The NEST code generation roadmap: rationale and methods

March 16, 2012 | Jochen Martin Eppler

BrainScaleS CodeJam #5, Edinburgh

JÜLICH
FORSCHUNGSZENTRUM

Outline

- Reasons for code generation
- The neural simulation tool NEST
- Problems with our current way of writing code
- State of code generation from lib9ML for NEST
- Performance considerations
- Open questions, outlook and discussion

”If syntactic sugar didn't count, we'd all be programming in assembly language.“

C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond - David Abrahams, Aleksey Gurtovoy

Why do we need code generation?

- Driven by advances in neuroscience, our simulation software becomes more and more complex
  - More neuron and synapse models with higher complexity
  - New modeling directions require more complex networks
  - Larger machines are required to simulate larger networks

- Programming languages with higher levels of abstraction are one possibility to master the complexity
- Model descriptions (e.g. NineML, NeuroML) are another
- High-level descriptions should not have to be interpreted at run-time, but should be compiled to or at least handed over to simulators

The neural simulation tool NEST

- NEST is a simulator for spiking neural networks
  - Distributed and multi-threaded simulation
  - Development driven by neuroscientific needs

- Neuron and synapse model types have to be written in C++
  - This requires users to know some internals of NEST
  - New models are often created from existing models
  - Custom models can be defined in modules and loaded at run-time

- The research focus shifts further towards large-scale simulations

Why model development in NEST is broken

- Neuron and synapse models are not simulator independent
- Updates of the internal API require changes in all standard models
  - This reduces maintainability and increases the risk of error
  - Most of the model code is boilerplate code
- NEST's architecture for coupling synapses and neurons only allows one synapse to be in between
  - It is impossible to combine synapse types, even though the code for the single types may be there
  - Code Generation allows a merge of multiple types
- Multiple code paths are currently defined in a single file using ifdefs

Python, PyNN, NineML and beyond

- With Python, the scientific community has a language with outstanding readability and superior productivity (cf. Prechelt, 2000)
- With PyNN, the computational neuroscience has a solid tool for specifying neural network models
- With NineML, a standard for neural network model descriptions is on its way

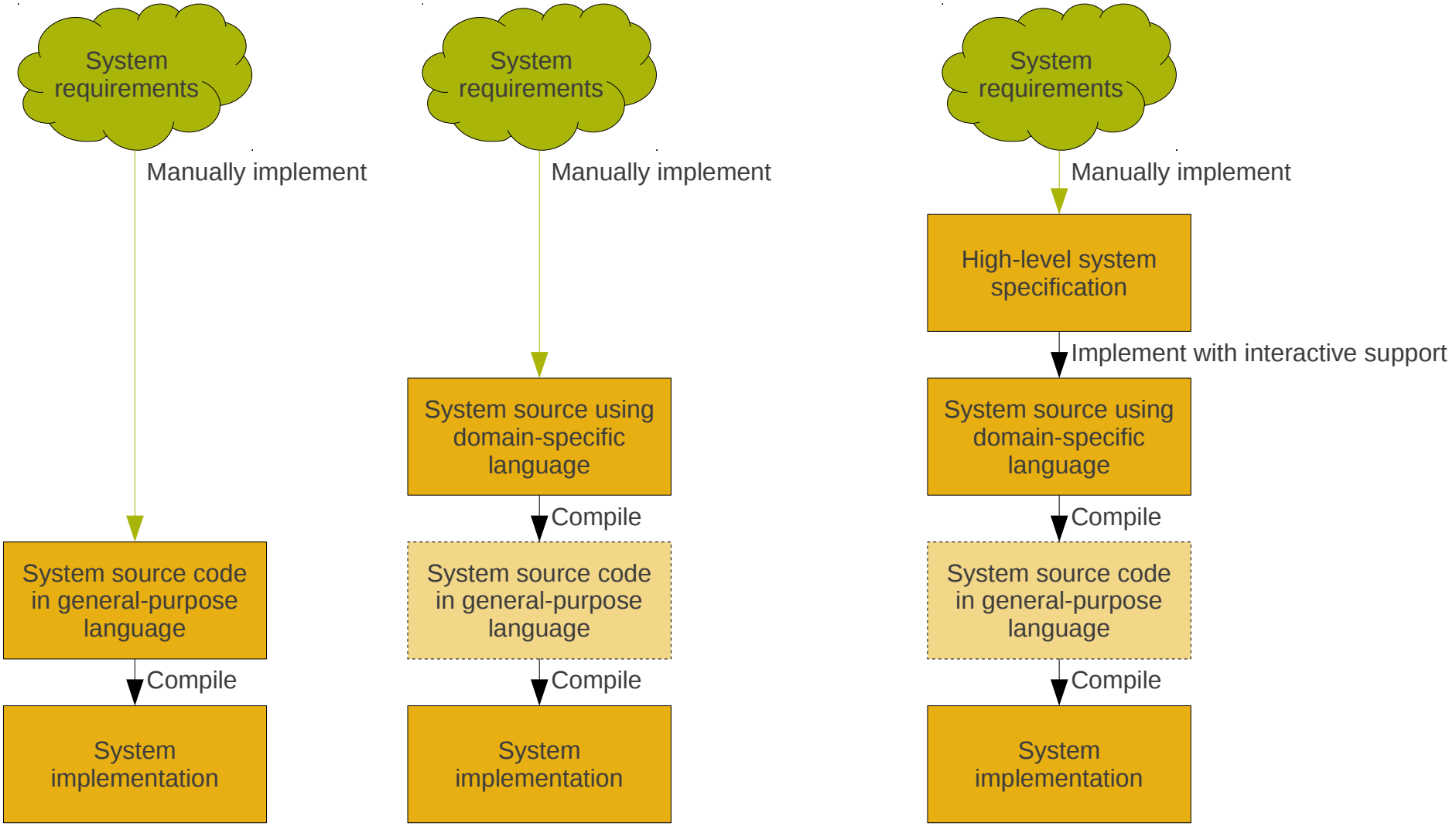
- The next logical step is to generate implementations from high-level model descriptions
- Code generation is the right step in this direction

Code generation

- Generative/automatic programming is all about bringing the benefits of automation to software development
- Automatic code generation allows to write code faster, as only a high-level description has to be provided

- Several approaches exist:
  - Generators
  - Computer-aided software engineering
  - Domain-specific languages
  - Metaprogramming (e.g. C++ templates)

Code generation



Code generation

- A *generator* is a program that takes a higher-level specification of a piece of software and produces its implementation

- Different kinds of generators exist:
  - Written from scratch (e.g. using bash or Python). NEST already uses this technique to write header files during configuration
  - Based on the metaprogramming facilities of a programming language. NEST heavily uses C++ templates
  - Using a generator infrastructure. For example a user interface designer
  - Using a template engine, which does text based replacements

Code generation

- Written from scratch

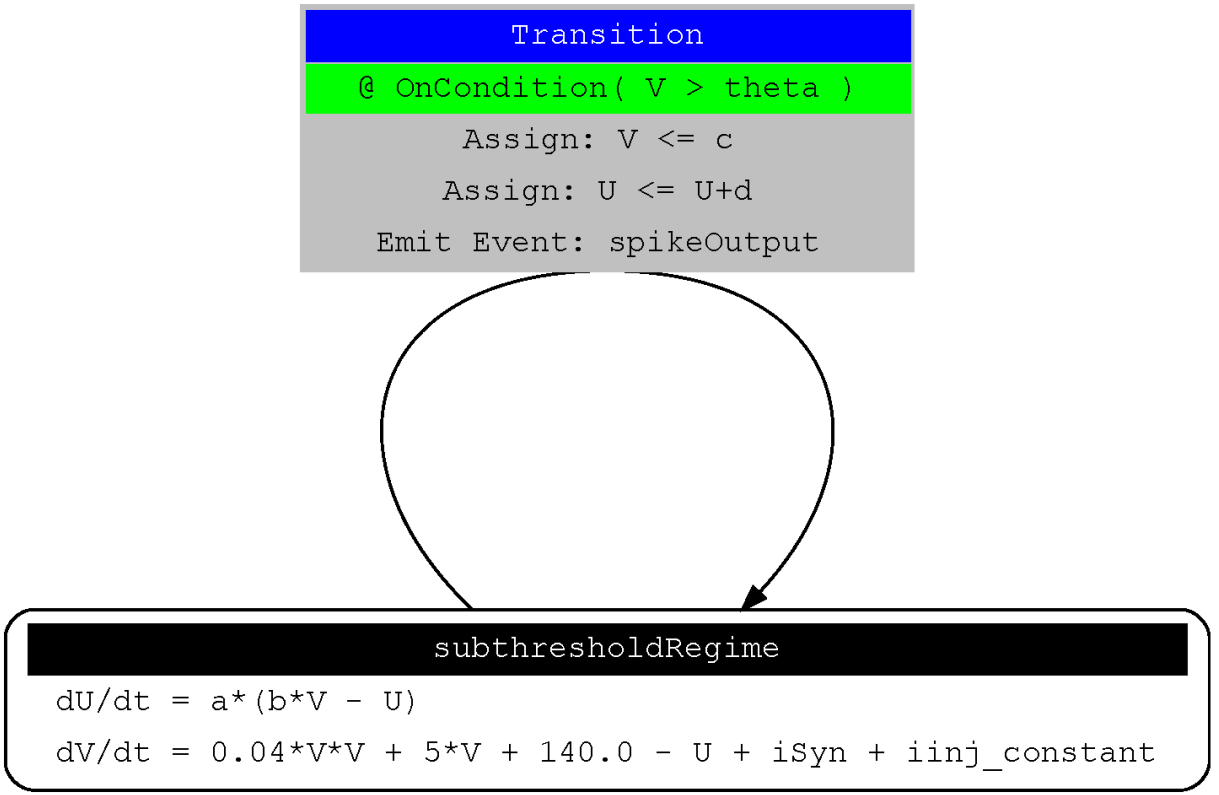
```
#! /bin/bash
echo "#include <iostream>"
echo "main() { std::cout << \"$1\" << std::endl; }"
```

- Using the metaprogramming facilities of a programming language

```
#include <iostream>
#include <string>
template <class T> T sum(T a, T b) { return a+b; }
main() {
  std::cout << sum(1.0, 2.0) << std::endl;
  std::cout << sum(1, 2) << std::endl;
}
```

Excursion to lib9ML

- lib9ML is a simulator independent object model describing the different elements of network models
- Dynamics are described by a set of state variables, regimes and transitions combined in a regime graph



State of lib9ML code generation for NEST

- A prototype for generating neuron models was provided by Eilif Muller
- Based on Cheetah, a text-based template engine for Python

- The notion of regimes and transitions maps nicely on how NEST likes to see its neuron models
- Susanne Kunkel and Abigail Morrison changed the template for neurons to make a compilable file for synapses
- For synapse models, the notion of regimes and transitions is a less good match
  - Regimes are expressed as ODEs, which are time-driven
  - NEST thinks about synapse in an event-driven fashion

State of lib9ML code generation for NEST

- A prototype for generating neuron models was provided by Eilif Muller
- Based on Cheetah, a text-based template engine for Python

- The notion of regimes and transitions maps nicely on how NEST likes to see its neuron models
- Susanne Kunkel and Abigail Morrison changed the template for neurons to make a compilable file for synapses
- For synapse models, the notion of regimes and transitions is a less good match
  - Regimes are expressed as ODEs, which are time-driven
  - NEST thinks about synapse in an event-driven fashion

State of lib9ML code generation for NEST

[Have a look at the templates]

State of lib9ML code generation for NEST

- Code generation still involves some manual steps
  - Creating the lib9ML description
  - Generating C++ code from the high-level description
  - Adding the code to MyModule
  - Dynamically load MyModule in NEST at run-time

- PyNEST needs functions to allow the user to specify models in a convenient way
- A just-in-time compilation infrastructure is needed

Problems with lib9ML and text-based code generation

- lib9ML is simulator agnostic by design

- Things like syntax and type checks are only performed later in the development cycle
- Decision for the right solvers can be hard, optimizations even harder
- Things like consistency and range checks for variables or variables defined relative to each other are hard
- Implementation details (e.g. different buffers for inh./exc. Spikes) cannot be expressed
- Searching for errors may become more complex as more levels of software are involved

Benefits of using code generation

- Speed: C/C++ is (much) faster than Python or interpreting XML
- Usability: Writing Python/XML is easier than C/C++ (Prechelt, 2000)

- Reduce the boiler-plate code that has to be written for each model
- Reduce the error-prone-ness of the code
- Improve the maintainability of the code

- Does a high-level description without much annotation allow the generation of optimal code at the level of machine code?