FACETS

FP6-2004-IST-FETPI 15879

Fast Analog Computing with Emergent Transient States

Concept of a common data model for neuroscience simulations

Report Version: 1.0 (*Revision* : 213)

Report Preparation: Andrew Davison, Eilif Müller and T. Viéville

Classification: PU

Contract Start Date: 01/09/2005 Duration: 4 Years

Project Co-ordinator: Karlheinz Meier (Heidelberg)

Partners: U Bordeaux, CNRS (Gif-sur-Yvette, Marseille), U Debrecen, TU Dresden, U Freiburg, TU Graz, U Heidelberg, EPFL Lausanne, Funetics S.a.r.l., U London, U Plymouth, INRIA, KTH Stockholm

Project funded by the European Community

under the "Information Society Technologies' Programme

DELIVERABLES TABLE

Project Number: FP6-2004-IST-FETPI 15879

Project Acronym: FACETS

Title: Fast Analog Computing with Emergent Transient States

Del. No.	Revision	Title	Type[1]	Classification[2]	Due Date	Issue Date
23	1.0	Concept of a common data model for neuroscience simulations	R	PU	Month 12	12/10/2006

[1] *R: Report; D: Demonstrator; P: Prototype; O: Other – Specify in footnote*

[2] *PU: Public*
PP: Circulation within programme participants, including the Commission Services
RE: Restricted circulation list (specify in footnote), including the Commission Services
CO: Confidential, only for members of the consortium, including the Commission Services

DELIVERABLE SUMMARY SHEET

Project Number: FP6-2004-IST-FETPI 15879

Project Acronym: FACETS

Title: Fast Analog Computing with Emergent Transient States

Deliverable N^o: 23

Due date: Month 12

Delivery Date: 12/10/2006

Short Description:

One FACETS goal is to build a common data model for describing neuroscience simulation models. The FACETS project provides an ideal infrastructure for this important task by :
- re-using existing specifications to simulate WP4 and WP6 outcomes at the 'neuron' level
- developing a new set of specifications to simulate WP5 and WP7 outcomes, restraining our development to event based (spiking) neuronal assemblies.

It was agreed that after the 1st year we will attempt to integrate this initiative with the NeuroML project (considering "event-based" network models).

After this first 12 months of the project the consortium has provided and evaluated a declarative (FacetsML) and a procedural (PyNN) description of neurons and networks within the scope of this project. Both specifications are available as cooperative open-source document bundles, FacetsML being in a software forge and PyNN being in an internal FACETS repository. As a step further, a prototype WYSIWYG editor for FacetsML has been developed, and requires evaluation not only by computer scientists but also by other colleagues. All specifications are computer language independent, written in XML (XSD schema and XSL transformation) and based on W3C standards. Utility tools are developed in Java for maximal portability. Technical tools are developed in Python which is the language used by most existing simulators within the consortium. Integration between Java and Python components is straightforward, using existing tools.

Partners owning: CNRS(a), UHEI and INRIA

Partners contributed: CNRS(a), UHEI, INRIA

Made available to: public

Contents

1

Introduction

1.1 The value of multiple simulators

There are many freely-available, open-source, well-documented tools for simulation of networks of spiking neurons. Within the FACETS project, seven different such tools are in active use.[1] There is considerable overlap in the classes of network that each is able to simulate, but each strikes a different balance between efficiency, flexibility, scalability and user-friendliness, and the different simulators encompass a range of simulation strategies. This makes the choice of which tool to use for a particular project a difficult one, and we would argue moreover that using just one simulator is an undesirable state of affairs. This follows from the general principle that scientific results must be reproducible to be valid, and that any given instrument may have flaws or introduce systematic bias: the simulators used in computational neuroscience are complex software packages, and may have hidden bugs or unexamined assumptions that may only be apparent in particular circumstances. Therefore it is desirable that any given model should be simulated using at least two different simulators and the results cross-checked.

This, however, is more easily said than done. The configuration files, scripting languages or graphical interfaces used for specifying model structure are very different for the different simulators, and this, together with subtle differences in the implementation of conceptually-identical ideas, makes the conversion of a model from one simulation environment to another an extremely non-trivial task; as such it is rarely undertaken.

We believe that the field of computational neuroscience in general, and the FACETS project in particular, would gain greatly from the ability to easily simulate a model with multiple simulators. First, it would greatly reduce implementation-dependent bugs, and possible subtle systematic biases due to use of an inappropriate simulation strategy. Second, it would facilitate communication between investigators and reduce the current segregation into simulator-specific communities; this, coupled to a willingness to publish actual simulation code as well as the more traditional model description, would perhaps lead to reduced fragmentation of research effort and an increased tendency to build on existing models rather than redevelop them *de novo*. Third, it would lead to a general improvement in simulator technology since bugs could be more easily identified, benchmarking greatly simplified, and hence best-practice more rapidly propagated.

1.2 Simulator-independent model specification

To easily simulate models with multiple simulators, we require a common language in which to express neuroscience models. In order to develop such a language, we need to itemize all of the concepts and objects that are used in computational neuroscience simulations, and the relationships between these them. The outcome of such a process of organizing the information about a particular domain is often referred to as a *data model*.

Existing data models are for instance:

[1] The seven are NEURON, NEST, CSIM, SPLIT, NCS, MVASpike and MONSTER.

- BrainML list of terms (http://brainml.org/) – tries to cover the whole of computational neuroscience, but is not yet published.
- NeuroML – more compact, fully defined for a limited scope, that scope is close to what we want to do.

We thus use the NeuroML standard as the basis of our data model and modify/extend it as required. NeuroML is introduced in more detail in Section 2.

1.3 General requirements

In order to be acceptable a system of specification has to be:

- Theoretically well founded. For example, algorithms provided by the system should have a formal specification (pre- and post-conditions). The complexity of the algorithms should be known. Data structures should also be described formally. Concurrency and real-time control should be based on established theory.
  Simplicity eases the development of a formal model of the system to be able to analyze its behaviour, predict its properties, etc..
- Stratified. It is crucial for such a system to be properly decomposed and layered, with well-defined interfaces to every part of the implementation, so that it will be possible to choose alternative implementations for some aspects. One reason for choosing an alternative implementation is increased efficiency, another is integration and compatibility with other systems.
  For the same reason, the implementation must not contain anything that could just as well be implemented in add-on libraries (here: stratification = hierarchy + modularity).
- Hierarchy-based. In order to organize and structure such a system, we require the concept of hierarchy (a system is hierarchical if each entity can be used as a component of another entity) where objects are only defined via their interface, while the physical implementation is independent (here abstraction = interface + implementation) and the interface is realized via parameterisation, i.e either values or pure functions.
- With maximal parameterisation. As much as possible, the system parameters have to be defined as a static set of structured data (instead of "put-in-the-code"), thus yielding to the design of generic software modules. Here, a parameter is not only a "datum" but also a symbolic function: e.g. the operator of a general filtering mechanism, the cost function of a criterion to be minimized, a measurement equation, etc.
- Really modular in practice. Many existing systems contain deep type hierarchies with a single root. In such systems, all useful function tends to drift towards the root, whose interface quickly gets fat. The single root also makes it virtually impossible to combine the type hierarchy with other systems. In contrast, new object-oriented systems tend to contain small, shallow, orthogonal type hierarchies which are combined with multiple inheritance (mix-in techniques).
- With weakly coupled components. In order to manage mechanisms of huge complexity, a system is modular if it is built of simple uniform entities (e.g. objects, cells, atoms, etc.) which are weakly coupled (e.g. independent, distributed, etc.).
  Modularity implies the ability to use what has been developed in the past and what will be developed in the future. It is important to create embedded applications for which you do not need to "download" more than what is actually used. The fact that each entity is simple and independent makes it more likely that implementation can be fast and easy to control.
- Open ended. There is no way that a system can provide support for every conceivable algorithm. Since such a system tries to cover applications, it will tend to be too large, monolithic, and hard

to use and support if not modular.
Therefore, such a system must consist of a small kernel to which problem-oriented code is added as modules.

- Well engineered. The æsthetic aspect of a system should not be underestimated. Most successful software packages are good compromises between generality, efficiency, and size. The concepts are often simple and straight-forward. The software is usually built with standard components, as far as possible, and uses wide-spread, well suited programming languages.
- With a progressive learning scheme. This is also important in order to create embedded applications where you do not need to "download" more than what is actually used. Moreover, it could enable a "progressive learning scheme" so that a newcomer does not need to learn more than is basically needed for his application to get started.
- Based on mainstream technology. Some systems invent their own specification languages, scripting languages, graphical interface, when there are existing, generally accepted alternatives. The use of mainstream components increases the probability that the system will be accepted and used.
- With support for development. Several existing systems lack adequate tools for testing and debugging new applications. The system must be robust and, e.g., detect crashed processes and communication problems. It must be possible to monitor program execution, especially in distributed systems, and to trace events.

In other words: small is beautiful, but not only this: it is also more manageable, easy to learn, more reliable, cheaper and faster.

Implementation of the common data model

Although NeuroML uses XML for model specification, we attempt to make our data model independent of the particular implementation technology. We therefore present two alternative/complementary implementations, a specification using XML-Schema, called FacetsML (Section 4), and an API using Python, called PyNN (Section 5).

2

NeuroML data model

As noted above, we have chosen to base our common data model on that developed by the NeuroML project, and to extend it where necessary. Therefore, we here summarise the NeuroML project and data model.[1]

2.1 Declarative model specification using NeuroML

The NeuroML project is an open-source collaboration whose stated aims are:

1. To support the use of declarative specifications for models in neuroscience using XML.

2. To foster the development of XML standards for particular areas of computational neuroscience modeling.

The following standards have so far been developed:

- **MorphML**: specification of neuroanatomy (i.e. neuronal morphology)
- **ChannelML**: specification of models of ion channels and receptors (see Figure 2.1 for an example)
- **Biophysics**: specification of compartmental cell models, building on MorphML and ChannelML
- **NetworkML**: specification of cell positions and connections in a network.

The common syntax of these specifications is XML (Extensible Markup Language). This has the advantages of being both human- and machine-readable, and standardized by an international organization, which in turn has led to wide uptake and developer participation.

Other XML-based specifications that have been developed in neuroscience and in biology more generally include BrainML for exchanging neuroscience data, CellML for models of cellular and subcellular processes and SMBL for representing models of biochemical reaction networks.

Although XML has become the most widely used technology for the electronic communication of hierarchically structured information, the real standardisation effort is orthogonal to the underlying technology, and concerns the structuring of domain-specific knowledge, i.e. a listing of the objects and concepts of interest in the domain and of the relationships between them, using a standardised terminology. To achieve this, NeuroML uses the XML Schema Language to define the allowed elements and structure of a NeuroML document. The validity of a NeuroML document may be checked with reference to the schema definitions. The NeuroML Validation service provides a convenient way to do this.

[1] Note that this section refers to version 1.3α of the NeuroML standards.

```
<?xml version="1.0" encoding="UTF-8"?>

<channelml xmlns="http://morphml.org/channelml/schema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:meta="http://morphml.org/metadata/schema"
  xsi:schemaLocation="http://morphml.org/channelml/schema  ../../Schemata/v1.1/Level2/ChannelML_v1.1.xsd"
  units="Physiological Units">

  <ion name="k"  default_erev="-77.0" charge="1"/>  <!-- phys units: mV -->

  <channel_type name="KChannel" density="yes">

    <meta:notes>Simple example of K conductance in squid giant axon.
        Based on channel from Hodgkin and Huxley 1952</meta:notes>

    <current_voltage_relation>
      <ohmic ion="k">
        <conductance default_gmax="36">  <!-- phys units: mS/cm2-->
          <gate power="4">
            <state name="n" fraction="1">
              <transition>
                <voltage_gate>
                  <alpha>
                    <parameterised_hh type="linoid" expr="A*(k*(v-d))/(1 - exp(-(k*(v-d))))">
                      <parameter name="A" value="0.1"/>
                      <parameter name="k" value="0.1"/>
                      <parameter name="d" value="-55"/>
                    </parameterised_hh>
                  </alpha>
                  <beta>
                    <parameterised_hh type="exponential" expr="A*exp(k*(v-d))">
                      <parameter name="A" value="0.125"/>
                      <parameter name="k" value="-0.0125"/>
                      <parameter name="d" value="-65"/>
                    </parameterised_hh>
                  </beta>
                </voltage_gate>
              </transition>
            </state>
          </gate>
        </conductance>
      </ohmic>
    </current_voltage_relation>
  </channel_type>
</channelml>
```

Figure 2.1. Example of Hodgkin-Huxley K^+ conductance specified in ChannelML, a component of NeuroML.

Using NeuroML for specifying network models

In order to use NeuroML to specify spiking neuronal network models we require detailed descriptions of

1. point spiking neurons (integrate and fire neurons and generalisations therefore)
2. compartmental models with Hodgkin-Huxley-like biophysics
3. large networks with structured internal connectivity related to a network topology (e.g.: full-connectivity, 1D or 2D map with local connectivity, synfire chains patterns, .. with/without randomness) and structured map to map connectivity (e.g.: point-to-point, point-to-many, ..)

At the time of writing, NeuroML supports the second and third items, but not the first. Specification of Hodgkin-Huxley-type models uses the MorphML, ChannelML and Biophysics standards of NeuroML (see Fig. 2.1 for an example), and for the FACETS project we adopt these in their entirety. Where we need to go beyond the NeuroML data model is in specification of networks, and so in the remainder of this section we introduce the NetworkML standard, with notes on its current limitations, while in the next section we present our proposed extensions.

A key point in understanding NetworkML is that a set of neurons and network connectivity may be defined either by *extension* (providing the list of all neurons, parameters and connections), for example:

```
<population name="PopulationA">
  <cell_type>CellA</cell_type>
  <instances>
    <instance id="0"><location x="0" y="0" z="0"/></instance>
    <instance id="1"><location x="0" y="10" z="0"/></instance>
    <instance id="2"><location x="0" y="20" z="0"/></instance>
    . . .
  </instances>
</population>
```

(note that `CellA` is a cell model described earlier in the NeuroML document) or by *specification*, i.e. an implicit enumeration, for example:

```
<population name="PopulationA">
  <cell_type>CellA</cell_type>
  <pop_location>
    <random_arrangement>
      <population_size>200</population_size>
      <spherical_location>
        <meta:center x="0" y="0" z="0" diameter="100"/>
      </spherical_location>
    </random_arrangement>
  </pop_location>
</population>
```

Similarly, for connectivity, one may define an explicit list of connections,

```
<projection name="NetworkConnection1">
  <source>PopulationA</source>
  <target>PopulationB</target>
  <connections>
    <connection id="0">
      <pre cell_id="0" segment_id = "0"/>
      <post cell_id="1" segment_id = "1"/>
    </connection>
    <connection id="1">
      <pre cell_id="2" segment_id = "0"/>
      <post cell_id="1" segment_id = "0"/>
    </connection>
    . . .
  </connections>
</projection>
```

or specify an algorithm to determine the connections:

```
<projection name="NetworkConnection1">
  <source>PopulationA</source>
  <target>PopulationB</target>
  <connectivity_pattern>
    <num_per_source>3</num_per_source>
    <max_per_target>2</max_per_target>
  </connectivity_pattern>
</projection>
```

2.2 Entity/object types

Here we list the types of object that are defined in the NetworkML specification, excluding those types (Populations, Projections, Instances) that essentially only contain lists of entities of other types.[2] The attributes of each type are given as *name* : *attribute type* pairs. Optional attributes are coloured pale blue, required attributes dark blue. Note that we do not use attributes in the narrow XML sense, but in the general sense of a named object or other data structure contained within a higher-level object. In the XML implementation, these attributes may be either XML attributes or XML elements.

We do not list the other NeuroML specifications (ChannelML, MorphML, etc.) for reasons of space, and because the NetworkML specification is most important in understanding our proposed extensions to the NeuroML data model.

2.2.1 NetworkML

Attributes

name	:	string
lengthUnits	:	meta:LengthUnits
volumeUnits	:	meta:VolumeUnits
populations	:	list of Populations
projections	:	list of Projections

This is the container for the entire model.

2.2.2 Population

Attributes

name	:	string
cell_type	:	string
instances	:	list of CellInstances
	or	
pop_location	:	PopulationLocation

A Population is a collection of cells all of the same type. The cell_type string must be the name of a cell model that has been previously defined using MorphML, ChannelML and Biophysics. The locations of the cells in space may be defined either through a list of CellInstances or through a PopulationLocation entity.

2.2.3 Projection

Attributes

[2] they may also contain meta-data, and the Populations type also has an attribute setting the system of units ("physiological" or SI) to be used.

name	:	string
from	:	string
to	:	string
synapse_properties	:	SynapseProperties
connections	:	list of Connections
	or	
connectivity_pattern	:	ConnectivityPattern

A Projection is a set of connections between two Population entities. from and to are the name attributes of Population entities. The individual connections may be specified either as a list of Connections, or through a ConnectivityPattern entity.

2.2.4 CellInstance

Attributes

id	:	integer
location	:	meta:Point

This specifies an integer id and a location in space for an individual cell in a Population.

2.2.5 PopulationLocation

Attributes

reference	:	string
random_arrangement	:	RandomArrangement
	or	
...	:	...

This specifies which algorithm to use for determining the location of cells in space. It would perhaps be better to make this an abstract type from which the types for specific algorithms (such as RandomArrangement) can inherit, and then the pop_location attribute of Population could have the specific types as values, rather than have this long list (random_arrangement,...) of alternate attributes.

2.2.6 RandomArrangement

Attributes

population_size	:	integer
spherical_location	:	meta:Sphere
	or	
...	:	...

The attributes are used to specify the number of cells in the Population and the boundaries of the region of space within which the cells are randomly distributed.

2.2.7 SynapseProperties

Attributes

synapse_type	:	string
delay	:	double
weight	:	double
threshold	:	double

Note that the units of these values is set by the units attribute of the containing Populations entity. delay and weight should be self-explanatory. The use of threshold assumes that synaptic transmission is triggered when the pre-synaptic membrane potential crosses a certain threshold.

2.2.8 ConnectivityPattern

Attributes

num_per_source	:	double
max_per_target	:	double
...	:	...

This is to be extended, but in principle is used to specify the algorithm to be used to determine the connections, and the parameters of that algorithm. Again, it may be better to have this as an abstract type and have types for specific algorithms inherit from this.

2.2.9 Connection

Attributes

source	:	SynapticLocation
target	:	SynapticLocation
id	:	integer

Specifies an integer id, and the pre- and post-synaptic entities.

2.2.10 SynapticLocation

Attributes

cell_id	:	integer
segment_id	:	integer
fraction_along	:	[0...1]

cell_id must correspond to the id attribute of a CellInstance. segment_id should correspond to an id defined using the MorphML specification. fraction_along specifies where on that segment the synapse should be located (the default is presumably the middle of the segment).

2.2.11 PotentialSynapticLocation

Attributes

synapse_type	:	string
group	:	string

This specifies a subset of sections on a cell where synaptic connection of a particular type is allowed. group is a list of groups of sections (from the Biophysics specification), and synapse_type is the name of a previously-defined synapse model (using the Biophysics specification).

3

Extensions to the NeuroML data model

The NetworkML standard is still at an alpha stage of development, and there are several improvements that could be made, even within its current scope of application (compartmental models with fixed parameter values). For the purposes of FACETS, we would like to extend its scope to include 'point' or 'threshold' models such as integrate-and-fire models and generalisations thereof, and to allow more sophisticated handling of parameters, including specification of the allowed range of values that parameters can take and the possibility of specifying a random distribution from which parameter values may be drawn.

3.1 Modifications to existing types

3.1.1 Connectivity patterns

It is proposed to make `ConnectivityPattern` an abstract type, and have types for specific connectivity algorithms inherit from this. Proposed specific types (and their attributes):

`AllToAll`

No attributes. Connect all cells in the presynaptic population to all cells in the postsynaptic population.

`OneToOne`

No attributes. Where the pre- and postsynaptic populations have the same size, connect cell i in the presynaptic population to cell i in the postsynaptic population for all i. In fact, despite the name, this should probably be generalised to the case where the pre and post populations have different dimensions, e.g., cell i in a 1D pre population of size n should connect to all cells in row i of a 2D post population of size (n, m).

`FixedProbability`

Attributes

p_connect : [0...1]

For each pair of pre-post cells, the connection probability is constant.

`DistanceDependentProbability`

Attributes

d_expression : string

For each pair of pre-post cells, the connection probability depends on distance. `d_expression` should be

the right-hand side of a valid Python[1] expression for probability, involving 'd', e.g. "`exp(-abs(d))`", or "`float(d<3)`".

`FixedNumberPre`

Attributes

n_conn	:	integer

Each presynaptic cell makes a fixed number of connections.

`FixedNumberPost`

Attributes

n_conn	:	integer

Each postsynaptic cell makes a fixed number of connections.

`FromFile`

Attributes

filename	:	string

Load connections from a file. The file format will need to be specified.

3.1.2 Spatial location of cells

It is proposed to make `PopulationLocation` an abstract type, and have types for specific location algorithms, such as `RandomArrangement` inherit from this. Proposed specific types and their attributes:

`GridStructure1D`

Attributes

size	:	integer
spacing	:	double

`GridStructure2D`

Attributes

sizex	:	integer
sizey	:	integer
spacingx	:	double
spacingy	:	double

`GridStructure3D`

Attributes

sizex	:	integer
sizey	:	integer
sizey	:	integer
spacingx	:	double
spacingy	:	double
spacingz	:	double

[1] Python was chosen because one of our implementations of the common data model is in Python, but mathematical expressions are anyway almost identical in most modern programming languages. In a later iteration of the data model, we may use MathML for mathematical expressions, but this would add unnecessary complexity at this stage.

`RandomArrangement`

Attributes

`size`	:	`integer`
`boundary`	:	`meta:Sphere or meta:Polyhedron`

`HexagonalClosePackedStructure`

Attributes

`size`	:	`integer`
`spacing`	:	`double`

3.2 Limits and units for parameters

Currently, units can only be specified on a model-wide basis. This has the advantage of simplicity, but may require the user to convert values manually. Allowing specification of units on a parameter-by-parameter basis allows the user to use the units that are most convenient for him/her, but adds complexity. The ideal situation would be to allow either model-wide or parameter-wise specification of units. This may or may not be achievable.

Other potentially useful attributes of a parameter besides its numerical value and units are the allowed range of values, notation, default value, and precision[2]. These can be used to:

- verify the relevance of a given numerical value,
- automatically generate a graphical user interface to browse/edit such values,
- provide numerical algorithms with required "numerical conditioning" information,

Some examples are given in Table 3.1.

We propose therefore to define a new type `PValue` that will be used in most places a numerical type, such as `double` or `integer` is used now.

[2] **Standard conditions for numbers** At the specification level, a "physical" parameter is always represented though a vector of bounded quantities, with a precision (so that there is a finite range of significant values). Such a precision is in practice very easy to estimate (e.g. 1 mm for a pupil ruler, 1 deg for a protractor, 1 pixel in an image, etc...) and so are bounds. These quantities are not precise numbers but orders of magnitude.
Following this track, two parameters can be considered as distinct only if their difference is higher than some "epsilon-value". Otherwise, we cannot decide whether these values are the same or differ by a quantity too small to be measurable. In the latter cases, we can simply state that they are indistinguishable. When several parameters are taken into consideration together the underlying assumption is that the related covariance between then has been – up to some linear transformation – diagonalized.
Although over-simple, such an specification is very useful at both the theoretical and implementation levels. It has been observed that there is a real gain in taking this experimental specification into account: with such a specification, "quasi-static" estimation methods, with step by step variations from an initial estimate towards the problem solution, are powerful strategies for local estimations (adaptations to limited range variations from a default value, interactive estimation where a user-provided initial estimate is to be refined, efficiency in tracking tasks, etc.), experimentally more efficient than the usual, standard methods, because the stability of the estimation process is easy to control in this case. Furthermore, the estimation is stopped as soon as the required precision is obtained, whereas for standard methods, convergence to a non-negligible precision only is not so easy to obtain, so that overhead occurs.

value-name	notation	physical-unit	default value	minimal value	maximal value	precision
membrane-time-constant	τ_{m}	milli-seconds	20.0	0	$+\infty$	1e-3
refractory-delay	δ_{r}	milli-seconds	0.0	0	$+\infty$	1e-3
refractory-time-constant	τ_{r}	milli-seconds	$+\infty$	0	$+\infty$	1e-3
reset-potential	V_{reset}	milli-volts	V_{rest}	$-\infty$	$+\infty$	1e-3
rest-potential	V_{rest}	milli-volts	-65.0	$-\infty$	$+\infty$	1e-3
threshold-potential	V_θ	milli-volts	-50.0	$-\infty$	$+\infty$	1e-3
excitatory-synaptic-conductance-time-constant	τ_{E}	milli-seconds	2.0	0	$+\infty$	1e-3
inhibitory-synaptic-conductance-time-constant	τ_{I}	milli-seconds	5.0	0	$+\infty$	1e-3
excitatory-reversal-potential	V_{E}	milli-volts	0.0	V_{rest}	$+\infty$	1e-3
inhibitory-reversal-potential	V_{I}	milli-volts	-75	$-\infty$	V_θ	1e-3
excitatory-synaptic-conductance-initial-value	g_0^{E}	micro-siemens	0.0	0	$+\infty$	1e-3
inhibitory-synaptic-conductance-initial-value	g_0^{I}	micro-siemens	0.0	0	$+\infty$	1e-3
membrane-potential-initial-value	V_{init}	milli-volts	V_{rest}	$-\infty$	$+\infty$	1e-3

Table 3.1. Examples of additional attributes of a numerical parameter besides the numerical value

`PValue`

Attributes

value	:	double or integer
units	:	Units
name	:	string
notation	:	string
default_value	:	double or integer
minimal_value	:	double or integer
maximal_value	:	double or integer
precision	:	double or integer

`notation` could be a unicode string, a LaTeXexpression, or a MathML expression. The `Units` type is described below.

The downside to this proposal is that it adds greatly to the verbosity of a model description. It may therefore be desirable to add subtypes of `PValue` for which `default_value`, `minimal_value`, `maximal_value` and `precision`, together with the dimensions of the quantity if not the specific units, are pre-defined and fixed, e.g. `MembranePotentialValue`, `TimeConstantValue`, `SynapticConductanceValue`, etc.

`Units`

Attributes

unitlist	:	list of Units
base_unit	:	Boolean

`Unit`

Attributes

units	:	idref
exponent	:	integer
multiplier	:	double
offset	:	double
prefix	:	UnitPrefix

The use of the `Units` and `Unit` types is best explained by an example:

```
<units name="volt">
```

```
  <unit exponent="2" units="metre" />
  <unit units="kilogram" />
  <unit exponent="-3" units="second" />
  <unit exponent="-1" units="ampere" />
</units>

<units name="ohm">
  <unit units="volt" />
  <unit exponent="-1" units="ampere" />
</units

<units name="siemenspersquarecentimetre">
  <unit exponent="-1" units="ohm" />
  <unit prefix="centi" exponent="-2" units="metre" />
</units>
```

i.e., a new unit is defined in terms of a list of existing units, where each component unit in the list is multiplied together.

The seven base units of the SI system[3] are defined as, for example,

```
<units name="ampere" base\_unit="True" />
```

An alternative, probably desirable, system would be to include the base units, plus the named derived units of the SI system (joule, coulomb, volt, ohm), in the schema definition as an enumeration, to avoid having to define these commonly-used units in every document.

The `base_unit` attribute has default value "False".

The `multiplier` attribute can be used to pre-multiply the quantity to be converted by any real-valued scale factor. For instance, a multiplier of 0.45359237 is used to define a pound in terms of a kilogram. The `multiplier` attribute has a default value of "1.0"

The `offset` attribute is used to represent the addition of a constant in the transformation between the current units and the base units. This should only be necessary for the definition of temperature scales. For instance, an offset value of "32.0" is needed to define Fahrenheit in terms of Celsius. The `offset` attribute has a default value of "0.0".

The `prefix` attribute can be used to indicate a scale for the referenced units:

yotta 10^{24}	zetta 10^{21}	exa 10^{18}	peta 10^{15}	tera 10^{12}	giga 10^{9}	mega 10^{6}	kilo 10^{3}	hecto 10^{2}	deka 10^{1}
yocto 10^{-24}	zepto 10^{-21}	atto 10^{-18}	femto 10^{-15}	pico 10^{-12}	nano 10^{-9}	micro 10^{-6}	milli 10^{-3}	centi 10^{-2}	deci 10^{-1}

3.3 Dynamic parameter and parameter variability

A step further, we must be able to allow parameters to be drawn from a random distribution, or to be specified by an algorithm (e.g. location-dependent).

[3] SI base units

Name	Symbol	Quantity
kilogram	kg	Mass
second	s	Time
meter	m	Length
ampere	A	Electrical current
kelvin	K	Temperature
mole	mol	Amount of substance
candela	cd	Luminous intensity

3.3.1 Random distribution

They are to be specified in this context by standard parameters, as sketched here:

Attributes

type	:	distribution kind
mean	:	distribution mean
standard-deviation	:	distribution standard-deviation
skewness	:	distribution 3rd order momentum (lack of symmetry)
kurtosis	:	distribution 4rd order momentum ("flatness" of the distribution)

using, e.g., a mixture of two Gaussian or uniform distributions.

3.3.2 Dynamic parameter

As stated in the FKB M8 reference document, values can also be defined dynamically using standard expressions.

3.4 Threshold models

We use 'threshold models' to describe spiking neuron models with an 'artificial' action potential, i.e. one in which an 'action potential' is triggered when some variable or variables (e.g. membrane potential, rate of change of membrane potential) pass(es) a threshold, rather than being produced from Hodgkin-Huxley-type equations.

We use this term rather than 'point' model or 'non-compartmental model' because it is possible to have multi-compartmental, multi-point and/or spatially-extended threshold models.[4]

The NeuroML standards define a biophysical cell model, which consists of a collection of ion channels, synaptic mechanisms, a compartmental cable model with a defined morphology, and a specification of the spatial distribution of channels/synapses within the morphological structure.

Threshold models share most of this structure. The most complex threshold models may have all of these, but with the action potential specified by a threshold crossing rather than Hodgkin-Huxley fast sodium and delayed-rectifier potassium channels. The simplest threshold models will have a morphology consisting of a single point, no ion channels, and one synaptic mechanism.

We propose, then, to reuse as much as possible of the NeuroML structure for biophysical models. The NeuroML `Level3Cell` type contains a `biophysics` attribute, of type `Level3Biophysics`, which is an extension of the `Biophysics` type defined in the Level 2 Biophysics specification. The `Biophysics` type has the following attributes:

`Biophysics`

Attributes

mechanismlist	:	list of Mechanisms
specificCapacitance	:	SpecCapacitance
specificAxialResistance	:	SpecAxialResistance
initialMembPotential	:	InitialMembPotential
ionProperties	:	IonProperties
units	:	Units

(Note that the `Units` type is not the same as the one defined below.) All of these are potentially relevant to threshold models. The `Mechanism` type has an attribute `type`, which may be either a `ChannelMechanism` or an `IonConcentration`. A spiking threshold and reset is arguably a `Mechanism`, and so we propose to add a third choice, `ThresholdMechanism`.

[4] e.g., a two-compartment integrate-and-fire model with a dendrite compartment and soma compartment, a resistive connection between dendrite and soma and a threshold mechanism only in the soma.

A ThresholdMechanism could be defined either by specifying equations, or simply by specifying the name of a standard model, e.g. "IntegrateAndFire", "Izhikevich", whose behaviour is not defined within FacetsML, but in the simulators. Allowing specification of equations gives the most flexibility and generality, but also adds great complexity. At least initially, therefore, we propose to define a library of standard models, each of which will extend the StandardCell type:

StandardCell

Attributes

v_rest	:	PValue
v_reset	:	PValue
t_refrac	:	PValue
cm	:	PValue
i_offset	:	PValue

For example, IFNeuron has a fixed firing threshold, and the sub-threshold membrane potential decays with a single exponential time course, and so adds attributes v_thresh and tau_m.

3.5 Plasticity mechanisms

3.5.1 Facilitation, depression

Models of facilitation, depression and other forms of short-term mono-synaptic plasticity are an extension of general synapse models, since they generally affect the peak conductance of the synapse. ChannelML documents may contain elements of type SynapseType:

SynapseType

Attributes

name	:	string
density	:	meta:YesNo
doub_exp_syn	:	DoubleExponentialSynapse

evidently it is intended to add other synapse models in addition to doub_exp_syn.

DoubleExponentialSynapse

Attributes

max_conductance	:	double
rise_time	:	double
decay_time	:	double
reversal_potential	:	double

There are two possible options for adding short-term plasticity models. Either define further classes of synapse, some with plasticity mechanisms, some without, or attempt to dissociate the 'static' part of the synapse model from the plastic part, and have an attribute (or set of attributes) for each part. This requires considerable further study, and we do not yet make a recommendation.

3.5.2 Long-term plasticity

We need here to distinguish between biophysical and phenomenological models of long-term plasticity mechanisms. The former can probably be incorporated into ChannelML, or may have to wait for the planned Level 4 NeuroML specification, to cover subcellular processes such as detailed calcium dynamics and signaling pathways. The latter probably does not belong in ChannelML, and so could perhaps go in NetworkML, and be associated with the Projection type.

Further complications may be anticipated for non-local/homeostatic mechanisms, which may not be associated with a single `Population` or `Projection`.

Again, further study is required.

3.6 Further Extensions

Higher-order structures So far, we have Populations and Projections as the highest-order structures. We could add columns, metacolumns, maps, etc.

Non-event inputs Inputs other than spiking ones, e.g., continuous currents are also to be considered

4

Implementation 1: FACETS-ML

4.1 Introduction to FacetsML

Many of the extensions to the NeuroML data model that were described in Section 3 have been implemented in the FacetsML language, a declarative implementation of the FACETS data-model described in the previous sections. FacetsML is an XML-based[1] language for describing and exchanging models of spiking neuron networks, based on NeuroML, for use within the FACETS. It may be considered both as a test-bed for possible future additions to the NeuroML specification, and a container for components that are needed in the FACETS project but may be too specific for the NeuroML standards, for example the standard cell models.

When the FACETS project was begun, there was no Level 3 (network) specification in NeuroML, and so we developed independently a schema for specifying neuronal networks on top of the NeuroML Levels 1 and 2 standards (FacetsML version 0.8). We are now reimplementing FacetsML to base it on the NetworkML specification.

FacetsML, like NeuroML, uses XML Schema[2] to define the allowed structure, content and semantics of XML neuronal network descriptions documents.

4.2 Fundamentals

This section of the FacetsML specification introduces some concepts that are used throughout the entire language.

4.2.1 Basic Structure

Definition of a valid FacetsML identifier

The most common use of a FacetsML identifier is the name attribute required on many basic elements in FacetsML. The value of this attribute can be used to reference that element from elsewhere in the model definition or from another model definition altogether. An object's name can generally be thought of as a unique identifier for that object. Although the XML specification defines a mechanism for specifying that the value of an attribute is unique across an entire document (with the ID attribute type), this

[1] eXtensible Markup Language (XML). XML is a standard published by the World Wide Web Consortium, the organization responsible for defining many internet-related standards, including HTML. XML is essentially a means of adding structure to text documents, allowing machines to unambiguously associate text or binary data with a particular component in a document's data model.
XML is an appropriate medium for FacetsML because it is both human and machine readable. A model author can create a FacetsML document with a text editor or with FacetsML authoring software. XML is a well-defined and widely used specification. Many free software utilities and libraries for the processing of XML already exist, simplifying the development of FacetsML processing software. XML has also been designed to be usable over the internet, making FacetsML suitable for the interchange of models between software and databases at different locations.

[2] XML Schema is a W3C-approved standard which provides a means for defining the structure, content and semantics of XML documents.

functionality is not used in FacetsML 0.8 because an object's name need only be unique across its own class of objects.

The generation of computer code for running simulations is one of the target applications for FacetsML. The value of an object's name attribute is intended to be a suitable name for the same object when it is represented in computer code. For this reason FacetsML identifiers must consist of only alphanumeric characters and the underscore character ("_"), and are subject to some additional constraints outlined below. These names will generally not be the most effective way of identifying the object to humans working with FacetsML models as it is not possible to include whitespace or formatting.

The XML specification is based on the Unicode standard, which defines a scheme for 16 bit character encoding. Thus it is possible to include, for instance, Japanese characters in a valid XML document. In the interests of making the code generation process as convenient as possible for those using mainstream programming languages, FacetsML identifiers are subject to the following constraints:

- An identifier must consist only of alphanumeric characters from the US-ASCII character set and underscore characters,
- An identifier must contain at least one letter, and
- An identifier must not begin with a digit.

Convenient code generation is also the reason why colons, periods, and hyphens may not appear in FacetsML identifiers. FacetsML identifiers are case sensitive: a variable with an identifier of ABC is different from a variable with an identifier of abc.

Namespaces in FacetsML

Namespaces in XML is a companion specification to the XML 1.0 specification. XML namespaces add a second level of naming to elements and attributes, allowing processing software to distinguish between elements and attributes from different languages. A namespace is identified by a Uniform Resource Identifier (URI), which has the feature of being unique. The value of a namespace URI need have nothing to do with the XML document that uses it. However, it typically points to a document that defines the rules for the language. The URI may be mapped to a prefix, which may then be used in front of element and attribute names, separated by a colon. If not mapped to a prefix, the URI sets the default namespace for the current element and all of its children.

The following table lists the names, URIs and recommended prefixes of the namespaces referenced in this specification. For interoperability, the root element of any FacetsML document should set the default namespace and map the fml prefix to the FacetsML 0.8 namespace URI.

Namespace Name	Namespace URI	Recommended Prefix
FacetsML	`http://facets-project.org/facetsml/schema`	fml
FacetsML Metadata	`http://facets-project.org/metadata/schema`	fmeta
NeuroML Metadata	`http://morphml.org/metadata/schema`	meta
NeuroML Morphology	`http://morphml.org/morphml/schema`	mml
NeuroML Biophysics	`http://morphml.org/biophysics/schema`	bio
NeuroML Channel	`http://morphml.org/channelml/schema`	cml
NeuroML Network	`http://morphml.org/networkml/schema`	net

4.2.2 Rules for FacetsML documents

Valid FacetsML identifiers

A valid FacetsML identifier must consist of only letters, digits and underscores, must contain at least one letter, and must not begin with a digit. This can be written using Extended Backus-Naur Form (EBNF) notation as follows:

letter ::= 'a'...'z', 'A'...'Z'
digit ::= '0'...'9'
identifier ::= ('_')* (letter) (letter | '_' | digit)*

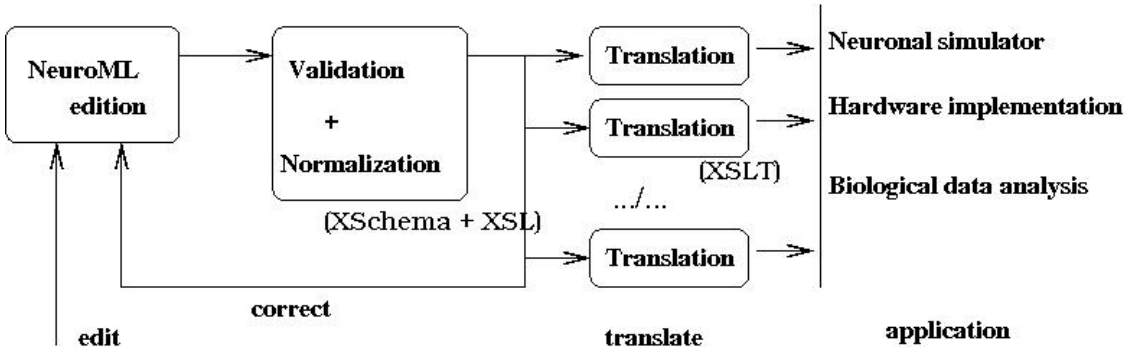

Figure 4.1. From NeuroML/FacetsML to simulator

Proper use of the FacetsML namespace

A document must not contain elements or attributes in the FacetsML namespace that are not defined in this specification.

Text nodes within FacetsML elements

Any characters that occur immediately within elements in the FacetsML namespace must be either space (`#x20`) characters, carriage returns (`#xA`), line feeds (`#xD`), or tabs (`#x9`).

4.2.3 Rules for Processor behavior

Treatment of FacetsML identifiers

FacetsML processing software must handle identifiers in a case-sensitive manner.

Treatment of attribute namespaces

FacetsML processing software must treat attributes without an explicit namespace declaration as if they were in the same namespace as their parent element.

4.3 Mapping to specific simulators

To go from FacetsML/NeuroML to code that can be run on a particular simulator, we proceed as follows:

1. edit data-structures using a suitable XML editor,
2. validate (i.e., verify that the related logical-structures are well-formed and valid with respect to the specification, conditions, etc..)
3. normalize (i.e., translate it an equivalent logical-structure but without redundancy, while some factorization allows to simplify subsequent manipulation) it.

From this valid norm form translation is efficient and safe. Translation may take at least two forms. Either a simulator may accept a NeuroML document as input, and translation from FacetsML/NeuroML elements to native simulator objects is performed by the simulator, or the XSL Transformation language may be used to generate native simulator code (e.g. `hoc` or `NMODL` in the case of NEURON). For example, the NeuroML Validator service provides translation of ChannelML and MorphML files to NEURON and GENESIS formats. This process is summarized in Figure 4.1.

Mapping to the MVASPIKE and NEST simulators has been designed and implemented while a partial mapping to NEURON is available in NeuroML.

5

Implementation 2: PyNN

5.1 Programmatic model specification using Python

For network simulations, we may well require more flexibility than can easily be obtained using a declarative model specification such as FacetsML, but we still wish to obtain simple conversion between simulators, i.e. to be able to write the simulation code for a model only once, then run the same code on multiple simulators. This requires first the definition of an API (Application Programming Interface) or meta-language, a set of functions/classes which provides a superset of the capabilities of the simulators we wish to run on[1]. There are two possible second steps: (i) each simulator implements a parser which can interpret the meta-language; (ii) a separate program either translates the meta-language into simulator-specific code or controls the simulator directly, giving simulator-specific function calls.

In our opinion, the second of these possibilities is the better one, since

1. it avoids replication of effort in writing parsers

2. we can then use a general purpose, state-of-the-art interpreted programming language, such as Python or Ruby, rather than a simulator-specific language, and thus leverage the effort of outside developers in areas that are not neuroscience specific, such as data analysis and visualisation[2]

The PyNN project[3] within FACETS has begun to develop both the API and the binding to individual simulation engines, for both purposes using the Python programming language. The API has two parts, a low-level, procedural API (functions `create()`, `connect()`, `set()`, `record()`), and a high-level, object-oriented API (classes `Population` and `Projection`, which have methods like `set()`, `record()`, `setWeights()`, etc.). The low-level API is good for small networks, and perhaps gives more flexibility. The high-level API is good for hiding the details and the book-keeping, and for expressing models with a higher level of abstraction.

The high-level API is intended to implement the FACETS common data model, i.e. to have a one-to-one mapping with FacetsML, i.e. a `population` element in FacetsML will correspond to a `Population` object in PyNN. This correspondence is not yet complete, but is one of the goals of the project.

The other thing that is required to write a model once and run it on multiple simulators is standard cell models, as discussed previously in §3.4. A complication is that although the synaptic model and the membrane model are independent, and are specified separately in FacetsML, most simulators define threshold models in terms of both membrane and synaptic behaviour, for reasons of efficiency. Therefore the standard cell models in PyNN are each a combination of a membrane model and a synaptic model, e.g. `IF_curr_alpha` is an `IFNeuron` standard membrane model with a `SynCurrAlpha` synaptic model.

[1] Note that since we choose a superset, the system must emit a warning/error if the underlying simulator engine does not support a particular feature.

[2] For Python, examples include efficient data storage and transfer (HDF5, ROOT), data analysis (SciPy), parallelisation (MPI), GUI toolkits (GTK, QT).

[3] pronounced 'pine'

```
cell_params = { 'tau_m' : 20.0,  'tau_syn' : 2.0,   'tau_refrac': 1.0,
                'v_rest': -65.0, 'v_thresh': -50.0, 'cm': 1.0}

populationA = Population((10,), "IF_curr_alpha", cell_params)
populationB = Population((5,5), "IF_curr_alpha", cell_params)
populationA.randomInit('uniform', v_reset, v_thresh)

connAtoB = Projection(populationA, populationB, 'fixedProbability', 0.2)
connAtoA = Projection(populationA, populationA, 'distanceDependentProbability', "exp(-abs(d))")
connBtoA = Projection(populationB, populationA, 'allToAll')

connAtoB.setWeights(w_AB)
connAtoA.setWeights(w_AA)
connBtoA.setWeights(w_BA)

populationA.record()
populationB.record()

run(1000.0)

populationA.printSpikes("populationA.spiketimes")
populationB.printSpikes("populationA.spiketimes")
```

Figure 5.1. Example of the use of the PyNN API to specify a network that can then be run on multiple simulators.

PyNN translates standard cell-model names and parameter names into simulator-specific names, e.g. standard model `IF_curr_alpha` is `iaf_neuron` in NEST and `StandardIF` in NEURON, while `SpikeSourcePoisson` is a `poisson_generator` in NEST and a `NetStim` in NEURON.

An example of the use of the API to specify a simple network is given in Figure 5.1.

Bindings currently exist to control NEST (PyNEST) and MVASpike, and Python can be used as an alternative interpreter for NEURON (nrnpython), although the level of integration (how easy it is to access the native functionality) is variable. There are future plans to extend this to other simulators, initially MVASpike and CSIM), and to add support for parallel computation.

5.2 API

5.2.1 Data

default_values = {

SpikeSourcePoisson	: {'duration': 1000000000.0, 'start': 0.0, 'rate': 0.0 }
IF_cond_alpha	: {'tau_refrac': 0.0, 'tau_m': 20.0, 'e_rev_E': 0.0, 'cm': 1.0, 'e_rev_I': -70.0, 'v_thresh': 'tau_syn_E': 5.0, 'v_rest': -65.0, 'tau_syn_I': 5.0 }
IF_curr_exp	: {'tau_refrac': 0.0, 'tau_m': 20.0, 'i_offset': 0.0, 'cm': 1.0, 'v_thresh': -50.0, 'tau_syn 'v_rest': -65.0, 'tau_syn_I': 5.0, 'v_reset': -65.0 }
IF_curr_alpha	: {'tau_refrac': 0.0, 'tau_m': 20.0, 'i_offset': 0.0, 'cm': 1.0, 'v_thresh': -50.0, 'v_rest': 'tau_syn': 5.0, 'v_reset': -65.0 }
SpikeSourceArray	: {'spike_times': [] }

}

dt = 0.10

5.2.2 Functions

connect(source, target, weight=1, delay=0, synapse_type=None, p=1) Connect a source of spikes to a synaptic target. source and target can both be individual cells or lists of cells, in which case all possible connections are made with probability p.

create(celltype, paramDict=None, n=1) Create n cells all of the same type. If n > 1, return a list of cell ids/references. If n==1, return just the single id.

end() Do any necessary cleaning up before exiting.

record(src, filename) Record spikes to a file. src can be an individual cell or a list of cells.

record_v(source, filename) Record membrane potential to a file. source can be an individual cell or a list of cells.

run(simtime) Run the simulation for simtime ms.

set(cells, celltype, param, val=None) Set one or more parameters of an individual cell or list of cells. param can be a dict, in which case val should not be supplied, or a string giving the parameter name, in which case val is the parameter value. celltype must be supplied for doing translation of parameter names.

setRNGseeds(seedList) Globally set rng seeds.

setup(timestep=0.1, min_delay=0.1, max_delay=0.1) Should be called at the very beginning of a script.

5.2.3 Classes

Random

Wrapper class for random number generators. The idea is to be able to use either simulator-native rngs, which may be more efficient, or a standard python rng, e.g. numpy.Random, which would allow the same random numbers to be used across different simulators, or simply to read externally-generated numbers from files.

__init__(self, type='default', distribution='uniform', label=None, seed=None)

next(self, n=1) Return n random numbers from the distribution.

Projection

A container for all the connections between two populations, together with methods to set parameters of those connections, including of plasticity mechanisms.

__init__(self, presynaptic_population, postsynaptic_population, method='allToAll', methodParameters=None, source=None, target=None, label=None, rng=None) Create Population object.

presynaptic_population and postsynaptic_population – Population objects.

source – string specifying which attribute of the presynaptic cell signals action potentials

target – string specifying which synapse on the postsynaptic cell to connect to If source and/or target are not given, default values are used.

method – string indicating which algorithm to use in determining connections. Allowed methods are 'allToAll', 'oneToOne', 'fixedProbability', 'distanceDependentProbability', 'fixedNumberPre', 'fixedNumberPost', 'fromFile', 'fromList'

methodParameters – dict containing parameters needed by the connection method, although we should allow this to be a number or string if there is only one parameter.

rng – since most of the connection methods need uniform random numbers, it is probably more convenient to specify a Random object here rather than within methodParameters, particularly since some methods also use random numbers to give variability in the number of connections per cell.

__len__(self) Return the total number of connections.

_allToAll(self, parameters=None, synapse_type=None) Connect all cells in the presynaptic population to all cells in the postsynaptic population.

_distanceDependentProbability(self, parameters, synapse_type=None) For each pair of pre-post cells, the connection probability depends on distance. d_expression should be the right-hand side of a valid python expression for probability, involving 'd', e.g. "exp(-abs(d))", or "float(d<3)"

_fixedNumberPost(self, parameters, synapse_type=None) Each postsynaptic cell receives a fixed number of connections.

_fixedNumberPre(self, parameters, synapse_type=None) Each presynaptic cell makes a fixed number of connections.

_fixedProbability(self, parameters, synapse_type=None) For each pair of pre-post cells, the connection probability is constant.

_fromFile(self, parameters) Load connections from a file.

_fromList(self, parameters) Read connections from a list of lists, or somesuch...

_oneToOne(self, synapse_type=None) Where the pre- and postsynaptic populations have the same size, connect cell i in the presynaptic population to cell i in the postsynaptic population for all i. In fact, despite the name, this should probably be generalised to the case where the pre and post populations have different dimensions, e.g., cell i in a 1D pre population of size n should connect to all cells in row i of a 2D post population of size (n,m).

printWeights(self, filename, format=None) Print synaptic weights to file.

randomizeDelays(self, rng) Set delays to random values taken from rng.

randomizeWeights(self, rng) Set weights to random values taken from rng.

saveConnections(self, filename) Save connections to file in a format suitable for reading in with the 'fromFile' method.

setDelays(self, d) d can be a single number, in which case all delays are set to this value, or an array with the same dimensions as the Projection array.

setMaxWeight(self, wmax) Note that not all STDP models have maximum or minimum weights.

setMinWeight(self, wmin) Note that not all STDP models have maximum or minimum weights.

setThreshold(self, threshold) Where the emission of a spike is determined by watching for a threshold crossing, set the value of this threshold.

setWeights(self, w) w can be a single number, in which case all weights are set to this value, or an array with the same dimensions as the Projection array.

setupSTDP(self, stdp_model, parameterDict) Set-up STDP.

toggleSTDP(self, onoff) Turn plasticity on or off.

weightHistogram(self, min=None, max=None, nbins=10) Return a histogram of synaptic weights. If min and max are not given, the minimum and maximum weights are calculated automatically.

Timer

For timing script execution.

elapsedTime() (static) Return the elapsed time but keep the clock running.

reset() (static) Reset the time to zero, and start the clock.

start() (static) Start timing.

Population

An array of neurons all of the same type. 'Population' is used as a generic term intended to include layers, columns, nuclei, etc., of cells.

__getitem__(self, id) Returns a representation of the cell with coordinates given by the tuple 'id', suitable for being passed to other methods that require a cell id. Note that __getitem__ is called when using [] access, e.g. p = Population(...) p[id] is equivalent to p.__getitem__(id)

__init__(self, dims, celltype, cellparams=None, label=None) dims should be a tuple containing the population dimensions, or a single integer, for a one-dimensional population. e.g., (10,10) will create a two-dimensional population of size 10x10. celltype should be a string - the name of the neuron model class that makes up the population. cellparams should be a dict which is passed to the neuron model constructor label is an optional name for the population.

__len__(self) Returns the total number of cells in the population.

call(self, methodname, arguments) Calls the method methodname(arguments) for every cell in the population. e.g. p.call("set_background","0.1") if the cell class has a method set_background().

meanSpikeCount(self) Returns the mean number of spikes per neuron.

printSpikes(self, filename) Prints spike times to file in the two-column format "spiketime cell_id" where cell_id is the index of the cell counting along rows and down columns (and the extension of that for 3-D). This allows easy plotting of a 'raster' plot of spiketimes, with one line for each cell.

print_v(self, filename) Write membrane potential traces to file.

randomInit(self, func, *params) Sets initial membrane potentials for all the cells in the population to random values.

record(self, record_from=None) If record_from is not given, record spikes from all cells in the Population. record_from can be an integer - the number of cells to record from, chosen at random - or a list containing the ids (e.g., (i,j,k) tuple for a 3D population) of the cells to record.

record_v(self, record_from=None) If record_from is not given, record the membrane potential for all cells in the Population. record_from can be an integer - the number of cells to record from, chosen at random - or a list containing the ids of the cells to record.

rset(self, parametername, randomobj) 'Random' set. Sets the value of parametername to a value taken from the randomobj Random object.

set(self, param, val=None) Set one or more parameters for every cell in the population. param can be a dict, in which case val should not be supplied, or a string giving the parameter name, in which case val is the parameter value. e.g. p.set("tau_m",20.0). p.set('tau_m':20,'v_rest':-65)

tcall(self, methodname, objarr) 'Topographic' call. Calls the method methodname() for every cell in the population. The argument to the method depends on the coordinates of the cell. objarr is an array with the same dimensions as the Population. e.g. p.tcall("memb_init",vinitArray) calls p.cell[i][j].memb_init(vInitArray[i] for all i,j.

tset(self, parametername, valueArray) 'Topographic' set. Sets the value of parametername to the values in valueArray, which must have the same dimensions as the Population.

6

Future developments

Since 15 scientific teams share this format and tool, the next step is to target a public dissemination of the PyNN/FML software and tools within the NeuroML initiative.

Version 1 of the PyNN software with version 1 of the FacetsML validator and editor will be published as soon as the present specifications have been completed and fully validated.

Software development and dissemination will use a 'software forge' to be able to maintain this open-source software and really construct a cooperative specification and implementation task-force.

This is to be continued with the following objective for the different FACETS partners involved in network modeling:

1. to obtain a fully transparent bridge between all the model formats of the different partners (the Facets Model Format, FMF);

2. to develop, by each partner, the necessary interface to be able to create a FMF version of their "local" model;

3. to continue the implementation within PyNN of interfaces to the different simulators used in FACETS (this work to be carried out by those partners involved in developing simulators), including analog hardware, to support running FMF models as widely as possible;

4. to continue the development of efficient and accurate simulation tools and algorithms, and produce an extensive review of the status of the currently available resources.

In addition, we would like to study the possibility of building a "meta-simulator", which will not only use simulator-independent code, but also include common input/output specifications, different plug-in capabilities, etc. This will allow to completely write the models, manipulate them and run them, completely independently of the exact simulator used.