frontiers in
NEUROINFORMATICS

ORIGINAL RESEARCH ARTICLE
published: 04 February 2014
doi: 10.3389/fninf.2014.00006

Equation-oriented specification of neural models for simulations

Marcel Stimberg[1,2]*, Dan F. M. Goodman[3,4], Victor Benichoux[1,2] and Romain Brette[1,2]

[1] Laboratoire Psychologie de la Perception, CNRS and Université Paris Descartes, Paris, France
[2] Département d'Etudes Cognitives, Ecole Normale Supérieure, Paris, France
[3] Eaton Peabody Laboratory, Massachusetts Eye and Ear Infirmary, Boston, MA, USA
[4] Department of Otology and Laryngology, Harvard Medical School, Boston, MA, USA

Edited by:
Andrew P. Davison, Centre national de la recherche scientifique, France

Reviewed by:
Jochen M. Eppler, Research Center Jülich, Germany
Thomas Nowotny, University of Sussex, UK

***Correspondence:**
Marcel Stimberg, Equipe Audition, Département d'Etudes Cognitives, Ecole Normale Supérieure, 29 rue d'Ulm, 75230 Paris Cedex 05, France
e-mail: marcel.stimberg@ens.fr

Simulating biological neuronal networks is a core method of research in computational neuroscience. A full specification of such a network model includes a description of the dynamics and state changes of neurons and synapses, as well as the synaptic connectivity patterns and the initial values of all parameters. A standard approach in neuronal modeling software is to build network models based on a library of pre-defined components and mechanisms; if a model component does not yet exist, it has to be defined in a special-purpose or general low-level language and potentially be compiled and linked with the simulator. Here we propose an alternative approach that allows flexible definition of models by writing textual descriptions based on mathematical notation. We demonstrate that this approach allows the definition of a wide range of models with minimal syntax. Furthermore, such explicit model descriptions allow the generation of executable code for various target languages and devices, since the description is not tied to an implementation. Finally, this approach also has advantages for readability and reproducibility, because the model description is fully explicit, and because it can be automatically parsed and transformed into formatted descriptions. The presented approach has been implemented in the Brian2 simulator.

Keywords: python, neuroscience, computational neuroscience, simulation, software

1. INTRODUCTION

Computational simulations have become an indispensable tool in neuroscience. Today, a large number of software packages are available to specify and carry out simulations (Brette et al., 2007). In specifying neural simulations, there is a tension between two often conflicting goals: a simulator should be able to express a wide range of possible models while at the same time allowing the creation of simulations rapidly and easily. Most simulators restrict the basic model components (e.g., neurons and synapses) to a set of predefined standard models, thereby severely limiting the range of possible models.

Defining new model components can be time consuming and technically challenging, requiring the user to implement them in a low-level language such as C++, as in the NEST simulator (Gewaltig and Diesmann, 2007; neuron models can also be specified in NEST's "simulation language interpreter" language), implement them in a special purpose language, as in the NEURON simulator (Carnevale and Hines, 2006) which uses NMODL (Hines and Carnevale, 2000) for this purpose, or to specify them in a complex simulator-independent description language such as NeuroML (Gleeson et al., 2010) or NineML (Raikov et al., 2011). In addition, the definition of model components may not be entirely transparent, as it may be necessary to inspect the simulator source code to know the details of the simulated model. This makes it difficult to assess and reproduce models, and to verify that they correspond to the description of them in a publication.

In this article, we present an approach that combines extensibility with ease-of-use by using mathematical equations to describe every aspect of the neural model, including membrane potential and synaptic dynamics. We implemented this approach in Brian2, the current development version of the Python-based Brian simulator (Goodman and Brette, 2008, 2009), extending previous work which introduced the use of equations-based definitions of membrane potential dynamics. This consistent use of equations to define all aspects of the model greatly extends the scope of Brian, making it possible to run on different computing devices as well as to automatically generate an accurate description of the model to be included in the methods section of a publication.

Simulating a neural model means tracking the change of neural variables such as membrane potential or synaptic weights over time. The rules governing these changes take two principal forms: continuous updates (e.g., the decay of the membrane potential back to a resting state in the absence of inputs) and event-based updates (e.g., the reset after a spike in an integrate-and-fire neuron, or the impact of a pre-synaptic spike on a post-synaptic cell). Generally, continuous updates can be described by deterministic or stochastic differential equations, while event-based updates can be described as a series of mathematical operations. In this unified framework, it is possible to specify a very wide range of model components. With different sets of equations, neuronal models can range from variants of integrate-and-fire neurons to arbitrarily complex descriptions referring to biophysical properties

such as ion channels. In the same way, a wide range of synaptic models can fit in this framework: from simple static synapses to plastic synapses implementing short- or long-term plasticity, including spike-timing dependent rules. Finally, mathematical expressions can also be used to describe neuronal threshold conditions or synaptic connections, weights and delays. In sum, this framework based on the mathematical definition of a neural network model seen as a hybrid system allows for expressiveness while at the same time minimizing the "cognitive load" for users, because they do not have to remember the names and properties of simulator-dependent objects and functions but can describe them in a notation similar to the notation used in analytical work and publications (Brette, 2012). In section 2, we describe this general framework and show how neural and synaptic models can be described in this way.

We then explain in section 3 how an equation-oriented description of models can be transformed into runnable code, using code generation. This code generation involves two steps. The first step applies to model components described by differential equations. In most simulators, the numerical integration method (such as forward Euler or Runge-Kutta) is either fixed or part of a model component definition itself. We propose instead to describe the integration method separately, in mathematical notation. Using the capabilities of the Python symbolic manipulation library SymPy (Joyner et al., 2012), we automatically combine this update rule with the actual model equations to yield a sequence of *abstract code* statements. These are high level pseudocode statements that define a sequence of mathematical operations which abstract away the details of how they should be computed (e.g., by loops or vectorised statements), for example `a = b + c`. The second code generation step then applies to this sequence of statements and to all other explicitly given statements and transforms abstract code statements into programming language code (for example in C++) using a general code generation mechanism (Goodman, 2010).

Our approach also has important implications for the issue of reproducibility of simulation results: by making the equations underlying the model fully explicit, the source code also acts as a readable documentation of the model. In addition, giving the neural simulator access to mathematical descriptions of model equations or connection patterns allows for straightforward semi-automatic generation of model descriptions (see e.g., Nordlie et al., 2009), which we describe in section 4.

2. MODEL DESCRIPTIONS

2.1. NEURAL MODELS

Neural models are described by state variables that evolve in time. Mathematically speaking, they are hybrid systems: variables evolve both continuously (e.g., the evolution of the membrane potential between action potentials) and discontinuously through events (e.g., the reset of the membrane potential after a spike in an integrate-and-fire model, or the effect of pre-synaptic spikes). To describe a model therefore requires a system that allows for both of these components. An event is a change in the state variables of the system that is triggered by a logical condition on these variables (possibly after a delay); spikes are the most obvious type of events in neural models. But more generally, there could be other types of events, for example changes triggered when some variable (e.g., intracellular calcium) reaches a threshold value. In addition, it is common that neural models, in particular integrate-and-fire models, have different states, typically excitable and non-excitable (refractory), with different sets of differential equations. An event can then not only trigger changes in state variables but also a transition in state.

It would be possible to make such a system extremely general by allowing for an arbitrary number of general states that a neuron can be in, conditions to change between the states and descriptions of the dynamic evolution within the states (as in NeuroML, Gleeson et al., 2010). Such a system would however have the disadvantage of being very complex to use and to simulate. Therefore, we imposed restrictions so as to simplify the description framework while supporting most neural models currently used.

In the following, we have made the following simplifying choices: (1) there are only two states, active (excitable) and refractory (non-excitable); (2) there is a single type of event per state. In the active state, the only type of event is spikes. It triggers changes in state variables of the neuron (reset) and its target synapses (propagation), and triggers a transition to the refractory state. In the refractory state, the only type of event is a condition that triggers the transition to the active state. This is indisputably restrictive, but was chosen as a reasonable compromise between simplicity and generality. However, the framework could be extended to more general cases (see Discussion).

Finally, another important aspect of neural models is that some state variables represent physical quantities (e.g., the membrane potential, the membrane capacitance, etc.) that have physical units while others correspond to abstract quantities that are unitless. Therefore, to be fully explicit and to avoid any errors when dealing with variables in various units and scales, a description system should allow the user to explicitly state the units in all expressions that deal with state variables.

In sum, the description of a neuron model consists of the following four components: the model equations, the threshold condition, the reset statements and the refractoriness condition. Model equations define all the state variables with their units and describe their continuous evolution in time. The threshold condition describes when an action potential should be emitted and when the reset statements should be executed. Finally, the refractoriness condition describes under which condition a neuron can exit the refractory state. We explain in section 2.1.3 how to specify different dynamics in the refractory state. We will show that this four-component description allows for flexible specification of most neuronal models while being still simple enough to be automatically transformable into executable code.

2.1.1. Model equations

Model equations of point neurons are most naturally defined as a system of ordinary differential equations, describing the evolution of state variables. The differential equations can be non-autonomous (depend on the time t) or autonomous, and deterministic or stochastic (referring to one or more stochastic

processes). To make the equations more readable, the formalism should also allow for named sub-expressions. Note that even though some models are presented in integral form as sums of post-synaptic potentials, they can be rewritten into a system of equivalent differential equations (see e.g., Destexhe et al., 1994).

As an example, consider the following equations defining a Hodgkin-Huxley model with a passive leak current and active sodium and potassium currents for action potential generation (omitting the equations for the rate variables h, m and n):

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{I_l + I_K + I_{Na}}{C_m}$$
$$I_l = g_l\,(E_l - v)$$
$$I_{Na} = g_{Na} h m^3\,(E_{Na} - v)$$
$$I_K = g_K n^4\,(E_K - v)$$

Since this model includes the action potential generation in the dynamics it does not need a threshold or reset, except for recording or transmitting spikes.

The model equations can be readily transformed into a string description (including information about the units and general comments), see **Figure 1A** for a translation into Brian2 syntax, which includes a specification of units after the colon and also uses units (e.g., mV) in the equations themselves.

If a state variable should evolve stochastically, this can be modelled by including a reference to a stochastic white noise process ξ (or several independent such processes $\xi_1, \xi_2, \ldots$). The inclusion of a stochastic process in the model equations means that the differential equations are now stochastic, with important consequences for their numerical integration (see section 3.1). **Figure 1B** shows an example for the description of a noisy membrane equation in Brian2.

2.1.2. Threshold and reset

Integrate-and-fire models require a threshold condition and one or more reset statements. A simple leaky integrate-and-fire neuron, for example, can be described as:

$$\frac{\mathrm{d}v}{\mathrm{d}t} = -(v - v_0)/\tau_m$$
$$\text{After } v > v_{th}: \qquad v \leftarrow v_0$$

where v_0 is the cell's resting and reset potential, v_{th} is the cell's threshold and τ_m is the membrane time constant (see **Figure 1C** for a translation into Brian2 syntax).

Reset statements are not restricted to resetting variables but can perform arbitrary updates of state variables. Similarly, the threshold condition is not restricted to comparing the membrane potential to a fixed value, it is more generally a logical

A Hodgkin-Huxley equations

```
G = NeuronGroup(number_of_neurons,
                '''dv/dt = (I_l+I_Na+I_K)/C_m   : volt # membrane potential
                   I_l = g_l*(-v+E_l)             : amp  # passive current
                   I_Na = g_Na*(m**3)*h*(-v+E_Na) : amp  # sodium current
                   I_K = g_K*(n**4)*(-v+E_K)      : amp  # potassium current
                   ...# equations for n, m, h''')
```

B Noisy membrane

```
G = NeuronGroup(number_of_neurons,
                'dv/dt = -(v-v_0)/tau_m +
                         tau_m**-0.5*3*mV*xi : volt # membrane potential')
```

C Leaky integrate-and-fire neuron

```
G = NeuronGroup(number_of_neurons,
                'dv/dt = -(v-v_0)/tau_m : volt # membrane potential',
                threshold='v > v_th', reset='v = v_0')
```

D Leaky integrate-and-fire neuron with adaptive threshold

```
G = NeuronGroup(number_of_neurons,
                '''dv/dt = -(v-v_0)/tau_m          : volt # membrane potential
                   dv_th/dt = -(v_th-v_th0)/tau_th : volt # threshold''',
                threshold='v > v_th', reset='''v = v_0
                                              v_th += 3*mV''')
```

FIGURE 1 | Examples for neuronal model descriptions in Brian2. (A) Differential equations and parameters are defined via multi-line strings, units are specified after the colon. Note that the units specify the units of the *variables* defined in the respective line which in the case of differential equations is not equivalent to the units of left-hand side and right-hand side of the equations (`volt` vs. `volt/second`). **(B)** The symbol `xi` is used to refer to a stochastic variable for defining stochastic equations. **(C,D)** Threshold conditions and reset statements are also defined by strings.

expression evaluated on the state variables. For example, a leaky integrate-and-fire neuron with an adaptive threshold could be described by the equations:

$$\frac{dv}{dt} = -(v - v_0)/\tau_m$$

$$\frac{dv_{th}}{dt} = -(v_{th} - v_{th0})/\tau_{th}$$

After $v > v_{th}$: $v \leftarrow v_0$

$v_{th} \leftarrow v_{th} + 3\,\mathrm{mV}$

This model increases the threshold after every spike by 3 mV, between spikes it decays back to the resting value v_{th0} (see **Figure 1D** for the Brian2 syntax).

2.1.3. Refractoriness

In integrate-and-fire models, the fact that a neuron is not able to generate a second action potential for a short time after a first one is modeled explicitly (and not following from channel dynamics described in the differential equations). In contrast to that, Hodgkin-Huxley type models only use a threshold for detecting spikes and the refractoriness to prevent detecting multiple spikes for a single threshold crossing (the threshold condition would evaluate to *true* for several time points). In this case, the refractory period is not easily expressed as a time span but more naturally as a condition that the neuron should remain refractory for as long as it stays above the threshold.

A simple formulation of refractoriness that allows some flexibility is to consider that the exit from refractoriness is defined by a logical expression that may depend on state variables and the time of the last spike. In Brian2, the latter is stored in the special variable *lastspike* and the condition evaluates to *false* when the neuron must exit the refractory state. For example, a fixed refractory period of 2 ms can be described as $(t - lastspike) \leq 2\,\mathrm{ms}$ (see **Figure 2A** for the Brian2 syntax). If the refractoriness should vary across neurons, the expression can refer to a neuronal state variable instead: $(t - lastspike) \leq refractoriness$. Since the state variable *refractoriness* can undergo dynamic changes as well (e.g., it could be described by a differential equation or change with every spike as part of the reset), this allows to model very complex refractoriness conditions (**Figure 2B**). For Hodgkin-Huxley models, the refractoriness condition could simply be $v \geq v_{\mathrm{th}}$, with no reference to the time of the last spike (**Figure 2C**).

Finally, the set of differential equations could be different in the refractory state. Most generally, this could be described by two different sets of equations. However, neural models generally implement refractoriness in only two ways: either some or all state variables are clamped to a fixed value, or the state variables are allowed to continue to change but threshold crossings are ignored. Only the former case (clamped variables) requires new syntax. We propose to simply mark certain differential equations as non-changing during the refractory period. This can be an explicit syntax feature of the description language (as in Brian2, see **Figure 2D**). Alternatively, an additional boolean variable representing refractoriness can be introduced, and clamping can then be implemented by multiplying the right hand side of the differential equation by that variable, interpreting the truth values *true* and *false* as 0 and 1, respectively.

2.2. SYNAPTIC MODELS

The description of synaptic models has very similar requirements to the description of neuronal models: synaptic state variables may evolve continuously in time and undergo instantaneous changes at the arrival of a pre-synaptic or post-synaptic spike. A synapse connects a pre-synaptic neuron to a post-synaptic neuron, and can have synapse-specific variable values. Events

A Simple refractoriness, threshold crossings are ignored for 2 ms after a spike:

```
G = NeuronGroup(..., threshold='v > v_th', reset='v = v_r',
                refractory='(t-lastspike) <= 2*ms' )
```

B Adaptive refractoriness (increases after every spike):

```
G = NeuronGroup(..., model='''dv/dt = -v/tau : volt
                              dref/dt = -(ref-3*ms) / (50*ms) : second''',
                threshold='v > v_th', reset='v = v_r; ref += 1*ms',
                refractory='(t-lastspike) <= ref')
```

C Use of threshold and refractoriness for HH-type models:

```
G = NeuronGroup(..., threshold='v > v_th', refractory='v > v_th')
```

D Membrane potential clamped during refractoriness:

```
G = NeuronGroup(..., model='dv/dt = -v/tau : volt (unless refractory)',
                refractory='(t-lastspike) <= 2*ms', ...)
```

FIGURE 2 | Examples for refractoriness formulations in Brian2. (A,B) Refractoriness condition based on time since last spike. **(C)** Refractoriness condition based on membrane potential threshold crossing. **(D)** Modifying differential equations based on refractoriness state, the keyword `unless refractory` clamps a variable x during the refractory period, i.e., $\frac{dx}{dt} = 0$.

can be pre-synaptic spikes or post-synaptic spikes, and they can trigger changes in synaptic variables, pre-synaptic neural variables or post-synaptic neural variables. With these specifications, describing synaptic models should follow a similar scheme to the one used for neural models: the continuous evolution of the synaptic state variables is described by differential equations, "pre" and "post" statements describe the effect of a pre- or post-synaptic spike. In contrast to neural models, there is no need for a threshold condition since action potentials are emitted from the pre-/post-synaptic neurons according to their threshold conditions.

A very simple synaptic model might not define any equation and only add a constant value to a post-synaptic conductance or current (or the membrane potential directly) on every pre-synaptic spike, for example $g_{\text{post}} \leftarrow g_{\text{post}} + 1\,\text{nS}$. Note that the index "post" is used to distinguish post-synaptic variables from synaptic variables (in a simple textual description as part of a neural simulator, a suffix `_post` can be used instead, see **Figure 3A**). Similarly, the index "pre" can be used for pre-synaptic neural variables. To model differing weights across synapses, a synaptic state variable w storing the weights can be introduced, which can then be referred to in the statement: $g_{\text{post}} \leftarrow g_{\text{post}} + w$ (see **Figure 3B**).

Probabilistic synapses can be modeled by introducing a source of randomness in the "pre" statement. If $\mathcal{U}(0, 1)$ is a random variable uniformly distributed between 0 and 1 and "int" a function converting a boolean expression into a value of 0 or 1, a synapse that transmits spikes with 50% probability can use the following "pre"' statement: $g_{\text{post}} \leftarrow g_{\text{post}} + \text{int}\,(\mathcal{U}(0, 1) < 0.5)\, w$ (see **Figure 3C**).

In the most general formulation, however, the evolution of the synapses' state variables have to be described by differential equations, in the same way as neuronal model equations. By allowing these equations, as well as the "pre" and "post"

A Simple synaptic connection with constant weights:

```
S = Synapses(source_group, target_group, pre='g_post += 1*nS')
```

B Simple synaptic connection with variable weights:

```
S = Synapses(source_group, target_group, 'w : siemens', pre='g_post += w')
```

C Synapse transmitting spikes with 50% probability:

```
S = Synapses(source_group, target_group, pre='g_post += int(rand() < 0.5)*w')
```

D Synapse with spike-timing-dependent-plasticity:

```
S = Synapses(source_group, target_group,
             '''w : siemens
                dA_source/dt = -A_source/tau_source : siemens (event-driven)
                dA_target/dt = -A_target/tau_target : siemens (event-driven)''',
             pre='''g_post += w
                    A_source += deltaA_source
                    w = clip(w+A_target, 0*siemens, w_max)''',
             post='''A_target += deltaA_target
                     w = clip(w+A_source , 0*siemens, w_max)'''
             )
```

E Synapse with spike-timing-dependent-plasticity, alternative formulation:

```
S = Synapses(source_group, target_group,
             '''w : siemens
                A_source : siemens
                A_target : siemens''',
             pre='''g_post += w
                    A_source = A_source*exp((lastupdate-t)/tau_source)+deltaA_source
                    A_target = A_target*exp((lastupdate-t)/tau_target)
                    w = clip(w+A_target, 0*siemens, w_max)''',
             post='''A_source = A_source*exp((lastupdate-t)/tau_source)
                     A_target = A_target*exp((lastupdate-t)/tau_target)+deltaA_target
                     w = clip(w+A_source, 0*siemens, w_max)''')
```

FIGURE 3 | Examples for synaptic model descriptions in Brian2. (A–C) Forward propagation is described by specifying statements to be executed on the arrival of a pre-synaptic spike (keyword argument `pre`), changing the value of post-synaptic variables (suffix `_post`). **(D,E)** Equivalent definitions of spike timing-dependent plasticity using forward and backward propagation. The `(event-driven)` at the end of the differential equations in **(D)** makes Brian2 automatically convert the equations into the alternative formulation given in **(E)**. The name `w_max` refers to a constant defined outside of the synaptic model. `clip` is a function that restricts the values given as the first argument to the range specified by the second and third argument.

statements, to refer to pre- and post-synaptic variables a variety of synaptic models can be implemented, including spike-timing-dependent-plasticity and short-term plasticity. For example, models of spike-timing-dependent-plasticity rules make use of abstract "traces" of pre- and post-synaptic activity. Such a model (Song et al., 2000) can be described by the following equations:

$$\frac{\mathrm{d}A_{\text{source}}}{\mathrm{d}t} = -A_{\text{source}}/\tau_{\text{source}}$$

$$\frac{\mathrm{d}A_{\text{target}}}{\mathrm{d}t} = -A_{\text{target}}/\tau_{\text{target}}$$

After a pre-synaptic spike:

$$g_{\text{post}} \leftarrow g_{\text{post}} + w$$

$$A_{\text{source}} \leftarrow A_{\text{source}} + \Delta A_{\text{source}}$$

$$w \leftarrow \min\left(\left[w + A_{\text{target}}\right]^{+}, w_{\max}\right)$$

After a post-synaptic spike:

$$A_{\text{target}} \leftarrow A_{\text{target}} + \Delta A_{\text{target}}$$

$$w \leftarrow \min\left(\left[w + A_{\text{source}}\right]^{+}, w_{\max}\right)$$

where $[x]^{+} = x$ for $x > 0$ and $[x]^{+} = 0$, otherwise. This assumes an additional state variable w, storing the synaptic weights, parameters ΔA_{source} and ΔA_{target} setting the change in the traces with every spike and a parameter $w_{\max}$, setting the maximum synaptic weight. The variable g_{post} refers to a state variable of the post-synaptic neurons.

Equivalently, the differential equations can be analytically integrated between spikes, allowing for an event-driven and therefore faster simulation (e.g., Brette, 2006; Morrison et al., 2007). It is possible in this case (but not in general) because the variables A_{source} and A_{target} do not have to be updated at every timestep (which could be very costly if the model involves a large number of synapses) since their values are only needed on the arrival of a spike. Their differential equations are linear, it is therefore possible to include the solutions directly in the "pre" and "post" statements. This requires an additional state variable (*lastupdate* in Brian2) that stores the time of the last state update (pre- or post-synaptic spike). The model can then be reformulated without differential equations (storing w, A_{source} and A_{target} as state variables, see **Figure 3E**), leading to the formulation of the STDP rule in its integrated form (Song and Abbott, 2001):

After a pre-synaptic spike:

$$g_{\text{post}} \leftarrow g_{\text{post}} + w$$

$$A_{\text{source}} \leftarrow A_{\text{source}} \cdot e^{(\text{lastupdate}-t)/\tau_{\text{source}}} + \Delta A_{\text{source}}$$

$$A_{\text{target}} \leftarrow A_{\text{target}} \cdot e^{(\text{lastupdate}-t)/\tau_{\text{target}}}$$

$$w \leftarrow \min\left(\left[w + A_{\text{target}}\right]^{+}, w_{\max}\right)$$

After a post-synaptic spike:

$$A_{\text{source}} \leftarrow A_{\text{source}} \cdot e^{(\text{lastupdate}-t)/\tau_{\text{source}}}$$

$$A_{\text{target}} \leftarrow A_{\text{target}} \cdot e^{(\text{lastupdate}-t)/\tau_{\text{target}}} + \Delta A_{\text{target}}$$

$$w \leftarrow \min\left(\left[w + A_{\text{source}}\right]^{+}, w_{\max}\right)$$

The transformation from differential equations to event-driven updates can be done automatically using symbolic manipulation. In Brian2, this is implemented for certain analytically solvable equations, in particular systems of independent linear differential equations (see **Figure 3D**).

The framework presented so far is insufficient for two important cases, however. One is non-linear synaptic dynamics such as models of NMDA receptors. In this case, individual synaptic conductances must be simulated separately and then summed into the total synaptic conductance. In such a model, the total NMDA conductance of a single neuron can be described as follows (e.g., Wang, 2002):

$$g_{\text{total}}^{\text{NMDA}} = \sum_i g_i^{\text{NMDA}}$$

$$g_i^{\text{NMDA}} = w_i s_i^{\text{NMDA}}$$

$$\frac{\mathrm{d}x_i^{\text{NMDA}}}{\mathrm{d}t} = \frac{-x_i^{\text{NMDA}}}{\tau_1^{NMDA}}$$

$$\frac{\mathrm{d}s_i^{\text{NMDA}}}{\mathrm{d}t} = \frac{-s_i^{\text{NMDA}}}{\tau_2^{NMDA}} + \alpha x_i^{\text{NMDA}}(1 - s_i^{\text{NMDA}})$$

After a pre-synaptic spike at synapse i:

$$x_i^{\text{NMDA}} \leftarrow x_i^{\text{NMDA}} + 1$$

Another important case is gap junctions. In this case, the synaptic current is a function of pre- and post-synaptic potential, which can be expressed in the previously introduced framework, and then all synaptic currents must be added in a neuronal variable representing the total current.

Both cases can be addressed by marking every relevant synaptic variable (NMDA conductance, gap junction current) so that the sum over this variable should be taken for all synapses connecting to a post-synaptic neuron and copied over to the corresponding post-synaptic state variable at each simulation time step. See **Figure 4** for the corresponding syntax in Brian2.

2.3. NETWORK STRUCTURE

We consider the following specifications for synaptic connections: each synapse is defined by its source (pre-synaptic neuron) and target (post-synaptic neuron); there is a transmission delay from source to synapse, and another one from target to synapse (for the backpropagation needed for example in spike-timing-dependent plasticity rules); there may be several synapses for a given pair of pre- and post-synaptic neurons.

There are several approaches to the problem of building the set of synaptic connections, each having certain advantages and disadvantages. A set of pre-defined connectivity patterns such as "fully connected," "one-to-one connections," "randomly connected," etc. does not allow us to capture the full range of possible connection patterns. In addition, such patterns are not always clearly defined: for example, if neurons in a group are connected

A Non-linear NMDA synapse

```
target_group = NeuronGroup(number_of_neurons, '''...
                                              g_nmda : siemens''', ...)
S = Synapses(source_group, target_group,
             model='''w : siemens  # weight of the connection
                      g_nmda_post = w*s_nmda : siemens (summed)
                      dx_nmda/dt = -x_nmda/tau_nmda_1 : 1
                      ds_nmda/dt = -s_nmda/tau_nmda_2+alpha*x_nmda*(1-s_nmda) : 1''',
             pre='x_nmda += 1')
```

B Gap-junction

```
neurons = NeuronGroup(number_of_neurons, '''...
                                         Igap : amp''', ...)
S = Synapses(neurons, neurons, '''w : siemens # gap junction conductance
                                  Igap_post = w*(v_pre-v_post) : amp (summed)''')
```

FIGURE 4 | Examples for synaptic model descriptions involving summed variables in Brian2. (A,B) The equation lines ending with `(summed)` specify that the post-synaptic variable indicated with the suffix `_post` is set by summing the corresponding synaptic values on each time step.

to each other randomly, does that include self-connections or not (cf. Crook et al., 2012)? Another approach is to specify connection patterns using a connectivity matrix, allowing for all possible connection patterns. The disadvantage of this approach is that it is not easy to see or report what the underlying connectivity pattern is.

An alternative approach that offers expressivity, explicitness and concise description of connectivity patterns is to use mathematical expressions that specify: (1) whether two neurons i and j are connected, (2) the probability of connections between neurons i and j, (3) the number of synapses between i and j (different types of expressions can be combined).

The first expression is a boolean expression that evaluates to *true* for a certain combination of indices i and j. For example, $i = j$ describes a one-to-one connectivity, $\lfloor i/N \rfloor = j$ describes a convergent pattern where N pre-synaptic neurons connect to one post-synaptic neuron and $|\left(i - j + N/2\right) \bmod N - N/2| = 1$ describes a ring structure of N neurons, where each neuron connects to its immediate neighbours. More complex patterns can be expressed in the same formalism, if the expression can also refer to state variables of the pre- and post-synaptic neurons. Similarly to the synaptic statements described in the previous section, we propose using an index or suffix such as "pre" and "post" to specify these variables.

For example, if the location of neurons in the 2d plane is stored as neural state variables x and y, spatial connectivity can be readily expressed. The following expression describes a connection from each neuron to all neurons in a 250 µm radius: $\sqrt{(x_{\text{pre}} - x_{\text{post}})^2 + (y_{\text{pre}} - y_{\text{post}})^2} < 250\,\mu\text{m}$.

The same framework can be used to specify connection probability, possibly in combination with conditions described above. For example, the condition $i \neq j$ combined with the probability expression $p_{\max} \cdot \exp\left(-\frac{(x_{\text{pre}}-x_{\text{post}})^2+(y_{\text{pre}}-y_{\text{post}})^2}{2(125\,\mu\text{m})^2}\right)$ unambiguously defines a structured random connectivity (as a 2-dimensional Gaussian with standard deviation 125 µm) that does not allow for self-connections.

Finally, the expression syntax can also be used to create more than one synapse for a pre-/post-synaptic pair of neurons (useful for example in models where a neuron receives several inputs from the same source but with different delays).

See **Figure 5** for the use of mathematical expressions to specify synaptic connectivity in Brian2.

2.4. ASSIGNING STATE VARIABLE VALUES

To make the description complete, the initial value of state variables must be set. Many models include state variables that differ across neurons from the start in a systematic (e.g., synaptic weights or delays might depend on the distance between two neurons) or random way (e.g., initial membrane potential values). Such descriptions can be expressed using the very same formalism that has been presented so far (for Brian2 syntax, see **Figure 6**). For example, initial membrane potential values might be set to random values as $v_0 + \mathcal{N}(0, 1) \cdot 3\,\text{mV}$.

For synaptic variables, references to pre- and post-synaptic state variables can be used to express values depending on the neurons that are connected via the synapse. For example, synaptic delays might be set to depend on the distance of the involved neurons as $\frac{\sqrt{(x_{\text{pre}}-x_{\text{post}})^2+(y_{\text{pre}}-y_{\text{post}})^2}}{\text{speed}}$.

Setting state variables with textual descriptions instead of assigning values directly using the programming language syntax may seem to be a questionable choice. However, it offers at least two advantages: firstly, it allows the generation of code setting the variable that then runs on another device, e.g., a GPU, instead of having to copy over the generated values (see section 3); secondly, it allows for a semi-automatic model documentation system to generate meaningful descriptions of the initial values of a state variable (see section 4).

Full connectivity:

```
S.connect('True')
```

One-to-one connectivity:

```
S.connect('i == j')
```

Convergent connectivity:

```
S.connect('(i/N) == j')
```

Ring structure, connecting to the immediate neighbours:

```
S.connect('abs((i - j + N/2)%N - N/2) == 1')
```

Connections to 2d neighbourhood:

```
S.connect('sqrt((x_pre-x_post)**2+(y_pre-y_post)**2) < 250*umeter')
```

Sparse random connectivity without self-connections:

```
S.connect('i !=j', p=0.1)
```

Random connectivity to a 2d neighbourhood without self-connections:

```
S.connect('i != j',
          p='p_max*exp(-(x_pre-x_post)**2+(y_pre-y_post)**2) / (2*(125*umeter)**2)')
```

One-to-one connectivity with two synapses per connection:

```
S.connect('i == j', n=2)
```

FIGURE 5 | Examples for synaptic connection descriptions in Brian2. Strings define conditions based on the pre-synaptic neuron index *i*, post-synaptic neuron index *j*, and pre- and post-synaptic neuron variables with suffix `_pre` and `_post`. Optionally, a value or expression for the probability of creating a synapse can be given as the keyword argument `p`. The number of synapses to create for each connection can be given as the keyword argument `n`.

A Random initial values for a membrane potential:

```
G.v = 'v_0+randn()*3*mV'
```

B Synaptic delays, depending on distance between cells:

```
S.delay = 'sqrt((x_pre-x_post)**2+(y_pre-y_post)**2)/speed'
```

C Synaptic weights (unitless), based on distance:

```
S.w = 'w_0*exp((-(x_pre-x_post)**2+(y_pre-y_post)**2)/(2*(125*umeter)**2))'
```

D Synaptic weights, based on presynaptic firing rates:

```
S.w = 'w_base+w_0/rates_pre'
```

FIGURE 6 | Examples for specifying initial state variable values in Brian2. (A) Neuronal variables. **(B–D)** Synaptic variables based on pre- and post-synaptic neuronal variables indicated with suffix `_pre` and `_post` respectively.

3. GENERATING CODE FROM MODEL DESCRIPTIONS

To simulate a neural model means to track the evolution of its variables in time. As shown in the previous section, these dynamical changes consist of three components: *continuous updates* (the model equations), *event-triggered updates* (e.g., the reset in an integrate-and-fire neuron or the synaptic transmission after a pre-synaptic spike), and *logical expressions* defining the events (e.g., a threshold condition).

Continuous updates are specified in the form of equations that first have to be combined with a numerical integration method to

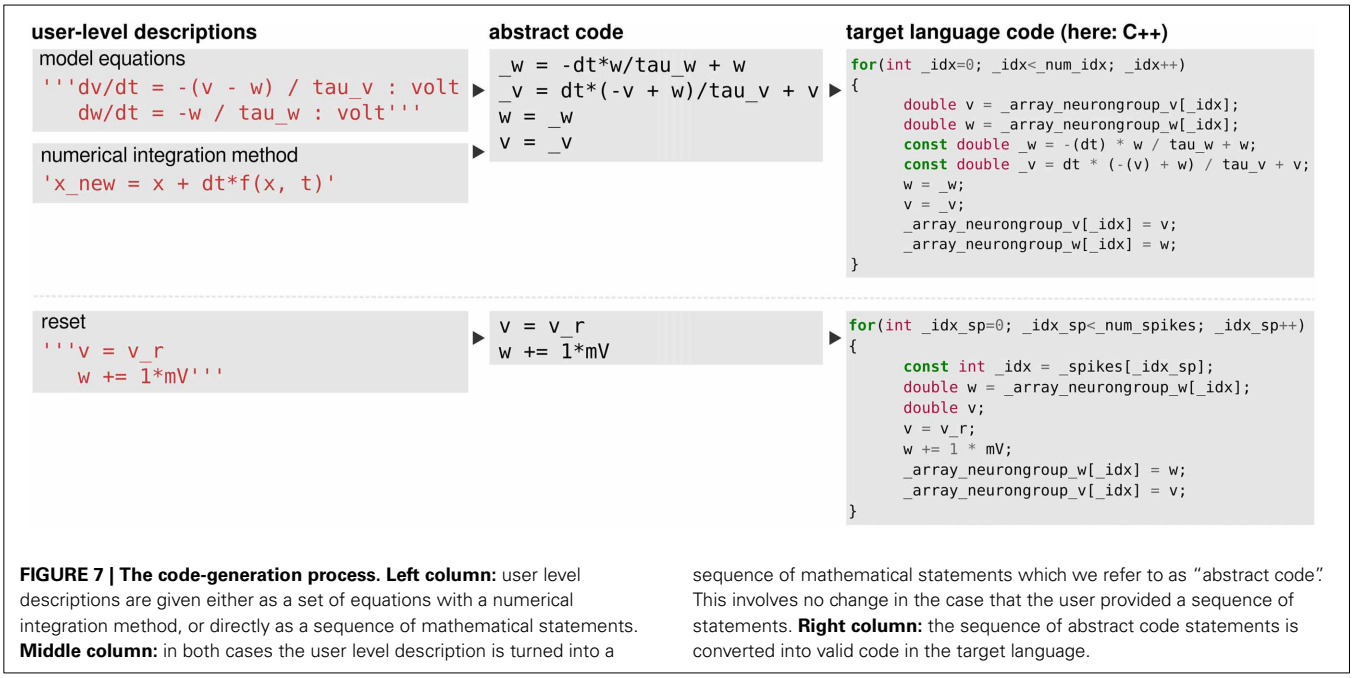

FIGURE 7 | The code-generation process. Left column: user level descriptions are given either as a set of equations with a numerical integration method, or directly as a sequence of mathematical statements. **Middle column:** in both cases the user level description is turned into a sequence of mathematical statements which we refer to as "abstract code". This involves no change in the case that the user provided a sequence of statements. **Right column:** the sequence of abstract code statements is converted into valid code in the target language.

yield abstract code (see **Figure 7** top) which is then transformed into programming language code. Event-triggered updates and logical expressions on the other hand are directly specified in the form of abstract code and only have to be transformed into programming language code (**Figure 7** bottom). These two steps of code generation will be described in the following.

3.1. NUMERICAL INTEGRATION OF CONTINUOUS STATE UPDATES

Most neural models are based on equations that are not analytically solvable. The standard approach is therefore to use numerical integration and calculate the values at discrete time points. Many well-studied integration methods exist, allowing for different trade-offs between computational complexity and precision. Often, the provided numerical integration methods are either an integral part of the simulation tool (e.g., in Neuron, Carnevale and Hines, 2006) or built into a specific neural or synaptic model (e.g., in NEST, Gewaltig and Diesmann, 2007).

Here we show a new approach implemented in Brian2, in which a mathematical formulation of an integration method can be combined with the description of the neural model to yield abstract code that is later transformed into target language code using a common "abstract code to language code" framework.

3.1.1. Deterministic equations

Explicit integration methods can be described by a recursive scheme providing the values at discrete time steps. For example, the "midpoint method" (second-order Runge-Kutta) calculates the nth value of a variable x (where individual values are spaced apart in time by dt) according to:

$$x_{n+1} = x_n + dt \cdot f\left(x_n + \frac{dt \cdot f\left(t_n, x_n\right)}{2}, t_n + \frac{dt}{2}\right) \qquad (1)$$

We specify this integration scheme using the following description, defining a name for a subexpression to avoid nested references to the function f (which would make later processing steps considerably more difficult):

```
'''k = dt*f(x, t)
   x_new = x+dt*f(x+k/2, t+dt/2)'''
```

The **x_new** line denotes the final new value for the variable x.

Let us consider a model equation with two state variables, describing a neuron with an adaptation current:

```
'''dv/dt = (-w-v)/tau_v : volt # the membrane equation
   dw/dt = -w/tau_w : volt # the adaptation current'''
```

The integration method and the model equations are combined and transformed into abstract code using SymPy, according to algorithm 1.

Combining the midpoint method and the neuronal equations from above according to this algorithm works as follows:

The model equations in vector form:

$$\underline{x} = (v, w)\text{, with } x_v = v, x_w = w$$

$$\underline{f}\left(\underline{x}, t\right) = \left(\frac{(-x_v - x_w)}{\tau_v}, \frac{-x_w}{\tau_w}\right)$$

The first statement σ:

$$\underline{k} = dt \cdot \underline{f}\left(\underline{x}, t\right)$$

Expanding it:

$$\underline{k} = \left(dt \cdot \frac{-x_v - x_w}{\tau_v}, dt \cdot \frac{-x_w}{\tau_w}\right)$$

Append to code:

Algorithm 1 | Combining model equations and numerical integration method description to yield abstract code. Here, the differential equations in vector form are $\mathrm{d}\underline{x}/\mathrm{d}t = \underline{f}(\underline{x}, t)$ where $\underline{x}$ is the vector of state variables with components x_v for variable v, and $\underline{f}$ is a vector function with corresponding components f_v. All statements in the numerical integration method should be understood as referring to vectors of variables and vector functions.

for all statements $\underline{\sigma}$ in the numerical integration method **do**
  Expand $\underline{f}(\hat{\underline{x}}, \hat{t})$ in $\underline{\sigma}$
  Replace $\underline{x}_{\text{new}}$ in $\underline{\sigma}$ by $\underline{y}$
  for all state variables v **do**
    Append component σ_v of the transformed $\underline{\sigma}$ to code
  end for
end for
for all state variables v **do**
  Append $x_v = y_v$ to code
end for

$$k_v = dt \cdot \frac{-x_v - x_w}{\tau_v}$$

Append to code:

$$k_w = dt \cdot \frac{-x_w}{\tau_w}$$

The second statement σ:

$$\underline{x}_{\text{new}} = \underline{x} + dt \cdot f\left(\underline{x} + \underline{k}/2, t + dt/2\right)$$

Expanding it:

$$\underline{x}_{\text{new}} = \left(x_v + dt \cdot \frac{-(x_v + k_v/2) - (x_w + k_w/2)}{\tau_v}, \; x_w + dt \cdot \frac{-x_w + k_w/2}{\tau_w}\right)$$

After replacing $\underline{x}_{\text{new}}$ by y:

Append to code:

$$y_v = x_v + dt \cdot \frac{-(x_v + k_v/2) - (x_w + k_w/2)}{\tau_v}$$

Append to code:

$$y_w = x_w + dt \cdot \frac{-x_w + k_w/2}{\tau_w}$$

Finally:

Append to code:

$$x_v = y_v \text{ and } x_w = y_w$$

The full abstract code then reads (using names starting with underscores to denote the variables k and y introduced by the algorithm):

```
_k_v = dt*(-v-w)/tau_v
_k_w = -dt*w/tau_w
_v = dt*(-_k_v/2-_k_w/2-v-w)/tau_v+v
_w = -dt*(_k_w/2+w)/tau_w+w
v = _v
w = _w
```

3.1.2. Stochastic equations

The same procedure can also be applied to stochastic differential equations, a description of a state updater in this case looks like (Euler-Maruyama method):

```
'x_new = x + dt*f(x, t) + g(x, t)*dW'
```

The function g in the above formulation corresponds to the factor of the stochastic variable. Note that in the specific case of the Euler-Maruyama method, the function g has to be a constant and is therefore not a function of time ("additive noise"), we use the notation $g(x, t)$ nevertheless for consistency. The symbol **dW** denotes a normally distributed random variable with variance dt.

In an equation defining a simple integrate-and-fire neuron with additive noise

```
'dv/dt = -v/tau + tau**-0.5*sigma*xi : volt'
```

f is identified as **-v/tau** and g as **tau**`**-0.5*`**sigma**, leading to the following abstract code:

```
xi = dt**.5*randn()
_v = -dt*v/tau + v + sigma*tau**(-0.5)*xi
v = _v
```

The **randn**`()` function generates a normal distributed random number.

In the case of more than one stochastic variable (which can be used to model shared noise between variables) the stochastic part of the state updater description is understood as being additive for all stochastic variables. For example, in the case of two stochastic variables, the above described integration method is read as

```
'x_new = x + dt*f(x, t) + g_1(x, t)*dW_1
        + g_2(x, t)*dW_2'
```

Therefore, the following equation with two stochastic variables

```
'''dv/dt = (-w-v)/tau_v + tau_v**-0.5*sigma_both*
             xi_both + tau_v**-0.5*sigma_v*xi_v : volt
   dw/dt = -w/tau_w + tau_w**-0.5*sigma_both*xi_both:
           volt'''
```

will be integrated as:

```
xi_both = dt**.5*randn()
xi_v = dt**.5*randn()
_w = -dt*w/tau_w + w + sigma_both*tau_w**(-0.5)*xi_both
_v = dt*(-v/tau_v - w/tau_v) + v + sigma_both*tau_v**
     (-0.5)*xi_both + sigma_v*tau_v**(-0.5)*xi_v
w = _w
v = _v
```

3.2. TURNING ABSTRACT CODE INTO RUNNABLE CODE

3.2.1. Abstract code

"Abstract code" is also used for updates that are triggered by specific events, typically a spike, either in a neuron itself or in a pre- or post-synaptic neuron in the context of synapses. In contrast to the model equations, this code is not a mathematical formulation

but an explicit statement on how variables should be changed. For example, the reset of an integrate-and-fire neuron might simply state `'v = v_r'` to reset the membrane potential to a specific value. The new value might also depend on the old value, e.g., for an adaptation current: `'w = w+3*mV'`. This is a programming language statement and not a mathematical equation, therefore it could also make use of an in-place operator: `'w += 3*mV'`. In Brian2, abstract code is used for the reset statements of an integrate-and-fire neuron (where the reset might include updating adaptation variables, changing the refractory period, etc.) and for the code executed on the arrival of pre- or post-synaptic spikes at a synapse. As shown in the previous section, the code generated from model equations via the numerical integration method also is abstract code that will be executed on every timestep.

Abstract code is currently restricted to simple statements of the form:

```
{variable} {operator} {expression}
```

where {**variable**} is the name of a state variable (or a temporary variable used later in the same abstract code block), {**operator**} is one of `=`, `+=`, `-=`, `*=`, `/=`, i.e., either an assignment or an inplace operation, and **expression** is an arbitrary expression in Python syntax (which is mostly indistinguishable from C++ syntax, except for rare exceptions such as the absence of a "power" operator in C++, where the **pow** function will be used as a replacement). Even though this code does not allow for control structures such as if statements, some conditional updating can be realized by using boolean expressions. For example, if a model should only update its adaptation variable w during an initial time period of 1 s, the reset code could be stated as `'w += 3*mV*int(t < 1*second)'` This makes use of an **int** function that converts a boolean expression into 0 or 1.

3.2.2. Expressions

The final remaining building block for model definitions are *expressions*, such as the threshold condition in integrate and fire models. It could refer to a fixed value (`'v > -30*mV'`), to another state variable (`'v > v_t'`) or use any other expression that would be admissible in the right-hand-side of an abstract code statement; in fact, the threshold condition is simply captured in a variable by transforming the expression {**expression**} into the statement **_cond** ={**expression**}. We now describe some specifics for expressions used in making synaptic connections and in assigning state variables.

Building connections from the synaptic connection descriptions presented previously is straightforward: two nested loops iterate over all possible values of i and j, evaluating the connection condition in turn, using the given i and j values and accessing any referenced pre-/post-synaptic variables with indices i and j respectively. If the condition evaluates to *true* and no probability is given, the given number of synapses is created (potentially after evaluating the expression determining the number of synapses, in the same way as the condition was evaluated). If a probability has been given, creating the synapse(s) is only done if a random number drawn from a uniform distribution in the interval $[0, 1)$ is smaller than the given probability (respectively the result of evaluating the given expression).

Note that in languages supporting vectorization (e.g., in Python with the NumPy libraries (Oliphant, 2007)), instead of looping over individual indices, it is more efficient to only loop over one index and use a vector of all values for the other index. In principle it would be possible to have two vectors of all possible index combinations i and j and not do any looping at all, but this would require generating temporary vectors that have size $N \times M$ (for group sizes N and M) which is infeasible for large networks.

Setting state variable values with string expressions does not require any specific mechanism and can use the existing code generation techniques. In particular, setting a state variable of a group of neurons can be implemented in the same way as the reset (it can be thought of as a reset that only happens once, not after every spike) and setting state variables of synapses can be implemented in the same way as the effect of a pre- or post-synaptic spike.

3.2.3. Code generation

The abstract code that is generated from the combination of model equations and state updater descriptions or directly provided for event-triggered state updates mostly follows Python syntax conventions but is not directly executable as such. It describes what operations should be executed in the context of a given neuron or synapse, but the implementation may use vectorization or parallelization over neurons/synapses (e.g., in Python, see Brette and Goodman, 2011) or loops (e.g., in C++). Therefore, there is an additional step to transform the abstract code into runnable code for a target language and/or machine.

Let us investigate a simple code statement, resulting from applying forward Euler integration to an integrate-and-fire model with an adaptive current (same example as in the beginning of section 3.1):

```
_w = -dt*w/tau_w + w
_v = dt*(-v-w)/tau_v + v
w = _w
v = _v
```

If we let **w** and **v** refer to the state variable arrays (NumPy arrays), these statements are directly executable in Python. However, they don't directly change the original arrays, **w** and **v** instead refer to new temporary arrays that need to be copied back to the original arrays. Python code generation therefore surrounds the above code with:

```
w = _array_w
v = _array_v
... # main state update code (see above)
_array_w[:] = w
_array_v[:] = v
```

Code in other languages, e.g., C++, does not have built-in support for vectorisation, therefore it has to loop explicitly. Still,

the main state update code can be left intact, by surrounding it with[1]:

```
for(int _neuron_idx=0; _neuron_idx<_num_neurons;
        _neuron_idx++)
{
    double w = _array_w[_neuron_idx];
    double v = _array_v[_neuron_idx];
    ... // main state update code (see above)
    _array_w[_neuron_idx] = w;
    _array_v[_neuron_idx] = v;
}
```

Thus, the transformation from abstract code to target code consists of a model-independent template (responsible for the `for` loop in the C++ code), statements for reading/writing state variables from/to the arrays they are stored in and small changes to the abstract code to yield syntactically correct code in the target language (in the case of C++ this includes adding semicolons at the end of statements and replacing `x**y` by `pow(x, y)`, for example).

More details on the code generation mechanism can be found in Goodman (2010).

4. AUTOMATIC MODEL DOCUMENTATION

Even though it is now considered best practice for publications in computational neuroscience to make the source code that was used to generate the published results available, a simulator-independent description of the simulation is still valuable. Firstly, it is more accessible, particularly for researchers not familiar with the given simulation environment and/or programming language. Secondly, it simplifies reproducing and verifying the result with other tools.

There are two main approaches to this issue: first, the whole model can be specified in an abstract specification language such as NeuroML (Gleeson et al., 2010) or NineML (Raikov et al., 2011) which then allows the generation of simulator code and textual descriptions, e.g., in the form of an HTML page (see http://opensourcebrain.org for examples). Second, the model may be documented in a standardized form (e.g., Nordlie et al., 2009) that can be directly included in the publication.

The techniques presented in this paper allow for a third approach: since the simulator operates on high-level descriptions of the model in the form of strings, it is possible to create model descriptions automatically. For example, by virtue of SymPy's LaTeX printing facilities, Brian2's `Equations` object can be automatically converted into mathematical descriptions in LaTeX code (shown here in an interactive Python session):

```
>>> eqs = Equations('''dv/dt = (g_L*(E_L-v) +
                       g_s*(E_s-v))/tau_m : volt
                   dg_s/dt = -g_s/tau_s : siemens''')
>>> print sympy.latex(eqs)
\begin{align*}
\frac{\mathrm{d}v}{\mathrm{d}t} &= \frac{1}{\tau_{m}}
\left(g_{L} \left(E_{L} - v\right) + g_{s}
\left(E_{s} -  v\right)\right) &&  \text{(unit:
$\mathrm{V}$)}\\ \frac{\mathrm {d}g_{s}}{\mathrm{d}t}
&= - \frac{g_{s}}{\tau_{s}} &&
\text{(unit: $\mathrm{S}$)}
\end{align*}
```

[1]The code generated by Brian2 is a bit more complicated, including optimisations using `const` and `__restrict__` keywords.

Included in a LaTeX document, this is rendered as:

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{1}{\tau_m}\left(g_L\left(E_L - v\right) + g_s\left(E_s - v\right)\right) \qquad \text{(unit: V)}$$

$$\frac{\mathrm{d}g_s}{\mathrm{d}t} = -\frac{g_s}{\tau_s} \qquad \text{(unit: S)}$$

This "rich representation" of models not only makes it easier to generate useful model descriptions but can also help in preventing mistakes when generating them; a description that is directly generated from code is always "in sync" with it.

Models are not only defined by their equations, but also by parameter values. For simple parameters, e.g., the time constant τ_m from above, most simulators would allow for a convenient read-out of the values and therefore be able to display them with name and value in a table, for example. The situation is different for values that have to be described as a vector of values instead of a scalar, e.g., a τ_m that varies across neurons. Suppose we have a group G, consisting of N neurons, where we want the membrane time constant τ_m to vary across neurons. This might be specified by doing:

```
G.tau_m = 20*ms + 5*ms*randn(N)
```

where `randn` generates normally distributed random numbers. All that the simulator "knows" in this case, is that the parameter τ_m should be set to a given array of N numbers. There is no way it can infer where these numbers came from and all it can do in an automatic fashion is either to display all values (which would be inconvenient for large N), display a subset of them, to give an "idea" of the numbers used or provide summary statistics, e.g., minimum, maximum, mean and standard deviation of the values. A useful description would have to be manually provided by the researcher, with all the possibilities for making a mistake that this entails.

In contrast, consider the following assignment, providing the expression as a string:

```
G.tau_m = '20*ms + 5*ms*randn()'
```

where `randn` is implicitly understood to vectorize over all neurons. Given such a description, the simulator can automatically generate a human-readable documentation of the parameter value, say "$\tau_m = 20\,\mathrm{ms} + 5\,\mathrm{ms}\,\mathcal{N}(0, 1)$," without any intervention from the researcher.

While a *completely* automatic documentation will not be feasible in all cases, and postprocessing by the researcher is often inevitable, a fully automatic documentation system also offers other advantages: interactive exploration, for example in ipython notebooks (Pérez and Granger, 2007), benefits from having a rich representation of model components. In addition, tools that are concerned with provenance tracking and aim to support the

workflow of researchers (e.g., Davison, 2012) could use such a mechanism to give quick and automatic overviews not only over the parameter values used in simulations but also about the model itself, i.e., the equations that define it.

5. DISCUSSION

We have described a general framework for defining neural network models, which is based essentially on mathematical equations. It consists of a formalism for defining state variables including their physical units, differential equations describing the dynamics of state variables, conditions on state variables to trigger events and event-triggered statements, changing the state variables discontinuously.

We think that such a mechanism has several advantages over the approach of writing models based on a fixed library of models and mechanisms that can only be extended by writing new descriptions in a low-level language: the equation-oriented framework allows for straightforward descriptions of models; it is explicit about details of the model; by relying on common mathematical notation it does not require the user to learn any special syntax or names used for models and mechanisms.

5.1. LIMITATIONS

Not all models can be expressed in the framework we have presented, and we will now try to list these limitations. Neuron models were constrained to have only two excitability states, active and refractory, instead of an arbitrary number of states with transitions between them. This is not a fundamental limitation of an equation-oriented approach, but rather a choice that substantially simplifies the syntax.

The framework also neglects the spatial dimension, which would be important to simulate the cable equation on models with an extended spatial morphology, or to simulate calcium dynamics. While a small number of compartments could be already simulated in the current framework (by using the equations for the equivalent electrical model), a complex multicompartment simulation can only be described in a simple way with an explicit support for this scenario.

Regarding synaptic connections, although a fairly diverse set of synaptic models can be expressed with our framework, there are at least two limitations: structural plasticity and development (requiring the addition and removal of connections during a simulation), and hetero-synaptic plasticity (which cannot be expressed in the presented framework that describes changes in individual synapses independently of other synapses). Extending the framework to cover these cases is not impossible but would require substantial additions.

Finally, the proposed method of specifying the numerical integration method is designed for the general case where the equations cannot be integrated analytically. In Brian2 we do however also allow for special integrators for specific cases such as linear equations that can be integrated exactly using a matrix exponential (Hirsch and Smale, 1974; Rotter and Diesmann, 1999). Even though such integration schemes cannot be expressed in the way described in this paper, they still fit with the general code generation framework: instead of combining the model equations with a textual description of the integration method to yield the abstract code, the numerical integration method is implemented as a Python function that converts model equations directly into abstract code.

5.2. RELATION TO OTHER WORK

The NineML description language uses string descriptions of differential integrations, conditions and statements in a similar way to the approach presented here. However, due to the use of XML-based definitions and the decision to allow an arbitrary number of states and transition conditions, it is much more verbose in the common use cases and therefore more difficult to use for interactive exploration and rapid development. NineML and the approach presented here are not incompatible but rather complementary. It would be possible to automate the creation of a Brian2 simulation from a NineML description or vice versa.

For describing connectivity patterns, Djurfeldt (2012) proposed the *connection-set algebra*. This is based on similar ideas as the approach presented here, notably it also allows for unambiguous, explicit descriptions and has a high expressiveness. Connection-set algebra builds on the concept of elementary connection patterns (e.g., one-to-one connectivity, full connectivity) that can then be combined to yield more complex patterns. One advantage of this approach over the one we presented is that it allows for an implementation that is very efficient, especially for sparse connectivity patterns such as one-to-one connectivity (it generates all pairs (i, i) instead of checking all pairs (i, j) for i being equal to j). However, it also has a few disadvantages compared to our approach: connectivity patterns that are based on arbitrary pre- or post-synaptic state variables are more difficult to specify, and it defines connectivities based on a system that is completely separate from the rest of the model definition and cannot make use of common features such as the unit system.

5.3. FUTURE WORK

The framework presented allows for a wide variety of models with a minimal and unobstrusive syntax. However, we also plan to further increase its expressivity: the restriction to two neural states can be lifted without sacrificing simplicity by supporting multiple event types, each with a condition and a list of statements to be executed. This indirectly allows for an arbitrary number of states, since the state could be represented by a neural variable and equations could then depend on this value. The textual descriptions of numerical integration methods are currently restricted to explicit methods that only refer to the previous simulation time step. The same formalism could be quite naturally extended to implicit methods (e.g., the backward Euler method: $x_{\text{new}} = x + dt \cdot f(t + dt, x_{\text{new}})$ or multistep methods (e.g., the two-step Adams-Bashforth method: $x_{\text{new}} = x + dt\left(\frac{3}{2}f(t, x) - \frac{1}{2}f(t - dt, x_{\text{prev}})\right)$). More efficient synapse creation for sparse connectivities (e.g., one-to-one connectivity) can be achieved by either analyzing the user-specified connectivity definition for common patterns (e.g., $i = j$), or by adding a new syntax explicitly stating which synapses to create (in contrast to a boolean condition for a synapse to exist). Finally, control statements such as `if` and `for` could be added to abstract code. This poses some challenges for code generation, particularly when using vectorised statements, but would allow for even greater expressivity in models, and the use of more

complex numerical integration schemes such as variable time step schemes. These additions would allow for a wider variety of expressible models without sacrificing the core principles of model descriptions being explicit, readable and easy to write.

ACKNOWLEDGMENTS

This work was supported by ANR-11-0001-02 PSL*, ANR-10-LABX-0087 and ERC StG 240132.

REFERENCES

Brette, R. (2006). Exact simulation of integrate-and-fire models with synaptic conductances. *Neural Comput.* 18, 2004–2027. doi: 10.1162/neco.2006.18.8.2004

Brette, R. (2012). On the design of script languages for neural simulation. *Netw. Comp. Neural* 23, 150–156. doi: 10.3109/0954898X.2012.716902

Brette, R., and Goodman, D. F. M. (2011). Vectorized algorithms for spiking neural network simulation. *Neural Comput.* 23, 1503–1535. doi: 10.1162/NECO_a_00123

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., et al. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *J. Comput. Neurosci.* 23, 349–398. doi: 10.1007/s10827-007-0038-6

Carnevale, N. T., and Hines, M. L. (2006). *The NEURON Book*. Cambridge: Cambridge University Press. doi: 10.1017/CBO9780511541612

Crook, S. M., Bednar, J. A., Berger, S., Cannon, R., Davison, A. P., Djurfeldt, M., et al. (2012). Creating, documenting and sharing network models. *Netw. Comp. Neural* 23, 131–149. doi: 10.3109/0954898X.2012.722743

Davison, A. (2012). Automated capture of experiment context for easier reproducibility in computational research. *Comput. Sci. Eng.* 14, 48–56. doi: 10.1109/MCSE.2012.41

Destexhe, A., Mainen, Z. F., and Sejnowski, T. J. (1994). Synthesis of models for excitable membranes, synaptic transmission and neuromodulation using a common kinetic formalism. *J. Comput. Neurosci.* 1, 195–230. doi: 10.1007/BF00961734

Djurfeldt, M. (2012). The connection-set algebra – a novel formalism for the representation of connectivity structure in neuronal network models. *Neuroinform* 10, 287–304. doi: 10.1007/s12021-012-9146-1

Gewaltig, M.-O., and Diesmann, M. (2007). NEST (NEural simulation tool). *Scholarpedia* 2:1430. doi: 10.4249/scholarpedia.1430

Gleeson, P., Crook, S., Cannon, R. C., Hines, M. L., Billings, G. O., Farinella, M., et al. (2010). NeuroML: a language for describing data driven models of neurons and networks with a high degree of biological detail. *PLoS Comput. Biol.* 6:e1000815. doi: 10.1371/journal.pcbi.1000815

Goodman, D., and Brette, R. (2008). Brian: a simulator for spiking neural networks in Python. *Front. Neuroinform.* 2:5. doi: 10.3389/neuro.11.005.2008

Goodman, D. F. M. (2010). Code generation: a strategy for neural network simulators. *Neuroinform* 8, 183–196. doi: 10.1007/s12021-010-9082-x

Goodman, D. F. M., and Brette, R. (2009). The Brian simulator. *Front. Neurosci.* 3, 192–197. doi: 10.3389/neuro.01.026.2009

Hines, M. L., and Carnevale, N. T. (2000). Expanding NEURON's repertoire of mechanisms with NMODL. *Neural Comput.* 12, 995–1007. doi: 10.1162/089976600300015475

Hirsch, M. W., and Smale, S. (1974). *Differential Equations, Dynamical Systems, and Linear Algebra*. New York, NY: Academic Press.

Joyner, D., Čertík, O., Meurer, A., and Granger, B. E. (2012). Open source computer algebra systems: SymPy. *ACM Commun. Comput. Algebra.* 45, 225–234. doi: 10.1145/2110170.2110185

Morrison, A., Aertsen, A., and Diesmann, M. (2007). Spike-timing-dependent plasticity in balanced random networks. *Neural Comput.* 19, 1437–1467. doi: 10.1162/neco.2007.19.6.1437

Nordlie, E., Gewaltig, M.-O., and Plesser, H. E. (2009). Towards reproducible descriptions of neuronal network models. *PLoS Comput. Biol.* 5:e1000456. doi: 10.1371/journal.pcbi.1000456

Oliphant, T. E. (2007). Python for scientific computing. *Comput. Sci. Eng.* 9, 10–20. doi: 10.1109/MCSE.2007.58

Pérez, F., and Granger, B. E. (2007). IPython: a system for interactive scientific computing. *Comput. Sci. Eng.* 9, 21–29. doi: 10.1109/MCSE.2007.53

Raikov, I., Cannon, R., Clewley, R., Cornelis, H., Davison, A., De Schutter, E., et al. (2011). NineML: the network interchange for neuroscience modeling language. *BMC Neurosci.* 12(Suppl. 1):P330. doi: 10.1186/1471-2202-12-S1-P330

Rotter, S., and Diesmann, M. (1999). Exact digital simulation of time-invariant linear systems with applications to neuronal modeling. *Biol. Cybern.* 81, 381–402. doi: 10.1007/s004220050570

Song, S., and Abbott, L. F. (2001). Cortical development and remapping through spike timing-dependent plasticity. *Neuron* 32, 339–350. doi: 10.1016/S0896-6273(01)00451-2

Song, S., Miller, K. D., and Abbott, L. F. (2000). Competitive Hebbian learning through spike-timing-dependent synaptic plasticity. *Nat. Neurosci.* 3, 919–926. doi: 10.1038/78829

Wang, X.-J. (2002). Probabilistic decision making by slow reverberation in cortical circuits. *Neuron* 36, 955–968. doi: 10.1016/S0896-6273(02)01092-9

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Received: 04 October 2013; paper pending published: 18 November 2013; accepted: 14 January 2014; published online: 04 February 2014.

*Citation: Stimberg M, Goodman DFM, Benichoux V and Brette R (2014) Equation-oriented specification of neural models for simulations. Front. Neuroinform. **8**:6. doi: 10.3389/fninf.2014.00006*

This article was submitted to the journal Frontiers in Neuroinformatics.