Expanding NEURON's Repertoire of Mechanisms with NMODL

M.L. Hines[1] and N.T. Carnevale[2]

Departments of [1]Computer Science and [2]Psychology

Yale University

michael.hines@yale.edu

ted.carnevale@yale.edu

Short title: Expanding NEURON with NMODL

An extended preprint of:
Hines, M.L. and Carnevale, N.T.
Expanding NEURON's repertoire of mechanisms with NMODL
Neural Computation 12:839-851, 2000

CONTENTS

ABSTRACT

Neuronal function involves the interaction of electrical and chemical signals that are distributed in time and space. The mechanisms that generate these signals and regulate their interactions are marked by a rich diversity of properties that precludes a "one size fits all" approach to modeling. This paper shows how the model description language NMODL enables the neuronal simulation environment NEURON to accommodate these differences.

INTRODUCTION

Recently we described the core concepts and strategies that are responsible for much of the utility of NEURON as a tool for empirically-based neuronal modeling (Hines and Carnevale 1997). That paper focused on the strategy used in NEURON to deal with the problem of mapping a spatially distributed system into a discretized (compartmental) representation in a manner that ensures conceptual control while at the same time maintaining numeric accuracy and computational efficiency. Now we shift our attention to another important feature of NEURON: its special facility for expanding and customizing its library of biophysical mechanisms.

The need for this facility stems from the fact that experimentalists are applying an ever-growing armamentarium of techniques to dissect neuronal operation at the cellular level. There is a steady increase in the number of phenomena that are known to participate in electrical and chemical signaling and that are characterized well enough to support empirically-based simulations. Since the mechanisms that underlie these phenomena differ across neuronal cell class, developmental stage, and species (e.g. chapter 7 in (Johnston and Wu 1995); also see (McCormick 1998)), a simulator that is useful in research must provide a flexible and powerful means for incorporating new biophysical mechanisms in models. It must also help the user remain focused on the model instead of programming. Such a means is provided to the NEURON simulation environment by NMODL, a high-level language that was originally implemented for NEURON by Michael Hines and later extended by him and Upinder Bhalla to generate code suitable for linking with GENESIS (Wilson and Bower 1989).

A brief overview of how NMODL is used will clarify its underlying rationale. The first step is to write a text file (a "`mod` file") that describes a mechanism as a set of nonlinear algebraic equations, differential equations, or kinetic reaction schemes. The description employs a syntax that closely resembles familiar mathematical and chemical notation. This text is passed to a translator that converts each statement into many statements in C, automatically generating code that handles details such as mass balance for each ionic species and producing code suitable for each of NEURON's integration methods. The output of the translator is then compiled for computational efficiency. This achieves tremendous conceptual leverage and savings of effort not only because the high-level mechanism specification is much easier to understand and far more compact than the equivalent C code, but also because it spares the user from having to bother with low-level programming issues like how to "interface" the code with other mechanisms and with NEURON itself.

Because of the unusual structure and features of the NMODL language, it would be futile to attempt explanation without illustration. Therefore this paper is organized around a sequence of examples of increasing complexity and sophistication that introduce important topics in the context of problems of

scientific interest. These examples show how to take advantage of the leverage provided by NMODL for creating representations of biophysical mechanisms.

DESCRIBING MECHANISMS WITH NMODL

NMODL is a descendant of the MOdel Description Language (MODL (Kohn et al. 1994)), which was developed at Duke University by the National Biomedical Simulation Resource project for the purpose of building models that would be exercised by the Simulation Control Program (SCoP (Kootsey et al. 1986)). NMODL has the same basic syntax and style of organizing model source code into named blocks as MODL. Variable declaration blocks, such as `PARAMETER`, `STATE`, and `ASSIGNED`, specify names and attributes of variables that are used in the model. Other blocks are directly involved in setting initial conditions or generating solutions at each time step (the equation definition blocks, e.g. `INITIAL`, `BREAKPOINT`, `DERIVATIVE`, `KINETIC`, `FUNCTION`, `PROCEDURE`). Furthermore, C code can be inserted inside the model source code to accomplish implementation-specific goals.

NMODL recognizes all the keywords of MODL, but we will limit this discussion to those that are relevant to NEURON simulations. We will also examine the changes and extensions that were necessary to endow NMODL with NEURON-specific features. To give these ideas real meaning, they will be presented in the context of NMODL text for models of the following mechanisms:

- a passive "leak" current and a localized transmembrane shunt (density mechanisms vs. point processes)
- an electrode stimulus (discontinuous parameter changes with variable time step methods)
- voltage-gated channels (differential equations vs. kinetic schemes)
- ion accumulation in a restricted space (extracellular K^+)
- buffering, diffusion, and active transport (Ca^{2+} pump)
- synaptic transmission

This paper makes extensive use of specialized concepts and terminology that pertain to NEURON itself; for definitive treatment of these the reader is referred to prior publications ((Hines 1984; Hines 1989; Hines 1993; Hines 1994; Hines and Carnevale 1995), but particularly (Hines and Carnevale 1997)) and NEURON's on-line help files, which are available through links at http://www.neuron.yale.edu.

Example 1: a passive "leak" current

A passive "leak" current is one of the simplest biophysical mechanisms. Because it is distributed over the surface of a cell, it is described in terms of conductance per unit area and current per unit area, and therefore belongs to the class of "density mechanisms" (Hines and Carnevale 1997). Other density mechanisms include ion accumulation in a restricted space and active transport.

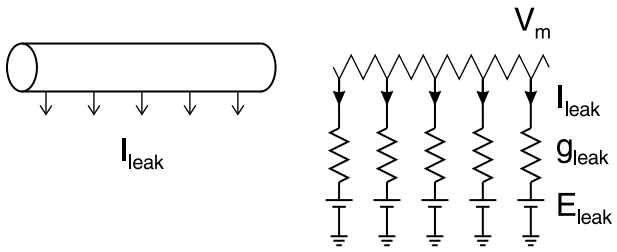

Figure 1

Figure 1 illustrates a branch of a neuron with a distributed leak current (left) and the equivalent circuit of a model of the passive current mechanism (right): a distributed constant conductance g_{leak} in series with a voltage source E_{leak} equal to the equilibrium potential for the ionic current. The leak current density is given by $i_{leak} = g_{leak} (V_m - E_{leak})$, where V_m is the membrane potential. Because this is a model of a physical system that is distributed in space, the variables i_{leak} and V_m and the parameters g_{leak} and E_{leak} are all functions of position.

Let us examine the NMODL text for an implementation of this model (Listing 1). Inline comments start with a colon and terminate at the end of the line. NMODL also allows comment blocks, which are demarcated by the keywords `COMMENT . . ENDCOMMENT`. In passing it should be noted that a similar syntax can be used to embed C code in a `mod` file, e.g.

```
VERBATIM
     /* c statements */
ENDVERBATIM
```

The statements between `VERBATIM` and `ENDVERBATIM` will appear without change in the output file that is written by the NMODL translator. Although this should be done only with great care, `VERBATIM` can be a convenient and effective way for individual users to add new features to NEURON or even to employ NEURON as a "poor man's C compiler."

```
: A passive leak current

NEURON {
     SUFFIX leak
     NONSPECIFIC_CURRENT i
     RANGE i, e, g
}

PARAMETER {
     g = 0.001  (siemens/cm2)  < 0, 1e9 >
     e = -65    (millivolt)
}

ASSIGNED {
     i  (milliamp/cm2)
     v  (millivolt)
}

BREAKPOINT { i = g*(v - e) }
```

Listing 1. `leak.mod`

Named blocks have the general form *KEYWORD* { *statements* }, and keywords are all upper case. User-defined variable names in NMODL can be up to 20 characters long. Each variable

must be defined before it is used. The variable names chosen for this example were `i`, `g`, and `e` for the leak current, its specific conductance, and its equilibrium potential, respectively. Some variables are not "owned" by any mechanism but are available to all mechanisms; these include `v`, `celsius`, `t`, `dt`, `diam`, and `area`.

As an aside, it should be noted that use of `dt` in NMODL is neither necessary nor good practice. Prior to the availability of variable time step methods in NEURON, analytic expressions involving `dt` were frequently used for efficient modeling of voltage sensitive channel states. This idiom is now built-in and employed automatically when such models are described in their underlying derivative form.

The `NEURON` block

The principal extension that differentiates NMODL from its MODL origins is that there are separate instances of mechanism data, with different values of states and parameters, in each segment (compartment) of a model cell. The `NEURON` block was introduced to make this possible by defining what the model of the mechanism looks like from the "outside" when there are many instances of the model sprinkled at different locations on the cell. The specifications entered in this block are independent of any particular simulator, but the detailed "interface code" requirements of a particular simulator determine whether the output C file is suitable for NEURON (NMODL) or GENESIS (GMODL). For this paper, we assume the translator is NMODL and that it produces code accepted by NEURON.

The actual name of the current NMODL translator is `nocmodl` (`nocmodl.exe` on the PC). This translator is consistent with the object-oriented extensions that were introduced with version 3 of NEURON. However, the older translator which predated these extensions was called `nmodl`, and we will use the generic name NMODL to refer to NEURON-compatible translators.

The `SUFFIX` keyword has two consequences. First, it identifies this to be a density mechanism, which can be incorporated into a NEURON cable section by an `insert` statement (see **Usage** below). Second, it tells the NEURON interpreter that the names for variables and parameters that belong to this mechanism will include the suffix `_leak`, so there will be no conflict with similar names in other mechanisms.

The stipulation that `i` is a `NONSPECIFIC_CURRENT` also has two consequences. First, the value of `i` will be reckoned in charge balance equations. Second, this current will make no direct contribution to mass balance equations (it will have no direct effect on ionic concentrations). We will show how to model mechanisms with specific ionic currents that can change concentrations in later examples.

The `RANGE` keyword asserts that the values of `i`, `e`, and `g` are functions of position. In other words, each of these variables can have a different value in each of the segments that make up a section. In the NEURON interpreter, manipulation of these variables uses the `RANGE` variable syntax (Hines and Carnevale 1997). The alternative to `RANGE` is `GLOBAL`, which is discussed below in ***The `PARAMETER` block***.

The membrane potential `v` is not mentioned in the `NEURON` block for two reasons. First, `v` is one of the variables that are available to all mechanisms. Second, it is not necessary to assert that `v` is a `RANGE` variable because membrane potential is a `RANGE` variable by default. However, for model completeness in non-NEURON contexts, and to enable units checking, `v` should be declared in the `ASSIGNED` block (see below).

Variable declaration blocks

As noted above, each user-defined variable must be declared before it is used. Even if it is named in the NEURON block, it still has to appear in a variable declaration block.

Mechanisms frequently involve expressions that contain a mix of constants and variables whose units belong to different scales of investigation and which may themselves be defined in terms of other, more "fundamental" units. This can easily lead to arithmetic errors that can be difficult to isolate and rectify. Therefore NMODL has special provisions for establishing and maintaining consistency of units. To facilitate unit checking, each variable declaration includes a specification of its units in parentheses. The names used for these specifications are based on the UNIX units database. A variable whose units are not specified is taken to be dimensionless.

The user may specify whatever units are appropriate except for variables that are defined by NEURON itself. These include v (millivolts), t (milliseconds), celsius (ºC), diam (μm), and area (μm^2). Currents, concentrations, and equilibrium potentials created by the USEION statement also have specific units (see **The NEURON block** in **Example 6: extracellular potassium accumulation** below). In this particular density mechanism, i and g are given units of current per unit area (milliamperes/cm^2) and conductance per unit area (siemens/cm^2), respectively.

The PARAMETER block

Variables whose values are normally specified by the user are parameters and are declared in a PARAMETER block. In the NEURON graphical user interface (GUI), a parameter is viewed using a special field editor which is designed to facilitate the entry of new values (see **Usage** below).

While parameters generally remain constant during a simulation, they can be changed in mid-run if necessary to emulate some external influence on the characteristic properties of a model. To avoid confusion, such changes should only be performed through the hoc interpreter or the GUI, and not by statements in the mod file.

The PARAMETER block in this example gives default values of 0.001 siemens/cm^2 and –65 mV to g and e, respectively. The pair of values in angle brackets specifies the default minimum and maximum values for g that can be entered into the field editor of the GUI. In this case, we merely ensure that conductance g cannot be negative.

Because g and e are PARAMETERs, their values are visible at the hoc level and can be overridden by hoc commands or altered through the GUI. PARAMETERs ordinarily have global scope, which means that changing the value of a PARAMETER affects every instance of that mechanism throughout an entire model. However, the NEURON block for this particular mechanism stipulates that g and e are RANGE variables, so they can be given different values in every segment where the leak current has been inserted.

The ASSIGNED block

The ASSIGNED block is used for declaring two kinds of variables: those that are given values outside the mod file, and those that appear on the left hand side of assignment statements within the mod file. The first group includes variables that are potentially available to every mechanism, such as v, celsius, t, and ionic variables (ionic variables are discussed in connection with **The NEURON block** in **Example 6: extracellular potassium accumulation** below). The second group specifically omits variables that are unknowns in a set of simultaneous linear or nonlinear

algebraic equations, or that are dependent variables in differential equations or kinetic reaction schemes, which are handled differently (see **Example 4: a voltage-gated current** below for a discussion of the `STATE` block).

Mechanism-specific `ASSIGNED` variables are `RANGE` variables by default. For a mechanism-specific `ASSIGNED` variable to be visible outside of the `mod` file, it must be declared as `RANGE` or `GLOBAL` in the `NEURON` block. `ASSIGNED` variables that are not "owned" by any mechanism (`v`, `celsius`, `t`, `dt`, `diam`, and `area`) are not mentioned in the `NEURON` block.

The current `i` is not a state variable because the model of the leak current mechanism does not define it in terms of a differential equation or kinetic reaction scheme; that is to say, `i` has no dynamics of its own. Furthermore it is not an unknown in a set of equations. Instead, it is calculated by direct assignment. Therefore it is declared in the `ASSIGNED` block.

For similar reasons membrane potential `v` is also declared in the `ASSIGNED` block. Although membrane potential is unquestionably a state variable in a model of a cell, to the leak current mechanism it is a driving force rather than a state variable.

Equation definition blocks

In this simple model there is only one equation, which is defined in the `BREAKPOINT` block.

The `BREAKPOINT` block

This is the main computation block in NMODL. Its name derives from SCoP, in which simulations are executed by incrementing an independent variable through a sequence of steps or "breakpoints" at which the dependent variables of the model are computed and displayed (Kohn et al. 1994).

At exit from the `BREAKPOINT` block, all variables should be consistent with the independent variable. The independent variable in NEURON is always time `t`, and neither `t` nor the time step `dt` should be changed in NMODL.

A single formula is all that is necessary for the leak current model. As we shall see later, more complicated models may require invoking NMODL's built-in routines to solve families of simultaneous algebraic equations or perform numeric integration.

Usage

The following hoc code illustrates how this mechanism might be used. Note the use of `RANGE` syntax to examine the value of `i_leak` near one end of `cable`.

```
cable {
      nseg = 5
      insert leak
      // override defaults
      g_leak = 0.002  // S/cm2
      e_leak = -70    // mV
}

// show leak current density near 0 end of cable
print cable.i_leak(0.1)
```

Because of the interface code generated as a consequence of the definitions in the `NEURON` block, the `leak` mechanism will appear with the other density mechanisms in the `Distributed Mechanism Manager` and `Viewer` windows. This is illustrated in Figure 2, which shows the `Distributed Mechanism Inserter`. The check mark signifies that the `leak` mechanism has been inserted into the section named `cable`.

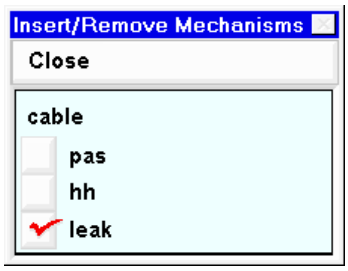

Figure 2

Example 2: a localized shunt

At the opposite end of the spatial scale from a distributed passive current is a localized shunt induced by microelectrode impalement (Durand 1984; Staley et al. 1992). A shunt is restricted to a small enough region that it can be described in terms of a net conductance (or resistance) and total current, i.e. it is a point process (Hines and Carnevale 1997). Most synapses are also best represented by point processes.

The localized nature of the shunt is emphasized in the cartoon of the neurite (Fig.3 left). The equivalent circuit of the shunt (right) is similar to the equivalent circuit of the distributed leak current (Fig.1 right), but here the resistance and current are understood to be concentrated in a single, circumscribed part of the cell. We will focus on how the NMODL code for this model differs from the density mechanism presented earlier.

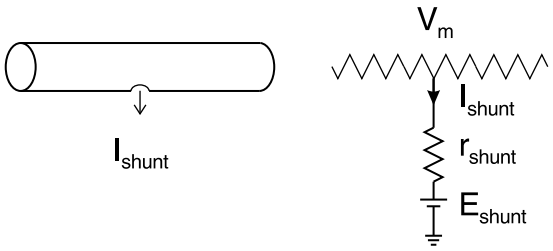

Figure 3

```
: A shunt current

NEURON {
      POINT_PROCESS Shunt
      NONSPECIFIC_CURRENT i
      RANGE i, e, r
}

PARAMETER {
      r = 1 (gigaohm)  < 1e-9, 1e9 >
      e = 0 (millivolt)
}

ASSIGNED {
      i  (nanoamp)
      v  (millivolt)
}

BREAKPOINT { i = (0.001)*(v - e)/r }
```

Listing 2. `shunt.mod`

The `NEURON` block

The `NEURON` block identifies this mechanism as a point process, which means that it will be managed in hoc using an object-oriented syntax (see **Usage** below). Making `i`, `e`, and `r` `RANGE` variables means that each instance of this point process can have separate values for these variables. If a variable is instead asserted to be `GLOBAL`, then its value would be shared among all instances of the mechanism.

Variable declaration blocks

These are nearly identical to the `PARAMETER` and `ASSIGNED` blocks of the `leak` mechanism. However, `Shunt` is a point process so all of its current flows at one site instead of being distributed over an area. Therefore its `i` and `r` are in units of nanoamperes (total current) and gigaohms (0.001 / total conductance in microsiemens), respectively.

This code specifies default values for the `PARAMETER`s `r` and `e`. Allowing a minimum value of 10^{-9} for `r` prevents an inadvertent divide by 0 error (infinite conductance) by ensuring that a user cannot set `r` to 0 in its GUI field editor. This protection, however, only holds for field editors and does not prevent an interpreter statement from setting `r` to 0 or even a negative value.

Equation definition blocks

Like the leak current mechanism, the shunt mechanism is extremely simple and involves no state variables. The single equation is defined in the `BREAKPOINT` block.

The `BREAKPOINT` block

The sole "complication" in this block is that the calculation of `i` includes a factor of 0.001 to reconcile the units on the left and right hand sides of this assignment (nanoamperes vs. millivolts divided by gigaohms). The parentheses surrounding this conversion factor are a convention that is necessary for units checking: they disambiguate it from mere multiplication by a number. When NEURON's unit checking utility `modlunit` is used to check the NMODL code in Listing 2, it will find no errors and will exit without an error message.

```
f:\modfils\leak\shunt>modlunit shunt.mod
model   $Revision: 1.1.1.1 $   $Date: 1994/10/12 17:22:51 $
Checking units of shunt.mod

f:\modfils\leak\shunt>
```

However if the parentheses were omitted, an error message would be emitted that reports inconsistent unit factors.

```
f:\modfils\leak\shunt>modlunit shunt.mod
model   $Revision: 1.1.1.1 $   $Date: 1994/10/12 17:22:51 $
Checking units of shunt.mod
The previous primary expression with units: 1-12 coul/sec
is missing a conversion factor and should read:
  (0.001)*()
 at line 20 in file shunt.mod
        i = 0.001*(v - e)/r<<ERROR>>
```

An error message would also result if parentheses surrounded a number which the user intended to be a quantity, since the unit factors would be inconsistent.

The convention of using single numbers enclosed in parentheses to signify unit conversion factors is simple and minimizes the possibility of mistakes either by the user or by the software. It is important to note that expressions that involve more than one number, such as "(1 + 1)", will *not* be interpreted as conversion factors.

Usage

This hoc code illustrates how the shunt mechanism might be applied to a section called `cable`; note the object syntax for specifying the shunt resistance and current (see (Hines and Carnevale 1997)).

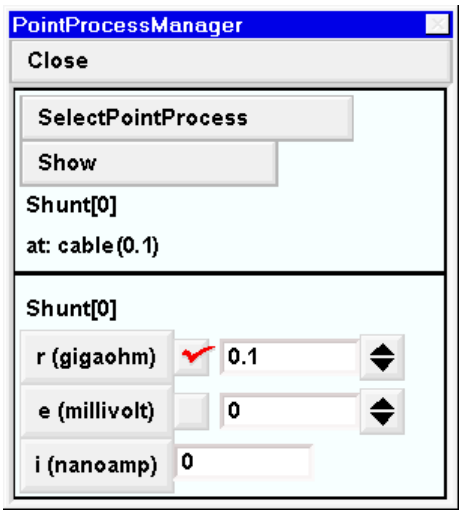

Figure 4

```
objref s
// put near 0 end of cable
cable s = new Shunt(0.1)
// not bad for a sharp electrode
s.r = 0.2
// show shunt current
print s.i
```

The definitions in the `NEURON` block of this particular model enable NEURON's graphical tools to include the `Shunt` object in the menus of its `Point Process Manager` and `Viewer` windows (Fig.4). The check mark on the button adjacent to the numeric field for `r` indicates that the shunt resistance has been changed from its default value (0.2 gigaohm when the shunt was created by the hoc code immediately above) to 0.1 gigaohm.

Example 3: an intracellular stimulating electrode

An intracellular stimulating electrode is similar to a shunt in the sense that both are localized sources of current that are modeled as point processes. However, the current from a stimulating electrode is not generated by an opening in the cell membrane but instead is injected directly into the cell. This particular model of a stimulating electrode has the additional difference that the current changes discontinuously, i.e. it is a pulse with distinct start and stop times.

```
: Current clamp

NEURON {
      POINT_PROCESS IClamp1
      RANGE del, dur, amp, i
      ELECTRODE_CURRENT i
}

UNITS { (nA) = (nanoamp) }
```

```
PARAMETER {
      del (ms)
      dur (ms)  < 0, 1e9 >
      amp (nA)
}

ASSIGNED { i (nA) }

INITIAL { i = 0 }

BREAKPOINT {
      at_time(del)
      at_time(del+dur)

      if (t < del + dur && t > del) {
            i = amp
      } else {
            i = 0
      }
}
```

Listing 3. `iclamp1.mod`

The `NEURON` block

This mechanism is identical to the built-in `IClamp` model. Calling it `IClamp1` allows the reader to test and modify it without conflict with the existing `IClamp` point process.

This model of a current clamp generates a rectangular current pulse whose amplitude `amp` in nanoamperes, start time `del` in milliseconds, and duration `dur` in milliseconds are all adjustable by the user. Furthermore, these parameters are individually adjustable for each separate instance of this mechanism. Therefore they are declared as `RANGE` variables in the `NEURON` block.

The current `i` delivered by `IClamp1` is declared in the `NEURON` block to make it available for examination. The `ELECTRODE_CURRENT` statement has two important consequences: positive values of `i` will depolarize the cell (in contrast to the hyperpolarizing effect of positive transmembrane currents), and when the `extracellular` mechanism is present there will be a change in the extracellular potential `vext`. Further discussion of extracellular fields is beyond the scope of this paper.

Equation definition blocks

The `BREAKPOINT` block

The logic for deciding whether `i` = 0 or `i` = `amp` is straightforward, but the `at_time()` calls need explanation. To work properly with variable time step methods, e.g. CVODE, models that change parameters discontinuously during a simulation must notify NEURON when such events take place. With fixed time step methods, users implicitly assume that events take place on time step boundaries (integer multiples of `dt`), and they would never consider defining a pulse duration narrower than `dt`. Neither eventuality can be left to chance with variable time step methods.

During a variable time step simulation, the first `at_time()` call guarantees that a time step boundary will be at `del` – ε, where ε is on the order of 10^{-9} ms. Integration will then restart from its new initial condition at `del` + ε. For more information, see **Discontinuities in PARAMETERs** below.

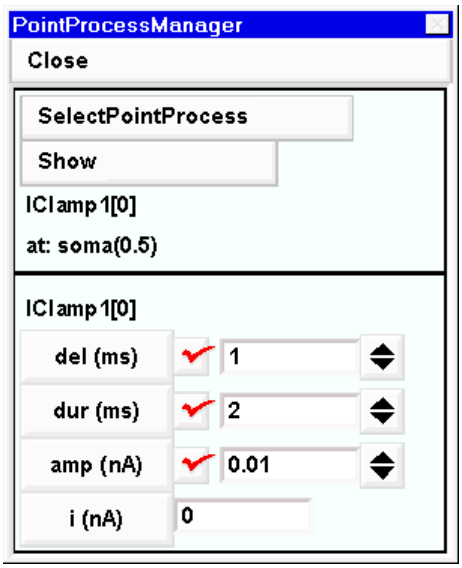

Figure 5

The `INITIAL` block

The code in the `INITIAL` block is executed when the hoc function `finitialize()` is called. Initialization of more complex mechanisms is discussed below in **Example 4: a voltage-gated current** and **Example 6: extracellular potassium accumulation**. The initialization here consists of making sure that `IClamp1.i` is 0 when `t` = 0.

Usage

Regardless of whether a fixed or variable time step integrator is chosen, `IClamp1` looks the same to the user. In either case, a current stimulus of 0.01 nA amplitude that starts at `t` = 1 ms and lasts for 2 ms would be created by this hoc code or through the GUI panel (Fig.5).

```
objref ccl
// put at middle of soma
soma ccl = new IClamp1(0.5)
ccl.del = 1
ccl.dur = 2
ccl.amp = 0.01
```

Example 4: a voltage-gated current

One of the particular strengths of NMODL is its flexibility in dealing with ion channels whose conductances are not constant but instead are regulated by factors such as the transmembrane potential gradient and/or the concentrations of ligands on one or both sides of the membrane. Here we will use the well-known Hodgkin-Huxley (HH) delayed rectifier to show how a voltage-gated current can be implemented, and later we will examine a model of a potassium (K^+) current that depends on both voltage and intracellular calcium concentration.

The delayed rectifier and all other voltage-gated channels that are distributed over the cell surface are density mechanisms. Therefore their NMODL representations and hoc usage will have many similarities to those of the passive leak current presented in Example 1. The following discussion focuses on the significant differences between the models of the delayed rectifier and the passive leak current.

In this example, membrane potential is in absolute millivolts, i.e. reversed in polarity from the original Hodgkin-Huxley convention and shifted to reflect a resting potential of –65 mV.

```
: HH voltage-gated potassium current

NEURON {
      SUFFIX kd
      USEION k READ ek WRITE ik
      RANGE gkbar, gk, ik
}

UNITS {
      (S)  = (siemens)
      (mV) = (millivolt)
      (mA) = (milliamp)
}

PARAMETER { gkbar = 0.036 (S/cm2) }

ASSIGNED {
      v     (mV)
      ek    (mV) : typically ~ -77.5
      ik    (mA/cm2)
      gk    (S/cm2)
}

STATE { n }

BREAKPOINT {
      SOLVE states METHOD cnexp
      gk = gkbar * n^4
      ik = gk * (v - ek)
}

INITIAL {
      : Assume v has been constant for a long time
      n = alpha(v)/(alpha(v) + beta(v))
}

DERIVATIVE states {
      : Computes state variable n at present v & t
      n' = (1-n)*alpha(v) - n*beta(v)
}

FUNCTION alpha(Vm (mV)) (/ms) {
      LOCAL x
      UNITSOFF
      x = (Vm+55)/10
      if (fabs(x) > 1e-6) {
            alpha = 0.1*x/(1 - exp(-x))
      }else{
            alpha = 0.1/(1 - 0.5*x)
      }
      UNITSON
}
```

```
FUNCTION beta(Vm (mV)) (/ms) {
      UNITSOFF
      beta = 0.125*exp(-(Vm+65)/80)
      UNITSON
}
```

Listing 4. `kd.mod`

The `NEURON` block

As with the passive model, `SUFFIX` marks this as a density mechanism, whose variables and parameters will be identified in hoc by a particular suffix. Three `RANGE` variables are declared in this block: the peak conductance density `gkbar` (the product of channel density and "open" conductance per channel), the macroscopic conductance `gk` (the product of `gkbar` and the fraction of channels that are open at any moment), and the current `ik` that passes through `gk`. At the level of hoc, these will be available as `gkbar_kd`, `gk_kd`, and `ik_kd`.

This model also has a fourth `RANGE` variable: the gating variable `n`, which is declared in the `STATE` block (see ***The `STATE` block*** below). `STATE` variables are automatically `RANGE` variables and do not need to be declared in the `NEURON` block.

A mechanism needs a separate `USEION` statement for each of the ions that it affects or is affected by. This example has one `USEION` statement, which includes `READ ek` because the potential gradient that drives `ik_kd` depends on the equilibrium potential for K^+. Since the resulting ionic flux may affect local $[K^+]$, this example also includes `WRITE ik` so that NEURON can keep track of the total outward current that is carried by an ion, its internal and external concentrations, and its equilibrium potential. We will return to this point in the context of a model with extracellular K^+ accumulation.

The UNITS block

The statements in the `UNITS` block define new names for units in terms of existing names in the UNIX units database. This can increase legibility and convenience, and is helpful both as a reminder to the user and as a means for automating the process of checking for consistency of units.

Variable declaration blocks

The `ASSIGNED` block

This is analogous to the `ASSIGNED` block of the `leak` mechanism. For the sake of clarity, variables whose values are computed outside this `mod` file are listed first. Note that `ek` is listed as an `ASSIGNED` variable, unlike `e` of the leak mechanism which was a `PARAMETER`. The reason for this difference is that mechanisms that produce K^+ fluxes may cause the equilibrium potential `ek` to change in the course of a simulation. However, the equilibrium potential for the leak current was not linked to a specific ionic species and therefore will remain fixed unless explicitly altered by hoc statements or the GUI.

The STATE block

If a model involves differential equations, families of algebraic equations, or kinetic reaction schemes, their dependent variables or unknowns are to be listed in the STATE block. Therefore gating variables such as the delayed rectifier's n are declared here.

In this paper we will refer to variables that are declared in the STATE block as STATE variables, or simply STATEs. This NMODL-specific terminology should not be confused with the physics or engineering concept of a "state variable" as a variable that describes the state of a system. While membrane potential is a "state variable" in the engineering sense, it would never be a STATE because its value is calculated only by NEURON and never by NMODL code. Likewise, the unknowns in a set of simultaneous equations (e.g. specified in a LINEAR or NONLINEAR block) would not be state variables in an engineering sense, yet they would all be STATEs.

All STATEs are automatically RANGE variables. This is appropriate, since channel gating can vary with position along a neurite.

Equation definition blocks

In addition to the BREAKPOINT block, this model also has INITIAL, DERIVATIVE, and FUNCTION blocks.

The BREAKPOINT block

This is the main computation block of the mechanism. By the end of the BREAKPOINT block, all variables are consistent with the new time. If a mechanism has STATEs, this block must contain one SOLVE statement that tell how the values of the STATEs will be computed over each time step. The SOLVE statement specifies a block of code that defines the simultaneous equations that govern the STATEs. Currents are set with assignment statements at the end of the BREAKPOINT block.

There are two major reasons why variables that depend on the number of times they are executed, such as counts or flags or random variables, should in general not be calculated in a BREAKPOINT block. First, the assignment statements in a BREAKPOINT block are usually called twice per time step. Second, with variable time step methods the value of t may not even be monotonically increasing. The metaphor to keep in mind is that the BREAKPOINT block is responsible for making all variables consistent at time t. Thus assignment statements in this block are responsible for trivially specifying the values of variables which depend *only* on the values of STATEs, t, and v, while the SOLVE statements perform the magic required to make the STATEs consistent at time t. It is not belaboring the point to reiterate that the assignment statements should produce the same result regardless of how many times BREAKPOINT is called with the same STATEs, t, and v. All too often errors have resulted from an attempt to explicitly compute what is conceptually a STATE in a BREAKPOINT block. Computations that must be performed only once per time step should be placed in a PROCEDURE, which in turn would be invoked by a SOLVE statement in a BREAKPOINT block.

In this connection it should be emphasized that the SOLVE statement is not a function call, and that the body of the DERIVATIVE block (or any other block specified by a SOLVE statement) will be executed asynchronously with respect to BREAKPOINT assignment statements. Therefore it is incorrect to invoke rate functions from the BREAKPOINT block; instead these must be called

from the block that is specified by the SOLVE statement (in this example, from within the DERIVATIVE block).

Models of active currents such as ik_kd are generally formulated in terms of ionic conductances that are functions of voltage- and time-dependent gating variables. The SOLVE statements at the beginning of the BREAKPOINT block specify the differential equations or kinetic schemes that govern the kinetics of the gating variables. The algebraic equations that compute the ionic conductances and currents follow the SOLVE statements.

For mechanisms whose STATEs are described by differential equations, it is often most convenient and efficient to use one of NEURON's built-in numerical integrators. A good choice for this particular mechanism is cnexp, which is described below in connection with the DERIVATIVE block.

The INITIAL block

The INITIAL block may contain any instructions that should be executed when the hoc function finitialize() is called. Though often overlooked, proper initialization of *all* STATEs is as important as correctly computing their temporal evolution. This is accomplished for the common case by finitialize(), which executes the initialization strategy defined in the INITIAL block for each mechanism. Prior to executing the INITIAL block, STATE values are set to their values in the STATE declaration block (or set to 0 if it was not given a specific value in the STATE declaration block).

For this delayed rectifier mechanism, n is set to its steady-state value for the membrane potential that exists in the compartment. This potential itself can be "left over" from a previous simulation run, or it can be specified by the user, e.g. on a compartment by compartment basis using statements such as dend.v(0.2) = -48 before calling finitialize(), or uniformly over the entire cell with a statement like finitialize(-55).

Initialization strategies. The INITIAL block should be used to initialize STATEs with respect to the initial values of membrane potential and ionic concentrations. It should be noted that there are several other ways to prepare STATEs for a simulation run. The most direct is simply to assign values explicitly using hoc statements such as cable.n_kd(0.3) = 0.9, but this can create arbitrary initial conditions that would be quite "unnatural."

A more "physiological" approach, which may be appropriate for models of oscillating or chaotic systems or whose mechanisms show other complex interactions, would be to perform an "initialization run" during which the model converges toward its limit cycle or attractor. A practical alternative for systems that settle to a stable equilibrium point when left undisturbed is to assign t a large negative value and then advance the simulation over several large time steps (keeping $t < 0$ prevents the initialization steps from triggering scheduled events such as stimulus currents or synaptic inputs). This tactic takes advantage of the strong stability properties of NEURON's implicit integration methods.

With either approach, once the initialization transients have decayed, the STATEs can be saved to a SaveState object that can then be kept in memory or written to a file for future re-use. The following example shows how to restore STATEs properly, assuming that they are contained in a SaveState object named mystates. When STATEs are restored, it is necessary to make sure that the variable order variable time step integrator is properly initialized; this is the purpose of cvode.re_init(), which has no effect if one is using a fixed time step method.

```
proc init() {
      // set Vm to v_init, t to 0,
      //   and call INITIAL block in all mechanisms
      finitialize(v_init)

      mystates.restore()

      // make all assigned variables (currents, conductances,
      //   equilibrium potentials) consistent with the STATEs
      fcurrent()

      //  initialize the cvode integrator
      cvode.re_init()  // no effect if cvode is not active
}
```

The `DERIVATIVE` block

This is used to assign values to the derivatives of those `STATE`s that are described by differential equations. The statements in this block are of the form $y' = expr$, where a series of apostrophes can be used to signify higher-order derivatives.

For NEURON's fixed time step integration method, these equations are integrated using the numerical method specified by the `SOLVE` statement in the `BREAKPOINT` block. The `SOLVE` statement should explicitly invoke one of the integration methods that is appropriate for systems in which state variables can vary widely during a time step (stiff systems). The `cnexp` method used in this example combines second-order accuracy with computational efficiency. It is appropriate when the right hand side of $y' = f(v,y)$ is linear in y, so it is well-suited to models with HH-style ionic currents. This method calculates the `STATE`s analytically under the assumption that all other variables are constant throughout the time step. If the variables change but are second-order correct at the midpoint of the time step, then the calculation of `STATE`s is also second-order correct.

If $f(v,y)$ is not linear in y, then the implicit integration method `derivimplicit` should be used. This provides first-order accuracy and is usable with general ODEs regardless of stiffness or nonlinearity.

With variable time step methods, *no* variable is assumed to be constant. These methods not only change the time step, but adaptively choose a numerical integration formula with local error that ranges from first-order up to $O(\Delta t^6)$. The present implementation of NMODL creates a diagonal Jacobian approximation for the block of `STATE`s. If $y_i' = f_i(v,y)$ is polynomial in y_i this is done analytically, otherwise by numerical differencing. In the rare case where this is inadequate, the user may supply an explicit Jacobian. Future versions of NMODL may attempt to deal with Jacobian evaluation in a more sophisticated manner. This illustrates a particularly important benefit of the NMODL approach: improvements in methods do not affect the high level description of the membrane mechanism.

The `FUNCTION` block

The functions defined by `FUNCTION` blocks are available at the hoc level and in other mechanisms by adding the suffix of the mechanism in which they are defined, e.g. `alpha_kd()` and `beta_kd()`. Functions or procedures can be simply called from hoc if they do not reference `RANGE` variables (references to `GLOBAL` variables are allowed). If a function or procedure does

reference a RANGE variable, then prior to calling the function from hoc it is necessary to specify the proper instance of the mechanism (its location on the cell). This is done by a setdata_ function that has the syntax

section_name { setdata_*suffix*(x) }

where *section_name* is the name of the section that contains the mechanism in question, *suffix* is the mechanism suffix, and x is the normalized distance along the section where the particular instance of the mechanism exists. The functions in our kd example do not use RANGE variables, so a specific instance is not needed.

The differential equation that describes the kinetics of n involves two voltage-dependent rate constants whose values are computed by the functions alpha() and beta(). The original algebraic form of the equations that define these rates is

$$\alpha = \frac{0.1\left(\frac{v+55}{10}\right)}{1-e^{-\left(\frac{v+55}{10}\right)}} \qquad \text{and} \qquad \beta = 0.125\,e^{-\left(\frac{v+65}{80}\right)}$$

The denominator for α goes to 0 when v = –55 mV, which could cause numeric overflow. The code used in alpha() avoids this by switching, when v is very close to –55, to an alternative expression that is based on the first three terms of the infinite series expansion of e^x.

As noted elsewhere in this paper, NMODL has features that facilitate establishing and maintaining consistency of units. Therefore the rate functions alpha() and beta() are introduced with the syntax

FUNCTION *f_name*(*arg1 (units1), arg2 (units2), . . .*) *(returned_units)*

to declare that their arguments are in units of millivolts and that their returned values are in units of inverse milliseconds (“/ms”). This allows automatic units checking on entry to and return from these functions. For the sake of legibility the UNITSOFF . . .UNITSON directives disable units checking just within the body of these functions. This is acceptable because the terms in the affected statements are mutually consistent. Otherwise the statements would have to be rewritten in a way that makes unit consistency explicit at the cost of legibility, e.g.

```
x = (Vm + 55 (millivolt))/(10 (millivolt))
```

Certain variables exist solely for the sake of computational convenience. These typically serve as scale factors, flags, or temporary storage for intermediate results, and are not of primary importance to the mechanism. Such variables are often declared as LOCAL variables within an equation block, e.g. x in this mechanism. LOCAL variables that are declared in an equation block are not “visible” outside the block and they do not retain their values between invocations of the block. LOCAL variables that are declared outside an equation block have very different properties and are discussed under **Variable declaration blocks** in **Example 8: calcium diffusion with buffering**.

Usage

The hoc code and graphical interface for using this distributed mechanism are similar to those for the `leak` mechanism (Fig.2). However, the `kd` mechanism involves more `RANGE` variables, and this is reflected in the choices available in the `RANGE` variable menu of NEURON's `Plot what?` tool for graph windows. Since `kd` uses potassium, the variables `ek` and `ik` (total K^+ current) appear in this list along with the variables that are explicitly declared as `RANGE` and `STATE` in `kd.mod` (see Fig.6). The total K^+ current `ik` will differ from `ik_kd` only if another mechanism that `WRITE`s `ik` is present in this section.

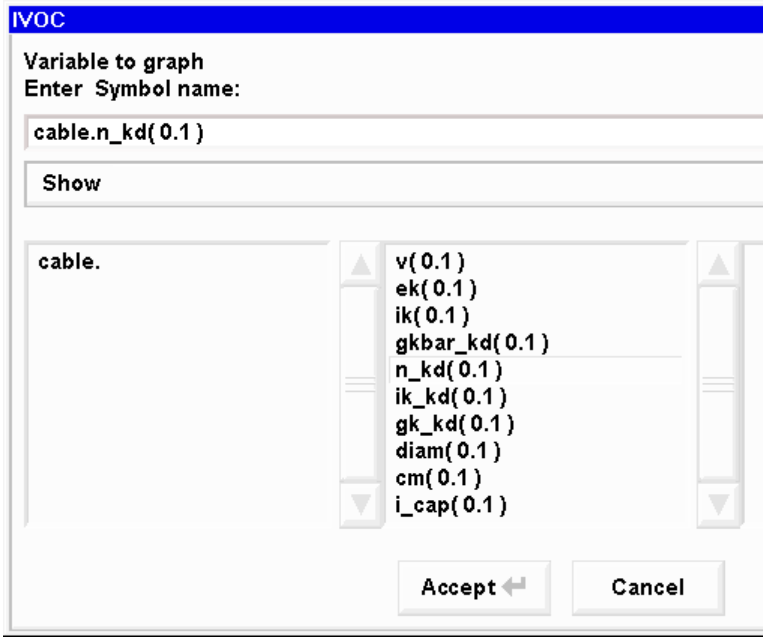

Figure 6

Example 5: a calcium-activated voltage-gated current

This model of a potassium current that depends on both voltage and intracellular calcium concentration $[Ca^{2+}]_i$. is based on the work of Moczydlowski and Latorre (1983). It is basically an elaboration of the HH mechanism in which the forward and backward rates depend jointly on membrane potential and $[Ca^{2+}]_i$.. Here we point out the salient implementational differences between this and the previous model.

```
: Calcium activated K channel

NEURON {
      SUFFIX cagk
      USEION ca READ cai
      USEION k READ ek WRITE ik
      RANGE gkbar
      GLOBAL oinf, tau
}

UNITS {
      (mV)    = (millivolt)
      (mA)    = (milliamp)
      (S)     = (siemens)
      (molar) = (1/liter)
      (mM)    = (millimolar)
      FARADAY = (faraday) (kilocoulombs)
      R       = (k-mole) (joule/degC)
}
```

```
PARAMETER {
      gkbar = 0.01  (S/cm2)
      d1    = 0.84
      d2    = 1.0
      k1    = 0.18  (mM)
      k2    = 0.011 (mM)
      bbar  = 0.28  (/ms)
      abar  = 0.48  (/ms)
}

ASSIGNED {
      cai      (mM)    : typically 0.001
      celsius  (degC)  : typically 20
      v        (mV)
      ek       (mV)
      ik       (mA/cm2)
      oinf
      tau      (ms)
}

STATE { o }    : fraction of channels that are open

BREAKPOINT {
      SOLVE state METHOD cnexp
      ik = gkbar*o*(v - ek)
}

DERIVATIVE state {
      rate(v, cai)
      o' = (oinf - o)/tau
}

INITIAL {
      rate(v, cai)
      o = oinf
}

: the following are all callable from hoc

FUNCTION alp(v (mV), ca (mM)) (/ms) {
      alp = abar/(1 + exp1(k1,d1,v)/ca)
}

FUNCTION bet(v (mV), ca (mM)) (/ms) {
      bet = bbar/(1 + ca/exp1(k2,d2,v))
}

FUNCTION exp1(k (mM), d, v (mV)) (mM) {
      : numeric constants in an addition or subtraction
      : expression automatically take on the unit values
      : of the other term
      exp1 = k*exp(-2*d*FARADAY*v/R/(273.15 + celsius))
}
```

```
PROCEDURE rate(v (mV), ca (mM)) {
      LOCAL a
      : LOCAL variable takes on units of right hand side
      a = alp(v,ca)
      tau = 1/(a + bet(v, ca))
      oinf = a*tau
}
```

Listing 5. `cagk.mod`

The `NEURON` block

Because the potassium conductance depends on $[Ca^{2+}]_i$, two `USEION` statements are required. The `RANGE` statement declares only the peak conductance density `gkbar`, so this mechanism's ionic conductance will not be visible from hoc (in fact, the activated ionic conductance density is not even calculated in this model). Likewise, there will be no `ik_cagk` that reports this particular current component separately, even though it will be added to the total K^+ current `ik` because of `WRITE ik`.

The variables `oinf` and `tau`, which govern the gating variable `o`, should be accessible in hoc for the purpose of seeing how they vary with membrane potential and $[Ca^{2+}]_i$. At the same time, the storage and syntax overhead required for a `RANGE` variable does not seem warranted because it appears unlikely to be necessary or useful to plot either `oinf` or `tau` as a function of space. Therefore they have been declared to be `GLOBAL` rather than `RANGE`. On first examination, this might seem to pose a problem. The gating of this K^+ current depends on membrane potential and $[Ca^{2+}]_i$, both of which may vary with location, so how can it be correct to use `GLOBAL`s for `oinf` and `tau`? And if some reason did arise to examine the values of these variables at a particular location, how could this be done? We shall see that the answers to these questions lie in the `DERIVATIVE` and `PROCEDURE` blocks.

The UNITS block

The last two statements in this block require some clarification. The first parenthesized item on the right hand side of the equal sign is the numeric value of a standard entry in the UNIX units database, which may be expressed on a scale appropriate for physics rather than membrane biophysics. The second parenthesized item acts like a scale factor that converts it to the specific units chosen for this model. Thus `(faraday)` appears in the units database in terms of coulombs/mole and has a numeric value of 96,485.309, but for this particular mechanism we prefer to use a constant whose units are kilocoulombs/mole. The statement

```
FARADAY = (faraday) (kilocoulombs)
```

results in `FARADAY` having units of kilocoulombs and a numeric value of 96.485309. The item `(k-mole)` in the statement

```
R       = (k-mole) (joule/degC)
```

is not kilomoles but instead is a specific entry in the units database equal to the product of Boltzmann's constant and Avogadro's number. The end result of this statement is that `R` has units of joules/°C and a numeric value of 8.313424. These special definitions of `FARADAY` and `R` pertain to this mechanism only; a different mechanism could assign different units and numeric values to these labels.

Another possible source of confusion is the interpretation of the symbol "e". This is always the electronic charge ($\sim 1.6 \cdot 10^{-19}$ coulombs), except outside the UNITS block where a *single* number in parentheses is treated as a conversion factor, e.g. the expression `(2e4)` is treated as a conversion factor of $2 \cdot 10^4$. Although errors involving "e" in a units expression are easy to make, they are always caught by `modlunit`.

Variable declaration blocks

The `ASSIGNED` block

Comments in this block can be helpful to the user as reminders of "typical" values or usual conditions under which a mechanism operates. For example, the `cagk` mechanism is intended for use in the context of $[Ca^{2+}]_i$ on the order of 0.001 mM. Similarly, the temperature sensitivity of this mechanism is accommodated by including the global variable `celsius`. NEURON's default value for `celsius` is 6.3ºC, but as the comment in this `mod` file points out, the parameter values for this particular mechanism were intended for an "operating temperature" of 20ºC. Therefore the user may need to change `celsius` through hoc or the GUI.

The variables `oinf` and `tau`, which were made accessible to NEURON by the `GLOBAL` statement in the `NEURON` block, are given values by the procedure `rate` and are declared as `ASSIGNED`.

The `STATE` block

Because `o`, the fraction of channels that are open, is described by a differential equation, this mechanism needs a `STATE` block.

Equation definition blocks

The `BREAKPOINT` block

This mechanism does not make its ionic conductance available to hoc, so the `BREAKPOINT` block just calculates the ionic current passing through these channels and doesn't bother with separate computation of a conductance.

The `DERIVATIVE` block

The gating variable o is governed by a first-order differential equation. The procedure `rate` assigns values to the voltage-sensitive parameters of this equation: the steady-state value `oinf`, and the time constant `tau`.

This provides the answer to the first question that was raised above in the discussion of the `NEURON` block. The procedure `rate` will be executed individually for each segment in the model that has the `cagk` mechanism. Each time `rate` is called, its arguments will equal the membrane potential and $[Ca^{2+}]_i$ of the segment that is being processed, since `v` and `cai` are both `RANGE` variables. Therefore `oinf` and `tau` can be `GLOBAL` without destroying the spatial variation of the gating variable `o`.

The FUNCTION and PROCEDURE blocks

The functions `alp()`, `bet()`, `exp1()`, and the procedure `rate()` implement the mathematical expressions that describe `oinf` and `tau`. To facilitate units checking, their arguments are tagged with the units that they use. The `rate()` procedure achieves some efficiency by calling `alp()` once and using the returned value twice; calculating `oinf` and `tau` separately would have required two calls to `alp()`.

The procedure `rate()` helps answer the second question that was raised in the discussion of the `NEURON` block: how to examine the variation of `oinf` and `tau` over space. This is easily done in hoc with code such as

```
forall { // iterate over all sections
     for (x) { // iterate over each segment
          rate(v(x), cai(x))
          // here put statements to plot
          //   or save oinf and tau
     }
}
```

Usage

This mechanism involves both K^+ and Ca^{2+}, so the list of `RANGE` variables displayed by `Plot what?` has more entries than it did for the `kd` mechanism (compare Figs.7 and 6). However, `cai`, `cao`, and `eca` will remain constant unless the section in which this mechanism has been inserted also includes something that can affect calcium concentration (e.g. a pump or buffer).

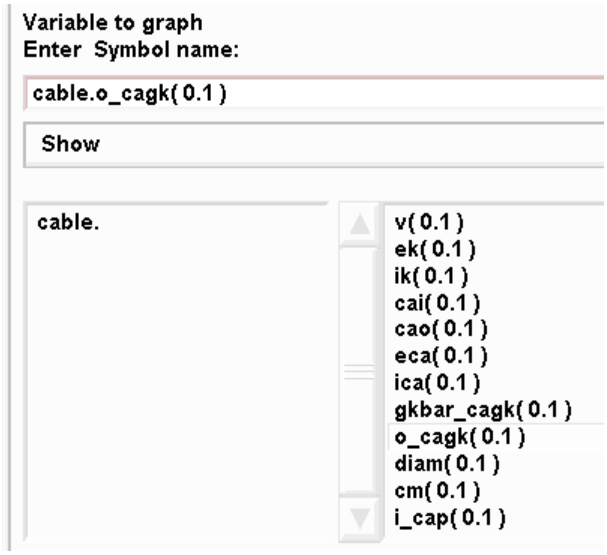

Figure 7

Example 6: extracellular potassium accumulation

Because mechanisms can generate transmembrane fluxes that are attributed to specific ionic species by the `USEION` *x* `WRITE` *ix* syntax, modeling the effects of restricted diffusion is straightforward. The `kext` mechanism described here emulates the accumulation of potassium in the extracellular space adjacent to squid axon (Fig.8). The experiments of Frankenhaeuser and Hodgkin (1956) indicated that satellite cells and other

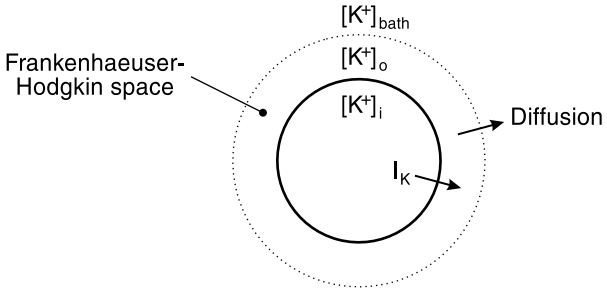

Figure 8

extracellular structures act as a diffusion barrier that prevents free communication between this space and the bath. Therefore, when there is a large efflux of K^+ ions from the axon, e.g. during the repolarizing phase of an action potential or in response to injected depolarizing current, K^+ builds up in the "Frankenhaeuser-Hodgkin space" (F-H space). This elevation of $[K^+]_o$ shifts E_K in a depolarized direction, which has two important consequences. First, it reduces the driving force for K^+ efflux and causes a decline of the outward I_K. Second, when the action potential terminates or the injected depolarizing current is stopped, the persistent elevation of E_K causes a slowly decaying depolarization or inward current. This depolarizing shift dissipates gradually as $[K^+]_o$ equilibrates with $[K^+]_{bath}$.

```
: Extracellular potassium ion accumulation

NEURON {
  SUFFIX kext
  USEION k READ ik WRITE ko
  GLOBAL kbath
  RANGE fhspace, txfer
}

UNITS {
  (mV)    = (millivolt)
  (mA)    = (milliamp)
  FARADAY = (faraday) (coulombs)
  (molar) = (1/liter)
  (mM)    = (millimolar)
}

PARAMETER {
  kbath   =  10 (mM)        : seawater (squid axon!)
  fhspace = 300 (angstrom)  : effective thickness of F-H space
  txfer   =  50 (ms)  : tau for F-H space <-> bath exchange = 30-100
}

ASSIGNED { ik  (mA/cm2) }

STATE { ko  (mM) }

BREAKPOINT { SOLVE state METHOD cnexp }

DERIVATIVE state {
  ko' = (1e8)*ik/(fhspace*FARADAY) + (kbath - ko)/txfer
}
```

Listing 6. `kext.mod`

The `NEURON` block

A compartment may contain several mechanisms that have direct interactions with ionic concentrations (e.g. diffusion, buffers, pumps). Therefore NEURON must be able to compute the total currents and concentrations consistently. The `USEION` statement sets up the necessary "bookkeeping" by automatically creating a separate mechanism that keeps track of four essential

variables: the total outward current carried by an ion, the internal and external concentrations of the ion, and its equilibrium potential. In this case the name of the ion is "k" and the automatically-created mechanism is called "k_ion" in the hoc interpreter. The k_ion mechanism has variables ik, ki, ko, and ek, which represent I_K, $[K^+]_i$, $[K^+]_o$, and E_K, respectively. These do not have suffixes; furthermore, they are RANGE variables so they can have different values in every segment of each section in which they exist. In other words, the K^+ current through Hodgkin-Huxley potassium channels near one end of the section cable would be cable.ik_hh(0.1), but the total K^+ current generated by all sources, including other ionic conductances and pumps, would be cable.ik(0.1).

This mechanism computes $[K^+]_o$ from the outward potassium current, so it READs ik and WRITEs ko. When a mechanism WRITEs a particular ionic concentration, this means that it sets the value for that concentration at all locations in every section into which it has been inserted. This has an important consequence: in any given section, no ionic concentration should be "written" by more than one mechanism.

The bath is assumed to be a large, well-stirred compartment that envelops the entire "experimental preparation." Therefore kbath is a GLOBAL variable so that all sections that contain the kext mechanism will have the same numeric value for $[K^+]_{bath}$. Since this would be one of the controlled variables in an experiment, the value of kbath is specified by the user and will remain constant during the simulation. The thickness of the F-H space is fhspace, the time constant for equilibration with the bath is txfer, and both are RANGE variables so they can vary along the length of each section.

Variable declaration blocks

The PARAMETER block

The default value of kbath is set to 10 mM, consistent with the composition of seawater (Frankenhaeuser and Hodgkin 1956). Since kbath is GLOBAL, a single hoc statement can change this to a new value that will affect all occurrences of the kext mechanism, e.g. kbath_kext = 8 would change it to 8 mM everywhere.

The STATE block

Ionic concentration is a STATE of a mechanism only if that mechanism calculates the concentration. This model computes ko, the potassium concentration in the F-H space, according to the dynamics specified by an ordinary differential equation.

Equation definition blocks

The BREAKPOINT block

This mechanism involves a single differential equation that tells the rate of change of ko, the K^+ concentration in the F-H space. The choice of integration method in NMODL is based on the recognition that the equation is linear in ko. The total K^+ current ik might also vary during a time step (see the DERIVATIVE block) if membrane potential, some K^+ conductance, or ko itself is changing rapidly. In a simulation where such rapid changes were likely to occur, proper modeling practice would lead one either to use NEURON with CVODE, or to use a fixed time step that would be short compared to the rate of change of ik.

The INITIAL block

The only STATE in this mechanism is the ionic concentration ko, so this mechanism does not have an INITIAL block. This is because the model translator for NEURON ignores default values for ionic concentrations. Any assignment to an ion concentration in an INITIAL block will result in an inconsistent initialization on return from finitialize(). Furthermore, in this particular model it is likely to be too limiting to set ko = kbath.

Instead, concentrations should be initialized in hoc. Choosing the best way to do this depends on the design and intended use of the model in which the mechanism has been embedded: is the concentration supposed to start at the same value in all sections where the mechanism has been inserted, or should it be nonuniform from the outset?

Take the case of a mechanism that WRITEs an ion concentration. Such a mechanism has an associated global variable that can be used to initialize the concentration to the same value in each section where the mechanism exists. These global variables have default values for na, k and ca that are "reasonable" but probably incorrect for any specific preparation. The default concentrations for ion names created by the user are 1 mM; these should be assigned correct values in hoc. A subsequent call to finitialize() will use this to initialize the ionic concentration.

The name of the global variable is formed from the name of the ion that the mechanism uses and the concentration that it WRITEs. For example, the kext mechanism uses k and WRITEs ko, so the corresponding global variable is ko0_k_ion. The sequence of instructions

```
ko0_k_ion = 10      // seawater, 4 x default value (2.5)
ki0_k_ion = 4*54.4  // 4 x default value, preserves ek
finitialize(v_init) // v_init is the starting Vm
```

will set ko to 10 mM and ki to 217.6 mM in every segment that has the kext mechanism.

What if one or more sections of the model are supposed to have different initial concentrations? For these particular sections the ion_style() function would be used to assert that the global variable is not to be used to initialize the concentration for this particular ion. The numeric arguments in the statement

```
dend ion_style("k_ion",3,2,1,1,0)
```

would have the following effects on the kext mechanism in the dend section (in sequence): treat ko as a STATE variable; treat ek as an ASSIGNED variable; on call to finitialize() use the Nernst equation to compute ek from the concentrations; compute ek from the concentrations on every call to fadvance(); do *not* use ko0_k_ion or ki0_k_ion to set the initial values of ko and ki. The proper initialization would now be to set ko and ki explicitly for this section, e.g.

```
ko0_k_ion = 10  // all sections start with ko = 10 mM
dend {ko = 5  ki = 2*54.4}  //  . . . except dend
finitialize(v_init)
```

A complete discussion of ion_style(), its arguments, and its actions is contained in NEURON's help system.

The `DERIVATIVE` block

At the core of this mechanism is a single differential equation that relates $d[K^+]_o/dt$ to the sum of two terms. The first term describes the contribution of `ik` to $[K^+]_o$, subject to the assumption that the thickness F-H space is much smaller than the diameter of the section. The unit conversion factor of 10^8 is required because `fhspace` is given in Ångstroms. The second term describes the exchange of K^+ between the bath and the F-H space.

Usage

If this mechanism is present in a section, the following `RANGE` variables will be accessible through hoc: $[K^+]$ inside the cell and within the F-H space (`ki` and `ko`); equilibrium potential and total current for K (`ek` and `ik`); thickness of the F-H space and the rate of equilibration between it and the bath (`fhspace_kext` and `txfer_kext`). The bath $[K^+]$ will also be available as the global variable `kbath_kext`.

General comments about kinetic schemes

Kinetic schemes provide a high level framework that is perfectly suited for compact and intuitively clear specification of models that involve discrete states in which "material" is conserved. The basic notion in such mechanisms is that flow out of one state equals flow into another. Almost all models of membrane channels, chemical reactions, macroscopic Markov processes, and ionic diffusion are elegantly expressed through kinetic schemes. It will be helpful to review some fundamentals before proceeding to specific examples of mechanisms implemented with kinetic schemes.

The unknowns in a kinetic scheme, which are usually concentrations of individual reactants, are declared in the `STATE` block. The user expresses the kinetic scheme with a notation that is very similar to a list of simultaneous chemical reactions. The NMODL translator converts the kinetic scheme into a family of ODEs whose unknowns are the `STATE`s. Hence the simple

```
STATE { mc    m }
KINETIC scheme1 {
      ~ mc <-> m (a(v), b(v))
}
```

is equivalent to

```
DERIVATIVE scheme1 {
      mc' = -a(v)*mc + b(v)*m
      m'  =  a(v)*mc - b(v)*m
}
```

The first character of a reaction statement is the tilde "~", which is used to immediately distinguish this kind of statement from other sequences of tokens that could be interpreted as an expression. The expression to the left of the three character reaction indicator "`<->`" specifies the reactants, and the expression immediately to the right specifies the products. The two expressions in parentheses specify the forward and reverse reaction rates (here the rate functions `a(v)` and

b(v)). After each reaction, the variables f_flux and b_flux are assigned the values of the forward and reverse fluxes respectively. These can be used in assignment statements such as

```
~ cai + pump <-> capump (k1,k2)
~ capump <-> pump + cao (k3,k4)
ica = (f_flux - b_flux)*2*Faraday/area
```

In this case, the forward flux is k3*capump, the reverse flux is k4*pump*cao, and the positive-outward current convention is consistent with the sign of the expression for ica (in the second reaction, forward flux means positive ions move from the inside to the outside).

More complicated reaction sequences such as the wholly imaginary

```
KINETIC scheme2 {
      ~ 2A + B <-> C (k1,k2)
      ~  C + D <-> A + 2B (k3,k4)
}
```

begin to show the clarity of expression and suggest the comparative ease of modification of the kinetic representation over the equivalent but stoichiometrically confusing

```
DERIVATIVE scheme2 {
      A' = -2*k1*A^2*B + 2*k2*C   + k3*C*D   - k4*A*B^2
      B' =   -k1*A^2*B   + k2*C + 2*k3*C*D - 2*k4*A*B^2
      C' =    k1*A^2*B   - k2*C   - k3*C*D   + k4*A*B^2
      D' =                        - k3*C*D   + k4*A*B^2
}
```

Clearly a statement such as

```
~ calmodulin + 3Ca <-> active (k1, k2)
```

would be easier to modify (e.g. so it requires combination with 4 calcium ions) than the relevant term in the three differential equations for the STATEs that this reaction affects. The kinetic representation is easy to debug because it closely resembles familiar notations and is much closer to the conceptualization of what is happening than the differential equations would be.

Another benefit of kinetic schemes is the simple polynomial nature of the flux terms, which allows the translator to easily perform a great deal of preprocessing that makes implicit numerical integration more efficient. Specifically, the nonzero elements $\partial y_i'/\partial y_j$ (partial derivatives of dy_i/dt with respect to y_j) of the sparse matrix are calculated analytically in NMODL and collected into a C function that is called by solvers to calculate the Jacobian. Furthermore, the form of the reaction statements determines if the scheme is linear, obviating an iterative computation of the solution. Voltage-sensitive rates are allowed, but to guarantee numerical stability the rate constants should not be functions of STATEs. Thus writing the calmodulin example as

```
~ calmodulin <-> active (k3*Ca^3, k2)
```

will work but is potentially unstable if `Ca` is a `STATE` in other simultaneous reactions in the same `mod` file. Variable time step methods such as CVODE will compensate by reducing `dt`, but this will make the simulation run more slowly.

Kinetic scheme representations provide a great deal of leverage because a single compact expression is equivalent to a large amount of C code. One special advantage from the programmer's point of view is the fact that these expressions are independent of the solution method. Different solution methods require different code, but the NMODL translator generates this code automatically. This saves the user's time and effort and ensures that all code expresses the same mechanism. Another advantage is that the NMODL translator handles the task of interfacing the mechanism to the remainder of the program. This is a tedious exercise that would require the user to have special knowledge that is not relevant to neurophysiology and which may change from version to version.

Special issues are raised by mechanisms that involve fluxes between compartments of different size, or whose reactants have different units. The first of the following examples has none of these complications, which are addressed later in models of diffusion and active transport.

Example 7: kinetic scheme for a voltage-gated current

This illustration of NMODL's facility for handling kinetic schemes implements a simple three-state model for the conductance state transitions of a voltage-gated potassium current

$$\mathrm{C}_1 \underset{kb_1}{\overset{kf_1}{\rightleftarrows}} \mathrm{C}_2 \underset{kb_2}{\overset{kf_2}{\rightleftarrows}} \mathrm{O}$$

The closed states are C_1 and C_2, the open state is O, and the rates of the forward and backward state transitions are calculated in terms of the equilibrium constants and time constants of the isolated reactions through the familiar expressions $K_i(v) = kf_i / kb_i$ and $\tau_i(v) = 1/(kf_i + kb_i)$. The equilibrium constants $K_i(v)$ are given by the Boltzmann factors $K_1 = e^{\left[k_2(d_2 - v) - k_1(d_1 - v)\right]}$ and $K_2 = e^{-k_2(d_2 - v)}$, where the energies of states C_1, C_2, and O are 0, $k_1(d_1 - v)$, and $k_2(d_2 - v)$ respectively.

The typical sequence of analysis is to determine the constants k_1, d_1, k_2, and d_2 by fitting the steady-state voltage clamp data, and then to find the voltage-sensitive transition time constants $\tau_1(v)$ and $\tau_2(v)$ from the temporal properties of the clamp current at each voltage pulse level. In this example the steady-state information has been incorporated in the NMODL code, and the time constants are conveyed by tables (arrays) that are created within the interpreter.

```
: Three state kinetic scheme for HH-like potassium channel
: Steady-state v-dependent state transitions have been fit
: Needs v-dependent time constants from tables created under hoc

NEURON {
   SUFFIX k3st
   USEION k READ ek WRITE ik
   RANGE g, gbar
}

UNITS { (mV) = (millivolt) }

PARAMETER {
   gbar = 33     (millimho/cm2)
   d1   = -38    (mV)
   k1   = 0.151  (/mV)
   d2   = -25    (mV)
   k2   = 0.044  (/mV)
}

ASSIGNED {
   v    (mV)
   ek   (mV)
   g    (millimho/cm2)
   ik   (milliamp/cm2)
   kf1  (/ms)
   kb1  (/ms)
   kf2  (/ms)
   kb2  (/ms)
}

STATE { c1 c2 o }

BREAKPOINT {
   SOLVE kin METHOD sparse
   g = gbar*o
   ik = g*(v - ek)*(1e-3)
}

INITIAL { SOLVE kin STEADYSTATE sparse }

KINETIC kin {
   rates(v)
   ~ c1 <-> c2     (kf1, kb1)
   ~ c2 <-> o      (kf2, kb2)
   CONSERVE c1 + c2 + o = 1
}

FUNCTION_TABLE tau1(v(mV)) (ms)
FUNCTION_TABLE tau2(v(mV)) (ms)
```

```
PROCEDURE rates(v(millivolt)) {
   LOCAL K1, K2
   K1 = exp(k2*(d2 - v) - k1*(d1 - v))
   kf1 = K1/(tau1(v)*(1+K1))
   kb1 = 1/(tau1(v)*(1+K1))
   K2 = exp(-k2*(d2 - v))
   kf2 = K2/(tau2(v)*(1+K2))
   kb2 = 1/(tau2(v)*(1+K2))
}
```

Listing 7. `k3st.mod`

The `NEURON` block

With one exception, the `NEURON` block of this model is essentially the same as for the delayed rectifier presented above in **Example 4: a voltage-gated current**. The difference is that, even though this model contributes to the total K^+ current `ik`, its own current is not available separately (i.e. there will be no `ik_k3st` at the hoc level) because `ik` is not declared as a `RANGE` variable.

Variable declaration blocks

The `STATE` block

The `STATE`s in this mechanism are the fractions of channels that are in closed states 1 or 2 or in the open state. Since the total number of channels in all states is conserved, the sum of the `STATE`s must be unity

$$c1 + c2 + o = 1$$

This conservation law means that the `k3st` mechanism really has only two independent state variables, a fact that underscores the difference between a `STATE` in NMODL and the concept of a state variable. It also affects how NMODL sets up the equations that are to be solved, as we will see in the discussion of the `KINETIC` block below.

Not all reactants or products need to be `STATE`s. If the reactant is an `ASSIGNED` or `PARAMETER` variable, then a differential equation is not generated for it, and it is treated as constant for the purposes of calculating the declared `STATE`s. Statements such as

```
PARAMETER {kbath (mM)}
STATE {ko (mM)}
KINETIC scheme3 {
      ~ ko <-> kbath (r, r)
}
```

are translated to the single ODE equivalent

```
ko' = r*(kbath - ko)
```

i.e. `ko` tends exponentially to the steady state value of `kbath`.

Equation definition blocks

The `BREAKPOINT` block

The recommended idiom for integrating a kinetic scheme is

```
BREAKPOINT {
      SOLVE scheme METHOD sparse
      . . .
}
```

which integrates the `STATE`s in the scheme one `dt` step per call to `fadvance()` in NEURON. The `sparse` method is generally faster than computing the full Jacobian matrix, though both use Newton iterations to advance the `STATE`s with a fully implicit method (first-order correct). Additionally, the `sparse` method separates the Jacobian evaluation from the calculation of the `STATE` derivatives, thus allowing variable time step methods, such as CVODE, to efficiently compute only what is needed to advance the `STATE`s. Non-implicit methods, such as Runge-Kutta or Euler, should be avoided since kinetic schemes commonly have very wide ranging rate constants that make these methods numerically unstable with reasonable `dt` steps. In fact, it is not unusual to specify equilibrium reactions such as

```
~ A <-> B (1e6*sqrt(K), 1e6/sqrt(K))
```

which can only be solved by implicit methods.

The `INITIAL` block

Initialization of a kinetic scheme to its steady state values is accomplished with

```
INITIAL {
      SOLVE scheme STEADYSTATE sparse
}
```

Appropriate `CONSERVE` statements should be part of the scheme (see the following discussion of the `KINETIC` block) so that the equivalent system of ODEs is linearly independent. It should be kept in mind that source fluxes (constant for infinite time) have a strong effect on the steady state. Finally, it is crucial to test the scheme in NEURON under conditions in which the correct behavior is known.

The `KINETIC` block

The voltage-dependent rate constants are computed in the separate procedure `rates()`. That procedure computes the equilibrium constants `K1` and `K2` from the constants `k1`, `d1`, `k2`, and `d2`, whose empirically-determined default values are given in the `PARAMETER`s block, and membrane potential `v`. The time constants `tau1` and `tau2`, however, are found from tables created under hoc (see ***The `FUNCTION_TABLE`s*** below).

The other item of note in this block is the `CONSERVE` statement. As mentioned above in **General comments about kinetic schemes**, the fundamental idea is to systematically account for conservation of material. When there is neither a source nor a sink reaction for a `STATE`, the differential equations are not linearly independent when calculating steady states (`dt` approaches

infinity). For example, in `scheme1` above the steady state condition `m' = mc' = 0` yields two identical equations. Steady states can be approximated by integrating for several steps from any initial condition with large `dt`, but roundoff error can be a problem if the Jacobian matrix is nearly singular. To solve the equations while maintaining strict numerical conservation throughout the simulation (no accumulation of roundoff error), the user is allowed to explicitly specify conservation equations with the `CONSERVE` statement. The conservation law for `scheme1` is expressed as

```
CONSERVE m + mc = 1
```

The `CONSERVE` statement does not add to the information content of a kinetic scheme and should be considered only as a hint to the translator. The NMODL translator uses this algebraic equation to replace the ODE for the last `STATE` on the left side of the equal sign. If one of the `STATE` names is an array, the conservation equation will contain an implicit sum over the array. If the last `STATE` is an array, then the ODE for the last `STATE` array element will be replaced by the algebraic equation. The choice of which `STATE` ODE is replaced by the algebraic equation is implementation-dependent and does not affect the solution (to within roundoff error). If a `CONSERVE`d `STATE` is relative to a compartment size, then compartment size is implicitly taken into account for the `STATE`s on the left hand side of the `CONSERVE` equation (see Example 8 for discussion of the `COMPARTMENT` statement). The right hand side is merely an expression, in which any necessary compartment sizes must be included explicitly.

Thus in a calcium pump model

$$\mathrm{Ca}_{\mathrm{int}} + \mathrm{Pump} \underset{k_2}{\overset{k_1}{\rightleftarrows}} \mathrm{Ca} \bullet \mathrm{Pump} \underset{k_4}{\overset{k_3}{\rightleftarrows}} \mathrm{Ca}_{\mathrm{ext}} + \mathrm{Pump}$$

the pump is conserved and one could write

```
CONSERVE pump + pumpca = total_pump * pumparea
```

The `FUNCTION_TABLE`s

As noted above, the steady-state clamp data define the voltage dependence of *K1* and *K2*, but a complete description of the K^+ current requires analysis of the temporal properties of the clamp current to determine the rate factors at each of the command potentials. The result would be a list or table of membrane potentials with associated time constants. One way of dealing with these numeric values would be to fit them with a pair of approximating functions, but the tactic used in this example is to leave them in tabular form for NMODL's `FUNCTION_TABLE` to deal with.

This is done by placing the numeric values in three hoc `Vector`s, say `v_vec`, `tau1_vec`, and `tau2_vec`, where the first is the list of voltages and the other two, at corresponding indices, give the time constants. These `Vector`s would be attached to the `FUNCTION_TABLE`s of this model with the hoc commands

```
table_tau1_k3st(tau1_vec, v_vec)
table_tau2_k3st(tau2_vec, v_vec)
```

Then whenever `tau1(x)` is called in the NMODL file, or `tau1_k3st(x)` is called from hoc, the interpolated value of the array is returned.

A useful feature of `FUNCTION_TABLE`s is that prior to developing the `Vector` database, they can be attached to a scalar value as in

```
table_tau1_k3st(100)
```

effectively becoming constant functions. Also `FUNCTION_TABLE`s can be declared with two arguments and doubly dimensioned hoc arrays attached to them. The latter is useful, for example, with voltage- and calcium-sensitive rates. In this case the table is linearly interpolated in both dimensions.

Usage

Inserting this mechanism into a section makes the `STATE`s `c1_k3st`, `c2_k3st`, and `o_k3st` available at the hoc level, as well as the conductances `gbar_k3st` and `g_k3st`.

Example 8: calcium diffusion with buffering

This mechanism illustrates how to use kinetic schemes to model intracellular Ca^{2+} diffusion and buffering. It differs from the prior example in several important aspects: Ca^{2+} is not conserved but instead enters as a consequence of the transmembrane Ca^{2+} current; diffusion involves the exchange of Ca^{2+} between compartments of unequal size; Ca^{2+} is buffered.

Only free Ca^{2+} is assumed to be mobile, whereas bound Ca^{2+} and free buffer are stationary. The Ca^{2+} buffer concentration and rate constants are based on the bullfrog sympathetic ganglion cell model described by Yamada et al. (1989). For a thorough treatment of numeric solution of the diffusion equations the reader is referred to Oran and Boris (1987).

Modeling diffusion with kinetic schemes

Diffusion is modeled as the exchange of Ca^{2+} between adjacent compartments. For radial diffusion, the compartments are a series of concentric shells around a cylindrical core, as shown in Fig.9 for `Nannuli` = 4. The index of the outermost shell is 0 and the index of the core is `Nannuli` – 1. The outermost shell is half as thick as the others so that $[Ca^{2+}]$ will be second-order correct with respect to space at the surface of the segment. Concentration is also second-order correct midway through the thickness of the other shells and at the center of the core. These depths are indicated by "x" in Fig.9. The radius of the cylindrical core equals the thickness of the outermost shell, and the intervening `Nannuli` – 2 shells each have thickness Δr = `diam` / 2 (`Nannuli` – 1), where `diam` is the diameter of the segment.

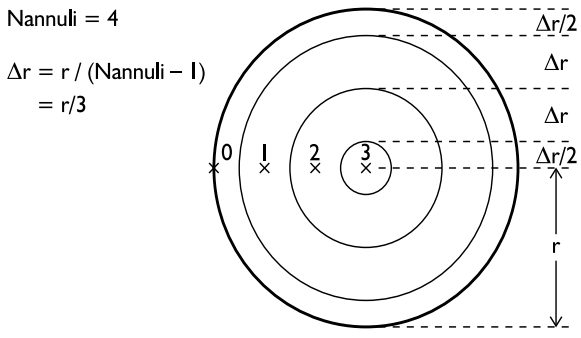

Figure 9

Because segment diameter and the number of shells affect the dimensions of the shells, they also affect the time course of diffusion. The flux between adjacent shells is $\Delta[Ca^{2+}]\ D_{Ca}\ A / \Delta r$,

where $\Delta[Ca^{2+}]$ is the concentration difference between the shell centers, D_{Ca} is the diffusion coefficient for Ca^{2+}, A is the area of the boundary between shells, and Δr is the distance between their centers. This suggests that diffusion can be described by the basic kinetic scheme

```
FROM i = 0 TO Nannuli-2 {
     ~ ca[i] <-> ca[i+1] (f[i+1], f[i+1])
}
```

where `Nannuli` is the number of shells, `ca[i]` is the concentration midway through the thickness of shell `i` (except for `ca[0]` which is the concentration at the outer surface of shell 0), and the rate constants `f[i+1]` equal $D_{Ca}\ A_{i+1} / \Delta r$. For each adjacent pair of shells, both A_{i+1} and Δr are directly proportional to segment diameter. Therefore the ratios $A_{i+1} / \Delta r$ depend only on shell index, i.e. once they have been computed for one segment, they can be used for all segments that have the same number of radial compartments regardless of segment diameter.

As it stands, this kinetic scheme is dimensionally incorrect. Dimensional consistency requires that the product of `STATE`s and rates be in units of `STATE` per time. In the present example the `STATE`s `ca[]` are intensive variables (concentration, or mass/volume), so the product of `f[]` and `ca[]` must be in units of concentration per time. However, the rates have units of volume per time, so this product is in units of mass per time, i.e. a flux that signifies the rate at which Ca^{2+} is entering or leaving a compartment. This flux is the time derivative of an extensive variable.

This disparity is corrected by specifying `STATE` volumes with the `COMPARTMENT` statement, as in

```
COMPARTMENT volume {state1 state2 . . . }
```

where the `STATE`s named in the braces have the same compartment volume given by the *`volume`* expression after the `COMPARTMENT` keyword. The volume merely multiplies the *d*`STATE`*/dt* left hand side of the equivalent differential equations, converting it to an extensive quantity and making it consistent with flux terms in units of absolute quantity per time.

The volume of each cylindrical shell depends on its index and the total number of shells, and is proportional to the square of segment diameter. Consequently the volumes can be computed once for a segment with unit diameter and then scaled by `diam^2` for use in each segment that has the same `Nannuli`.

The equations that describe the radial movement of Ca^{2+} are independent of segment length. Therefore it is convenient to express shell volumes and surface areas in units of μm^2 (volume/length) and μm (area/length), respectively.

```
: Calcium ion accumulation with radial and longitudinal diffusion

NEURON {
   SUFFIX cadifus
   USEION ca READ cai, ica WRITE cai
   GLOBAL vrat, TotalBuffer  : vrat must be GLOBAL--see INITIAL block
                             : however TotalBuffer may be RANGE
}

DEFINE Nannuli 4
```

```
UNITS {
   (molar) = (1/liter)
   (mM)    = (millimolar)
   (um)    = (micron)
   (mA)    = (milliamp)
   FARADAY = (faraday)  (10000 coulomb)
   PI      = (pi)       (1)
}

PARAMETER {
   DCa   = 0.6 (um2/ms)
   k1buf = 100 (/mM-ms) : Yamada et al. 1989
   k2buf = 0.1 (/ms)
   TotalBuffer = 0.003  (mM)
}

ASSIGNED {
   diam      (um)
   ica       (mA/cm2)
   cai       (mM)
   vrat[Nannuli]  : numeric value of vrat[i] equals the volume
                  : of annulus i of a 1um diameter cylinder
                  : multiply by diam^2 to get volume per um length
   Kd        (/mM)
   B0        (mM)
}

STATE {
   : ca[0] is equivalent to cai
   : ca[] are very small, so specify absolute tolerance
   ca[Nannuli]       (mM) <1e-10>
   CaBuffer[Nannuli] (mM)
   Buffer[Nannuli]   (mM)
}

BREAKPOINT { SOLVE state METHOD sparse }

LOCAL factors_done

INITIAL {
   if (factors_done == 0) {  : flag becomes 1 in the first segment
      factors_done = 1       :   all subsequent segments will have
      factors()              :   vrat = 0 unless vrat is GLOBAL
   }

   Kd = k1buf/k2buf
   B0 = TotalBuffer/(1 + Kd*cai)

   FROM i=0 TO Nannuli-1 {
      ca[i] = cai
      Buffer[i] = B0
      CaBuffer[i] = TotalBuffer - B0
   }
}
```

```
LOCAL frat[Nannuli]  : scales the rate constants for model geometry

PROCEDURE factors() {
   LOCAL r, dr2
   r = 1/2                : starts at edge (half diam)
   dr2 = r/(Nannuli-1)/2  : full thickness of outermost annulus,
                          : half thickness of all other annuli
   vrat[0] = 0
   frat[0] = 2*r
   FROM i=0 TO Nannuli-2 {
      vrat[i] = vrat[i] + PI*(r-dr2/2)*2*dr2  : interior half
      r = r - dr2
      frat[i+1] = 2*PI*r/(2*dr2)  : outer radius of annulus
                                  : div by distance between centers
      r = r - dr2
      vrat[i+1] = PI*(r+dr2/2)*2*dr2  : outer half of annulus
   }
}

LOCAL dsq, dsqvol  : can't define local variable in KINETIC block
                   :   or use in COMPARTMENT statement

KINETIC state {
   COMPARTMENT i, diam*diam*vrat[i] {ca CaBuffer Buffer}
   LONGITUDINAL_DIFFUSION i, DCa*diam*diam*vrat[i] {ca}
   ~ ca[0] << (-ica*PI*diam/(2*FARADAY))  : ica is Ca efflux
   FROM i=0 TO Nannuli-2 {
      ~ ca[i] <-> ca[i+1]  (DCa*frat[i+1], DCa*frat[i+1])
   }
   dsq = diam*diam
   FROM i=0 TO Nannuli-1 {
      dsqvol = dsq*vrat[i]
      ~ ca[i] + Buffer[i] <-> CaBuffer[i]  (k1buf*dsqvol, k2buf*dsqvol)
   }
   cai = ca[0]
}
```

Listing 8. `cadif.mod`

The `NEURON` block

This model `READ`s `cai` to initialize the buffer (see ***The `INITIAL` block***), and it `WRITE`s `cai` because it computes [Ca^{2+}] in the outermost shell during a simulation run. It also `READ`s `ica`, which is the Ca^{2+} influx into the outermost shell.

There are two `GLOBAL`s. One is the total buffer concentration `TotalBuffer`, which is assumed to be uniform throughout the cell. The other is `vrat`, an array whose elements will be the numeric values of the (volume/length) of the shells for a segment with unit diameter. These values are computed by `PROCEDURE factors()` near the end of Listing 8. As noted above, a segment with diameter `diam` has shells with volume/length equal to `diam^2 * vrat[i]`.

Because each instance of this mechanism has the same number of shells, the same `vrat[i]` can be used to find the shell volumes at each location in the model cell where the mechanism exists.

The `DEFINE` statement sets the number of shells to 4. Many of the variables in this model are arrays, and NMODL arrays are not dynamic. Instead, their lengths must be specified when the NMODL code is translated to C.

The `UNITS` block

Faraday's constant is scaled here in order to avoid having to include this scale factor as a separate term in the statement in the `KINETIC` block where transmembrane current `ica` is reckoned as the efflux of Ca^{2+} from the outermost shell. Since each statement in a `UNITS` block must include an explicit assertion of the units that are involved, the statement that assigns the value 3.141 . . . to `PI` includes a `(1)` which signifies that this is a dimensionless constant.

Variable declaration blocks

The `ASSIGNED` block

The variable `vrat` is declared to be an array with `Nannuli` elements. As with C, array indices run from 0 to `Nannuli` – 1. The variables `Kd` and `B0` are the dissociation constant for the buffer and the initial value of free buffer, which are computed in the `INITIAL` block (see below). Both the total buffer and the initial concentration of Ca^{2+} are assumed to be uniform throughout all shells, so a scalar is used for `B0`.

The `STATE` block

In addition to diffusion, this mechanism involves Ca^{2+} buffering that follows the reaction

$$\mathrm{Ca} + \mathrm{Buffer} \underset{k_2^{buf}}{\overset{k_1^{buf}}{\rightleftarrows}} \mathrm{Ca} \bullet \mathrm{Buffer}$$

This takes place in each of the shells, so `ca`, `Buffer` and `CaBuffer` are all arrays.

The declaration of `ca[]` uses the syntax *state* (*units*) <*absolute_tolerance*> to specify the absolute tolerance that will be employed by CVODE. The solver attempts to use a step size for which the local error ε_i for each $state_i$ satisfies at least one of these two inequalities:

$$\varepsilon_i < relative\_tolerance \cdot |state_i|$$

or

$$\varepsilon_i < absolute\_tolerance$$

The default values for these tolerances are 0 and 10^{-2}, respectively, so only a `STATE` that is extremely small (such as intracellular $[Ca^{2+}]$) needs to have its absolute tolerance specified. As an alternative to specifying a smaller absolute tolerance, `ca[]` could have been defined in terms of units such as micromolar or nanomolar, which would have increased the numeric value of these variables. This would necessitate a change of scale factors in many of the statements that involve `ca[]`. For example, the assignment for `cai` (which is required to be in mM) would be `cai = (1e-6)*ca[0]`.

LOCAL variables declared outside of equation definition blocks

A LOCAL variable that is declared outside of an equation definition block is equivalent to a static variable in C. That is, it is visible throughout the mechanism (but not at the hoc level), it retains its value, and it is shared between all instances of a given mechanism. The initial value of such a variable is 0.

This particular mechanism employs four variables of this type: factors_done, frat[], dsq, and dsqvol. The meaning of each of these is discussed below.

Equation definition blocks

The INITIAL block

Initialization of this mechanism is a two step process. The first step is to use PROCEDURE factors() (see below) to set up the geometry of the model by computing the scale factor arrays vrat[] and frat[] that are applied to the shell volumes and rate constants. This only has to be done once because the same scale factors are used for all segments that have the same number of shells, as noted above in **Modeling diffusion with kinetic schemes**. The variable factors_done is a flag that indicates whether vrat[] and frat[] have been computed. The NMODL keyword LOCAL means that the value of factors_done will be the same in all instances of this mechanism, but that it will not be visible at the hoc level. Therefore factors() will be executed only once, regardless of how many segments contain the cadifus mechanism.

The second step is to initialize the mechanism's STATEs. This mechanism assumes that the total buffer concentration and the initial free calcium concentration are uniform in all shells, and that buffering has reached its steady-state. Therefore the initial concentration of free buffer is computed from the initial [Ca^{2+}] and the buffer's dissociation constant. It should be noted that the value of cai will be set to cai0_ca_ion just prior to executing the code in the INITIAL block (see also ***The INITIAL block*** in **Example 6: extracellular potassium accumulation**).

It may be instructive to compare this initialization strategy with the approach that was used for the voltage-gated current of Listing 7 (k3st.mod). That previous example initialized the STATEs through numeric solution of a kinetic scheme, so its KINETIC block required a CONSERVE statement to ensure that the equivalent system of ODEs would be linearly independent. Here, however, the STATEs are initialized by explicit algebraic assignment, so no CONSERVE statement is necessary.

PROCEDURE factors()

The arrays vrat[] and frat[], which are used to scale the shell volumes and rate constants to ensure consistency of units, are computed here. The elements of vrat[] are the volumes of a set of concentric cylindrical shells, whose total volume equals the volume of a cylinder with diameter and length of 1 μm. These values are computed in two stages by the FROM i=0 TO Nannuli-2 { } loop. The first stage finds the volume of the outer half and the second finds the volume of the inner half of the shell.

The frat array is declared to be LOCAL because it applies to all segments that have the cadifus mechanism, but it is unlikely to be of interest to the user and therefore does not need to be visible at the hoc level. This contrasts with vrat, which is declared as GLOBAL within the NEURON block so that the user can see its values. The values frat[i+1] equal $A_{i+1} / \Delta r$, where

A_{i+1} is the surface area between shells `i` and `i+1` for $0 \le$ `i` $<$ `Nannuli`, and Δr is the distance between shell centers (radius / (`Nannuli` – 1)).

The `KINETIC` block

The first statement in this block specifies the shell volumes for the `STATE`s `ca`, `CaBuffer`, and `Buffer`. As noted above in **Modeling diffusion with kinetic schemes**, these volumes equal the elements of `vrat[]` multiplied by the square of the segment diameter. Because this mechanism involves many compartments whose relative volumes are specified by the elements of an array, this example takes care of all compartments with a single statement of the form

`COMPARTMENT` *index*`,` *volume*`[`*index*`] { `*state1 state2 . . .* `}`

where the `STATE`s that are diffusing are listed inside the braces.

Next in this block is a `LONGITUDINAL_DIFFUSION` statement, which specifies that this mechanism includes nonlocal diffusion, i.e. longitudinal diffusion along a section and into connecting sections. The syntax for scalar `STATE`s is

`LONGITUDINAL_DIFFUSION` *flux_expr* `{` *state1 state2 . . .* `}`

where *flux_expr* is the product of the diffusion constant and the cross-sectional area between adjacent compartments. Units of the *flux_expr* must be (micron^4/ms), i.e. the diffusion constant has units of (micron^2/ms) and the cross-sectional area has units of (micron^2). For cylindrical shell compartments, the cross-sectional area is just the volume per unit length. If the states are arrays then all elements are assumed to diffuse between corresponding volumes in adjacent segments and the iteration variable must be specified as in

`LONGITUDINAL_DIFFUSION` *index*`,` *flux_expr*`(`*index*`) {` *state1 state2 . . .* `}`

A `COMPARTMENT` statement is also required for the diffusing `STATE`s and the units must be (micron^2), i.e. (micron^3/micron).

The compactness of `LONGITUDINAL_DIFFUSION` specification contrasts nicely with the great deal of trouble imposed on the computational methods used to solve the equations. The standard fixed time step implicit method, historically the default method used by NEURON, can no longer find steady states with extremely large (e.g. 10^9 ms) steps since not every Jacobian element for both flux and current with respect to voltage and concentration is presently accurately computed. The CVODE method works well for these problems since it does not allow `dt` to grow beyond the point of numerical instability. In the presence of these occasional limitations on numerical efficiency, it is satisfying that, as methods evolve to handle these problems more robustly, the specification of the models does not change.

The third statement in this block is equivalent to a differential equation that describes the contribution of transmembrane calcium current to Ca^{2+} in the outermost shell. The `<<` signifies an explicit flux. Because of the `COMPARTMENT` statement, the left hand side of the differential equation is not $d[Ca^{2+}]_0/dt$ but d(*total* Ca^{2+} in the outermost shell)$/dt$. This is consistent with the right hand side of the equation, which is in units of mass per time.

Next is the kinetic scheme for radial diffusion. The rate constants in this scheme equal the product of `DCa` and the factor `frat[]` for reasons that were explained above in **Modeling diffusion with kinetic schemes**.

It may not be immediately clear why the rate constants in the kinetic scheme for Ca^{2+} buffering are scaled by the compartment volume `dsqvol`; however, the reason will become obvious when one recalls that the `COMPARTMENT` statement at the beginning of the `KINETIC` block has converted the units of the *d*`STATE`*/dt* on the left hand side of the equivalent differential equations from concentration per time to mass per time. If the reaction rate constants were left unchanged, the right hand side of the differential equations for buffering would have units of concentration per time, which is inconsistent. Multiplying the rate constants by compartment volume removes this inconsistency by changing the units of the right hand side to mass per time.

The last statement in the `KINETIC` block updates the value of `cai` from `ca[0]`. This is necessary because intracellular [Ca^{2+}] is known elsewhere in NEURON as `cai`, e.g. to other mechanisms and to NEURON's internal routine that computes E_{Ca}.

When developing a new mechanism or making substantive changes to an existing mechanism, it is generally advisable to check for consistency of units with `modlunit`. Given the dimensional complexity of this model, such testing is absolutely indispensable.

Usage

If this mechanism is inserted in a section, the concentrations of Ca^{2+} and the free and bound buffer in all compartments will be available through hoc as `ca_cadifus[]`, `Buffer_cadifus[]`, and `CaBuffer_cadifus[]`. These `STATE`s will also be available for plotting and analysis through the GUI.

The `PARAMETER`s `DCa`, `k1buf`, `k2buf`, and `TotalBuffer` will also be available for inspection and modification through both the graphical interface and hoc statements (with the `_cadifus` suffix). All `PARAMETER`s are `GLOBAL`s by default, i.e. they will have the same values in each location where the `cadifus` mechanism has been inserted. Therefore in a sense it is gratuitous to declare in the `NEURON` block that `TotalBuffer` is `GLOBAL`. However, this declaration does serve the purpose of underscoring the nature of this important variable which is likely to be changed by the user.

In some cases it might be useful for one or more of the `PARAMETER`s to be `RANGE` variables. For example, `TotalBuffer` and even `DCa` or the buffer rate constants might not be uniform throughout the cell. To make `TotalBuffer` and `DCa` `RANGE` variables only requires replacing the line

```
GLOBAL vrat, TotalBuffer
```

in the `NEURON` block with

```
GLOBAL vrat
RANGE TotalBuffer, DCa
```

The `GLOBAL` volume factors `vrat[]` are available through hoc for inspection, but it is inadvisable to change their values because they would likely be inconsistent with the `frat[]` values and thereby cause errors in the simulation.

All occurrences of this mechanism will have the same number of shells, regardless of the physical diameter of the segments in which the mechanism has been inserted. With `Nannuli` = 4, the thickness of the outermost shell will be ≤ 1 μm in segments with `diam` ≤ 6 μm. If this spatial resolution is inadequate, or if the model has segments with larger diameters, then `Nannuli` may have to be increased. NMODL does not have dynamic arrays, so in order to change the number of shells one must recompile the mechanism after assigning a new value to `Nannuli` by editing the NMODL source code.

Example 9: a calcium pump

This mechanism involves a calcium pump that is based on the reaction scheme outlined in the description of the `KINETIC` block of **Example 7: kinetic scheme for a voltage-gated current**. It is a direct extension of the model of calcium diffusion with buffering in **Example 8: calcium diffusion with buffering**, the principal difference being that a calcium pump is present in the cell membrane. The following discussion focuses on the requisite changes in Listing 8, and the operation and use of the resulting new mechanism. For all other details the reader should refer to Example 8.

The `NEURON` block

Changes in the `NEURON` block are marked in **bold**. The first nontrivial difference from the prior example is that this mechanism `READ`s the value of `cao`, which is used in the pump reaction scheme.

```
NEURON {
      SUFFIX cdp
      USEION ca READ cao, cai, ica WRITE cai, ica
      RANGE ica_pmp
      GLOBAL vrat, TotalBuffer, TotalPump
}
```

The mechanism `WRITE`s a pump current that is attributed to `ica` so that its transmembrane Ca^{2+} flux will be factored into NEURON's calculations of $[Ca^{2+}]_i$. This current, which is a `RANGE` variable known as `ica_pmp_cdp` to the hoc interpreter, constitutes a net movement of positive charge across the cell membrane, and it follows the usual sign convention (outward current is "positive"). The pump current has a direct effect on membrane potential, which, because of the rapid activation of the pump, is manifest by a distinct delay of the spike peak and a slight increase of the postspike hyperpolarization. This mechanism could be made electrically "silent" by having it `WRITE` an equal but opposite `NONSPECIFIC` current or perhaps a current that involves some other ionic species, e.g. Na^+, K^+, or Cl^-.

The variable `TotalPump` is the total density of pump sites on the cell membrane, whether free or occupied by Ca^{2+}. Making it `GLOBAL` means that it is user adjustable, and that the pump is assumed to have uniform density wherever the mechanism has been inserted. If local variation is required, this should be a `RANGE` variable.

The UNITS block

This mechanism includes the statement `(mol) = (1)` because the density of pump sites will be specified in units of `(mol/cm2)`. The term `mole` cannot be used here because it is already defined in the units database as $6.022169 \cdot 10^{23}$.

Variable declaration blocks

The `PARAMETER` block

Five new statements have been added because this mechanism uses the rate constants of the pump reactions and the density of pump sites on the cell membrane.

```
k1 = 1      (/mM-ms)
k2 = 0.005  (/ms)
k3 = 1      (/ms)
k4 = 0.005  (/mM-ms)
: to eliminate pump, set TotalPump to 0 in hoc
TotalPump = 1e-14  (mol/cm2)
```

These particular rate constant values were chosen to satisfy two criteria: the pump influx and efflux should be equal at $[Ca^{2+}] = 50$ nM, and the rate of transport should be slow enough to allow a slight delay in accelerated transport following an action potential that included a voltage-gated Ca^{2+} current. The density `TotalPump` is sufficient for the pump to have a marked damping effect on $[Ca^{2+}]_i$ transients; lower values will reduce the ability of the pump to regulate $[Ca^{2+}]_i$.

The `ASSIGNED` block

These three additions have been made.

```
cao      (mM)
ica_pmp  (mA/cm2)
parea    (um)
```

This mechanism makes use of $[Ca^{2+}]_o$ as a constant. The pump current and the surface area over which the pump is distributed are also clearly necessary.

The CONSTANT block

Consistency of units requires explicit mention of an extracellular volume in the kinetic scheme for the pump.

```
CONSTANT { volo = 1e10  (um2) }
```

The value used here is equivalent to 1 liter of extracellular space per micron length of the cell, but the actual value is irrelevant to this mechanism because `cao` will be treated as a constant. Since the value of `volo` is not important for this mechanism, there is no need for it to be accessible through hoc commands or the GUI so it is not a `PARAMETER`. On the other hand, there is a sense in which it is an integral part of the pump mechanism, which implies that it would not be appropriate to make `volo` be a `LOCAL` variable since `LOCAL`s are intended for temporary

storage of “throwaway” values. Finally, the value of `volo` would never be changed in the course of a simulation. Therefore `volo` is declared in a `CONSTANT` block.

The `STATE` block

The densities of pump sites that are free or have bound Ca^{2+}, respectively, are represented by the two new `STATE`s

```
pump    (mol/cm2)
pumpca  (mol/cm2)
```

Equation definition blocks

The `BREAKPOINT` block

This block has one additional statement

```
BREAKPOINT {
      SOLVE state METHOD sparse
      ica = ica_pmp
}
```

The assignment `ica = ica_pmp` is needed to ensure that the pump current is reckoned in NEURON’s calculation of $[Ca^{2+}]_i$.

The `INITIAL` block

The statement

```
parea = PI*diam
```

must be included to specify the area per unit length over which the pump is distributed.

If it is correct to assume that $[Ca^{2+}]_i$ has been equal to `cai0_ca_ion` (default = 50 nM) for a long time, the initial levels of `pump` and `pumpca` can be set by using the steady-state formula

```
pump = TotalPump/(1 + (cai*k1/k2))
pumpca = TotalPump - pump
```

An alternative to this style of initialization would be to place

```
ica = 0
SOLVE state STEADYSTATE sparse
```

at the end of the `INITIAL` block, where the `ica = 0` statement is needed because the kinetic scheme interprets transmembrane Ca^{2+} currents as a source of Ca^{2+} flux. This idiom can be particularly convenient for mechanisms whose steady state solutions are difficult or impossible to express in analytical form. As noted in the discussion of the `INITIAL` block of the previous example (**Example 8: calcium diffusion with buffering**), this would require adding a `CONSERVE` statement to the `KINETIC` block to insure that the equations that describe the free and bound buffer are independent.

Both of these initializations make the explicit assumption that the net Ca^{2+} current generated by other sources equals 0, so the net pump current following initialization will also be 0. If this assumption is incorrect, as is almost certainly the case if one or more voltage-gated Ca^{2+} currents are included in the model, then $[Ca^{2+}]_i$ will start to change immediately when a simulation is started. Most often this will not be what is desired. The proper initialization of a model that contains mechanisms with complex interactions may involve performing an "initialization run" and using `SaveState` objects, as described in the discussion of the `INITIAL` block of **Example 4: a voltage-gated current**.

The STATE block

Changes in this block are marked in **bold**. The new `COMPARTMENT` statements and the scale factor `(1e10)` are required for dimensional consistency in the pump scheme.

```
KINETIC state {
   COMPARTMENT i, diam*diam*vrat[i] {ca CaBuffer Buffer}
   COMPARTMENT (1e10)*parea {pump pumpca}
   COMPARTMENT volo {cao}
   LONGITUDINAL_DIFFUSION DCa {ca}

   :pump
   ~ ca[0] + pump <-> pumpca  (k1*parea*(1e10), k2*parea*(1e10))
   ~ pumpca <-> pump + cao    (k3*parea*(1e10), k4*parea*(1e10))
   CONSERVE pump + pumpca = TotalPump * parea * (1e10)
   ica_pmp = 2*FARADAY*(f_flux - b_flux)/parea

   : all currents except pump
   ~ ca[0] << (-(ica - ica_pmp)*PI*diam/(2*FARADAY))
   FROM i=0 TO Nannuli-2 {
      ~ ca[i] <-> ca[i+1] (DCa*frat[i+1], DCa*frat[i+1])
   }
   dsq = diam*diam
   FROM i=0 TO Nannuli-1 {
      dsqvol = dsq*vrat[i]
      ~ ca[i] + Buffer[i] <-> CaBuffer[i] (k1buf*dsqvol, k2buf*dsqvol)
   }

   cai = ca[0]
}
```

The pump reaction statements implement the scheme outlined in the description of the `KINETIC` block of **Example 7: kinetic scheme for a voltage-gated current**. Also as described in that section, the `CONSERVE` statement ensures strict numerical conservation, which is helpful for convergence and accuracy.

In the steady state, the net forward flux in the first and second reactions must be equal. Even during physiologically-relevant transients, these fluxes track each other effectively instantaneously. Therefore the transmembrane Ca^{2+} flux generated by the pump is taken to be the net forward flux in the second reaction. This mechanism `WRITE`s `ica` in order to affect $[Ca^{2+}]_i$. The total transmembrane Ca^{2+} flux is the sum of the pump flux and the flux from all other

sources. Thus to make sure that `ica_pmp` is not counted twice, it is subtracted from total Ca^{2+} current `ica` in the expression that relates Ca^{2+} current to Ca^{2+} flux.

Usage

The `STATE`s and `PARAMETER`s that are available through hoc and the GUI are directly analogous to those of the `cadifus` mechanism, but they will have the suffix `_cdp` rather than `_cadifus`. The additional pump variables `pump_cdp`, `pumpca_cdp`, `ica_pmp_cdp`, and `TotalPump_cdp` will also be available and are subject to similar concerns and constraints as their counterparts in the diffusion reactions (see **Usage** in **Example 7: kinetic scheme for a voltage-gated current**).

Models with discontinuities

Discontinuities in `PARAMETER`s

In the past, abrupt changes in `PARAMETER`s and `ASSIGNED` variables, such as the sudden change in current injection during a current pulse, have been implicitly assumed to take place on a time step boundary. This is inadequate with variable time step methods because it is unlikely that a time step boundary will correspond to the onset and offset of the pulse. Worse, the time step may be longer than the pulse itself, which may thus be entirely ignored.

For these reasons, a model description must explicitly notify NEURON, via the `at_time()` function, of the times at which any discontinuities occur. The statement `at_time(`*event_time*`)` guarantees that, during simulation with a variable time step method, as `t` advances past *event_time*, the integrator will reduce the step size so that it completes at `t` = *event_time* – ε, where $\varepsilon \sim 10^{-9}$ ms. The next step resets the integrator to first order, thereby discarding any previous solution history, and immediately returns after computing all the dy_i/dt at `t` = *event_time* + ε. This is how the built-in current clamp model `IClamp` notifies NEURON of the time of onset of the pulse and its offset (see the `BREAKPOINT` block of **Example 3: an intracellular stimulating electrode**). Note that `at_time()` returns a value of 1 (“true”) only during the “infinitesimal” step that ends at `t` = *event_time* + ε; otherwise it returns 0.

During a variable time step simulation, a missing `at_time()` call may cause one of two symptoms. If a `PARAMETER` changes but returns to its original value within the same interval, the pulse may be entirely missed. More often a single discontinuity will take place within a time step interval, in which case what seems like a binary search will start for the location of the discontinuity in order to satisfy the error tolerance on the step; this, of course, is very inefficient.

Time dependent `PARAMETER` changes at the hoc interpreter level are highly discouraged because they cannot currently be properly computed in the context of variable time steps. For instance, with fixed time steps it was convenient to change `PARAMETER`s prior to `fadvance()` calls, as in

```
proc advance() {
      IClamp[0].amp = imax*sin(w*t)
      fadvance()
}
```

With variable time step methods, all time-dependent changes must be described explicitly in a model, in this case with

```
BREAKPOINT { i = imax*sin(w*t) }
```

A future version of NEURON may provide a facility to specify time dependent and discontinuous PARAMETER changes safely at the hoc level in the context of variable time step methods.

Discontinuities in STATEs

Some kinds of synaptic models process an event as a discontinuity in one or more of their STATE variables. For example, a synapse whose conductance follows the time course of an alpha function (for more detail about the alpha function itself see Rall (1977) and Jack et al. (1983)) can be implemented as a kinetic scheme in the two state model

```
KINETIC state {
      ~ a <-> g (k, 0)
      ~ g -> (k)
}
```

where a discrete synaptic event is handled as an abrupt increase of STATE a. This formulation has the attractive property that it can handle multiple streams of events with different weights, so that g will be the sum of the individual alpha functions with their appropriate onsets.

However, because of the special nature of states in variable time step ODE solvers, it is necessary not only to notify NEURON about the time of the discontinuity with the at_time(onset) call, but also to notify NEURON about any discontinuities in STATEs. If onset is the time of the synaptic event and gmax is the desired maximum conductance change, this would be accomplished by including a state_discontinuity() call in the BREAKPOINT block as follows:

```
BREAKPOINT {
      if (at_time(onset)) {
            : scale factor exp(1) = 2.718... ensures
            : that peak conductance will be gmax
            state_discontinuity(a, a + gmax*exp(1))
      }
      SOLVE state METHOD sparse
      i = g*(v - e)
}
```

The first argument to state_discontinuity() will be assigned the value of its second argument just *once* for any time step. This is important, since for several integration methods BREAKPOINT assignment statements are often executed twice to calculate the *di/dv* terms of the Jacobian matrix.

Although this synaptic model works well with deterministic stimulus trains, it is difficult for the user to supply the administrative hoc code for managing the onset and gmax variables to take advantage of the promise of “multiple streams of events with different weights.” The most important problem is how to save events that have significant delay between their generation and

their handling at time `onset`. As is, an event can be passed to this model by assigning values to `onset` and `gmax` only after the previous onset event has been handled.

Discussion of the details of how NEURON now treats streams of synaptic events with arbitrary delays and weights is beyond the scope of this paper. Let it suffice that from the local view of the postsynaptic model, the state discontinuity should no longer be handled in the `BREAKPOINT` block, and the above synaptic model is more properly written in the form

```
BREAKPOINT {
      SOLVE state METHOD sparse
      i = g*(v - e)
}

NET_RECEIVE(weight (microsiemens)) {
      state_discontinuity(a, a + weight*exp(1))
}
```

in which event distribution is handled internally from a specification of network connectivity (see next section).

General comments about synaptic models

The examples so far have been of mechanisms that are “local” in the sense that an instance of a mechanism at a particular location on the cell depends only on `STATE`s and `PARAMETER`s of the model *at that location*. Of course they normally depend on voltage and ionic variables as well, but these also are *at that location* and automatically available to the model. Synaptic models have an essential distinguishing characteristic that sets them apart: in order to properly compute their contribution to membrane current at the postsynaptic site, they require information from another place, e.g. presynaptic voltage. Models that contain `LONGITUDINAL_DIFFUSION` are perhaps also an exception, but their dependence on adjacent compartment ion concentration is handled automatically by the translator.

In the past, model descriptions could only use `POINTER` variables to obtain their presynaptic information. A `POINTER` in NMODL holds a reference to another variable; the specific reference is defined by a hoc statement such as

```
setpointer postcell.synapse.vpre, precell.axon.v(1)
```

in which `vpre` is a `POINTER`, declared in the indicated `POINT_PROCESS` synapse instance, which references the value of a specific membrane voltage, in this case at the distal end of the presynaptic axon. Gap junctions or ephaptic synapses can be handled by a pair of `POINT_PROCESS`es on the two sides of the junction that point to each other’s voltage, as in

```
section1 gap1 = new Gap(x1)
section2 gap2 = new Gap(x2)
setpointer gap1.vpre, section2.v(x2)
setpointer gap2.vpre, section1.v(x1)
```

This kind of detailed piecing together of individual components is acceptable for models with only a few synapses, but larger network models have required considerable administrative effort from users to 1) create mechanisms that handle synaptic delay, 2) exploit very great simulation efficiencies available with simplified models of synapses, and 3) maintain information about the connectivity of the network.

The experience of NEURON users — especially Alain Destexhe and William Lytton — in creating special models and procedures for managing network simulations has been incorporated in a new built-in network connection (`NetCon`) class, whose instances manage the delivery of presynaptic threshold events to postsynaptic `POINT_PROCESS`es. It is very important to note that the `NetCon` class works for all NEURON integrators, including a local variable time step method in which each cell is integrated with a time step appropriate to the state changes occurring in that cell. With this event delivery system, model descriptions of synapses never need to queue events, and they do not have to make heroic efforts to work properly with variable time step methods. These features offer enormous convenience to the user.

`NetCon` connects a presynaptic variable such as voltage to a synapse with arbitrary (individually specified on a per `NetCon` instance) delay and weight. If the presynaptic variable passes threshold at time `t`, a special `NET_RECEIVE` procedure in the postsynaptic `POINT_PROCESS` is called at time `t + delay`. The only constraint on `delay` is that it be nonnegative. Events always arrive at the postsynaptic object at the interval `delay` after the time they were generated, and there is no loss of events under any circumstances.

This new class also reduces the computational burden of network simulations, because the event delivery system for `NetCon` objects supports unlimited fan-in and fan-out (convergence and divergence). That is, many `NetCon` objects can be connected to the same postsynaptic `POINT_PROCESS` (fan-in). This yields large efficiency improvements because a single set of equations for synaptic conductance change can be shared by many streams of inputs (one input stream per connecting `NetCon` instance). Likewise, many `NetCon` objects can be connected to the same presynaptic variable (fan-out), thus providing additional efficiency improvement since the presynaptic variable is checked only once per time step and, when it crosses threshold in the positive direction, events are generated for each connecting `NetCon` object. The next example shows how a `NetCon` object might be used to establish the connection between two model neurons.

Example 10: synapse with exponential decay

The simplest useful synapse consists of an abrupt change in conductance, triggered by arrival of an event, which then decays with a single time constant. We imagine not only that the conductance summates when events arrive from different places, but that a single stream of events will also summate. The following model handles both these situations by defining a single conductance state g which is governed by a differential equation with the solution $g(t) = g(t_0)e^{(t-t_0)/\tau}$ where $g(t_0)$ is the conductance at the time of the most recent event.

```
: expsyn.mod

NEURON {
      POINT_PROCESS ExpSyn
      RANGE tau, e, i
      NONSPECIFIC_CURRENT i
}

PARAMETER {
      tau = 0.1  (ms)
      e   = 0    (millivolt)
}

ASSIGNED {
      v  (millivolt)
      i  (nanoamp)
}

STATE { g (microsiemens) }

INITIAL { g = 0 }

BREAKPOINT {
      SOLVE state METHOD cnexp
      i = g*(v - e)
}

DERIVATIVE state { g' = -g/tau }

NET_RECEIVE(weight (microsiemens)) {
      state_discontinuity(g, g + weight)
}
```

Listing 9. `expsyn.mod`

The `NET_RECEIVE` block

The new feature in this model is the `NET_RECEIVE` block, which is called by the `NetCon` event delivery system when an event arrives at this postsynaptic point process. In this case the value of the weight is specified by the particular `NetCon` object delivering the event, and this value increments the conductance state.

As noted above in **Discontinuities in `STATE`s**, `state_discontinuity()` must be called if discontinuous `STATE` changes are to work properly with the variable time step methods. The first argument of `state_discontinuity()` is interpreted as a reference to the `STATE`, and the second argument is an expression for its new value. If the variable to be changed is not a `STATE` variable, then it is safe to specify its new value with an ordinary assignment statement (see **Example 12: Use-dependent synaptic plasticity** below). Just before entry to `NET_RECEIVE` with an event to be delivered at time `t`, all `STATE`s, `v`, and values assigned in the `BREAKPOINT` block are consistent at time `t`.

Usage

Suppose we wanted to set up an `ExpSyn` synaptic connection between the two cells portrayed in Fig.10. This could be done with the following hoc code, which also illustrates the use of a `List` of `NetCon` objects as a means for keeping track of the synaptic connections in a network.

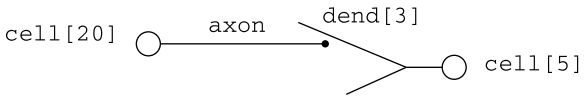

Figure 10

```
// the network will be represented
//   by a list of NetCon objects
objref ncl
ncl = new List()

// make an ExpSyn point process called syn
//   that is located on cell[5]
//   just to one side of the midpoint of dend[3]
objref syn
cell[5].dend[3] syn = new ExpSyn(0.3)

// cell[20].axon.v(1) is voltage at the presynaptic site
// connect the presynaptic cell to the ExpSyn instance syn
// via a new NetCon object
// and add the NetCon object to the list ncl
cell[20].axon ncl.append(new NetCon(&v(1), \
   syn, threshold, delay, weight)
```

Figure 11 shows graphs saved from a simulation of two input streams converging onto postsynaptic cell. The top graph indicates the presynaptic firing times (traces labeled `precell[0]` and `precell[1]`). The conductance of the `ExpSyn` mechanism and the membrane potential of the postsynaptic cell are shown in the middle and bottom graphs. For this example, the decay time constant for the synaptic conductance has been arbitrarily set to 3 ms. Temporal summation is evident in the synaptic conductance and postsynaptic membrane potential for inputs within an individual stream and between inputs on multiple streams.

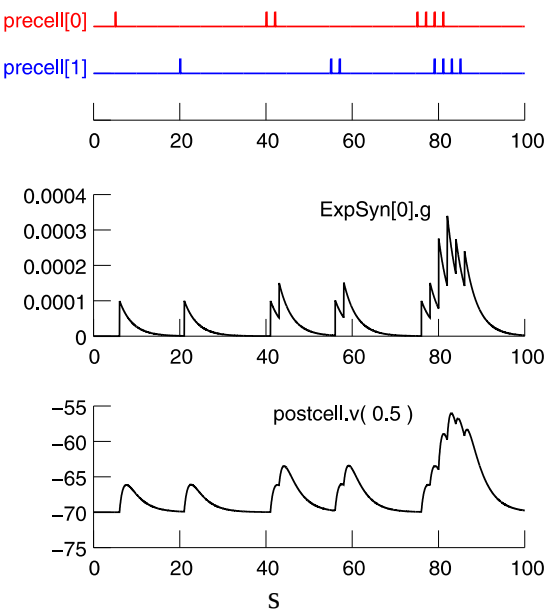

Figure 11

Example 11: alpha function synapse

It is a simple matter to extend ExpSyn to implement an alpha function synapse by replacing the differential equation with the two state kinetic scheme.

```
STATE { a (microsiemens) g (microsiemens) }
KINETIC state {
      ~ a <-> g (1/tau, 0)
      ~ g -> (1/tau)
}
```

and changing the discontinuity statement to

```
state_discontinuity(a, a + weight*exp(1))
```

The factor exp(1) = *e* is included so that an isolated event produces a peak conductance of magnitude weight, which occurs at time tau after the event. Since this mechanism involves a KINETIC block instead of a DERIVATIVE block, the integration method specified by the SOLVE statement must be changed from cnexp to sparse.

The extra computational complexity of using a kinetic scheme is offset by the fact that, no matter how many NetCon streams connect to this model, the computation time required to integrate STATE g remains constant. The only extra time is the potentially greater number of calls to the NET_RECEIVE block, which is called only when events are to be delivered. This illustrates a very useful tactic which will reappear in subsequent models: always move as much computational complexity as possible from temporal integration blocks (DERIVATIVE or KINETIC blocks) to the NET_RECEIVE block. The potential benefits are very large, since BREAKPOINT and SOLVE blocks are executed — sometimes repeatedly — at each time step, whereas statements in the NET_RECEIVE block are executed only once per delivered event. Indeed, with NEURON's variable time step methods it is possible to carry out what are essentially discrete event simulations, in which dt is always the interval between events. Since most steps reduce to an interpolation step followed by a single ODE function evaluation, this reduces the time step integration overhead to a fraction of a normal single integration step per event.

Some increase of efficiency can be gained by recasting the kinetic scheme as two linear differential equations

```
DERIVATIVE state {
      a' = -a/tau1
      b' = -b/tau
      g = b - a
}
```

which are solved efficiently by the cnexp method. As tau1 approaches tau from below, g approaches an alpha function (although the factor by which weight must be multiplied approaches infinity). Also, there are now two state discontinuities in the NET_RECEIVE block

```
state_discontinuity(a, a + weight*factor)
state_discontinuity(b, b + weight*factor)
```

Example 12: Use-dependent synaptic plasticity

Here the alpha function synapse is extended to implement a form of use-dependent synaptic plasticity. Each presynaptic event initiates two distinct processes: direct activation of ligand-gated channels, which causes a transient conductance change, and activation of a mechanism that in turn can have a modulatory effect on the conductance change produced by successive synaptic activations. Here we presume that synaptic strength is modulated by the postsynaptic increase of a second messenger, which we will call “G protein” for illustrative purposes. We must point out that this example is entirely hypothetical, and that it is quite different from models described by others (Destexhe and Sejnowski 1995) in which the G protein itself gates the ionic channels.

In this mechanism it is essential to distinguish each stream into the generalized synapse, since each stream has to maintain its own [G] (concentration of activated G protein). That is, streams are independent of each other in terms of the effect on [G], but their effects on synaptic conductance show linear superposition.

```
: gsyn.mod

NEURON {
   POINT_PROCESS GSyn
   RANGE tau1, tau2, e, i
   RANGE Gtau1, Gtau2, Ginc
   NONSPECIFIC_CURRENT i
   RANGE g
}

UNITS {
   (nA)   = (nanoamp)
   (mV)   = (millivolt)
   (umho) = (micromho)
}

PARAMETER {
   tau1   = 1  (ms)
   tau2   = 1.05    (ms)
   Gtau1  = 20   (ms)
   Gtau2  = 21   (ms)
   Ginc   = 1
   e      = 0    (mV)
}

ASSIGNED {
   v  (mV)
   i  (nA)
   g  (umho)
   factor
   Gfactor
}
```

```
STATE {
   A  (umho)
   B  (umho)
}

INITIAL {
   LOCAL tp
   A = 0
   B = 0
   tp = (tau1*tau2)/(tau2 - tau1) * log(tau2/tau1)
   factor = -exp(-tp/tau1) + exp(-tp/tau2)
   factor = 1/factor
   tp = (Gtau1*Gtau2)/(Gtau2 - Gtau1) * log(Gtau2/Gtau1)
   Gfactor = -exp(-tp/Gtau1) + exp(-tp/Gtau2)
   Gfactor = 1/Gfactor
}

BREAKPOINT {
   SOLVE state METHOD cnexp
   g = B - A
   i = g*(v - e)
}

DERIVATIVE state {
   A' = -A/tau1
   B' = -B/tau2
}

NET_RECEIVE(weight (umho), w, G1, G2, t0 (ms)) {
   G1 = G1*exp(-(t-t0)/Gtau1)
   G2 = G2*exp(-(t-t0)/Gtau2)
   G1 = G1 + Ginc*Gfactor
   G2 = G2 + Ginc*Gfactor
   t0 = t

   w = weight*(1 + G2 - G1)
   state_discontinuity(A, A + w*factor)
   state_discontinuity(B, B + w*factor)
}
```

Listing 10. `gsyn.mod`

The conductance of the ligand-gated ion channel uses the differential equation approximation for an alpha function synapse. The peak synaptic conductance depends on the value of [G] at the moment of synaptic activation. A similar, albeit much slower, alpha function approximation describes the time course of [G]. These processes peak approximately `tau1` and `Gtau1` after delivery of an event, respectively.

The peak synaptic conductance of an active `NetCon` is specified in the `NET_RECEIVE` block, where `w = weight*(1 + G2 - G1)` describes how the effective weight of the synapse is modified by [G]. Even though conductance is integrated, [G] is needed only at discrete event times so it can be computed analytically from the elapsed time since the prior synaptic activation.

The INITIAL block performs the tedious task of setting up the factors which are needed to make the peak changes equal to the values of w and Ginc.

Note that G1 and G2 do not need a state_discontinuity() to change them because they are not STATEs in this mechanism. They are not even variables in this mechanism, but instead are "owned" by the particular NetCon instance that delivered the event.

A NetCon object instance keeps an array of size equal to the number of arguments to NET_RECEIVE, and the arguments to NET_RECEIVE are really references to the elements of this array. The fact that the arguments are "call by reference," instead of the normal "call by value," is what allows this model to work: it allows assignment statements in gsyn.mod to change the values of variables that belong to the NetCon object. Since there is a separate array for each NetCon object that connects to this model, [G] can be different for different connections. However the individual NetCon objects all contribute linearly to the synaptic conductance.

Example 13: saturating synapses

Several authors (e.g. Destexhe et al. (1994a), Lytton (1996)) have found it useful to approximate a wide range of synaptic behavior by explicitly parameterizing the conductance change as a single time constant onset with specific duration (Cdur, interpreted as the duration of a transmitter pulse) followed by a separate time constant offset. The conductance changes elicited by separate streams summate, whereas repetitive impulses on one stream produce a saturating conductance change (steady state for a long onset time). We resolve the ambiguity of what to do when multiple spikes arrive on a single stream during the Cdur onset of an earlier spike (i.e. ignore, concatenate Cdur to make the transmitter pulse longer without increasing its concentration, or summate the transmitter) by choosing concatenation. Summation of transmitter is outside the scope of the Destexhe/Lytton model since that formulation demands identical onset time constants for all conductance changes and the onset time constant is proportional to transmitter concentration.

Although the idea of saturation can be captured with a model of the form used in the previous example, the separate onset/offset formulation requires keeping track of how much "material" in each stream is in the offset or onset state. The wrinkle here is that when an event arrives at time t to start an onset, another event must be generated to occur at time t+Cdur to start turning it off. To complicate matters further, other spikes on the same input line (same NetCon) may arrive before t+Cdur, which means that the offset event at t+Cdur should be ignored. The only time an offset event takes effect is if no other spikes occurred in the previous Cdur interval.

The NMODL implementation for this mechanism is given in Listing 11.

```
: ampa.mod

NEURON {
   POINT_PROCESS AMPA_S
   RANGE R, g
   NONSPECIFIC_CURRENT i
   GLOBAL Cdur, Alpha, Beta, Erev, Rinf, Rtau
}
```

```
UNITS {
   (nA)   = (nanoamp)
   (mV)   = (millivolt)
   (umho) = (micromho)
   (mM)   = (milli/liter)
}

PARAMETER {
   Cdur  = 0.3   (ms)  : transmitter duration (rising phase)
   Alpha = 0.94  (/ms) : forward (binding) rate
   Beta  = 0.18  (/ms) : backward (dissociation) rate
   Erev  = 0     (mV)  : equilibrium potential
}

ASSIGNED {
   v     (mV)   : postsynaptic voltage
   i     (nA)   : current = g*(v - Erev)
   g     (umho) : conductance
   Rinf         : steady state channels open
   Rtau  (ms)   : time constant of channel binding
   synon
}

STATE { Ron Roff }  : initialized to 0 by default

INITIAL {
   Rinf = Alpha / (Alpha + Beta)
   Rtau = 1 / (Alpha + Beta)
   synon = 0
}

BREAKPOINT {
   SOLVE release METHOD cnexp
   g = (Ron + Roff)*1(umho)
   i = g*(v - Erev)
}

DERIVATIVE release {
   Ron' = (synon*Rinf - Ron)/Rtau
   Roff' = -Beta*Roff
}
```

```
: on initialization, all arguments after the first one
:   are set to 0
NET_RECEIVE(weight, on, nspike, r0, t0 (ms)) {
   : flag is an implicit argument of NET_RECEIVE, normally 0
   if (flag == 0) {
      : a spike, so turn on if not already in a Cdur pulse
      nspike = nspike + 1
      if (!on) {
            r0 = r0*exp(-Beta*(t - t0))
            t0 = t
            on = 1
            synon = synon + weight
            state_discontinuity(Ron, Ron + r0)
            state_discontinuity(Roff, Roff - r0)
      }
      : come again in Cdur with flag = current value of nspike
      net_send(Cdur, nspike)
   }
   if (flag == nspike) {
      : if this associated with last spike then turn off
      r0 = weight*Rinf + (r0 - weight*Rinf)*exp(-(t - t0)/Rtau)
      t0 = t
      synon = synon - weight
      state_discontinuity(Ron, Ron - r0)
      state_discontinuity(Roff, Roff + r0)
      on = 0
   }
}
```

Listing 11. `ampa.mod`

Details of saturating mechanisms *per se* are covered by Destexhe et al. (1994a; 1994b) and Lytton (1996). Here we focus on how the `NET_RECEIVE` block is used to manage multiple input streams. An onset event, generated by the system when the connecting `NetCon`'s source passed threshold `t - delay` ago, always has an implicit argument called `flag` which is set to 0 and is *call by value* as opposed to the explicit arguments, which are "call by reference." The `nspike` variable counts the spikes that have taken place on the individual `NetCon` lines. A spike onset event (`flag` = 0) results in a `net_send()` call, which will generate an event with delay given by the first argument and flag value given by the second argument. All the explicit arguments will have the value of this particular `NetCon`, and therefore `flag` will only match `nspike` when there is no intervening spike event (on this `NetCon` line).

DISCUSSION

The model description framework has proven to be a useful, efficient, and flexible way to implement computational models of biophysical mechanisms. The leverage that NMODL provides to the user is amplified by its platform-independence, since it runs in the MacOS, MSWindows, and UNIX/Linux environments. Another important factor is consistency of high-level syntax, which allows it to incorporate advances in numerical methods in a way that is transparent to the user.

NMODL continues to undergo revision and improvement in response to the evolving needs of computational neuroscience, particularly in the domain of empirically-based modeling. One recent example of the extension of NMODL to encompass new kinds of mechanisms is longitudinal diffusion. Another is kinetic schemes in a form that can be interpreted as Markov processes (Colquhoun and Hawkes 1981), i.e. linear schemes, which are now translated into single channel models. By removing arbitrary limits related to programming complexity, such advances give NEURON the ability to accommodate insights derived from new experimental findings, and enable modeling to keep pace with the broad arena of “wet-lab” neuroscience.

ACKNOWLEDGMENTS

This work was supported in part by NIH grant NS11613. We wish to thank John Moore for inspiration and encouragement, Alain Destexhe and William Lytton for invaluable contributions to the convenient and efficient simulation of networks, Ragnhild Halvorsrud for helpful suggestions regarding this manuscript, and the many users of NEURON who have provided indispensable feedback, presented challenging problems that have stimulated new advances in the program, and developed their own enhancements.

REFERENCES

Colquhoun, D. and Hawkes, A.G. On the stochastic properties of single ion channels. *Philosophical Transactions of the Royal Society of London Series B* 211:205-235, 1981.

Destexhe, A., Mainen, Z.F., and Sejnowski, T.J. An efficient method for computing synaptic conductances based on a kinetic model of receptor binding. *Neural Computation* 6:14-18, 1994a.

Destexhe, A., Mainen, Z.F., and Sejnowski, T.J. Synthesis of models for excitable membranes, synaptic transmission, and neuromodulation using a common kinetic formalism. *J. Comput. Neurosci.* 1:195-231, 1994b.

Destexhe, A. and Sejnowski, T.J. G-protein activation kinetics and spillover of γ-aminobutyric acid may account for differences between inhibitory responses in the hippocampus and thalamus. *Proc. Nat. Acad. Sci.* 92:9515-9519, 1995.

Durand, D. The somatic shunt cable model for neurons. *Biophys. J.* 46:645-653, 1984.

Frankenhaeuser, B. and Hodgkin, A.L. The after-effects of impulses in the giant nerve fibers of *Loligo*. *J. Physiol.* 131:341-376, 1956.

Hines, M. Efficient computation of branched nerve equations. *Int. J. Bio-Med. Comput.* 15:69-76, 1984.

Hines, M. A program for simulation of nerve equations with branching geometries. *Int. J. Bio-Med. Comput.* 24:55-68, 1989.

Hines, M. NEURON—a program for simulation of nerve equations. In: *Neural Systems: Analysis and Modeling*, edited by F. Eeckman. Norwell, MA: Kluwer, 1993, p. 127-136.

Hines, M. The NEURON simulation program. In: *Neural Network Simulation Environments*, edited by J. Skrzypek. Norwell, MA: Kluwer, 1994, p. 147-163.

Hines, M. and Carnevale, N.T. Computer modeling methods for neurons. In: *The Handbook of Brain Theory and Neural Networks*, edited by M.A. Arbib. Cambridge, MA: MIT Press, 1995, p. 226-230.

Hines, M.L. and Carnevale, N.T. The NEURON simulation environment. *Neural Computation* 9:1179-1209, 1997.

Jack, J.J.B., Noble, D., and Tsien, R.W. *Electric Current Flow in Excitable Cells*. London: Oxford University Press, 1983.

Johnston, D. and Wu, S.M.-S. *Foundations of Cellular Neurophysiology*. Cambridge, MA: MIT Press, 1995.

Kohn, M.C., Hines, M.L., Kootsey, J.M., and Feezor, M.D. A block organized model builder. *Mathematical and Computer Modelling* 19:75-97, 1994.

Kootsey, J.M., Kohn, M.C., Feezor, M.D., Mitchell, G.R., and Fletcher, P.R. SCoP: an interactive simulation control program for micro- and minicomputers. *Bulletin of Mathematical Biology* 48:427-441, 1986.

Lytton, W.W. Optimizing synaptic conductance calculation for network simulations. *Neural Computation* 8:501-509, 1996.

McCormick, D.A. Membrane properties and neurotransmitter actions. In: *The Synaptic Organization of the Brain*, edited by G.M. Shepherd. New York: Oxford University Press, 1998, p. 37-75.

Moczydlowski, E. and Latorre, R. Gating kinetics of Ca^{2+}-activated K^{+} channels from rat muscle incorporated into planar lipid bilayers. *Journal of General Physiology* 82:511-542, 1983.

Oran, E.S. and Boris, J.P. *Numerical Simulation of Reactive Flow*. New York: Elsevier, 1987.

Rall, W. Core conductor theory and cable properties of neurons. In: *Handbook of Physiology, vol. 1, part 1: The Nervous System*, edited by E.R. Kandel. Bethesda, MD: American Physiological Society, 1977, p. 39-98.

Staley, K.J., Otis, T.S., and Mody, I. Membrane properties of dentate gyrus granule cells: comparison of sharp microelectrode and whole-cell recordings. *J. Neurophysiol.* 67:1346-1358, 1992.

Wilson, M.A. and Bower, J.M. The simulation of large scale neural networks. In: *Methods in Neuronal Modeling*, edited by C. Koch and I. Segev. Cambridge, MA: MIT Press, 1989, p. 291-333.

Yamada, W.M., Koch, C., and Adams, P.R. Multiple channels and calcium dynamics. In: *Methods in Neuronal Modeling*, edited by C. Koch and I. Segev. Cambridge, MA: MIT Press, 1989, p. 97-133.

INDEX