateX' )
  xA = numpy.array( [ 320e3 * (0.0131 + EREST_ACT),
    -320e3, -1.0, -1.0 * (0.0131 + EREST_ACT), -0.004,
    -280e3 * (0.0401 + EREST_ACT), 280e3, -1.0,
    -1.0 * (0.0401 + EREST_ACT), 5.0e-3,
    3000, -0.1, 0.05 ] )
  xgate.setupAlpha( xA )

  ygate = moose.element( 'Na/gateY' )
  yA = numpy.array( [ 128.0, 0.0, 0.0, -1.0 * (0.017 + EREST_ACT), 0.018,
    4.0e3, 0.0, 1.0, -1.0 * (0.040 + EREST_ACT), -5.0e-3,
    3000, -0.1, 0.05 ] )
  ygate.setupAlpha( yA )
```

Tabchannel K(DR) Hippocampal cell channel

12a ⟨*Tabchannel K-DR Hippocampal cell channel* 12a⟩≡ (6a)

```
def make_K_DR():
  if moose.exists( 'K_DR' ):
    return
  K_DR = moose.HHChannel( 'K_DR' )
  K_DR.Ek = EK        #  V
  K_DR.Gbar = 150 * SOMA_A  #  S
  K_DR.Gk = 0        #  S
  K_DR.Xpower = 1
  K_DR.Ypower = 0
  K_DR.Zpower = 0

  xgate = moose.element( 'K_DR/gateX' )
  xA = numpy.array( [ 16e3 * (0.0351 + EREST_ACT),
    -16e3, -1.0, -1.0 * (0.0351 + EREST_ACT), -0.005,
    250, 0.0, 0.0, -1.0 * (0.02 + EREST_ACT), 0.04,
    3000, -0.1, 0.05 ] )
  xgate.setupAlpha( xA )
```

Tabchannel K(A) Hippocampal cell channel

12b ⟨*Tabchannel K-A Hippocampal cell channell* 12b⟩≡ (6a)

```
def make_K_A():
  if moose.exists( 'K_A' ):
    return
  K_A = moose.HHChannel( 'K_A' )
  K_A.Ek = EK        #  V
  K_A.Gbar = 50 * SOMA_A  #  S
  K_A.Gk = 0        #  S
  K_A.Xpower = 1
  K_A.Ypower = 1
  K_A.Zpower = 0

  xgate = moose.element( 'K_A/gateX' )
  xA = numpy.array( [ 20e3 * (0.0131 + EREST_ACT),
    -20e3, -1.0, -1.0 * (0.0131 + EREST_ACT), -0.01,
    -17.5e3 * (0.0401 + EREST_ACT),
    17.5e3, -1.0, -1.0 * (0.0401 + EREST_ACT), 0.01,
    3000, -0.1, 0.05 ] )
  xgate.setupAlpha( xA )

  ygate = moose.element( 'K_A/gateY' )
  yA = numpy.array( [ 1.6, 0.0, 0.0, 0.013 - EREST_ACT, 0.018,
    50.0, 0.0, 1.0, -1.0 * (0.0101 + EREST_ACT), -0.005,
    3000, -0.1, 0.05 ] )
  ygate.setupAlpha( yA )
```

SynChan: Glu receptor

12c ⟨*Glu receptor* 12c⟩≡ (5a)

```
def make_glu():
  if moose.exists( 'glu' ):
    return
  glu = moose.SynChan( 'glu' )
  glu.Ek = 0.0
  glu.tau1 = 2.0e-3
  glu.tau2 = 9.0e-3
  glu.Gbar = 40 * SOMA_A
```

SynChan: NMDA receptor

13a ⟨*NMDA receptor* 13a⟩≡ (5a) 13b▷

```
def make_NMDA():
  if moose.exists( 'NMDA' ):
    return
  NMDA = moose.SynChan( 'NMDA' )
  NMDA.Ek = 0.0
  NMDA.tau1 = 20.0e-3
  NMDA.tau2 = 20.0e-3
  NMDA.Gbar = 5 * SOMA_A

  block = moose.MgBlock( '/library/NMDA/block' )
  block.CMg = 1.2    #  [Mg] in mM
  block.Zk = 2
  block.KMg_A = 1.0/0.28
  block.KMg_B = 1.0/62

  moose.connect( NMDA, 'channelOut', block, 'origChannel', 'OneToOne' )
  addmsg1 = moose.Mstring( '/library/NMDA/addmsg1' )
  addmsg1.value = '.. channel  ./block  channel'
  #Here we want to also tell the cell reader to _remove_ the original
  #Gk, Ek term going from the channel to the compartment, as this is
  # now handled by the MgBlock.
  #addmsg2 = moose.Mstring( 'NMDA/addmsg2'
  #addmsg2.value = 'DropMsg  ..  channel'
  addmsg3 = moose.Mstring( '/library/NMDA/addmsg3' )
  addmsg3.value = '.. VmOut  .  Vm'
```

The `Ca_NMDA` channel is a subset of the NMDA channel that carries Ca. It is identical to above, except that the Ek for Ca is much higher: 0.08 V from the consts at the top of this file. This is about the reversal potl for 1 uM `Ca_in`, 2 mM out. Also we do not want this channel to contribute to the current, which is already accounted for in the main channel. So there is no CHANNEL message to the parent compartment. I would like to have used the Nernst to do the Ca potential, and Synchans now take Ek messages but I haven't yet used this.

13b ⟨*NMDA receptor* 13a⟩+≡ (5a) ◁13a

```
def make_Ca_NMDA():
  if moose.exists( 'Ca_NMDA' ):
    return
  Ca_NMDA = moose.SynChan( 'Ca_NMDA' )
  Ca_NMDA.Ek = ECA
  Ca_NMDA.tau1 = 20.0e-3
  Ca_NMDA.tau2 = 20.0e-3
  Ca_NMDA.Gbar = 5 * SOMA_A

  block = moose.MgBlock( '/library/Ca_NMDA/block' )
  block.CMg = 1.2    #  [Mg] in mM
  block.Zk = 2
  block.KMg_A = 1.0/0.28
  block.KMg_B = 1.0/62

  moose.connect( Ca_NMDA, 'channelOut', block, 'origChannel', 'OneToOne' )
  addmsg1 = moose.Mstring( '/library/Ca_NMDA/addmsg1' )
  addmsg1.value = '.. VmOut  ./block  Vm'
  addmsg2 = moose.Mstring( '/library/Ca_NMDA/addmsg2' )
  addmsg2.value = './block  IkOut ../NMDA_Ca_conc current'
  # The original model has the Ca current also coming here.
```

Ca pool for influx through `Ca_NMDA` This pool used to set up Ca info coming to it. Now we insist that the originating channel should specify the deferred message.

14a ⟨*Functions to maniputate property of channels* 8⟩+≡ (5a) ◁8

```
def make_NMDA_Ca_conc():
  if moose.exists( 'NMDA_Ca_conc' ):
    return
  NMDA_Ca_conc = moose.CaConc( 'NMDA_Ca_conc' )
  NMDA_Ca_conc.tau = 0.004     # sec. Faster in spine than dend
  NMDA_Ca_conc.B = 17.402e12  # overridden by cellreader.
  NMDA_Ca_conc.Ca_base = 0.0
```

Spike Detector

14b ⟨*Spike detector* 14b⟩≡ (5a)

```
def make_axon():
  if moose.exists( 'axon' ):
    return
  axon = moose.SpikeGen( 'axon' )
  axon.threshold = -40e-3         # V
  axon.abs_refract = 10e-3        # sec
```

3.1 Database to keep the XML models

We use sqlite3 database. Let import it and add a section in our class to handle this database.

14c ⟨*Local imports* 14c⟩≡ (4a) 18a▷

```
import sqlite3 as sql
```

14d ⟨*methods* 14d⟩≡ (4b)

⟨*helper functions* 18b⟩
⟨*methods to deal with database* 15b⟩

And lets open a database and initialize it. And add code to clean up the connection before exiting the class.

14e ⟨*initialize members* 14e⟩≡ (4b)

```
self.dbdir = 'db'
self.dbname = 'models.db'
self.dbpath = os.path.join(self.dbdir, self.dbname)
self.includedFiles = list()

if not os.path.exists(self.dbpath) :
  try :
    os.makedirs(self.dbdir)
  except Exception as e :
    debug.printDebug("ERROR"
        , "Faild to create directory {0} with error {1}".format(self.dbdir, e))
    sys.exit(0)
#self.conn = sql.connect(self.dbpath)
self.conn = sql.connect(":memory:")
self.cursor = self.conn.cursor()
```

14f ⟨*clean up the mess* 14f⟩≡ (4b)

```
self.cursor.commit()
self.conn.close()
sys.exit()
```

3.2 Populate database

We have the parsed models and we would be searching them extensively when combining them together to map onto moose. Populate the sqlite3 database such that we can query it easily.

15a ⟨*flow of executation* 15a⟩≡ (4b)
⟨*initialize database* 17c⟩
⟨*populate database with models* 17b⟩
⟨*build queries from adaptorML* (never defined)⟩
⟨*run queries and generate moose scripts* (never defined)⟩

Populating database Should be directly translated XML to `sqlite3`? No, that would defeat the purpose of using sqlite3 in the middle. We must transform the XML as much as we can to create a well-defined database which we can simply query and build moose scripts. Let's me describe the flow. ETL stands for standard practise of **E**xtract, **T**ransform, and **L**oad.

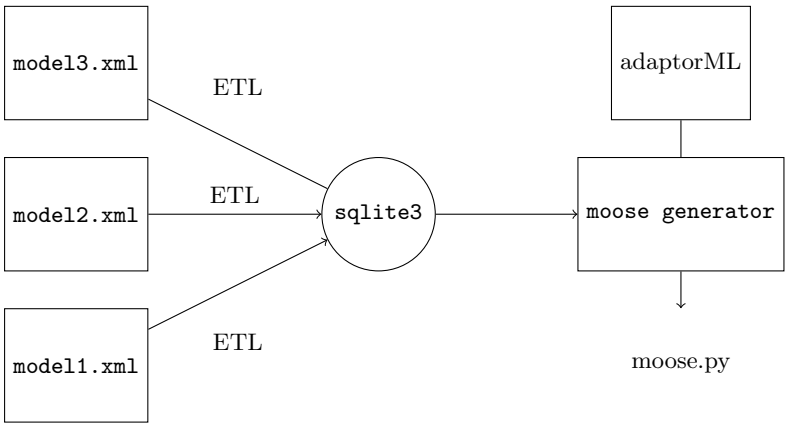

Figure 1: Flow of multi-scale modelling. ETL stands for **E**xtract **T**ransform and **L**oad.

Now we are ready to do ELT. We are interested in some of XML elements given in models which we transform (in Python) and load into database. To transform, we extract a part of required XML and build a query which we run to populate the database.

3.3 Transform XML to sqlite3 queries

15b ⟨*methods to deal with database* 15b⟩≡ (14d) 17d▷
⟨*segmentToQuery* 16⟩
⟨*segmentGroupToQuery* 17a⟩

Insert a segment into database Element `segment` has the following schema in neuroML. While attribute `id` is required, `name` is optional. Let's build a query to insert a `segment` into database.

```
<xs:complexType name="Segment">
    <xs:complexContent>
        <xs:extension base="BaseWithoutId">  <!-- Dont want to allow string value as with NmlId,
             want just non negative integer-->

            <xs:sequence>
                <xs:element name="parent" type="SegmentParent" minOccurs="0"/>
                <xs:element name="proximal" type="Point3DWithDiam" minOccurs="0"/>
                <xs:element name="distal" type="Point3DWithDiam" minOccurs="1"/>
            </xs:sequence>

            <xs:attribute name="id" type="SegmentId" use="required"/>
            <xs:attribute name="name" type="xs:string" use="optional"/>

        </xs:extension>
    </xs:complexContent>

</xs:complexType>
```

Listing 1: XML schema of element `segment`

16 ⟨*segmentToQuery* 16⟩≡ (15b 18d) 19▷

```
def segmentToQuery(self, segXML) :
    values = dict()
    for k in segXML.keys() :
      values[k] = segXML.get(k)
    # get parent, distal and proximal.
    for elem in segXML :
      if self.isTag('parent', elem) :
        values['parent'] = elem.get('segment')
        if elem.get('fractionAlong') :
          values['fractionAlong'] =  elem.get('fractionAlong')
      elif self.isTag('proximal', elem) :
        for k in elem.keys() :
          values["proximal_"+k.strip()] = elem.get(k)
      elif self.isTag('distal', elem) :
        for k in elem.keys() :
          values["distal_"+k.strip()] = elem.get(k)
    # build query
    query = "INSERT OR REPLACE INTO "+ self.segmentTable + ' ('
    query += ",".join(values.keys())
    query += ') VALUES (' + ", ".join(["'"+v.strip()+"'" for v in values.values()]) + ')'
    return query
```

Segment-group to query Schema for this element is following.

```
<xs:complexType name="SegmentGroup">


    <!--.... dendrite_group  ...-->
    <xs:complexContent>
        <xs:extension base="Base">

            <xs:sequence>
                <xs:element name="member" type="Member" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="include" type="Include" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="path" type="Path" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="subTree" type="SubTree" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="inhomogeneousParam" type="InhomogeneousParam" minOccurs="0"
                    maxOccurs="unbounded"/>
            </xs:sequence>

        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

Listing 2: XML schema of element `segmentGroup`

17a ⟨*segmentGroupToQuery* 17a⟩≡ (15b 18d) 20▷

```
def segmentGroupToQuery(self, segGrpXML) :
    query = "INSERT OR REPLACE INTO segment_prop (id) VALUES "
    groupId = segGrpXML.get('id')
    query += " ("+"'"+groupId.strip()+"')"
    self.executeQuery(query)
    for k in segGrpXML :
      query = "UPDATE OR REPLACE " + self.segmentTable + " SET "
      query += "segment_group='"+groupId+"' WHERE id='" + k.get('segment') + "'"
      self.executeQuery(query)

      if 'include' in k.keys() :
        debug.printDebug("WARN", "Element include is not implemented")
      if 'path' in k.keys() :
        debug.printDebug("WARN", "Element path is not implemented")
      if 'subTree' in k.keys() :
        debug.printDebug("WARN", "Element subTree is not implemented")
      if 'inhomogeneousParam' in k.keys() :
        debug.printDebug("WARN", "Element inhomogeneousParam is not implemented")
```

3.4 Extract, Transform, and Load

17b ⟨*populate database with models* 17b⟩≡ (15a)

```
for xml in self.xmlDict :
  xmlRootNodeList = self.xmlDict[xml]
  for xmlRootNode in xmlRootNodeList :
    self.extractTransformLoad(xml, xmlRootNode)
```

Now the hard part of populating the database starts. We need to initialize the database first before we can use it. Its a serious task. Let's initialize tables first and document them. The description of database is available in section 4.

17c ⟨*initialize database* 17c⟩≡ (15a)

```
self.initDB()
```

17d ⟨*methods to deal with database* 15b⟩+≡ (14d) ◁15b 18c▷

```
def extractTransformLoad(self, modelType, xmlRootNode) :
  if modelType == 'nml' :
    self.etlNMLModel(xmlRootNode)
  else :
      pass
```

ETL a NML model ETL an neuroML model. Model specified in `neuroML` other files too. As soon as we get to know the names of these files, we must demand them from the user after parsing of `neuroML` model is over. If these files are given from command line, ETL them silently, else demand these files from command-line. We'd need `re` library to for pattern matching.

18a ⟨*Local imports* 14c⟩+≡ (4a) ◁14c

```
import re
```

Since most of the tags have namespace in them, a function to search a tag would be handy.

18b ⟨*helper functions* 18b⟩≡ (14d 23b) 24b▷

```
def isTag(self, tagName, nmElem) :
  if re.search(r'^\{(?P<namesapce>[^}^{]+)\}'+tagName+'\s*$', nmElem.tag) :
    return True
  else :
    return False
```

18c ⟨*methods to deal with database* 15b⟩+≡ (14d) ◁17d 18d▷

```
def etlNMLModel(self, nmlTree) :
  debug.printDebug("STEP", "ETLing a nml model")
  nmlRootNode = nmlTree.getroot()
  for c in  [ child  for child in nmlRootNode if type(child.tag) == str] :
    if self.isTag('include', c) :
      self.includedFiles.append(c.get('href'))
    elif self.isTag('cell', c) :
      self.insertCellIntoDB(c)
    else :
      debug.printDebug("WARN", "{0} is not implemented yet.".format(child.tag))

# Function to insert a cell into database
def insertCellIntoDB(self, cell) :
  for c in cell.iterchildren(tag=etree.Element) :
    if self.isTag('morphology', c) :
      for elem in c.iterchildren(tag=etree.Element) :
        if self.isTag('segment', elem) :
           segmentDict = dict()
           debug.printDebug("SEGMENT", "Processing {0}".format(elem.tag))
           self.executeQuery(self.segmentToQuery(elem))
        elif self.isTag('segmentGroup', elem) :
          debug.printDebug("SEGGRP", "Segment group");
          print self.segmentGroupToQuery(elem)
        else :
          debug.printDebug("INFO" , "This element {0} is not supported".format(elem))
    else :
      debug.printDebug("WARN", "{0} not implemented".format(c.tag))
```

3.5 Transform XML to sqlite3 queries

18d ⟨*methods to deal with database* 15b⟩+≡ (14d) ◁18c 21▷

⟨*segmentToQuery* 16⟩
⟨*segmentGroupToQuery* 17a⟩

Insert a segment into database Element `segment` has the following schema in neuroML. While attribute `id` is required, `name` is optional. Let's build a query to insert a `segment` into database.

```
    <xs:complexType name="Segment">
        <xs:complexContent>
            <xs:extension base="BaseWithoutId">  <!-- Dont want to allow string value as with NmlId,
                 want just non negative integer-->

                <xs:sequence>
                    <xs:element name="parent" type="SegmentParent" minOccurs="0"/>
                    <xs:element name="proximal" type="Point3DWithDiam" minOccurs="0"/>
                    <xs:element name="distal" type="Point3DWithDiam" minOccurs="1"/>
                </xs:sequence>

                <xs:attribute name="id" type="SegmentId" use="required"/>
                <xs:attribute name="name" type="xs:string" use="optional"/>

            </xs:extension>
        </xs:complexContent>

    </xs:complexType>
```

Listing 3: XML schema of element `segment`

19 ⟨*segmentToQuery* 16⟩+≡ (15b 18d) ◁16

```
  def segmentToQuery(self, segXML) :
      values = dict()
      for k in segXML.keys() :
        values[k] = segXML.get(k)
      # get parent, distal and proximal.
      for elem in segXML :
        if self.isTag('parent', elem) :
          values['parent'] = elem.get('segment')
          if elem.get('fractionAlong') :
            values['fractionAlong'] =  elem.get('fractionAlong')
        elif self.isTag('proximal', elem) :
          for k in elem.keys() :
            values["proximal_"+k.strip()] = elem.get(k)
        elif self.isTag('distal', elem) :
          for k in elem.keys() :
            values["distal_"+k.strip()] = elem.get(k)
      # build query
      query = "INSERT OR REPLACE INTO "+ self.segmentTable + ' ('
      query += ",".join(values.keys())
      query += ') VALUES (' + ", ".join(["'"+v.strip()+"'" for v in values.values()]) + ')'
      return query
```

Segment-group to query Schema for this element is following.

```
<xs:complexType name="SegmentGroup">


    <!--.... dendrite_group  ...-->
    <xs:complexContent>
        <xs:extension base="Base">

            <xs:sequence>
                <xs:element name="member" type="Member" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="include" type="Include" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="path" type="Path" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="subTree" type="SubTree" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="inhomogeneousParam" type="InhomogeneousParam" minOccurs="0"
                    maxOccurs="unbounded"/>
            </xs:sequence>

        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

Listing 4: XML schema of element `segmentGroup`

20 ⟨*segmentGroupToQuery* 17a⟩+≡ (15b 18d) ◁17a

```
def segmentGroupToQuery(self, segGrpXML) :
    query = "INSERT OR REPLACE INTO segment_prop (id) VALUES "
    groupId = segGrpXML.get('id')
    query += " ("+"'"+groupId.strip()+"')"
    self.executeQuery(query)
    for k in segGrpXML :
      query = "UPDATE OR REPLACE " + self.segmentTable + " SET "
      query += "segment_group='"+groupId+"' WHERE id='" + k.get('segment') + "'"
      self.executeQuery(query)

      if 'include' in k.keys() :
        debug.printDebug("WARN", "Element include is not implemented")
      if 'path' in k.keys() :
        debug.printDebug("WARN", "Element path is not implemented")
      if 'subTree' in k.keys() :
        debug.printDebug("WARN", "Element subTree is not implemented")
      if 'inhomogeneousParam' in k.keys() :
        debug.printDebug("WARN", "Element inhomogeneousParam is not implemented")
```

4 Describe and initialize database

We have four tables in our database.

`segment`	One entry for each segment along with its chemical and electrical properties. Since `parent` is specified for each segment, we can figure out the topology from this table only.
`cells`	Types of cell available in this network and their properties.
`mapping`	Map a segment to another segment. Specify the relation in terms of $lhs = rhsExpr$.

See the chunk `methods to deal with database` for table documentation. **All units must be in S.I.**.

21 ⟨*methods to deal with database* 15b⟩+≡ (14d) ◁18d 22▷

```
def initDB(self, dropOldTables = False) :
  self.segmentTable = 'segment'
  self.cellTable = 'cell'
  self.mappingTable = 'mapping'

  # Table describing segments
  query = '''DROP TABLE IF EXISTS {0}'''.format(self.segmentTable)
  if dropOldTables :
    self.cursor.execute(query)
  query = '''CREATE TABLE IF NOT EXISTS {0}
    (id INTEGER PRIMARY KEY ASC
    , name VARCHAR
    , parent INTEGER
    , fractionAlong REAL default '0.0'
    , proximal_x REAL
    , proximal_y REAL
    , proximal_z REAL
    , proximal_diameter REAL
    , distal_x REAL
    , distal_y REAL
    , distal_z REAL
    , distal_diameter REAL
    , x REAL
    , y REAL
    , z REAL
    , segment_group VARCHAR
    , remark TEXT
    , FOREIGN KEY (segment_group) REFERENCES segment_group(id)
    )'''
  self.cursor.execute(query.format(self.segmentTable))

  query = '''CREATE TABLE IF NOT EXISTS segment_prop
    (id VARCHAR
    , key VARCHAR
    , value VARCHAR
    , PRIMARY KEY (id, key, value))'''
  self.cursor.execute(query)


  # Table describing cells in network.
  query = 'DROP TABLE IF EXISTS {0}'.format(self.cellTable)
  if dropOldTables :
    self.cursor.execute(query)

  query = '''CREATE TABLE IF NOT EXISTS {0}
    (type VARCHAR PRIMARY KEY -- Type of cell
    , leakReversal REAL       -- leakReversal potential
    , threshold REAL          -- Threshold voltage
```

```
    , reset REAL
    , tau REAL
    , refract REAL
    , capacitance REAL
    , leakConductance REAL
    , a REAL                  -- Izhikenvich Cell model
    , b REAL                  -- Izhikenvich cell model
    , c REAL                  -- Izhikenvich cell model
    , d REAL                  -- Izhikenvich cell model
    , gL REAL
    , EL REAL
    , VT REAL
    , delT REAL
    , tauw REAL
    , Idel REAL
    , Idur REAL
    )'''
  self.cursor.execute(query.format(self.cellTable))

  # Table specifying mapping.
  query = 'DROP TABLE IF EXISTS {0}'.format(self.mappingTable)
  if dropOldTables :
    self.cursor.execute(query)

  query = '''CREATE TABLE IF NOT EXISTS {0}
    (id INTEGER
    , _from INTEGER, fromType VARCHAR
    , _to INTEGER,  toType VARCHAR
    , lhsVar VARCHAR
    , rhsExpr VARCHAR
    , comment TEXT
    , PRIMARY KEY (_from, _to, lhsVar)
    )'''
  self.cursor.execute(query.format(self.mappingTable))
  self.conn.commit()
```

Execute query This method make sure to roll-back if the query is not executed successfully.

22 ⟨*methods to deal with database* 15b⟩+≡ (14d) ◁21

```
def executeQuery(self, query) :
  with self.conn :
    try :
      self.cursor.execute(query)
    except Exception as e :
      debug.printDebug("ERR", "Failed to execute query with error {0}".format(e))
      print("+ QUERY: {0}".format(query))
```

5 AdaptorML for Moose

This adaptor example was writeen by an intern during GSoC.

.

Wr
tor

23a ⟨*adaptor.xml* 23a⟩≡

```
<?xml version="1.0"?>
  <adapterML>
    <listOfAdaptors>
    <adaptor name="adaptK" id="/n/chem/neuroMesh/adaptK" scale="0.05">
      <inElement name="chemK" id="/n/chem/neuroMesh/kChan" field="get_conc"
            adapt_type="requestField" mode="OneToAll"/>
      <outElement name="elecK" id="/n/elec/compt/K" field="set_Gbar"
            adapt_type="outputSrc" mode="OneToAll"/>
    </adaptor>
    <adaptor name="adaptCa" id="/n/chem/neuroMesh/adaptCa" outputOffset="0.0001" scale="0.05">
      <inElement name="elecCa" id="/n/elec/compt/ca" field="concOut"
            adapt_type="input" mode="OneToAll"/>
      <outElement name="chemCa" id="/n/chem/neuroMesh/Ca" field="set_conc"
            adapt_type="outputSrc" mode="OneToAll"/>
    </adaptor>
  </listOfAdaptors>
</adapterML>
```

6 XML parser

This section deals with parser of XML models. In `multiscale.nw`, we call function `parseModels` belonging to this module. This function receives its arguments, a dictionary of paths of XML models. These file-paths are already verified; they exists and are readable. We now parse the XMLs and return a dictionary, with keys as path of XML models and value as top-most XML element.

Following captures what this module suppose to do.

23b ⟨*parser.py* 23b⟩≡

```
⟨Import 1⟩
import collections
⟨helper functions 18b⟩
⟨function parseModels for parsing models 24a⟩
```

Parse models Function `parseModels` parses the model, and it creates a dictionary containing root elements of parsed files to be returned. This function first read the file and validate it with a given schema. Validation can be turned on/off by setting the optional argument `validate` to `False`.

24a ⟨*function* `parseModels` *for parsing models* 24a⟩≡ (23b)

```
def parseModels(commandLineArgs, validate=False) :
  xmlRootElemDict = collections.defaultdict(list)
  models = vars(commandLineArgs)
  for model in models :
    if models[model] :
      for modelPath in models[model] :
        debug.printDebug("INFO", "Parsing {0}".format(models[model]))
        if validate :
          # parse model and valid it with schama
          modelXMLRootElem = parseAndValidateWithSchema(model, modelPath)
        else :
          # Simple parse the model without validating it with schema.
          modelXMLRootElem = parseWithoutValidation(model, modelPath)
        if modelXMLRootElem :
          xmlRootElemDict[model].append(modelXMLRootElem)
  return xmlRootElemDict
```

Validating with schema We need two helper functions, `parseAndValidateWithSchema` to parse a given XML when a schema is available and `parseWithoutValidation` validation is off **schema is not available**. Ideally, schema should be provided as an argument to this module, but we can fix their location. Folder `./moose_xml` contains the schema we are going to build and use in this application. Its path is `./moose_xml/moose.xsd`.

Wri
mo
sch
./m
pat

24b ⟨*helper functions* 18b⟩+≡ (14d 23b) ◁18b 25a▷

```
def parseAndValidateWithSchema(modelName, modelPath) :

    prefixPath = ''
    if modelName == 'xml' :
      schemaPath = os.path.join(prefixPath, 'moose_xml/moose.xsd')

    try :
      schemaH = open(schemaPath, "r")
      schemaText = schemaH.read()
      schemaH.close()
    except Exception as e :
      debug.printDebug("WARN", "Error reading schema for validation."+
        " Falling back to validation-disabled parser."
        + " Failed with error {0}".format(e))
      return parseWithoutValidation(modelName, modelPath)
    # Now we have the schema text
    schema = etree.XMLSchema(etree.XML(schemaText))
    xmlParser = etree.XMLParser(schema=schema, remove_comments=True)
    with open(modelPath, "r") as xmlTextFile :
        return etree.parse(xmlTextFile, xmlParser)
```

25a ⟨*helper functions* 18b⟩+≡ (14d 23b) ◁24b

```
def parseWithoutValidation(modelName, modelPath) :
    xmlParser = etree.XMLParser(remove_comments=True)
    try :
      xmlRootElem = etree.parse(modelPath, xmlParser)
    except Exception as e :
      debug.printDebug("ERROR", "Parsing failed. {0}".format(e))
      return
    return xmlRootElem
```

This ends our parser module and we can now go back to section 2 to do some real programming related stuff.

7 Debug module, print debugging messages

Different type of messages are printed in different colors.

25b ⟨*debug.py* 25b⟩≡

```
HEADER = '\033[95m'
OKBLUE = '\033[94m'
OKGREEN = '\033[92m'
WARNING = '\033[93m'
ERR = '\033[31m'
ENDC = '\033[0m'
RED = ERR
WARN = WARNING
INFO = OKBLUE
TODO = OKGREEN
DEBUG = HEADER
ERROR = ERR

prefix = dict(
    ERR = ERR
    , ERROR = ERR
    , WARN = WARN
    , FATAL = ERR
    , INFO = INFO
    , TODO = TODO
    , NOTE = HEADER
    , DEBUG = DEBUG
    )

def colored(msg, label) :
    """
    Return a colored string. Formatting is optional.
    """
    global prefix
    if label in prefix :
        color = prefix[label]
    else :
        color = ""
    return "[{0}] {1} {2}".format(label, color+msg, ENDC)

def printDebug(label, msg):
    print(colored(msg, label))
```

⟨*adaptor.xml* 23a⟩
⟨*argument parser* 2b⟩
⟨*build queries from adaptorML* (never defined)⟩
⟨*Ca-dependent K-AHP channel* 9⟩
⟨*Ca-dependent K-C channel* 10⟩
⟨*clean up the mess* 14f⟩
⟨*debug.py* 25b⟩
⟨*Define constants* 5b⟩
⟨*Definition of class* `Multiscale` 4b⟩
⟨*flow of executation* 15a⟩
⟨*function* `parseModels` *for parsing models* 24a⟩
⟨*functions in main* 3a⟩
⟨*Functions to create channels* 6a⟩
⟨*Functions to maniputate property of channels* 8⟩
⟨*Glu receptor* 12c⟩
⟨*hand over control to class in multiscale module* 3c⟩
⟨*helper functions* 18b⟩
⟨*Import* 1⟩
⟨*initialize database* 17c⟩
⟨*initialize members* 14e⟩
⟨*Local imports* 14c⟩
⟨*main.py* 2a⟩
⟨*methods* 14d⟩
⟨*methods to deal with database* 15b⟩
⟨*multiscale* 4a⟩
⟨*NMDA receptor* 13a⟩
⟨*Ordinary Ca channel* 6b⟩
⟨*parse xml models and handover control to main class* 3b⟩
⟨*parser.py* 23b⟩
⟨*populate database with models* 17b⟩
⟨*proto.py* 5a⟩
⟨*put information into cell-reader* 7c⟩
⟨*run queries and generate moose scripts* (never defined)⟩
⟨*segmentGroupToQuery* 17a⟩
⟨*segmentToQuery* 16⟩
⟨*setup X-gate using Traub model* 7a⟩
⟨*setup Y-gate using Traub model* 7b⟩
⟨*Spike detector* 14b⟩
⟨*Tabchannel K-A Hippocampal cell channell* 12b⟩
⟨*Tabchannel K-DR Hippocampal cell channel* 12a⟩
⟨*Tabchannel Na Hippocampal cell channel* 11⟩
⟨*tests* (never defined)⟩

Todo list