NEURON Extension to NMODL

This section describes the special NEURON block that has been added to the standard model description language in order to allow translation of a model into a form suitable for linking with NEURON.

The keyword NEURON introduces a special block which contains statements that tell NMODL how to organize the variables for access at the NEURON user level. It declares:

- Which names are to be treated as range variables.
- Which names are to be treated as global variables.
- The names of all the ions used in the model and how the corresponding concentrations, current, and reversal potential are to be treated.
- The suffix to be used for all variables in the model so that they do not conflict with variables in other models.
- Whether the model is for a point process such as a synapse or a distributed process with density along an entire section such as a channel density.
- Which names will be connected to external variables. (See “Importing variables from other mechanisms”.)

The syntax is (each statement can occur none or more times) :

Neuron

Description:

```
NEURON{
   SUFFIX ...
   RANGE ...
   GLOBAL ...
   NONSPECIFIC_CURRENT ...
   USEION ... READ ... WRITE ... VALENCE real
   POINT_PROCESS ...
   POINTER ...
   EXTERNAL ...
}
```

Suffix

Description:

The suffix, “`_name`” is appended to all variables, functions, and procedures that are accessible from the user level of NEURON. If the `SUFFIX` statement is absent, the file name is used as the suffix (with the addition of an underscore character). If there is a Point Processes and Artificial Cells statement, that name is used as the suffix. Suffixes prevent overloading of names at the user level of NEURON. At some point in the future I may add something similar to the access

statement which will allow the omission of the suffix for a specified mechanism. Note that suffixes are not used within the model description file itself. If the SUFFIX *name* is the word, "`nothing`", then no suffix is used for variables, functions, and procedures explicitly declared in the `.mod` file. However, the mechanism name will be the base file name. This is useful if you know that no conflict of names will exist or if the `.mod` file is primarily used to create functions callable from NEURON by the user and you want to specify those function names exactly.

Range

Description:

These names will be become range variables. Do not add suffixes here. The names should also be declared in the normal `PARAMETER` or `ASSIGNED` statement outside of the `NEURON` block. Parameters that do not appear in a `NEURON RANGE` statement will become global variables. Assigned variables that do not appear in this statement or in the `NEURON GLOBAL` statement will be hidden from the user. When a mechanism is inserted in a section, the values of these range variables are set to the values specified in the normal `PARAMETER` statement outside the `NEURON` block.

Global

Description:

These names, which should be declared elsewhere as `ASSIGNED` or `PARAMETER` variables, become global variables instead of range variables. Notice here that the default for a `PARAMETER` variable is to become a global variable whereas the default for an `ASSIGNED` variable is to become hidden at the user level.

Nonspecific Current

Description:

This signifies that we are calculating local currents which get added to the total membrane current but will not contribute to any particular ionic concentration. This current should be assigned a value after any `SOLVE` statement but before the end of the `BREAKPOINT` block. This name will be hidden at the user level unless it appears in a `NEURON RANGE` statement.

Useion

Description:

This statement declares that a specific ionic species will be used within this model. The built-in HH channel uses the ions `na` and `k`. Different models which

deal with the same ionic species should use the same names so that total concentrations and currents can be computed consistently. The ion, `Na`, is different from `na`. The example models using calcium call it, `ca`. If an ion is declared, suppose it is called, `ion`, then a separate mechanism is internally created within NEURON, denoted by `ion`, and automatically inserted whenever the “using” mechanism is inserted. The variables of the mechanism called `ion` are outward total current carried by this ion, `iion`; internal and external concentrations of this ion, `ioni` and `iono`; and reversal potential of this ion, `eion`. These ion range variables do NOT have suffixes. Prior to 9/94 the reversal potential was not automatically calculated from the Nernst equation but, if it was *used* it had to be set by the user or by an assignment in some mechanism (normally the Nernst equation). The usage of ionic concentrations and reversal potential has been changed to more naturally reflect their physiological meaning while remaining reasonably efficient computationally.

The new method governs the behaviour of the reversal potential and concentrations with respect to their treatment by the GUI (whether they appear in PARAMETER, ASSIGNED, or STATE panels; indeed, whether they appear at all in these panels) and when the reversal potential is automatically computed from the concentrations using the Nernst equation. The decision about what style to use happens on a per section basis and is determined by the set of mechanisms inserted within the section. The rules are defined in the reference to the function ion_style(). Three cases are noteworthy.

Read

Assume only one model is inserted in a section.

```
        USEION ca READ eca
```

Then eca will be treated as a PARAMETER and cai/cao will not appear in the parameter panels created by the gui.

Now insert another model at the same section that has

```
        USEION ca READ cai, cao
```

Then 1) eca will be “promoted” to an ASSIGNED variable, 2) cai/cao will be treated as constant PARAMETER’s, and 3) eca will be computed from the Nernst equation when finitialize() is called.

Write

Lastly, insert a final model at the same location in addition to the first two.

```
        USEION ca WRITE cai, cao
```

Then eca will still be treated as an ASSIGNED variable but will be computed not only by finitialize but on every call to fadvance(). Also cai/cao will be initialized to the global variables cai0_ca_ion and cao0_ca_ion respectively and treated as STATE's by the graphical interface.

The idea is for the system to automatically choose a style which is sensible in terms of dependence of reversal potential on concentration and remains efficient.

Since the nernst equation is now automatically used as needed it is necessary to supply the valence (charge carried by the ion) except for the privileged ions: na, k, ca which have the VALENCE 1, 1, 2 respectively.

Only the ion names `na`, `k`, and `ca` are initialized to a physiologically meaningful value — and those may not be right for your purposes. Concentrations and reversal potentials should be considered parameters unless explicitly calculated by some mechanism.

Valence

The `READ` list of a `USEION` specifies those ionic variables which will be used to calculate other values but is not calculated itself. The `WRITE` list of a `USEION` specifies those ionic variables which will be calculated within this mechanism. Normally, a channel will read the concentration or reversal potential variables and write a current. A mechanism that calculates concentrations will normally read a current and write the intracellular and/or extracellular; it is no longer necessary to ever write the reversal potential as that will be automatically computed via the nernst equation. It usually does not make sense to both read and write the same ionic concentrations. It is possible to READ and WRITE currents. One can imagine, a large calcium model which would `WRITE` all the ion variables (including current) and READ the ion current. And one can imagine models which `READ` some ion variables and do not `WRITE` any. It would be an error if more than one mechanism at the same location tried to WRITE the same concentration.

A bit of implementation specific discussion may be in order here. All the statements after the `SOLVE` statement in the `BREAKPOINT` block are collected to form a function which is called during the construction of the charge conservation matrix equation. This function is called several times in order to compute the current and conductance to be added into the matrix equation. This function is never called if you are not writing any current. The `SOLVE` statement is executed after the new voltages have been

computed in order to integrate the states over the time step, `dt`. Local static variables get appropriate copies of the proper ion variables for use in the mechanism. Ion variables get updated on exit from these functions such that WRITE currents are added to ion currents.

Point_Process

Description:

The `READ` list of a `USEION` specifies those ionic variables which will be used to calculate other values but is not calculated itself. The `WRITE` list of a `USEION` specifies those ionic variables which will be calculated within this mechanism. Normally, a channel will read the concentration or reversal potential variables and write a current. A mechanism that calculates concentrations will normally read a current and write the intracellular and/or extracellular; it is no longer necessary to ever write the reversal potential as that will be automatically computed via the nernst equation. It usually does not make sense to both read and write the same ionic concentrations. It is possible to READ and WRITE currents. One can imagine, a large calcium model which would `WRITE` all the ion variables (including current) and READ the ion current. And one can imagine models which `READ` some ion variables and do not `WRITE` any. It would be an error if more than one mechanism at the same location tried to WRITE the same concentration.

A bit of implementation specific discussion may be in order here. All the statements after the SOLVE statement in the BREAKPOINT block are collected to form a function which is called during the construction of the charge conservation matrix equation. This function is called several times in order to compute the current and conductance to be added into the matrix equation. This function is never called if you are not writing any current. The SOLVE statement is executed after the new voltages have been computed in order to integrate the states over the time step, `dt`. Local static variables get appropriate copies of the proper ion variables for use in the mechanism. Ion variables get updated on exit from these functions such that WRITE currents are added to ion currents.

Pointer

Description:

These names are pointer references to variables outside the model. They should be declared in the body of the description as normal variables with units and are used exactly like normal variables. The user is responsible for setting these pointer variables to actual variables at the hoc interpreter level. Actual variables are normal variables in other mechanisms, membrane potential, or any hoc variable. See below for how this connection is made. If a POINTER variable is

ever used without being set to the address of an actual variable, NEURON may crash with a memory reference error, or worse, produce wrong results. Unfortunately the errors that arise can be quite subtle. For example, if you set a POINTER correctly to a mechanism variable in section a. And then change the number of segments in section a, the POINTER will be invalid because the memory used by section a is freed and might be used for a totally different purpose. It is up to the user to reconnect the POINTER to a valid actual variable.

External

Description:

These names, which should be declared elsewhere as ASSIGNED or PARAMETER variables allow global variables in other models or NEURON c files to be used in this model. That is, the definition of this variable must appear in some other file. Note that if the definition appeared in another mod file this name should explicitly contain the proper suffix of that model. You may also call functions from other models (but do not ignore the warning; make sure you declare them as

```
extern double fname_othermodelsuffix();
```

in a VERBATIM block and use them with the proper suffix.

Connecting Mechanisms Together

Occasionally mechanisms need information from other mechanisms which may be located elsewhere in the neuron. Connecting pre and post synaptic point mechanisms is an obvious example. In the same vein, it may be useful to call a function from hoc which modifies some mechanism variables at a specific location. (Normally, mechanism functions callable from HOC should not modify range variables since the function does not know where the mechanism data for a segment is located. Normally, the pointers are set when NEURON calls the BREAKPOINT block and the associated SOLVE blocks.)

One kind of connection between mechanisms at the same point is through ionic mechanisms invoked with the USEION statement. In fact this is entirely adequate for local communication although treating an arbitrary variable as an ionic concentration may be conceptually strained. However, it does not solve the problem of communication between mechanisms at different points.

Pointer-Communication

Description:

Basically what is needed is a way to implement the hoc statement

```
section1.var1_mech1(x1) =  section2.var2_mech2(x2)
```

efficiently from within a mechanism without having to explicitly connect them through assignment at the HOC level everytime the *var2* might change.

First of all, the variables which point to the values in some other mechanism are declared within the NEURON block via

```
NEURON {
   POINTER var1, var2, ...
}
```

These variables are used exactly like normal variables in the sense that they can be used on the left or right hand side of assignment statements and used as arguments in function calls. They can also be accessed from HOC just like normal variables. It is essential that the user set up the pointers to point to the correct variables. This is done by first making sure that the proper mechanisms are inserted into the sections and the proper point processes are actually "located" in a section. Then, at the hoc level each POINTER variable that exists should be set up via the command:

```
        setpointer pointer, variable
```

where pointer and variable have enough implicit/explicit information to determine their exact segment and mechanism location. For a continuous mechanism, this means the section and location information. For a point process it means the object. The variable may also be any hoc variable or voltage, *v*.

For example, consider a synapse which requires a presynaptic potential in order to calculate the amount of transmitter release. Assume the declaration in the presynaptic model

```
NEURON { POINTPROCESS Syn   POINTER vpre }
```

Then

```
objref syn
somedendrite {syn = new Syn(.8)}
setpointer syn.vpre, axon.v(1)
```

will allow the syn object to know the voltage at the distal end of the axon section. As a variation on that example, if one supposed that the synapse needed the presynaptic transmitter concentration (call it *tpre*) calculated from a point process model called "release" (with object reference *rel*, say) then the statement would be

```
setpointer syn.tpre, rel.AcH_release
```

The caveat is that tight coupling between states in different models may cause numerical instability. When this happens, merging models into one larger model may eliminate the instability.