NEST Modeling Language:
A modeling language for spiking neuron and synapse models for NEST

I. Blundell, D. Plotnikov, J. M. Eppler

Software Engineering
RWTH Aachen
FZ Jülich
Wait, yet another modeling language?

How standards proliferate:
(See: A/C chargers, character encodings, instant messaging, etc.)

**SITUATION:**
There are 14 competing standards.

14?! Ridiculous!
We need to develop one universal standard that covers everyone’s use cases.

Yeah!

**SOON:**

**SITUATION:**
There are 15 competing standards.
Wait, yet another modeling language?
The neural simulation tool NEST

- NEST is a hybrid parallel (OpenMP+MPI) simulator for spiking neural networks, written in C++, but with a Python frontend
- Neuron models are mainly point neurons and phenomenological synapse models (STDP, STP, neuromodulation)
- NEST supports large-scale models on the largest supercomputers
- Still the code also runs fine on laptops and workstations

- Get publication and source code on http://nest-simulator.org

1.73 billion neurons
10.4 trillion synapses
82,944 processors
1 PB main memory
40 minutes / second
The zoo of models

NEST 2.10.0 has 36 neuron models built in
   19 are simple integrate-and-fire models
   2 are based on the Hodgkin&Huxley formalism
   11 have alpha-shaped post-synaptic responses
   10 use exponentially decaying post-synaptic responses
   15 with current-based dynamics solved exactly
   9 conductance-based neurons using different solvers
   plus some more exotic specimen

   … and the situation gets worse each release
   and each new modelling study
The zoo of models

NEST 2.10.0 has 36 neuron models built in
19 are simple integrate-and-fire models
2 are based on the Hodgkin&Huxley formalism
11 have alpha-shaped post-synaptic responses
10 use exponentially decaying post-synaptic responses
15 with current-based dynamics solved exactly
9 conductance-based neurons using different solvers
plus some more exotic specimen

… and the situation gets worse each release and each new modelling study
1. Copy & paste
2. Modify parts of the code
3. Ideally adapt the comments ;-)
4. Add to Makefiles
5. Re-compile and test
6. Goto 2…
The current process for model creation and the diversity leads to problems

- Copy & paste leads to errors and bad maintainability
- Implementation by non-programmers, often by trial and error

Basic NESTML features

- Semantic model checking and automatic choice of solver
- Automatic adaptation to new API versions
- Library for commonly used neuron dynamics and synaptic responses
- Ease of use
The current process for model creation and the diversity leads to problems

- Copy & paste leads to errors and bad maintainability
- Implementation by non-programmers, often by trial and error

Basic NESTML features

- Semantic model checking and automatic choice of solver
- Automatic adaptation to new API versions
- Library for commonly used neuron dynamics and synaptic responses
- Ease of use
Introductory Example: An IaF PSC model with alpha shape

neuron iaf_neuron:

state:
    y0, y1, y2, V_m mV [V_m >= -99.0]
    # Membrane potential
    alias V_rel mV = V_m + E_L

end

function set_V_rel(v mV):
    V_m = v - E_L

end

parameter:
    # Capacity of the membrane.
    C_m pF = 250 [C_m > 0]

end

internal:
    h ms = resolution()
    P11 real = exp(-h / tau_syn)
    ...
    P32 real = 1 / C_m * (P33 - P11)
    / (-1/tau_m - -1/tau_syn)

end

input:
    spikeInh <- inhibitory spike
    spikeExc <- excitatory spike
    currentBuffer <- current

end

output: spike

dynamics timestep(t ms):
    if r == 0: # not refractory
        V_m = P30 * (y0 + I_e) + P31 * y1 + P32 * y2 + P33 * V_m
    else:
        r = r - 1
    end

    # alpha shape PSCs
    V_m = P21 * y1 + P22 * y2
    y1 = y1 * P11
    y0 = currentBuffer.getSum(t);

end

end

Introductory Example:

Fist class domain concepts

Gaurds

SI Units
Major Building blocks Blocks 1/2

- **State block**
  - Variables describing the neuron’s state
  - *alias* to express a dependency (also in another block possible)

- **Parameter block**
  - Values adjustable during instantiation
  - Guard checks

- **Internal**
  - Capture helper variables

```plaintext
state:
  V_m mV [V_m >= -99.0]
  # Membrane potential
  alias V_rel mV = V_m + E_L
end

parameter:
  # Capacity of the membrane.
  C_m pF = 250 [C_m > 0]
end

internal:
  h ms = resolution()
P11 real = exp(-h / tau_syn)
...
P32 real = 1 / C_m * (P33 - P11)
  / (-1/tau_m - -1/tau_syn)
end
```
**Major Building blocks Blocks 2/2**

- Neuron’s dynamic is modelled in a predefined dynamics function
  - timestamp, event based

- Auxiliary helper functions

- Buffers:
  - First-order language concept
  - Semantic checks

- ODE Blocks
  - Dynamics can be defined declaratively

```plaintext
dynamics timestep(t ms):
    if r == 0: # not refractory
        V_m = P30 * (y0 + I_e) ...
    else:
        r = r - 1
    end
end

function set_V_rel(v mV):
    V_m = v - E_L
end

input:
    spikeInh <- inhibitory spike
    spikeExc <- excitatory spike
    cur <- current
end

output: spike

ODE:
    G := E/tau_syn) * t * exp(-1/tau_syn*t)
    d/dt V := -1/Tau * V + 1/C_m * G + I_e + cur
end
```
An IaF PSC model with alpha shape
ODE Approach

neuron iaf_neuron:
  internal:
    h ms = resolution()
    P11 real = exp(-h / tau_syn)
    ...
    P32 real = 1 / C_m * (P33 - P11)
    / (-1/tau_m - -1/tau_syn)
  end

dynamics timestep(t ms):
  if r == 0:  # not refractory
    V_m = P30 * (y0 + I_e) + P31 *
        z1 + P32 * y2 + P33 * y3
  else:
    r = r - 1
  end
  # alpha shape PSCs
  V_m = P21 * y1 + P22 * V_m
  y1 = y1 * P11
  end

neuron iaf_neuron_ode:
  internal:
    h ms = resolution()
  end

dynamics timestep(t ms):
  if r == 0:  # not refractory
    ODE:
      G := E/tau_syn) * t * exp(-1/tau_syn*t)
      d/dt V:=-1/Tau * V + 1/C_m * G + I_e +cur
  else:
    r = r - 1
  end
  # alpha shape PSCs
  V_m = P21 * y1 + P22 * V_m
  y1 = y1 * P11
  end

Current equations
Membran potential
Model cross-referencing

```python
import PSPHelpers

neuron iaf_neuron:
  use PSPHelpers as PSP

  dynamics timestep(t ms):
    PSP.computePSPStep(t)
    # alpha shape PSCs
    y2 = P21 * y1 + P22 * y2
    y1 = y1 * P11
  end

  ...
end
```

```python
component PSPHelpers:
  state:
    - y0, y1, y2, V_m mV [V_m >= -99]
    alias V_rel mV = y3 + E_L
  end

  function computePSPStep(t ms):
    if r == 0: # not refractory
      y3 = P30 * (y0 + I_e) + P31 *
           y1 + P32 * y2 + P33 * y3
    else:
      r = r - 1
    end

  end
  ...
end
```
MontiCore Language Workbench

- Opensource and free github project
- Grammar based
- Definition of modular language fragments
- Assistance for analysis, transformations
- Generates: parsers, symbol tables, language processing infrastructure

Composition of languages:
- independent language development
- composition of languages and tools
- Language extension
- Language inheritance (allows replacement)

Quick definition of domain specific languages (DSLs)
- by reusing existing languages
- variability in syntax, context conditions, generation, semantics
Language Architecture of NESTML

NESTML
Nest Modeling Language
Description of the neuron models

PL
Precedural Language:
Description of the imperative parts (e.g. definition of the dynamics function)

ODEDSL
Definition of Ordinary Differential Equations

UnitDSL:
definition and automatic conversion of physical units
Generator Architecture for NEST

- Templated based code generation
  - Based on well founded mathematical theory
- Traceable model transformations
  - After transformations altered NESTML model is produced

**SYMPY Solver**
Computation of the exact solution

**NESTML2NEST Generator**

**NEST Code**

**Bootstrapping Code**

**NESTML**

- **neuron.h**
- **neuron.cpp**
- **bootstrap.sh**
- **Makefile.am**
- **module.h**

**H/CPP**

**SUNDIALS**

**GSL**

**NAG**
For Comfort: Editor in Eclipse for NESTML

package codegeneration.iaf_cond_alpha_implicit_module:

neuron iaf_cond_alpha_implicit_neuron:

state:
  V_m real = 0
  DGI real = 1
  GI real = 1
  DGE real = 1
  GE real = 1
  - r integer
end

parameters:
  V_th mV = -55.0 # Threshold Potential in mV
  V_reset mV = -60.0 # Reset Potential in mV
  t_ref ms = 2.0 # Refractory period in ms
  g_L ms = 16.666 # Leak Conductance in mS
  C_m pf = 250.0 # Membrane Capacitance in pF
  alias Tau ms = (1 / g_L) * C_m

  V reversal E mV = 0 # Excitatory reversal Potential in mV
  V reversal I mV = -85.0 # Inhibitory reversal Potential in mV
  E_L mV = -70.0 # Leak reversal Potential (aka resting potential) in mV
  tau_synE ms = 0.2 # Synaptic Time Constant Excitatory Synapse in ms
  tau_synI ms = 2.0 # Synaptic Time Constant Inhibitory Synapse in ms
  I_e pA = 0 # Constant Current in pA
end

function set_Tau(v ms):
  end

internal:
  h ms = resolution()

  # Impulse to add to DG_EXC on spike arrival to evoke unit-amplitude
Current State and Future Work

- Open-source github project

- First evaluation during a community workshop
  - Participant wrote NESTML models and ran them in NEST under 30 minutes
  - Also without preliminary experience with NEST or NESTML

- Publication: NESTML: a modeling language for spiking neurons
  - (to appear in spring 2016)

- Support for:
  - Explicit solvable models
    - E.g. PSC models in the NEST context
  - Numerical solvers
    - For now the GSL solver is already integrated

- New modeling concepts and optimisations
  - E.g. struct of arrays
  - Multi-compartment models

- Targeting new platforms
  - GPU
  - SpiNNaker
Backup
ODE Processing Workflow

\[ G = \left(\frac{E}{\tau_{\text{syn}}} \right) * t * \exp\left(-\frac{1}{\tau_{\text{syn}}} \cdot t\right) \]
\[ \frac{d}{dt} V = -\frac{1}{\tau} \cdot V + \frac{1}{C_m} \cdot G \]

For the ODE a SymPy-Solver is generated and executed.

The matrix is parsed and a new NESTML Model with the solution matrix is created.
Every another unit is defined as a combination of base units:

\[ Q = L^\alpha \cdot M^\beta \cdot T^\gamma \cdot I^\delta \cdot \Theta^\varepsilon \cdot N^\zeta \cdot J^\eta \]

E.g. volt is defined as.

\[ V = m^2 \cdot kg \cdot s^{-3} \cdot A^{-1} \cdot K^0 \cdot mol^0 \cdot cd^0 \]