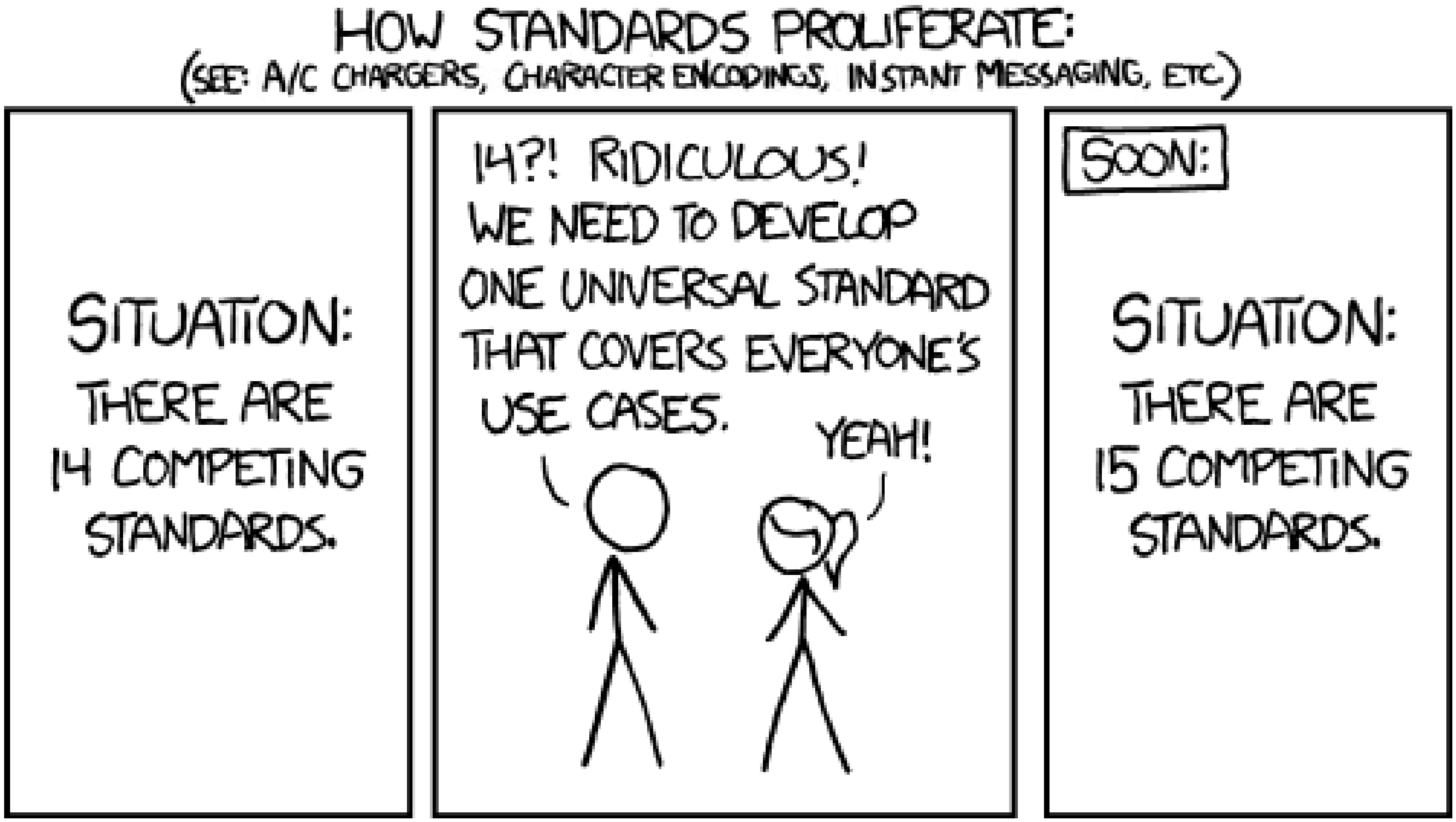
NEST Modeling Language:
A modeling language for spiking neuron and synapse models for NEST

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

Software Engineering
RWTH Aachen
FZ Jülich

AN INITIATIVE OF

Wait, yet another modeling language?

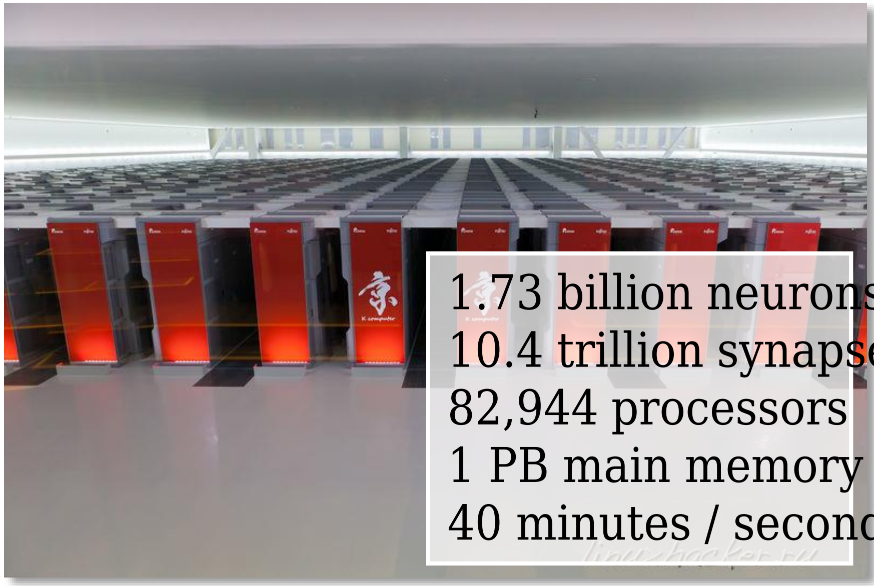


Blundell, Plotnikov
Lehrstuhl für
Software Engineering
RWTH Aachen

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



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

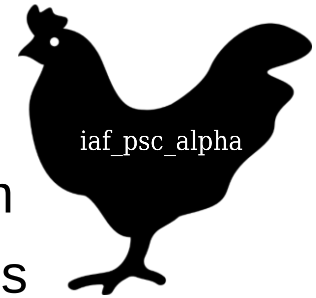
Blundell, Plotnikov
Lehrstuhl für
Software Engineering
RWTH Aachen

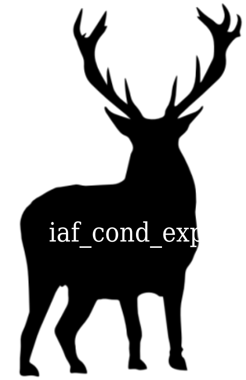
The zoo of models

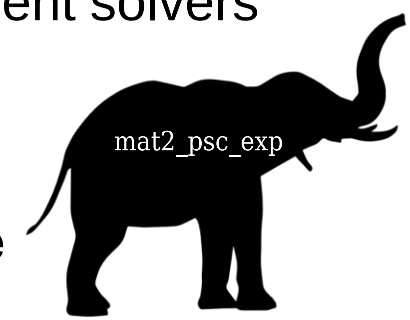
NEST 2.10.0 has 36 neuron models built in

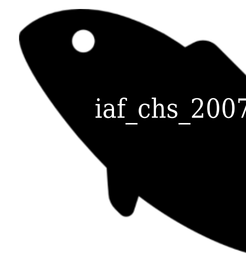
- 19 are simple integrate-and-fire models
- 2 are based on the Hodgkin&Huxley formalism
- 11 have alpha-shaped post-synaptic responses
- 10 use exponentially decaying post-synaptic responses
- 15 with current-based dynamics solved exactly
- 9 conductance-based neurons using different solvers
- plus some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

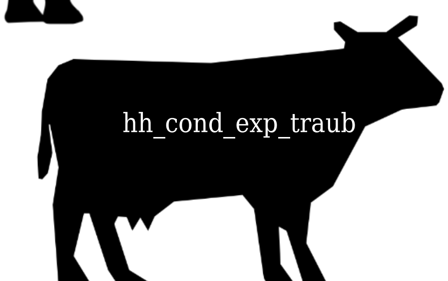
… and the situation gets worse each release
and each new modelling study

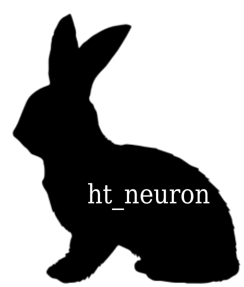












Blundell, Plotnikov
Lehrstuhl für
Software Engineering
RWTH Aachen

Creating neuron models

```
void iaf_psc_alpha::update(Time const & origin, const long_t from, const long_t to)
{
  assert(to >= 0 && (delay) from < Scheduler::get_min_delay());
  assert(from < to);

  for ( long_t lag = from ; lag < to ; ++lag )
  {
    if ( S_.r_ == 0 )
    {
      // neuron not refractory
      S_.y3_ = V_.P30_*(S_.y0_ + P_.I_e_)
             + V_.P31_ex_ * S_.y1_ex_ + V_.P32_ex_ * S_.y2_ex_
             + V_.P31_in_ * S_.y1_in_ + V_.P32_in_ * S_.y2_in_
             + V_.expm1_tau_m_ * S_.y3_ + S_.y3_;

      // lower bound of membrane potential
      S_.y3_ = ( S_.y3_ < P_.LowerBound_ ? P_.LowerBound_ : S_.y3_);
    }
    else // neuron is absolute refractory
      --S_.r_;

    // alpha shape EPSCs
    S_.y2_ex_  = V_.P21_ex_ * S_.y1_ex_ + V_.P22_ex_ * S_.y2_ex_;
    S_.y1_ex_ *= V_.P11_ex_;

    // Apply spikes delivered in this step; spikes arriving at T+1 have
    // an immediate effect on the state of the neuron
    V_.weighted_spikes_ex_ = B_.ex_spikes_.get_value(lag);
    S_.y1_ex_ += V_.EPSCInitialValue_ * V_.weighted_spikes_ex_;

    // alpha shape EPSCs
    S_.y2_in_  = V_.P21_in_ * S_.y1_in_ + V_.P22_in_ * S_.y2_in_;
    S_.y1_in_ *= V_.P11_in_;

    // Apply spikes delivered in this step; spikes arriving at T+1 have
    // an immediate effect on the state of the neuron
    V_.weighted_spikes_in_ = B_.in_spikes_.get_value(lag);
    S_.y1_in_ += V_.IPSCInitialValue_ * V_.weighted_spikes_in_;

    // threshold crossing
    if ( S_.y3_ >= P_.Theta_)
    {
      S_.r_  = V_.RefractoryCounts_;
      S_.y3_ = P_.V_reset_;
      // A supra-threshold membrane potential should never be observable.
      // The reset at the time of threshold crossing enables accurate integration
      // independent of the computation step size, see [2,3] for details.

      set_spiketime(Time::step(origin.get_steps()+lag+1));
      SpikeEvent se;
      network()->send(*this, se, lag);
    }

    // set new input current
    S_.y0_ = B_.currents_.get_value(lag);

    // log state data
    B_.logger_.record_data(origin.get_steps() + lag);
  }
}
```

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…

```
void nest::iaf_cond_alpha::update(Time const & origin, const long_t from, const long_t to)
{

  assert(to >= 0 && (delay) from < Scheduler::get_min_delay());
  assert(from < to);

  for ( long_t lag = from ; lag < to ; ++lag )
  {

    double t = 0.0;

    // numerical integration with adaptive step size control:
    // ------------------------------------------------------
    // gsl_odeiv_evolve_apply performs only a single numerical
    // integration step, starting from t and bounded by step;
    // the while-loop ensures integration over the whole simulation
    // step (0, step] if more than one integration step is needed due
    // to a small integration step size;
    // note that (t+IntegrationStep > step) leads to integration over
    // (t, step] and afterwards setting t to step, but it does not
    // enforce setting IntegrationStep to step-t; this is of advantage
    // for a consistent and efficient integration across subsequent
    // simulation intervals
    while ( t < B_.step_ )
    {
      const int status = gsl_odeiv_evolve_apply(B_.e_, B_.c_, B_.s_,
                         &B_.sys_,             // system of ODE
                         &t,                   // from t
                          B_.step_,            // to t <= step
                         &B_.IntegrationStep_, // integration step size
                          S_.y);               // neuronal state


      if ( status != GSL_SUCCESS )
        throw GSLSolverFailure(get_name(), status);
    }

    // refractoriness and spike generation
    if ( S_.r )
    {// neuron is absolute refractory
        --S_.r;
        S_.y[State_::V_M] = P_.V_reset;  // clamp potential
    }
    else
      // neuron is not absolute refractory
      if ( S_.y[State_::V_M] >= P_.V_th )
      {
     S_.r              = V_.RefractoryCounts;
     S_.y[State_::V_M] = P_.V_reset;

     // log spike with Archiving_Node
     set_spiketime(Time::step(origin.get_steps()+lag+1));

     SpikeEvent se;
     network()->send(*this, se, lag);
      }

    // add incoming spikes
    S_.y[State_::DG_EXC] += B_.spike_exc_.get_value(lag) * V_.PSConInit_E;
    S_.y[State_::DG_INH] += B_.spike_inh_.get_value(lag) * V_.PSConInit_I;

    // set new input current
    B_.I_stim_ = B_.currents_.get_value(lag);

    // log state data
    B_.logger_.record_data(origin.get_steps() + lag);
  }
}
```

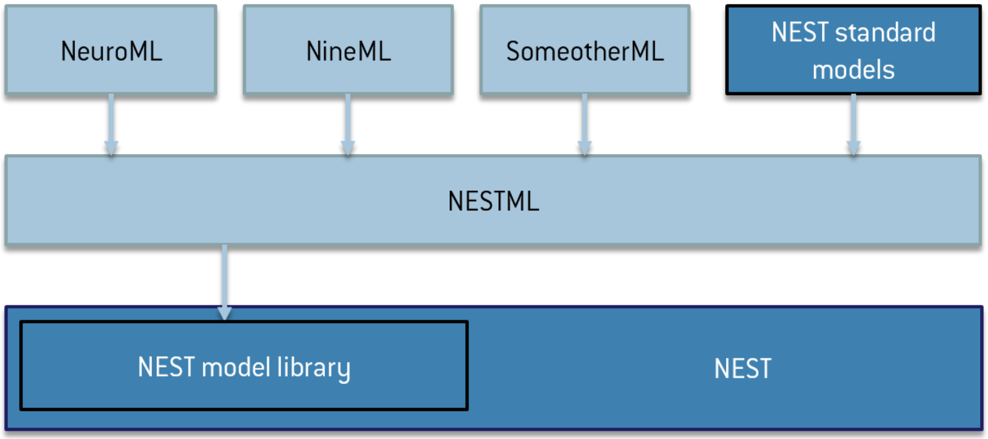
NESTML

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



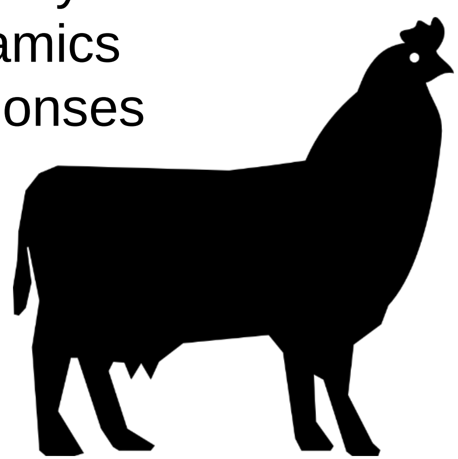
NESTML

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



Blundell, Plotnikov
Lehrstuhl für
Software Engineering
RWTH Aachen

Introductory Example:
An IaF PSC model with alpha shape

Fist class domain concepts

SI Units

Gaurds

```
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
```

Major Building blocks Blocks 1/2

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

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

- Parameter block
  - Values adjustable during instantiation
  - Guard checks

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

- Internal
  - Capture helper variables

```
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

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

- Auxiliary helper functions

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

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

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

output: spike
```

- ODE Blocks
  - Dynamics can be defined declaratively

```
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

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

     end
   else:
     r = r - 1
   end
   ...
 end
 ...
end
```

Current equations

Membran potential

Model cross-referencing

Imports a component

Uses imported component

```
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
```

```
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

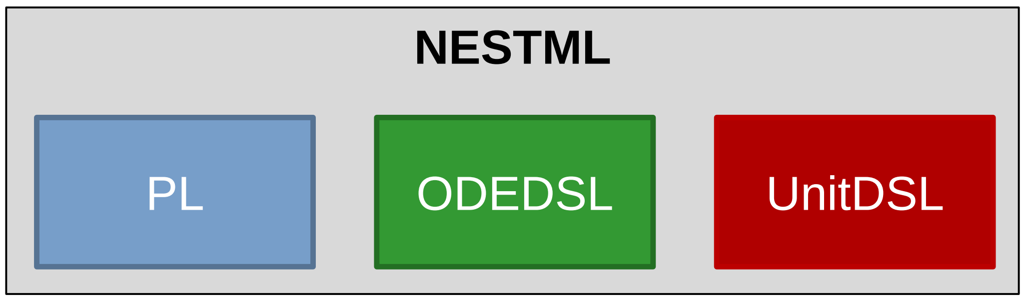
- Composition of languages:
  - independent language development
  - composition of languages and tools
  - Language extension
  - Language inheritance (allows replacement)

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

Blundell, Plotnikov
Lehrstuhl für
Software Engineering
RWTH Aachen

Language Architecture of NESTML

NESTML
Nest Modeling Language
Description of the neuron models



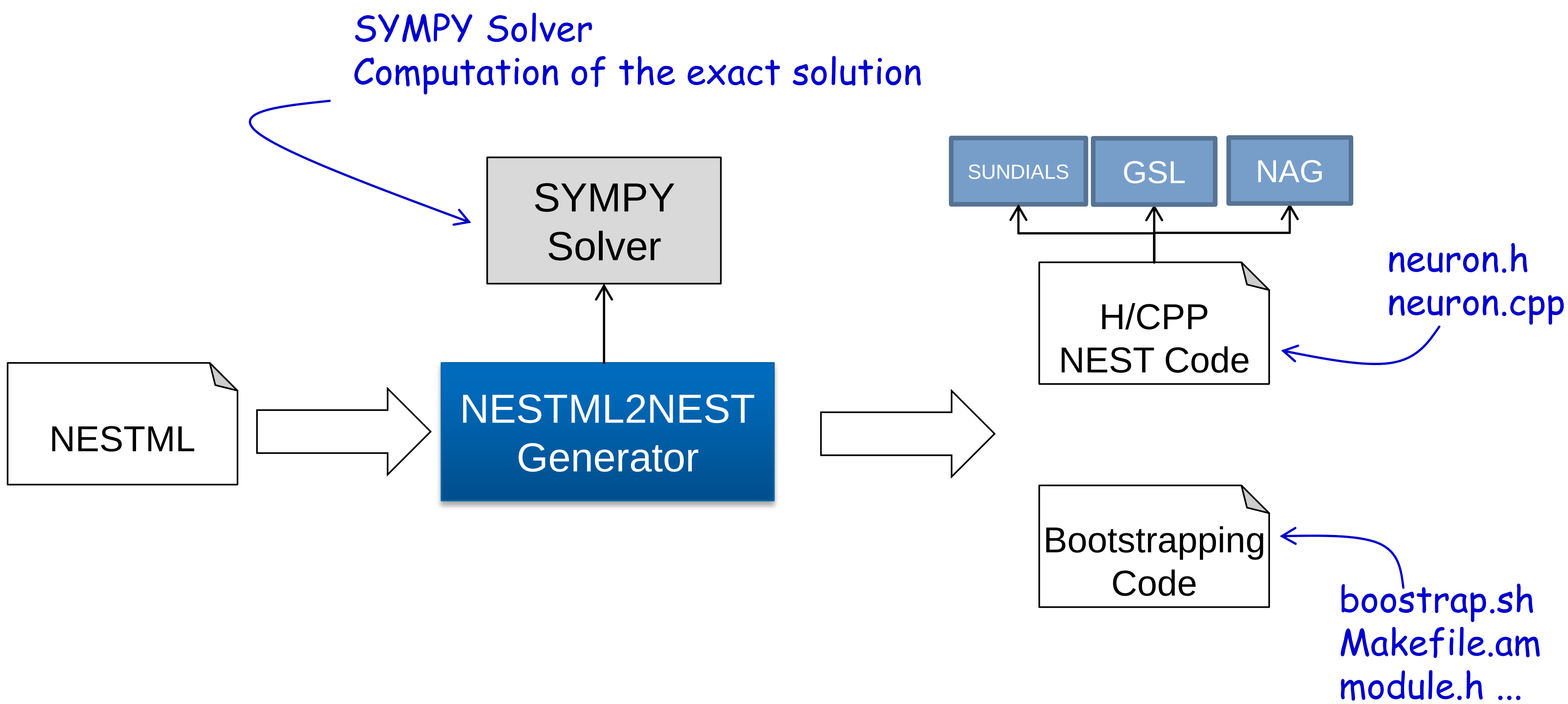
PL
Precedural Language:
Description of the imperative parts (e.g. definition of the dynamics function)

ODEDSL
Definition of
Ordinary Differential Equations

UnitDSL:
definition and automatic conversion of physical units

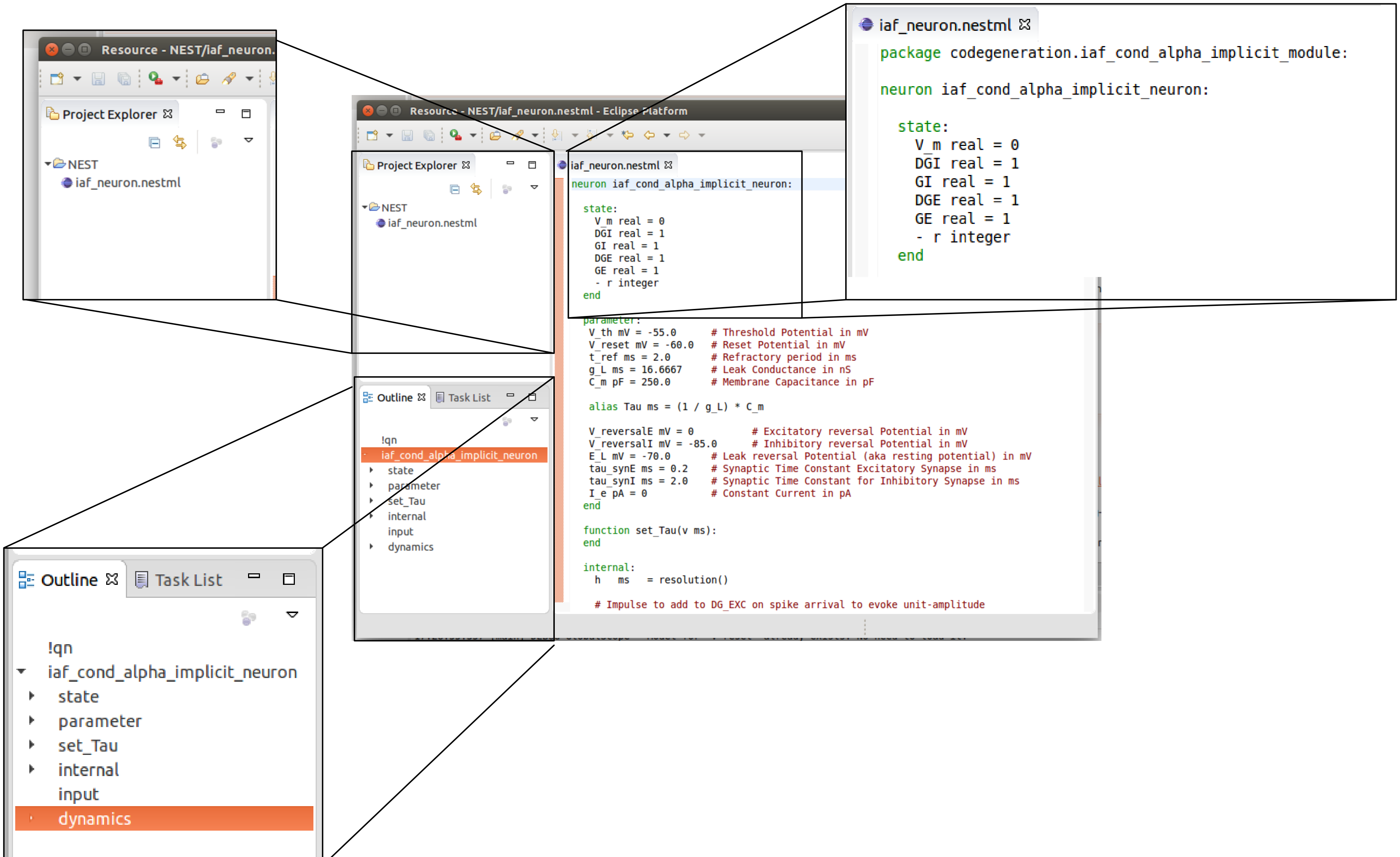
Generator Architecture for NEST

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



Blundell, Plotnikov
Lehrstuhl für
Software Engineering
RWTH Aachen

For Comfort: Editor in Eclipse for NESTML



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

NESTML

```
G ===(E/tau_syn) * t * exp(-1/tau_syn*t)
d/dt V === -1/Tau * V + 1/C_m * G
```

SymPy

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

Text

```
...
h*exp(-h/tau_in)# P10
exp(-h/tau_in)# P11
...
```

The matrix is parsed and a new NESTML Model with the solution matrix is created

NESTML

```
internal:
...
  P10 = h*exp(-h/tau_in)
  P11 = exp(-h/tau_in)
...
end
```

H/CPP
NEST Code

SI Units Specification

Größen-Name	Größen-Zeichen	Einheiten-Name	Einheiten-Zeichen
Länge	l	Meter	m
Masse	m	Kilogramm	kg
Zeit	t	Sekunde	s
Stromstärke	I	Ampere	A
Temperatur	T	Kelvin	K
Stoffmenge	n	Mol	mol
Lichtstärke	I_V	Candela	cd

- 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.

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