International Neuroinformatics Coordinating Facility

Network Interchange for Neuroscience Modeling Language (NineML) version 1.1

Release 1

The NineML Committee

Feb 03, 2018

Contents

1 About **1**
- 1.1 NineML Committee . . . 1
- 1.2 Licence . . . 1
- 1.3 Website . . . 1

2 Introduction **3**
- 2.1 Scope . . . 3
- 2.2 Design considerations . . . 4
- 2.3 Identifiers . . . 4

3 General Elements **5**
- 3.1 Document Layout . . . 5
- 3.2 Units and Dimensions . . . 6
- 3.3 Annotating Elements . . . 8

4 Abstraction Layer **9**
- 4.1 Component Classes and Parameters . . . 9
- 4.2 Mathematical Expressions . . . 11
- 4.3 Ports . . . 14
- 4.4 Dynamic Regimes . . . 16
- 4.5 Transitions . . . 19
- 4.6 Random Distributions . . . 22
- 4.7 Network Connectivity . . . 23

5 User Layer **25**
- 5.1 Components and Properties . . . 25
- 5.2 Values . . . 28
- 5.3 Populations . . . 30
- 5.4 Projections . . . 31
- 5.5 Selections: combining populations and subsets . . . 35

6 Serialization **37**

7 Examples **43**
- 7.1 Single Cell Models . . . 43
- 7.2 Network Models . . . 48

8 Acknowledgments **53**

Bibliography **55**

CHAPTER 1

About

The NineML specification is edited by the NineML Committee.

1.1 NineML Committee

- Thomas G. Close
- Alexander J. Cope
- Andrew P. Davison
- Jochen M. Eppler
- Erik De Schutter
- Ivan Raikov
- Paul Richmond

1.2 Licence

1.3 Website

See http://nineml.net for more information on the committee and NineML developments.

Last Updated: Feb 03, 2018

CHAPTER 2

Introduction

The increasing diversity of neuronal network models and the software/hardware platforms used to simulate them presents a significant challenge for sharing, replicability and reusability of models in computational neuroscience. To address this problem, we propose a common description language, *Network Interchange for Neuroscience Modeling Language (NineML)*, to facilitate the exchange neuronal network models between researchers and simulator platforms.

NineML is based on a common object model describing the different elements of neuronal network models. It was initiated and supported by the International Neuroinformatics Coordinating Facility (INCF) (http://www.incf.org) as part of the Multiscale Modeling Program, and has benefitted from wide-ranging input from computational neuroscientists, simulator developers and developers of simulator-independent languages (e.g. NeuroML, PyNN) (see Acknowledgements) *[Goddard2001]*, *[Gleeson2010]*, *[Davison2008]*.

2.1 Scope

The purpose of NineML is to provide a computer language for succinct and unambiguous description of computational neuroscience models of neuronal networks. NineML is intended to describe the network architecture, parameters and equations that govern the dynamics of a neuronal network, without taking into account model implementation details such as numerical integration methods.

The following neuronal network objects can be described in NineML,

1. spiking and non-spiking neurons
2. synapses
   (a) Post-synaptic membrane current mechanisms
   (b) Short-term synaptic dynamics (depression, facilitation)
   (c) Long-term synaptic modifications (STDP, learning, etc.)
   (d) Gap-junctions
3. populations of neurons
4. synaptic projections between populations of neurons

2.2 Design considerations

As one of the goals of NineML is to provide a means to exchange models between simulator platforms, it is important to maintain a clear distinction between the role of NineML and the role of a simulator. Therefore, NineML only contains the necessary information to describe the model not how to simulate it, although suggestions can be supplied in annotations to the model (see [sec:Annotations_Section]). For example, NineML should specify the neuron membrane equation to solve, but not how to solve it. In addition, for implementation and performance reasons, it is important to keep the language layer "close" to the simulator – such that the language layer is not responsible for maintaining separate representations of all the instantiated elements in the network.

A NineML object model representation can take multiple forms. A program can employ a concrete representation of the NineML objects in a specific programming language, convert an internal model representation to and from hierarchical data formats (see *Serialization*), or use code generation to produce a model representation for a target simulation environment.

The design of NineML is divided into two semantic layers:

1. An **Abstraction Layer** that provides the core concepts and mathematical descriptions with which model variables and state update rules are explicitly described in *parametrized* form, and
2. A **User Layer** that provides a syntax to specify the instantiation and the value of parameters of all these components of a network model.

Since the User Layer provides the instantiation and parametrization of model elements that have been defined in the Abstraction Layer, the two layers should share a complementary and compatible design philosophy. Which aspects of a model are defined in the Abtraction Layer and which are in the User Layer Layer are clearly defined (each element type belongs to only one layer with the exception of units and dimensions). In order to simplify their interpretation and maintain compatibility with a wide range of data formats (e.g. JSON, Python objects), NineML documents are not sensitive to the order that objects appear in.

2.3 Identifiers

Elements are identified by *names*, which are unique in the scope they are enclosed by (either within a component class or in the document scope of the file). For a name to be a valid NineML identifier, it must meet the requirements for a ANSI C89 identifiers. Additionally, identifiers are not permitted to begin or end with an underscore character (i.e. '_') to allow special variables to be defined in the same scope as identified variables/objects in generated code.

NineML identifiers are case-sensitive in the sense that they must be referred to with the same case as they are defined. However, two identifiers that are identical with the exception of case, e.g. 'v_threshold' and 'v_Threshold', are not permitted within the same scope. Identifiers used within component classes also cannot be the same (case-insensitive) as one of the built-in symbols or functions (see *MathInline*).

CHAPTER 3

General Elements

3.1 Document Layout

NineML documents must be enclosed within an NineML element, which should be in the 'http://nineml.net/9ML/1.0' namespace.

3.1.1 NineML

Attribute	Type/Format	Required
namespace (i.e. xmlns in XML)	'http://nineml.net/9ML/1.0'	yes

Children	Multiplicity	Required
Component	set	no
ComponentClass	set	no
Unit	set	no
Dimension	set	no
Population	set	no
Projection	set	no
Selection	set	no

Seven *document-level* elements are allowed to reside directly within NineML elements: *Component*, *ComponentClass*, *Unit*, *Dimension*, *Population*, *Projection* and *Selection*. Each element should be uniquely identified by its *name* attribute within the scope of the document (see).

Unit and *Dimension* elements must be defined within the document they are referenced, whereas the remaining element types can also be referenced from other NineML documents (see *Reference* and *Definition*).

Namespace attribute

The *namespace* attribute (xmlns in XML) is required and should refer to the URL corresponding to the correct NineML version, which for version 1.0 is 'http://nineml.net/9ML/1.0' (see http://www.w3.org/TR/REC-xml-names/).

3.2 Units and Dimensions

Dimensions are associated with parameters, analog ports and state variables in component class definitions. Each dimension can give rise to a family of unit declarations, each of which has the same dimensionality but a different multiplier. For example, typical units for a quantity with dimensionality voltage include millivolts (multiplier = 10^{-3}), microvolts (multiplier = 10^{-6}) and volts (multiplier = 1). To express a dimensional quantity both a numerical factor and a unit are required.

Except where physical constants are required, abstraction layer definitions generally only contain references to dimensions and are independent of any particular choice of units. Conversely, the user layer only refers to units. Internally, dimensional quantities are to be understood as rich types with a numerical factor and exponents for each of the base dimensions. They are independent of the particular choice of units by which they are assigned.

Note: The format for units and dimensions is the same as is used for LEMS/NeuroML v2.0 (http://www.neuroml.org) *[Cannon2014]*.

3.2.1 Dimension

Attribute	Type/Format	Required
name	identifier	yes
m	`integer`	no
l	`integer`	no
t	`integer`	no
i	`integer`	no
n	`integer`	no
k	`integer`	no
j	`integer`	no

Dimension objects are constructed values from the powers for each of the seven SI base units: length (*l*), mass (*m*), time (*t*), electric current (*i*), temperature (*k*), luminous intensity (*l*) and amount of substance (*n*). For example, acceleration has dimension lt^{-2} and voltage is $ml^2t^3i^{-1}$. *Dimension* objects must be declared in the top-level scope of the NineML document where they are referenced.

Name attribute

Each *Dimension* requires a *name* attribute, which should be a valid and uniquely identify the *Dimension* in current the scope.

M attribute

The *m* attribute specifies the power of the mass dimension in the *Dimension*. If omitted the power is zero.

L attribute

The *l* attribute specifies the power of the length dimension in the *Dimension*. If omitted the power is zero.

T attribute

The *t* attribute specifies the power of the time dimension in the *Dimension*. If omitted the power is zero.

I attribute

The *i* attribute specifies the power of the current dimension in the *Dimension*. If omitted the power is zero.

N attribute

The *n* attribute specifies the power of the amount-of-substance dimension in the *Dimension*. If omitted the power is zero.

K attribute

The *k* attribute specifies the power of the temperature dimension in the *Dimension*. If omitted the power is zero.

J attribute

The *j* attribute specifies the power of the luminous-intensity dimension in the *Dimension*. If omitted the power is zero.

3.2.2 Unit

Attribute	Type/Format	Required
symbol	`string`	yes
dimension	*Dimension*.name	yes
power	`integer`	no
offset	`integer`	no

Unit objects specify the dimension multiplier and the offset of a unit with respect to a defined *Dimension* object. *Unit* objects must be declared in the top-level scope of the NineML documents where they are referenced.

Symbol attribute

Each *Unit* requires a *symbol* attribute, which should be a valid and uniquely identify the *Unit* in current the scope.

Dimension attribute

Each *Unit* requires a *dimension* attribute. This attribute specifies the dimension of the units and should refer to the name of a *Dimension* element in the document scope.

Power attribute

Each *Unit* requires a *power* attribute. This attribute specifies the relative scale of the units compared to the equivalent SI units in powers of ten. If omitted the power is zero.

Offset attribute

A *Unit* can optionally have an *offset* attribute. This attribute specifies the zero offset of the unit scale. For example,

```
<Unit name="degC" dimension="temperature" power="0" offset="273.15"/>
```

If omitted, the offset is zero.

3.3 Annotating Elements

Annotations are provided to add semantic information about the model, preserving structure that is lost during conversion from an extended format to core NineML, and provide suggestions for the simulation of the model. It is highly recommended to add references to all publications on which the model or property values are based in the annotations. For adding semantic structure to the model it is recommended to use the Resource Description Framework (RDF) although it is not a strict requirement.

In order to be compliant with the NineML specification any tool handling NineML descriptions must preserve all existing annotations, except where a user explicitly edits/deletes them. In future versions of this section will be expanded to include suggested formats for commonly used annotations.

3.3.1 Annotations

Children	Multiplicity	Required
*	set	no

The *Annotations* element is the top-level of the annotations attached to a NineML element. They can be included at the top level of a document and within any NineML element (User Layer or Abstraction Layer), and may contain any object hierarchy that can be serialized to valid XML (although other hierarchical formats are supported, see *Serialization*).

CHAPTER 4

Abstraction Layer

4.1 Component Classes and Parameters

The main building block of the Abstraction Layer is the *ComponentClass*. The *ComponentClass* is intended to package together a collection of objects that relate to the definition of a model (e.g. cells, synapses, synaptic plasticity rules, random spike trains, inputs). All equations and event declarations that are part of particular entity model, such as neuron model, belong in a single *ComponentClass*. A *ComponentClass* can be used to represent either a specific model of a neuron or a composite model, including synaptic mechanisms.

The interface is the *external* view of the *ComponentClass* that defines what inputs and outputs the component exposes to other *ComponentClass* elements and the parameters that can be set for the *ComponentClass*. The interface consists of instances of ports and *Parameter* (see [fig:component_class_overview]).

As well as being able to specify the communication of continuous values, *ComponentClass* elements are also able to specify the emission and the reception of events. Events are discrete notifications that are transmitted over event ports. Since Event ports have names, saying that we transmit 'event1' for example would mean transmitting an event on the EventPort called 'event1'. Events can be used for example to signal action potential firing.

4.1.1 ComponentClass

Attribute	Type/Format	Required
name	identifier	yes

Children	Multiplicity	Required
Parameter	set	no
AnalogSendPort	set	no
AnalogReceivePort	set	no
AnalogReducePort	set	no
EventSendPort	set	no
EventReceivePort	set	no
[*Dynamics*,*ConnectionRule*,*RandomDistribution*]	singleton	yes

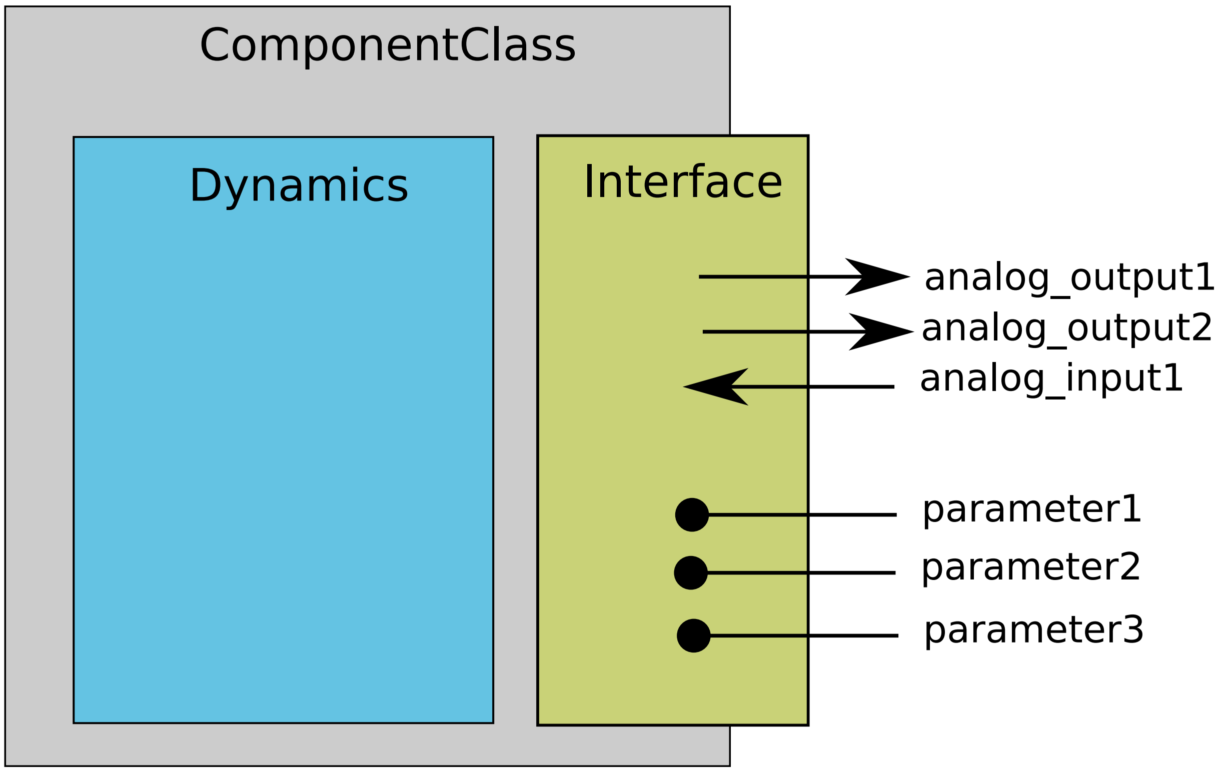

Fig. 4.1: ComponentClass Overview

A *ComponentClass* is composed of:

- *Parameter* objects for the *ComponentClass*, which specify which values are required to be provided in the User Layer.
- An unordered collection of port objects, which either publish or read state variables or derived values published from other components in the case of analog send and receive ports, or emit events or listen for events emitted from components. *EventSendPort* and *EventReceivePort* objects raise and listen for events passed between dynamic components.
- A 'main' block, which specifies the nature of the component class:
  - *Dynamics*, the component class defines a dynamic element such as neutron or post-synaptic response.
  - *ConnectionRule*, the component class defines a rule by which populations are connected in projections.
  - *RandomDistribution*, the component class defines random distribution.

Name attribute

Each *ComponentClass* requires a *name* attribute, which should be a valid and uniquely identify the *ComponentClass* in the document scope.

4.1.2 Parameter

Attribute	Type/Format	Required
name	identifier	yes
dimension	*Dimension*.name	yes

Parameter objects are placeholders for numerical values within a *ComponentClass*. They define particular qualities of the model, such as the firing threshold, reset voltage or the decay time constant of a synapse model. By definition, *Parameter*s are set at the start of the simulation, and remain constant throughout.

Name attribute

Each *Parameter* requires a *name* attribute, which is a valid and uniquely identifies the *Parameter* within the *ComponentClass*.

Dimension attribute

Parameter elements must have a *dimension* attribute. This attribute specifies the dimension of the units of the quantity that is expected to be passed to the *Parameter* and should refer to the name of a *Dimension* element in the document scope. For a dimensionless parameters a *Dimension* with all attributes of power 0 can be used.

4.2 Mathematical Expressions

As of NineML version 1.0, only inline mathematical expressions, which have similar syntax to the ANSI C89 standard, are supported. In future versions it is envisaged that inline expressions will be either augmented or replaced with MathML (http://mathml.org) expressions.

4.2.1 MathInline

Body format	Required
Inline-maths expression	yes

MathInline blocks are used to specify mathematical expressions. Depending on the context, *MathInline* blocks should return an expression that evaluates to either a bool (when used as the trigger for *OnCondition* objects) or a real number (when used as a right-hand-side for *Alias*, *TimeDerivative* and *StateAssignment* objects). All numbers/variables in inline maths expressions are assumed to be real numbers.

Body

The following arithmetic operators are supported in all inline maths expressions and have the same interpretation and precedence levels as in the ANSI C89 standard,

- Addition $+$
- Subtraction $-$
- Division $/$
- Multiplication $*$

The following inequality and logical operators are only supported in inline maths expressions within *Trigger* elements. They also have the same interpretation and precedence levels as in ANSI C89 standard.

- Greater than >
- Lesser than <
- Logical And: `&&`
- Logical Or: `||`
- Logical Not: `!`

The following functions are built in and are defined as per ANSI C89:

- `exp(x)`
- `sin(x)`
- `cos(x)`
- `log(x)`
- `log10(x)`
- `pow(x, p)`
- `sinh(x)`
- `cosh(x)`
- `tanh(x)`
- `sqrt(x)`
- `atan(x)`
- `asin(x)`
- `acos(x)`
- `asinh(x)`
- `acosh(x)`
- `atanh(x)`
- `atan2(x)`

The following symbols are built in, and cannot be redefined,

- pi
- t

where pi is the mathematical constant π, and t is the elapsed simulation time within a *Dynamics* block.

The following random distributions are available in *StateAssignment* elements via the `random` namespace, :

- `random.uniform` (see http://uncertml.org/distributions/uniform)
- `random.normal` (see http://uncertml.org/distributions/normal)
- `random.binomial(N,P)` (see http://uncertml.org/distributions/binomial)
- `random.poisson(L)` (see http://uncertml.org/distributions/poisson)
- `random.exponential(L)` (see http://uncertml.org/distributions/exponential)

4.2.2 Alias

Attribute	Type/Format	Required
name	identifier	yes

Children	Multiplicity	Required
MathInline	singleton	yes

An alias corresponds to an alternative name for a variable or part of an expression.

Aliases are motivated by two use cases:

- **substitution**: rather than writing long expressions for functions of state variables, we can split the expressions into a chain of *Alias* objects, e.g.:

```
m_alpha = (alphaA + alphaB * V)/(alphaC + exp((alphaD + V / alphaE)))
m_beta = (betaA + betaB * V)/(betaC + exp((betaD + V / betaE)))
minf = m_alpha / (m_alpha + m_beta)
mtau = 1.0 / (m_alpha + m_beta)
dm/dt = (1 / C) * (minf - m) / mtau
```

  In this case, `m_alpha`, `m_beta`, `minf` and `mtau` are all alias definitions. There is no reason we couldn't expand our $\mathrm{d}m/\mathrm{d}t$ description out to eliminate these intermediate *Alias* objects, but the expression would be very long and difficult to read.

- **Accessing intermediate variables**: if we would like to communicate a value other than a simple *StateVariable* to another *ComponentClass*. For example, if we have a component representing a neuron, which has an internal *StateVariable*, 'V', we may be interested in transmitting a current, for example $i = g * (E - V)$.

Name attribute

Each *Alias* requires a *name* attribute, which is a valid and uniquely identifies the *Alias* from all other elements in the *ComponentClass*.

4.2.3 Constant

Attribute	Type/Format	Required
name	identifier	yes
units	*Unit*.symbol	yes

Body format	Required
`float`	yes

Constant objects are used to specify physical constants such as the Ideal Gas *Constant* (i.e. 8.314462175 $\mathrm{JK}^{-1}\mathrm{mol}^{-1}$) or Avogadro's number (i.e. $6.0221412927 \times 10^{23}\mathrm{mol}^{-1}$), and to convert unit dimensions between abstract mathematical quantities.

The use of *Constant* elements to hold fixed model parameters is *strongly discouraged* since this breaks the division of semantic layers (abstraction and user), which is a key feature of NineML (see [sec:scope]).

Name attribute

Each *Constant* requires a *name* attribute, which should be a valid and uniquely identify the *Dimension* in current the scope.

Units attribute

Each *Constant* requires a *units* attribute. The *units* attribute specifies the units of the property and should refer to the name of a *Unit* element in the document scope.

Body

Any valid numeric value, including shorthand scientific notation e.g. 1e-5 (1×10^{-5}).

4.3 Ports

Ports allow components to communicate with each other during a simulation. Ports can either transmit discrete events or continuous streams of analog data. Events are typically used to transmit and receive spikes between neutron model, whereas analog ports can be used to model injected current and gap junctions between neuron models.

Ports are divided into sending, *EventSendPort* and *AnalogSendPort*, and receiving objects, *EventReceivePort*, *AnalogReceivePort* and *AnalogReducePort*. With the exception of *AnalogReducePort* objects, each receive port must be connected to exactly one matching (i.e. analog→analog, event→event) send port, where as a send port can be connected any number of receive ports. *AnalogReducePort* objects can be connected to any number of *AnalogSendPort* objects; the values of the connected ports are then “reduced” to a single data stream using the *operator* provided to the *AnalogReducePort*.

4.3.1 AnalogSendPort

Attribute	Type/Format	Required
name	[*StateVariable*,*Alias*].name	yes
dimension	*Dimension*.name	yes

AnalogSendPort objects allow variables from the current component to be published externally so they can be read by other *ComponentClass* objects. Each *AnalogSendPort* can be connected to multiple *AnalogReceivePort* and *AnalogReducePort* objects.

Name attribute

Each *AnalogSendPort* requires a *name* attribute, which should refer to a *StateVariable* or *Alias* within the current *ComponentClass*.

Dimension attribute

Each *AnalogSendPort* requires a *dimension* attribute. This attribute specifies the dimension of the units of the quantity that is expected to be passed through the *AnalogSendPort* and should refer to the name of a *Dimension* element in the document scope.

4.3.2 AnalogReceivePort

Attribute	Type/Format	Required
name	identifier	yes
dimension	*Dimension*.name	yes

*AnalogReceivePort*s allow variables that have been published externally to be used within the current component. Each *AnalogReceivePort* must be connected to exactly *one* *AnalogSendPort*.

Name attribute

Each *AnalogReceivePort* requires a *name* attribute, which is a valid and uniquely identifies the *AnalogReceivePort* from all other elements in the *ComponentClass*.

Dimension attribute

Each *AnalogReceivePort* requires a *dimension* attribute. This attribute specifies the dimension of the units of the quantity that is expected to be passed through the *AnalogReceivePort* and should refer to the name of a *Dimension* element in the document scope.

4.3.3 AnalogReducePort

Attribute	Type/Format	Required
name	identifier	yes
dimension	*Dimension*.name	yes
operator	+	yes

Reduce ports can receive data from any number of *AnalogSendPort* objects (including none). An *AnalogReducePort* takes an additional operator compared to an *AnalogReceivePort*, operator, which specifies how the data from multiple analog send ports should be combined to produce a single value. Currently, the only supported operation is $+$, which calculates the sum of the incoming port values.

The motivation for *AnalogReducePort* is that it allows us to make our *ComponentClass* definitions more general. For example, if we are defining a neuron, we would define an *AnalogReducePort* called *InjectedCurrent*. This allows us to write the membrane equation for that neuron as

$\mathrm{d}V/\mathrm{d}t = (1/C) * InjectedCurrent.$

Then, when we connect this neuron to synapses, current-clamps, etc, we simply need to connect the send ports containing the currents of these ComponentClass_es to the *InjectedCurrent* reduce port, without having to change our original *ComponentClass* definitions.

Name attribute

Each *AnalogReducePort* requires a *name* attribute, which is a valid and uniquely identifies the *AnalogReducePort* from all other elements in the *ComponentClass*.

Dimension attribute

Each *AnalogReducePort* requires a *dimension* attribute. This attribute specifies the dimension of the units of the quantity that is expected to be communicated through the *AnalogReducePort* and should refer to the name of a *Dimension* element in the document scope.

Operator attribute

Each *AnalogReducePort* requires an *operator* attribute. The operator reduces the connected inputs to a single value at each time point. For example the following port,

```
<AnalogReducePort name="total_membrane_current" dimension="current" operator="+"/>
```

will take all of the electrical currents that have been connected to it via *AnalogSendPort*s and sum them to get the total current passing through the membrane.

4.3.4 EventSendPort

Attribute	Type/Format	Required
name	identifier	yes

An *EventSendPort* specifies a channel over which events can be transmitted from a component. Each *EventSendPort* can be connected any number of *EventReceivePort* objects.

Name attribute

Each *EventSendPort* requires a *name* attribute, which is a valid and uniquely identifies the *EventSendPort* from all other elements in the *ComponentClass*.

4.3.5 EventReceivePort

Attribute	Type/Format	Required
name	identifier	yes

An *EventReceivePort* specifies a channel over which events can be received by a component. Each *EventReceivePort* must be connected to exactly *one* *EventSendPort*.

Name attribute

Each *EventReceivePort* requires a *name* attribute, which is a valid and uniquely identifies the *EventReceivePort* from all other elements in the *ComponentClass*.

4.4 Dynamic Regimes

Dynamics blocks define the dynamic equations of models such as neurons, post-synaptic responses or plasticity of synaptic weights. In *Dynamics* blocks, state variables are evolved by one or more sets of ordinary differential equations (ODE). Each set of equations is called a regime, and only one regime can be active at a particular point in time. The currently active regime can be changed by a transition event, which is represented as a logical expression on the state variables. When the logical expression evaluates to true, the transition must occur.

[fig:simple_regime_graph] illustrates a hypothetical transition graph for a system with three state variables, X, Y and Z, which transitions between three ODE regimes, *regime1*, *regime2* and *regime3*. At any time, the model will be in one and only one of these regimes, and the state variables will evolve according to the ODE of that regime.

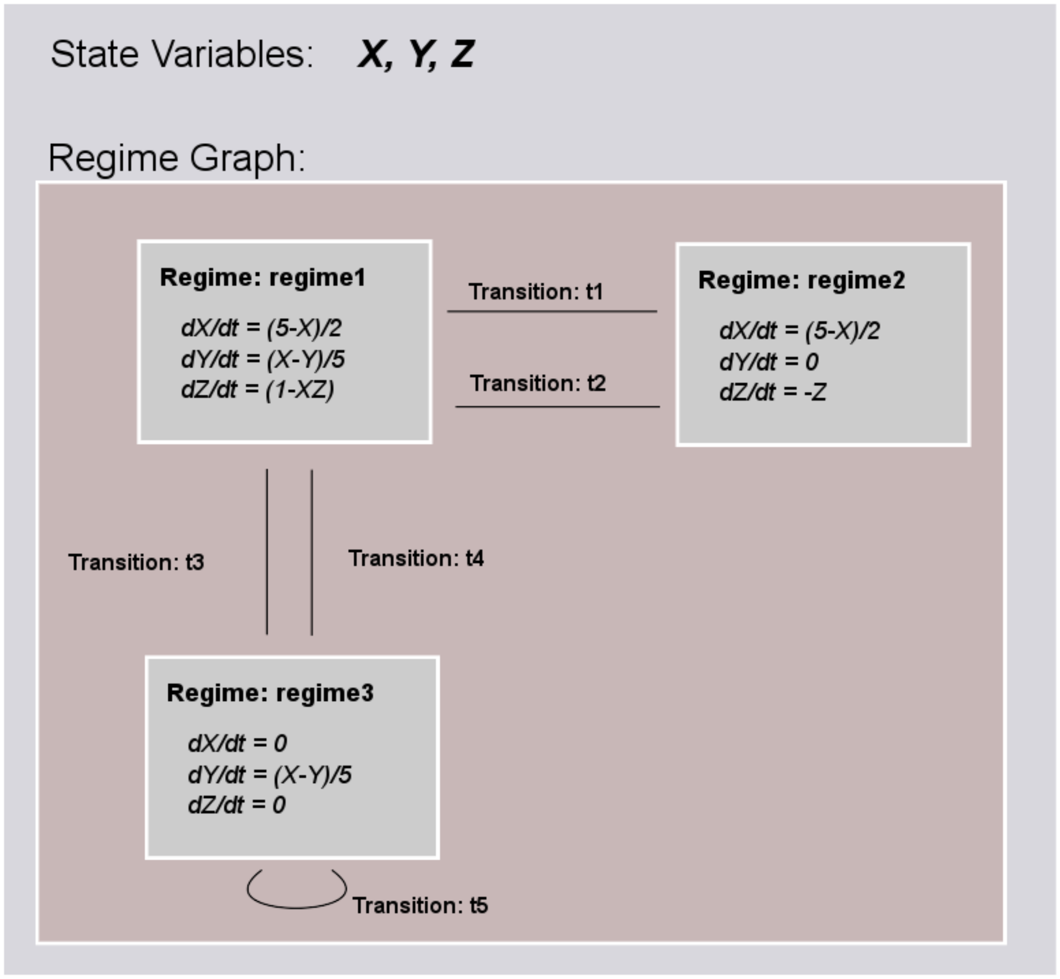

Fig. 4.2: The dynamics block for an example component.

4.4.1 Dynamics

Children	Multiplicity	Required
StateVariable	set	no
Regime	set	yes
Alias	set	no
Constant	set	no

The *Dynamics* block represents the *internal* mechanisms governing the behaviour of the component. These dynamics are based on ordinary differential equations (ODE) but may contain non-linear transitions between different ODE regimes. The regime graph (e.g. [fig:simple_regime_graph]) must contain at least one *Regime* element, and contain no regime islands. At any given time, a component will be in a single regime, and can change which regime it is in through transitions.

Note: *Alias* objects are defined in *Dynamics* blocks, not *Regime* blocks. This means that aliases are the same across all regimes.

4.4.2 StateVariable

Attribute	Type/Format	Required
name	identifier	yes
dimension	*Dimension*.name	yes

The state of the model is defined by a set of *StateVariable* objects. The value of a *StateVariable* can change in two ways:

- continuously through *TimeDerivative* elements (in *Regime* elements), which define how the *StateVariable* evolves over time, e.g. $dX/dt = 1 - X$.
- discretely through *StateAssignment* (in *OnCondition* or *OnEvent* transition elements), which make discrete changes to a *StateVariable* value, e.g. $X = X + 1$.

Name attribute

Each *StateVariable* requires a *name* attribute, which is a valid and uniquely identifies the *StateVariable* from all other elements in the *ComponentClass*.

Dimension attribute

Each *StateVariable* requires a *dimension* attribute. This attribute specifies the dimension of the units of the quantities that *StateVariable* is expected to be initialised and updated with and should refer to the name of a *Dimension* element in the document scope.

4.4.3 Regime

Attribute	Type/Format	Required
name	identifier	yes

Children	Multiplicity	Required
TimeDerivative	set	no
OnCondition	set	no
OnEvent	set	no

A *Regime* element represents a system of ODEs in time on *StateVariable*. As such, *Regime* defines how the state variables change (propagate in time) between subsequent transitions.

Name attribute

Each *Regime* requires a *name* attribute, which is a valid and uniquely identifies the *Regime* from all other elements in the *ComponentClass*.

4.4.4 TimeDerivative

Attribute	Type/Format	Required
variable	*StateVariable*.name	yes

Children	Multiplicity	Required
MathInline	singleton	yes

TimeDerivative elements contain a mathematical expression for the right-hand side of the ODE

$$\frac{\mathrm{d}variable}{\mathrm{d}t} = expression$$

which can contain of references to any combination of *StateVariable*, *Parameter*, *AnalogReceivePort*, *AnalogReducePort* and *Alias* elements with the exception of aliases that are derived from *RandomDistribution* components. Therefore, only one *TimeDerivative* element is allowed per *StateVariable* per *Regime*. If a *TimeDerivative* for a *StateVariable* is not defined in a *Regime*, it is assumed to be zero.

Variable attribute

Each *TimeDerivative* requires a *variable* attribute. This should refer to the name of a *StateVariable* in the *ComponentClass*. Only one *TimeDerivative* is allowed per *variable* in each *Regime*.

4.5 Transitions

The currently active dynamic regime can be changed via transitions. Transitions have instantaneous temporal extent (i.e. they are event-like). There are two types of transitions, condition-triggered transitions (see *OnCondition*), which are evoked when an associated trigger expression becomes true, or event-triggered transitions (see *OnEvent*), which are evoked when an associated event port receives an event from an external component. Multiple state assignments can be defined and multiple events can be sent within a single transition block.

During either type of transition three instantaneous actions can occur:

- The component transitions to a target regime (can be the same as the current regime)
- State variables can be assigned new values (see *StateAssignment*)
- The component can send events (see *OutputEvent*).

There is no order defined in transitions; this means that the order of resolution of state assignments can be ambiguous. If, for example, we have two transitions, T1 and T2, originating from the same *Regime*, in which T1 contains the state assignment *V=V+1* and T2 contains the assignment *V=V*V*, and both transitions are triggered simultaneously, then there is no guarantee about the value of V. It is left to the user to ensure such situations do not occur. Implementations should emit a warning when they are detected.

4.5.1 OnCondition

Attribute	Type/Format	Required
target_regime	*Regime*.name	no

Children	Multiplicity	Required
Trigger	singleton	yes
StateAssignment	set	no
OutputEvent	set	no

OnCondition blocks are activated when the mathematical expression in the *Trigger* block becomes true. They are typically used to model spikes in spiking neuron models, potentially emitting spike events and/or transitioning to an explicit refractory regime.

Target_regime attribute

An *OnEvent* can have a *target_regime* attribute, which should refer to the name of a *Regime* element in the *ComponentClass* that the dynamics block will transition to when the trigger condition is met. If the *target_regime* attribute is omitted the regime will transition to itself.

4.5.2 OnEvent

Attribute	Type/Format	Required
target_regime	*Regime*.name	no
port	*EventReceivePort*.name	yes

Children	Multiplicity	Required
StateAssignment	set	no
OutputEvent	set	no

OnEvent blocks are activated when the dynamics component receives an event from an external component on the port the *OnEvent* element is “listening” to. They are typically used to model the transient response to spike events from incoming synaptic connections.

Cascading of events, i.e. events triggering subsequent events, are permitted, which in theory could be recursive through components depending on their connectivity. It is the user’s responsibility to ensure that infinite recursion does not occur with zero delay. Implementations may decide to terminate after a given number of recursive cascades of zero delay (say 1000) to prevent infinite loops, but such limits should be modifiable by the user.

Port attribute

Each *OnEvent* requires a *port* attribute. This should refer to the name of an *EventReceivePort* in the *ComponentClass* interface.

Target_regime attribute

OnEvent can have a *targetRegime* attribute, which should refer to the name of a *Regime* element in the *ComponentClass* that the dynamics block will transition to when the *OnEvent* block is triggered by an incoming event. If the *targetRegime* attribute is omitted the regime will transition to itself.

4.5.3 Trigger

Children	Multiplicity	Required
MathInline	singleton	yes

Trigger objects define when an *OnCondition* transition should occur. The *MathInline* block of a *Trigger* can contain any arbitrary combination of 'and', 'or' and 'negation' *logical operations* ('&&', '||' and '!' respectively) on the result of pure inequality *relational operations* ('>' and '<'), which follow the syntax and semantics of ANSI C89. The inequality expression may contain references to *StateVariable*, *AnalogReceivePort*, *AnalogReducePort*, *Parameter* and *Alias* elements, with the exception of *Alias* elements derived from random distributions. The *OnCondition* block is triggered when the boolean result of the *Trigger* statement changes from *false* to *true*.

4.5.4 StateAssignment

Attribute	Type/Format	Required
variable	*StateVariable*.name	yes

Children	Multiplicity	Required
MathInline	singleton	yes

StateAssignment elements allow discontinuous changes in the value of state variables. Only one state assignment is allowed per variable per transition block. The assignment expression may contain references to *StateVariable*, *AnalogReceivePort*, *AnalogReducePort*, *Parameter* and *Alias* elements, including *Alias* elements derived from random distributions. State assignments are typically used to reset the membrane voltage after an outgoing spike event or update post-synaptic response states after an incoming spike event.

Variable attribute

Each *StateAssignment* requires a *variable* attribute. This should refer to the name of a *StateVariable* in the *ComponentClass*. Only one *StateAssignment* is allow per *variable* in each *OnEvent* or *OnCondition* block.

4.5.5 OutputEvent

Attribute	Type/Format	Required
port	*EventSendPort*.name	yes

OutputEvent elements specify events to be raised during a transition. They are typically used to raise spike events from within *OnCondition* elements.

Port attribute

Each *OutputEvent* requires a *port* attribute. This should refer to the name of an *EventSendPort* in the *ComponentClass* interface.

4.6 Random Distributions

Values for a property across all elements in a container (e.g. cells in a population, post-synaptic responses, plasticity rules or delays in a projection) can be defined as a random distribution by a *Component* within a RandomDistribution_Value element. A random distribution component must parameterize a *ComponentClass* with a *RandomDistribution* block; the component class defines the random distribution family (e.g. normal, cauchy, gamma, etc…). As of version 1.0, the only random distributions available to the user are those defined in the standard library, however, derived distributions are planned for future versions.

4.6.1 RandomDistribution

Attribute	Type/Format	Required
standard_library	URL	yes

The names and parameters of the random distribution in the standard library match the UncertML definitions that can be found at http://www.uncertml.org/distributions. The subset of the UncertML distributions that should be implemented are by NineML compliant packages are,

- BernoulliDistribution
- BetaDistribution
- BinomialDistribution
- CauchyDistribution
- ChiSquareDistribution
- DirichletDistribution
- ExponentialDistribution
- FDistribution
- GammaDistribution
- GeometricDistribution
- HypergeometricDistribution
- LaplaceDistribution
- LogisticDistribution
- LogNormalDistribution
- MultinomialDistribution
- NegativeBinomialDistribution
- NormalDistribution
- ParetoDistribution
- PoissonDistribution

- UniformDistribution
- WeibullDistribution

Note: Note: C implementations of these distributions are available in the GNU Scientific Library, http://www.gnu.org/software/gsl/

Standard_library attribute

The *standard_library* attribute is required and should point to a URLin the http://www.uncertml.org/distributions/ directory.

4.7 Network Connectivity

The connection rule for cells in the source and destination populations of a *Projection* (i.e. the rule that determines which source cells are connected to which destination cells) is defined by a connection-rule component within the *Connectivity* element of the *Projection*. This component must parameterize a *ComponentClass* with a *ConnectionRule* block, which describes the connection algorithm. As of version 1.0, the only connection rules available to the user are those defined in the standard library (e.g. all-to-all, one-to-one, probabilistic, etc…), however, custom connectivity rules are planned for future versions.

4.7.1 ConnectionRule

Attribute	Type/Format	Required
standard_library	URL	yes

Connection rules must be one of 6 standard library types, *all-to-all*, *one-to-one*, *probabilistic*, *explicit*, *random-fan-out* and *random-fan-in*, provided to the *standard_libarary* attribute.

Note: In future versions, built-in connectivity rules are to be replaced with mathematically expressed connection rules.

Standard_library attribute

The *standard_library* attribute is required and should point to the URLin the http://nineml.net/9ML/1.0/connectionrules/directory that corresponds to the desired connection rule.

All cells in the source population are connected to all cells in the destination population.

Each cell in the source population is connected to the cell in the destination population with the corresponding index. Note that this requires that the source and destination populations be the same size.

All cells in the source population are connected to cells in the destination population with a probability defined by a parameter, which should be named *probability*. The properties supplied to the *probability* parameter should either be a *SingleValue* representing the probability of a connection between all source and destination cell pairs, or a *ArrayValue* or *ExternalArrayValue* of size $M \times N$, where M and N are the size of the source and destination populations respectively. For array probabilities, the data in the *ArrayValue* or *ExternalArrayValue* are ordered by the indices

$$i_{\text{prob}} = i_{\text{source}} * N_{\text{dest}} + i_{\text{dest}}$$

where i_{prob}, i_{source} and i_{dest} are the indices of the probability entry, and the source and destination cells respectively, and N_{dest} is the size of the destination population.

*Cell*s in the source population are connected to cells in the destination population as specified by an explicit arrays. The source and destination are defined via parameters, which should be named *sourceIndicies* and *destinationIndicies* parameters respectively.

The properties supplied to the *sourceIndicies* parameter should be a *ArrayValue* or *ExternalArrayValue* drawn from the set $\{1, \ldots, M\}$ where M is the size of the source population and be the same length as the property supplied to the *target-indices* parameter.

The properties supplied to the *destinationIndicies* parameter should be a *ArrayValue* or *ExternalArrayValue* drawn from the set $\{1, \ldots, N\}$ where N is the size of the source population and be the same length as the property supplied to the *source-indices* parameter.

Each cell in the source population is connected to a fixed number of randomly selected cells in the destination population. The number of cells is specified by the parameter *number*. The property supplied to the *number* parameter should be a *SingleValue*.

Each cell in the destination population is connected to a fixed number of randomly selected cells in the source population. The number of cells is specified by the parameter *number*. The property supplied to the *number* parameter should be a *SingleValue*.

CHAPTER 5

User Layer

5.1 Components and Properties

5.1.1 Component

Attribute	Type/Format	Required
name	identifier	yes

Children	Multiplicity	Required
[*Definition*,*Prototype*]	singleton	yes
Property	set	no

Component elements instantiate Abstraction Layer component classes by providing properties for each of the parameters defined the class. Each *Component* is linked to a *ComponentClass* class by a *Definition* element, which locates the component class. A *Component* that instantiates a *ComponentClass* directly must supply matching *Property* elements for each *Parameter* in the *ComponentClass*. Alternatively, a *Component* can inherit a *ComponentClass* and set of *Property* elements from an existing component by substituting the *Definition* for a *Prototype* element, which locates the reference *Component*. In this case, only the properties that differ from the reference component need to be specified.

Name attribute

Each *Component* requires a *name* attribute, which should be a valid and uniquely identify the *Component* from all other elements in the document scope.

5.1.2 Definition

Attribute	Type/Format	Required
url	URL	no

Body format	Required
ComponentClass.name	yes

The *Definition* element establishes a link between a User Layer component and Abstraction Layer *ComponentClass*. This *ComponentClass* can be located either in the current document or in another file if a *url* attribute is provided.

Url attribute

If the *ComponentClass* referenced by the definition element is defined outside the current document, the *url* attribute specifies a URLfor the file which contains the *ComponentClass* definition. If the *url* attribute is omitted the *ComponentClass* is referenced from the current document.

Body

The name of the *ComponentClass* to be referenced *ComponentClass* needs to be provided in the body of the *Definition* element.

5.1.3 Prototype

Attribute	Type/Format	Required
url	URL	no

Body format	Required
Component.name	yes

The *Prototype* element establishes a link to an existing User Layer *Component*, which defines the *ComponentClass* and default properties of the *Component*. The reference *Component* can be located either in the current document or in another file if a *url* attribute is provided.

Url attribute

If the prototype *Component* is defined outside the current file, the *URL* attribute specifies a URLfor the file which contains the prototype *Component*. If the *url* attribute is omitted the *Component* is referenced from the current document.

Body

The name of the *Component* to be referenced *Component* needs to be provided in the body of the *Prototype* element.

5.1.4 Property

Attribute	Type/Format	Required
name	*Parameter*.name	yes
units	*Unit*.symbol	yes

Children	Multiplicity	Required
[*SingleValue*,*ArrayValue*,*ExternalArrayValue*,*RandomDistributionValue*]	singleton	yes

Property elements provide values for the parameters defined in the *ComponentClass* of the *Component*. Their *name* attribute should match the name of the corresponding *Parameter* element in the *ComponentClass*. The *Property* should be provided units that match the dimensionality of the corresponding *Parameter* definition.

Name attribute

Each *Property* requires a *name* attribute. This should refer to the name of a *Parameter* in the corresponding *ComponentClass* of the *Component*.

Units attribute

Each *Property* element requires a *units* attribute. The *units* attribute specifies the units of the quantity and should refer to the name of a *Unit* element in the document scope. For a dimensionless units a *Unit* with no SI dimensions can be used. The SI dimensions of the *Unit* should match the SI dimensions of the corresponding *Parameter*.

Note: "Dimensionless" parameters can be defined by referring to an empty Dimension object, i.e. one without any power or offset attributes

5.1.5 Reference

Attribute	Type/Format	Required
url	URL	no

Body format	Required
*.name	yes

Reference elements are used to locate User Layer elements in the document scope of the current separate documents. In most cases, User Layer elements (with the exception of *Population* elements supplied to *Projection*) can be specified inline, i.e. within the element they are required. However, it is often convenient to define a component in the document scope as this allows it to be reused at different places within the model. The *url* attribute can be used to reference a component in a separate document, potentially one published online in a public repository (e.g. ModelDB or Open Source Brain).

Url attribute

The *url* attribute specifies a URLfor the file which contains the User Layer element to be referenced. If the *url* attribute is omitted the element is referenced from the current document.

Body

The name of the User Layer element to be referenced should be included in the body of the *Reference* element.

5.2 Values

In NineML, "values" are arrays that implicitly grow to fill the size of the container (i.e. *Population* or *Projection*) they are located within. Values can be one of four types

- *SingleValue*, a consistent value across the container
- *ArrayValue*, an explicit array defined in NineML
- *ExternalArrayValue*, an explicit array defined in text (space delimited) or HDF5 format.
- *RandomDistributionValue*, an array of values derived from a random distribution.

5.2.1 SingleValue

Body format	Required
`integer`	yes

A *SingleValue* element represents an array filled with a single value.

Body

Any valid numeric value in ANSI C89, including shorthand scientific notation e.g. 1e-5 (1×10^{-5}).

5.2.2 ArrayValue

Children	Multiplicity	Required
ArrayValueRow	set	no

ArrayValue elements are used to represent an explicit array of values. *ArrayValue* elements contain a set of ArrayValue_Row elements (i.e. unordered, since they are explicitly ordered by their *index* attribute) in hierarchical data formats (see *Serialization*). Since is significantly slower to parse than plain text and binary formats it is not recommended to use *ArrayValue* for large arrays, preferring *ExternalArrayValue* instead.

5.2.3 ArrayValueRow

Attribute	Type/Format	Required
index	`integer`	yes

Body format	Required
`integer`	yes

ArrayValue_Row elements represent the numerical values of the explicit *ArrayValue* element.

Index attribute

The *index* attribute specifies the index of the ArrayValue_Row in the *ArrayValue*. It must be non-negative, unique amongst the set of ArrayValue_Row.index in the list, and the set of indices must be contiguous for a single *ArrayValue*.

Body

Any valid numeric value in ANSI C89, including shorthand scientific notation e.g. 1e-5 (1×10^{-5}).

Note: The order of ArrayValue_Row elements within an *ArrayValue* element does not effect the interpreted order of the values in the array in keeping with the order non-specific design philosophy of NineML.

5.2.4 ExternalArrayValue

Attribute	Type/Format	Required
url	URL	yes
mimeType	MIME type	yes
columnName	Data column name in external file	yes

ExternalArrayValue elements are used to explicitly define large arrays of values. The array data are not stored within the hierarchical data format but more efficient text or binary HDF5 (http://www.hdfgroup.org/HDF5/) formats. As of version 1.0, the data in the external files are stored as dense float or integer arrays. However, sparse-array formats are planned for future versions.

The *columnName* attribute of the *ExternalArrayValue* elements allows multiple arrays of equal length (and therefore typically relating to the same container) to be stored in the same external file.

Url attribute

The *url* attribute specifies the URLof the external data file.

MimeType attribute

The *mimetype* attribute specifies the data format for the external value list in the MIME type syntax. Currently, only two formats are supported `application/vnd.nineml.valuelist.text` and `application/vnd.nineml.valuelist.hdf5`.

- `application/vnd.nineml.externalvaluearray.text` - an ASCII text file with a single row of white-space separated column names, followed by arbitrarily many white-space separated data rows of numeric values. Each numeric value is associated with the column name corresponding to the same index the along the row. Therefore, the number of items in each row must be the same.
- `application/vnd.nineml.externalvaluearray.hdf5` - a HDF5 data file containing a single level of named members of `array->float` or `array->int` type.

ColumnName attribute

Each *ExternalArrayValue* must have a *columnName* attribute, which refers to a column header in the external data file.

5.2.5 RandomDistributionValue

Children	Multiplicity	Required
[*Component*,*Reference*]	singleton	yes

RandomDistributionValue elements represent arrays of values drawn from random distributions, which are defined by a *Component* elements. The size of the generated array is determined by the size of the container (i.e. *Population* or *Projection*) the *RandomDistributionValue* is nested within.

5.3 Populations

5.3.1 Population

Attribute	Type/Format	Required
name	identifier	yes

Children	Multiplicity	Required
Size	singleton	yes
Cell	singleton	yes

A *Population* defines a set of dynamic components of the same class. The size of the set is specified by the *Size* element. The properties of the dynamic components are generated from value types, which can be constant across the population, randomly distributed or individually specified (see *Values*).

Name attribute

Each *Population* requires a *name* attribute, which should be a valid and uniquely identify the *Population* from all other elements in the document scope.

5.3.2 Cell

Children	Multiplicity	Required
[*Component*,*Reference*]	singleton	yes

The *Cell* element specifies the dynamic components that will make up the population. The *Component* can be defined inline or via a *Reference* element.

5.3.3 Size

Body format	Required
int	yes

The number of cells in the population is specified by the integer provided in the body of the *Size* element. In future versions this may be extended to allow the size of a population to be derived from other features of the *Population*.

Body

The text of the *Size* element contains an integer representing the size of the population.

5.4 Projections

Projections define the synaptic connectivity between two populations, the post-synaptic response of the connections, the plasticity rules that modulate the post-synaptic response and the transmission delays. Synaptic and plasticity dynamic components are created if the connection rule determines there is a connection between a particular source and destination cell pair. The synaptic and plasticity components are then connected to and from explicitly defined ports of the cell components in the source and projection populations

SingleValue and *RandomDistributionValue* elements used in properties of a projection (in the *Connectivity*, *Response*, *Plasticity* and *Delay* elements) take the size of the number of connections made. Explicitly array values, *ArrayValue* and *ExternalArrayValue*, are only permitted with connection rules (as defined by the *Connectivity* element) where the number of connections is predetermined (i.e. *one-to-one*, *all-to-all* and *explicit*). Explicit arrays are ordered by the indices

$$i_{\text{value}} = i_{\text{source}} * N_{\text{dest}} + i_{\text{dest}}$$

where i_{value}, i_{source} and i_{dest} are the indices of the array entry, and the source and destination cells respectively, and N_{dest} is the size of the destination population. Value indices that do not correspond to connected pairs are omitted, and therefore the arrays are the same size as the number of connections.

5.4.1 Projection

Attribute	Type/Format	Required
name	identifier	yes

Children	Multiplicity	Required
Source	singleton	yes
Destination	singleton	yes
Connectivity	singleton	yes
Response	singleton	yes
Plasticity	singleton	no
Delay	singleton	yes

The *Projection* element contains all the elements that define a projection between two populations and should be uniquely identified in the scope of the document.

Name attribute

Each *Projection* requires a *name* attribute, which should be a valid and uniquely identify the *Projection* from all other elements in the document scope.

5.4.2 Connectivity

Children	Multiplicity	Required
Component	singleton	yes

Each *Connectivity* element contains a *Component*, which defines the connection pattern of the cells in the source population to cells in the destination population (i.e. binary 'connected' or 'not connected' decisions). For each connection that is specified, a synapse, consisting of a post-synaptic response and plasticity dynamic components, is created to model the synaptic interaction between the cells.

5.4.3 Source

Children	Multiplicity	Required
[*Component*,*Reference*]	singleton	yes
FromDestination	set	no
FromPlasticity	set	no
FromResponse	set	no

The *Source* element specifies the pre-synaptic population or selection (see *Selection*) of the projection and all the port connections it receives. The source population is specified via a *Reference* element since it should not be defined within the *Projection*. The source population can receive incoming port connections from the post-synaptic response (see *FromResponse*), the plasticity rule (see *FromPlasticity*) or the post-synaptic population directly (see *FromDestination*). Connections with these ports are only made if the Connectivity_determines that the source and destination cells should be connected.

5.4.4 Destination

Children	Multiplicity	Required
[*Component*,*Reference*]	singleton	yes
FromSource	set	no
FromPlasticity	set	no
FromResponse	set	no

The *Destination* element specifies the post-synaptic or selection (see *Selection*) population of the projection and all the port connections it receives. The destination population is specified via a *Reference* element since it should not be defined within the *Projection*. The source population can receive incoming port connections from the post-synaptic response (see *FromResponse*), the plasticity rule (see *FromPlasticity*) or the pre-synaptic population directly (see *FromSource*). Connections with these ports are only made if the Connectivity_determines that the source and destination cells should be connected.

5.4.5 Response

Children	Multiplicity	Required
[*Component*,*Reference*]	singleton	yes
FromSource	set	no
FromDestination	set	no
FromPlasticity	set	no

The *Response* defines the effect on the post-synaptic cell dynamics of an incoming synaptic input. The additional dynamics are defined by a *Component* element, which can be defined inline or referenced. For static connections (i.e. those without a *Plasticity* element), the magnitude of the response (i.e. synaptic weight) is typically passed as a property of the *Response* element.

The post-synaptic response dynamics can receive incoming port connections from the plasticity rule (see *FromPlasticity*) or the pre or post synaptic populations (see *FromSource* and *FromDestination*). The post-synaptic response object is implicitly created and connected to these ports if the Connectivity_determines that the source and destination cells should be connected.

5.4.6 Plasticity

Children	Multiplicity	Required
[*Component*,*Reference*]	singleton	yes
FromSource	set	no
FromDestination	set	no
FromResponse	set	no

The *Plasticity* element describes the dynamic processes that modulate the dynamics of the post-synaptic response, typically the magnitude of the response (see *Response*). If the synapse is not plastic the *Plasticity* element can be omitted.

The plasticity dynamics can receive incoming port connections from the post-synaptic response rule (see *FromResponse*) or the pre or post synaptic populations (see *FromSource* and *FromDestination*). The plasticity object is implicitly created and connected to these ports if the Connectivity_determines that the source and destination cells should be connected.

5.4.7 FromSource

Attribute	Type/Format	Required
sender	[*AnalogSendPort*,*EventSendPort*].name	yes
receiver	[*AnalogReceivePort*,*EventReceivePort*,*AnalogReducePort*].name	yes

The *FromSource* element specifies a port connection to the projection component (either the destination cell, post-synaptic response or plasticity dynamics) inside which it is inserted from the source cell dynamics.

Sender attribute

Each *FromSource* element requires a *sender* attribute. This should refer to the name of a *AnalogSendPort* or *EventSendPort* in the Cell_of the source population. The transmission mode of the port (i.e. analog or event) should match that of the port referenced by the *receiver* attribute.

Receiver attribute

Each *FromSource* element requires a *receiver* attribute. This should refer to the name of a *AnalogReceivePort*, *EventReceivePort* or *AnalogReducePort* in the *Component* in the enclosing *Source*/*Destination*/*Plasticity*/*Response* element. The transmission mode of the port (i.e. analog or event) should match that of the port referenced by the *sender* attribute.

5.4.8 FromDestination

Attribute	Type/Format	Required
sender	[*AnalogSendPort*,*EventSendPort*].name	yes
receiver	[*AnalogReceivePort*,*EventReceivePort*,*AnalogReducePort*].name	yes

The *FromDestination* element specifies a port connection to the projection component (either the source cell, post-synaptic response or plasticity dynamics) inside which it is inserted from the destination cell dynamics.

Sender attribute

Each *FromDestination* element requires a *sender* attribute. This should refer to the name of a *AnalogSendPort* or *EventSendPort* in the Cell_of the source population. The transmission mode of the port (i.e. analog or event) should match that of the port referenced by the *receiver* attribute.

Receiver attribute

Each *FromDestination* element requires a *receiver* attribute. This should refer to the name of a *AnalogReceivePort*, *EventReceivePort* or *AnalogReducePort* in the *Component* in the enclosing *Source*/*Destination*/*Plasticity*/*Response* element. The transmission mode of the port (i.e. analog or event) should match that of the port referenced by the *sender* attribute.

5.4.9 FromPlasticity

Attribute	Type/Format	Required
sender	[*AnalogSendPort*,*EventSendPort*].name	yes
receiver	[*AnalogReceivePort*,*EventReceivePort*,*AnalogReducePort*].name	yes

The *FromPlasticity* element specifies a port connection to the projection component (either the source cell, destination cell or post-synaptic response dynamics) inside which it is inserted from the plasticity dynamics.

Sender attribute

Each *FromPlasticity* element requires a *sender* attribute. This should refer to the name of a *AnalogSendPort* or *EventSendPort* in the *Cell*->Component_ of the source population. The transmission mode of the port (i.e. analog or event) should match that of the port referenced by the *receiver* attribute.

Receiver attribute

Each *FromPlasticity* element requires a *receiver* attribute. This should refer to the name of a *AnalogReceivePort*, *EventReceivePort* or *AnalogReducePort* in the *Component* in the enclosing *Source*/*Destination*/ *Plasticity*/*Response* element. The transmission mode of the port (i.e. analog or event) should match that of the port referenced by the *sender* attribute.

5.4.10 FromResponse

Attribute	Type/Format	Required
sender	[*AnalogSendPort*,*EventSendPort*].name	yes
receiver	[*AnalogReceivePort*,*EventReceivePort*,*AnalogReducePort*].name	yes

The *FromResponse* element specifies a port connection to the projection component (either the source cell, destination cell or plasticity dynamics) inside which it is inserted from the post-synaptic response dynamics.

Sender attribute

Each *FromResponse* element requires a *sender* attribute. This should refer to the name of a *AnalogSendPort* or *EventSendPort* in the *Cell*->Component_ of the source population. The transmission mode of the port (i.e. analog or event) should match that of the port referenced by the *receiver* attribute.

Receiver attribute

Each *FromResponse* element requires a *receiver* attribute. This should refer to the name of a *AnalogReceivePort*, *EventReceivePort* or *AnalogReducePort* in the *Component* in the enclosing *Source*/*Destination*/ *Plasticity*/*Response* element. The transmission mode of the port (i.e. analog or event) should match that of the port referenced by the *sender* attribute.

5.4.11 Delay

Attribute	Type/Format	Required
units	*Unit*.symbol	yes

Children	Multiplicity	Required
[*SingleValue*,*ArrayValue*,*ExternalArrayValue*,*RandomDistributionValue*]	singleton	yes

In version 1.0, the *Delay* element specifies the delay between the pre-synaptic cell port and both the Plasticity_and *Response*. In future versions, it is planned to include the delay directly into the port-connection objects (i.e. *FromSource*, *FromDestination*, etc. . .) to allow finer control of the delay between the different components.

Units attribute

The *units* attribute specifies the units of the delay and should refer to the name of a *Unit* element in the document scope. The units should be temporal, i.e. have $t = 1$ and all other SI dimensions set to 0.

5.5 Selections: combining populations and subsets

Selections are designed to allow sub and super-sets of cell populations to be projected to/from other populations (or selections thereof). In version 1.0, the only supported operation is the concatenation of multiple populations into super-sets but in future versions it is planned to provide “slicing” operations to select sub sets of populations.

5.5.1 Selection

Attribute	Type/Format	Required
name	identifier	yes

Children	Multiplicity	Required
Concatenate	singleton	yes

The *Selection* element contains the operations that are used to select the cells to add to the selection.

Name attribute

Each *Selection* requires a *name* attribute, which should be a valid and uniquely identify the *Selection* from all other elements in the document scope.

5.5.2 Concatenate

Children	Multiplicity	Required
Item	set	yes

The *Concatenate* element is used to add populations to a selection. It contains a set of *Item* elements which reference the *Population* elements to be concatenated. The order of the *Item* elements does not effect the order of the concatenation, which is determined by the *index* attribute of the *Item* elements. The set of Item_@*index* attributes must be non-negative, contiguous, not contain any duplicates and contain the index 0 (i.e. $i = 0, \ldots, N-1$).

5.5.3 Item

Attribute	Type/Format	Required
index	`integer`	yes

Children	Multiplicity	Required
Reference([*Population*,*Selection*])	singleton	yes

Each *Item* element references as a *Population* or *Selection* element and specifies their order in the concatenation.

Index attribute

Each *Item* requires a *index* attribute. This attribute specifies the order in which the *Population*s in the *Selection* are concatenated and thereby the indices of the cells within the combined *Selection*.

Note: This preserves the order non-specific nature of elements in NineML

CHAPTER 6

Serialization

There are four officially supported data formats for serializing NineML: XML, JSON, YAML, and HDF5 (although it is possible to use other data formats). When referenced from another NineML document, the format of a NineML file is recognised by the extension of its filename, i.e:

Format	Extension
XML	.xml
JSON	.json
YAML	.yml
HDF5	.h5

Note: Tools that plan to support NineML only need to support one data format since the officially supported NineML Python Library can be used to convert between the data formats listed above.

NineML is intended to be an abstract object model that is independent of the choice of hierarchical data format used to serialize it. However, some aspects of NineML were designed with XML in mind and there are some subtle differences between hierarchical data formats that prevent general mappings from XML. Therefore, in order to map the NineML object model onto non-XML data formats some additional conventions are required.

Several features of XML that are used in the NineML specification and are not present in JSON/YAML (JSON and YAML are equivalent representations), and/or HDF5 are:

Namespaces (xmlns): There is no concept of namespaces in JSON/YAML or HDF5, which are used in NineML to distinguish the document version and annotations.

Attributes: In JSON/YAML there is no concept of attributes. This does not pose a problem if a given NineML type does not have body text as attributes can be treated as separate children. However, for NineML types that do, such as *Constant* and *Definition*, both the body text and attributes can't be represented without additional conventions.

Sets of child elements: While there are list structures in JSON/YAML, which can be used to represent arbitrarily sized sets of child elements (e.g. parameters, properties, regimes), HDF5 does not have an equivalent structure for storing sets of objects of the same type.

Fortunately, JSON, YAML and HDF5 all permit arbitrary strings as field names, whereas element/attribute names in XML must start with an alphabetic character. Therefore we can use non alphanumeric characters, in this case the '@' symbol, to escape the following special fields.

@namespace: Holds the namespace of the element as the special attribute xmlns does in XML.

@body: Used to differentiate body text from other attributes in JSON/YAML and HDF5 iff there are other attributes (Datasets could technically be used as body elements in HDF5 but they are designed to hold array data not single values). Note that for JSON/YAML and HDF5 if the serial form of an element only contains body text (e.g. *MathInline*) then this is "flattened" to be the sole value of the element.

@multiple: A **HDF_** group that has a @multiple attribute equal to 'true', contains multiple child elements of the given NineML type, which are stored as sub-groups named by arbitrary integer indices. Note that this is not strictly required for elements in the NineML specification (although it simplifies code to read them), where the multiplicity of children of a given type is defined, but is for parsing arbitrary object hierarchies in annotations.

Note: Future versions of NineML will be designed to minimise the need for the the *@body* field within the NineML object model. However, it will still be required to represent arbitrary annotations and language extensions designed in XML.

ArrayValues should also be stored within native data array structures of the format (e.g. HDF5 datasets) instead of within ArrayValueRow elements.

The following model of a Izhikevich neuron uses both shows an example of how namespaces and body elements are represented natively in XML.

```
<?xml version='1.0' encoding='UTF-8'?>
<NineML xmlns="http://nineml.net/9ML/1.0">
  <ComponentClass name="Izhikevich">
    <Parameter name="C_m" dimension="capacitance"/>
    <Parameter name="a" dimension="per_time"/>
    <Parameter name="alpha" dimension="per_time_voltage"/>
    <Parameter name="b" dimension="per_time"/>
    <Parameter name="beta" dimension="per_time"/>
    <Parameter name="c" dimension="voltage"/>
    <Parameter name="d" dimension="voltage_per_time"/>
    <Parameter name="theta" dimension="voltage"/>
    <Parameter name="zeta" dimension="voltage_per_time"/>
    <AnalogReducePort name="Isyn" dimension="current" operator="+"/>
    <EventSendPort name="spike"/>
    <AnalogSendPort name="V" dimension="voltage"/>
    <Dynamics>
      <StateVariable name="U" dimension="voltage_per_time"/>
      <StateVariable name="V" dimension="voltage"/>
      <Regime name="subthreshold_regime">
        <TimeDerivative variable="U">
          <MathInline>a*(-U + V*b)</MathInline>
        </TimeDerivative>
        <TimeDerivative variable="V">
          <MathInline>-U + V*beta + alpha*(V*V) + zeta + Isyn/C_m</MathInline>
        </TimeDerivative>
        <OnCondition target_regime="subthreshold_regime">
          <Trigger>
            <MathInline>V &gt; theta</MathInline>
          </Trigger>
          <StateAssignment variable="U">
            <MathInline>U + d</MathInline>
          </StateAssignment>
          <StateAssignment variable="V">
            <MathInline>c</MathInline>
          </StateAssignment>
          <OutputEvent port="spike"/>
        </OnCondition>
      </Regime>
    </Dynamics>
    <Annotations>
      <Validation xmlns="http://github.com/INCF/nineml-python" dimensionality="True"/>
    </Annotations>
  </ComponentClass>
  <Component name="SampleIzhikevich">
    <Definition url="./izhikevich.xml">Izhikevich</Definition>
    <Property name="C_m" units="pF">
      <SingleValue>1.0</SingleValue>
    </Property>
    <Property name="a" units="per_ms">
      <SingleValue>0.2</SingleValue>
    </Property>
    <Property name="alpha" units="per_mV_ms">
      <SingleValue>0.04</SingleValue>
    </Property>
    <Property name="b" units="per_ms">
      <SingleValue>0.025</SingleValue>
    </Property>
    <Property name="beta" units="per_ms">
      <SingleValue>5.0</SingleValue>
    </Property>
    <Property name="c" units="mV">
      <SingleValue>-75.0</SingleValue>
    </Property>
    <Property name="d" units="mV_per_ms">
      <SingleValue>0.2</SingleValue>
    </Property>
    <Property name="theta" units="mV">
      <SingleValue>-50.0</SingleValue>
    </Property>
    <Property name="zeta" units="mV_per_ms">
      <SingleValue>140.0</SingleValue>
    </Property>
    <Initial name="U" units="mV_per_ms">
      <SingleValue>-1.625</SingleValue>
    </Initial>
    <Initial name="V" units="mV">
      <SingleValue>-70.0</SingleValue>
    </Initial>
  </Component>
  <Dimension name="capacitance" m="-1" l="-2" t="4" i="2"/>
  <Dimension name="current" i="1"/>
  <Unit symbol="mV" dimension="voltage" power="-3"/>
  <Unit symbol="mV_per_ms" dimension="voltage_per_time" power="0"/>
  <Unit symbol="pF" dimension="capacitance" power="-12"/>
  <Unit symbol="per_mV_ms" dimension="per_time_voltage" power="6"/>
  <Unit symbol="per_ms" dimension="per_time" power="3"/>
  <Dimension name="per_time" t="-1"/>
  <Dimension name="per_time_voltage" m="-1" l="-2" t="2" i="1"/>
  <Dimension name="voltage" m="1" l="2" t="-3" i="-1"/>
  <Dimension name="voltage_per_time" m="1" l="2" t="-4" i="-1"/>
```

```
</NineML>
```

whereas in YAML the `@namespace` and `@body` fields must be used in place of the `xmlns` attribute and body text.

```
NineML:
  '@namespace': http://nineml.net/9ML/1.0
  ComponentClass:
  - name: Izhikevich
    Parameter:
    - {name: C_m, dimension: capacitance}
    - {name: a, dimension: per_time}
    - {name: alpha, dimension: per_time_voltage}
    - {name: b, dimension: per_time}
    - {name: beta, dimension: per_time}
    - {name: c, dimension: voltage}
    - {name: d, dimension: voltage_per_time}
    - {name: theta, dimension: voltage}
    - {name: zeta, dimension: voltage_per_time}
    AnalogReducePort:
    - {name: Isyn, dimension: current, operator: +}
    EventSendPort:
    - {name: spike}
    AnalogSendPort:
    - {name: V, dimension: voltage}
    Dynamics:
      StateVariable:
      - {name: U, dimension: voltage_per_time}
      - {name: V, dimension: voltage}
      Regime:
      - name: subthreshold_regime
        TimeDerivative:
        - {MathInline: a*(-U + V*b), variable: U}
        - {MathInline: -U + V*beta + alpha*(V*V) + zeta + Isyn/C_m, variable: V}
        OnCondition:
        - Trigger: {MathInline: V > theta}
          target_regime: subthreshold_regime
          StateAssignment:
          - {MathInline: U + d, variable: U}
          - {MathInline: c, variable: V}
          OutputEvent:
          - {port: spike}
    Annotations:
      Validation:
      - {'@namespace': 'http://github.com/INCF/nineml-python', dimensionality: 'True
↪'}
  Component:
  - Definition: {'@body': Izhikevich, url="./izhikevich.yml"}
    name: SampleIzhikevich
    Property:
    - {name: C_m, SingleValue: 1.0, units: pF}
    - {name: a, SingleValue: 0.2, units: per_ms}
    - {name: alpha, SingleValue: 0.04, units: per_mV_ms}
    - {name: b, SingleValue: 0.025, units: per_ms}
    - {name: beta, SingleValue: 5.0, units: per_ms}
    - {name: c, SingleValue: -75.0, units: mV}
    - {name: d, SingleValue: 0.2, units: mV_per_ms}
    - {name: theta, SingleValue: -50.0, units: mV}
    - {name: zeta, SingleValue: 140.0, units: mV_per_ms}
    Initial:
    - {name: U, SingleValue: -1.625, units: mV_per_ms}
    - {name: V, SingleValue: -70.0, units: mV}
  Dimension:
  - {name: capacitance, m: -1, l: -2, t: 4, i: 2}
  - {name: current, i: 1}
  - {name: per_time, t: -1}
  - {name: per_time_voltage, m: -1, l: -2, t: 2, i: 1}
  - {name: voltage, m: 1, l: 2, t: -3, i: -1}
  - {name: voltage_per_time, m: 1, l: 2, t: -4, i: -1}
  Unit:
  - {symbol: mV, dimension: voltage, power: -3}
  - {symbol: mV_per_ms, dimension: voltage_per_time, power: 0}
  - {symbol: pF, dimension: capacitance, power: -12}
  - {symbol: per_mV_ms, dimension: per_time_voltage, power: 6}
  - {symbol: per_ms, dimension: per_time, power: 3}
```

Example representation of sets of *Parameter* elements in HDF5 format:

```
/NineML/ComponentClass/Parameter/@multiple = true
/NineML/ComponentClass/Parameter/0/name = 'C_m'
/NineML/ComponentClass/Parameter/0/dimension = 'capacitance'
/NineML/ComponentClass/Parameter/1/name = 'a'
/NineML/ComponentClass/Parameter/1/dimension = 'per_time'
...
```

CHAPTER 7

Examples

7.1 Single Cell Models

7.1.1 Izhikevich Model

In this first example, we are describing how to represent the Izhikevich model in NineML *[Izhikevich2003]*. The model is composed of single *ComponentClass*, containing a single *Regime*, *subthresholdRegime*, and two state variables, U & V.

The ODEs defined for the *Regime* are:

$$\frac{dV}{dt} = 0.04 * V * V + 5 * V + 140.0 - U + i_{\text{synapse}} + i_{\text{injected}}$$
$$\frac{dU}{dt} = a * (b * V - U)$$

The *ComponentClass* has a single *OnCondition* transition, is triggered when $V > theta$. When triggered, It causes an Event called *spikeOutput* to be emitted, and two *StateAssignment*s to be made:

$$U \leftarrow U + d$$
$$V \leftarrow c$$

The target-regime of the *OnCondition* transition is not declared explicitly in the XML, implying that the target-regime is the same as the source-regime, i.e. *subthresholdRegime*.

The RegimeGraph is shown in Figure [fig:EX1_RegimeGraph]

Using this *Abstraction Layer* definition, as well as suitable parameters from the user layer; $a = 0.02, b = 0.2, c = -65, d = 8, i_{\text{injected}} = 5.0$, we can simulate this, giving output as shown in Figure [fig:Ex1_Output].

In Figure [fig:Ex1_Output], we can see the value of the *StateVariable* V over time. We can also see that when the value of $V > theta$ triggers the condition, we emit a spike, and the *StateAssignment* of $V \leftarrow c$ resets the value of V. The corresponding *Abstraction Layer* description for this model is:

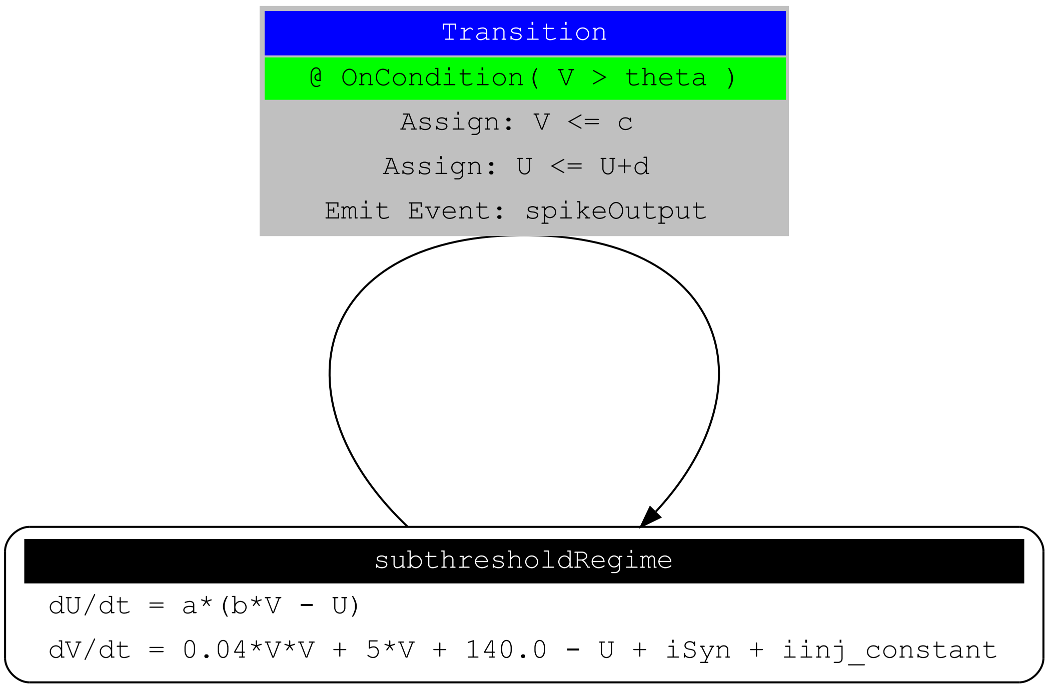

Fig. 7.1: RegimeGraph for the XML model in this section.

```
NineML:
  '@namespace': http://nineml.net/9ML/1.0
  ComponentClass:
  - name: Izhikevich
    Parameter:
    - {name: a, dimension: per_time}
    - {name: b, dimension: per_voltage}
    - {name: c, dimension: voltage}
    - {name: d, dimension: dimensionless}
    - {name: iInj, dimension: current}
    - {name: theta, dimension: voltage}
    AnalogReducePort:
    - {name: iSyn, dimension: current, operator: +}
    EventSendPort:
    - {name: spikeOutput}
    AnalogSendPort:
    - {name: V, dimension: voltage}
    Dynamics:
      StateVariable:
      - {name: U, dimension: dimensionless}
      - {name: V, dimension: voltage}
      Regime:
      - name: subthresholdRegime
        TimeDerivative:
        - {variable: U, MathInline: a*(-U + V*b)}
        - {variable: V, MathInline: (5*V + 0.04*(V*V)/unitV + unitR*(iInj + iSyn)
            + unitV*(-U + 140.0))/unitT}
        OnCondition:
        - Trigger: {MathInline: V > theta}
          target_regime: subthresholdRegime
          StateAssignment:
          - {variable: U, MathInline: U + d}
          - {variable: V, MathInline: c}
          OutputEvent:
          - {port: spikeOutput}
      Constant:
      - {name: unitR, units: Ohm, '@body': 1.0}
      - {name: unitT, units: s, '@body': 1.0}
      - {name: unitV, units: V, '@body': 1.0}
  Dimension:
  - {name: dimensionless}
  - {name: per_time, t: -1}
  - {name: per_voltage, m: -1, l: -2, t: 3, i: 1}
  - {name: voltage, m: 1, l: 2, t: -3, i: -1}
  - {symbol: Ohm, dimension: resistance, power: 0}
  - {symbol: V, dimension: voltage, power: 0}
  - {symbol: s, dimension: time, power: 1}
```

User Layer description for the above example is:

```
NineML:
  '@namespace': http://nineml.net/9ML/1.0
  ComponentClass:
  - name: IzhikevichProperties
    Definition: {'@body': Izhikevich}
    Property:
    - {name: a, SingleValue: 0.02, units: per_s}
    - {name: b, SingleValue: 0.2, units: per_V}
    - {name: c, SingleValue: -65.0, units: mV}
    - {name: d, SingleValue: 8.0, units: unitless}
    - {name: iInj, SingleValue: 10.0, units: nA}
    - {name: theta, SingleValue: 50.0, units: mV}
  Unit:
  - {symbol: mV, dimension: voltage, power: -3}
  - {symbol: per_V, dimension: per_voltage, power: 0}
  - {symbol: per_s, dimension: per_time, power: 0}
  - {symbol: unitless, dimension: dimensionless, power: 0}
  Dimension:
  - {name: dimensionless}
  - {name: per_time, t: -1}
  - {name: per_voltage, m: -1, l: -2, t: 3, i: 1}
  - {name: voltage, m: 1, l: 2, t: -3, i: -1}
```

Here, we show the simulation results of this XML representation with an initial V=-60mV and U=0.

7.1.2 Leaky Integrate and Fire model

In this example, we build a representation of a integrate-and-fire neuron, with an attached input synapse *[Abbott1999]*. We have a single *StateVariable*, *iaf_V*. This time, the neuron has an absolute refractory period; which is implemented by using 2 regimes. *RegularRegime* & *RefractoryRegime* In *RegularRegime*, the neuron voltage evolves as:

$$\frac{d(iaf\_V)}{dt} = \frac{iaf\_gl * (iaf\_vrest - iaf\_V) + iaf\_ISyn + cobaExcit\_I}{iaf\_cm}$$

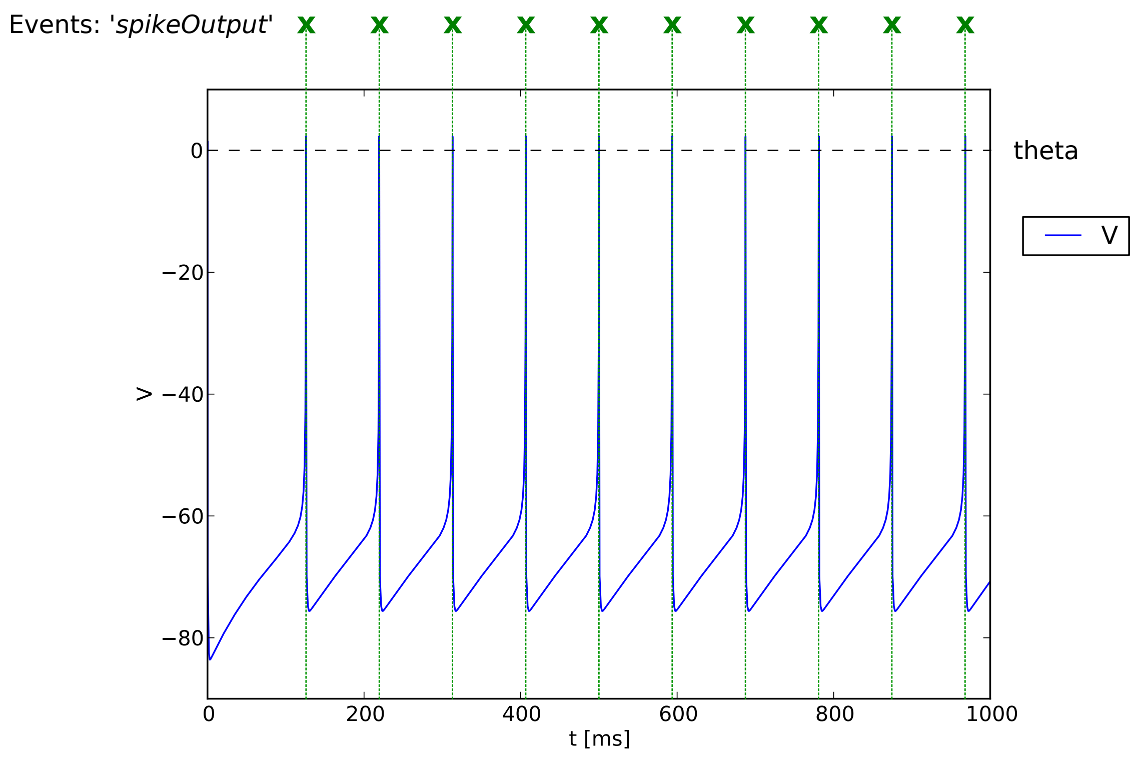

Fig. 7.2: Result of simulating of the XML model in this section

In *RefractoryRegime*, the neuron voltage does not change in response to any input:

$$\frac{d(iaf\_V)}{dt} = 0$$

In both *Regimes*, the synapses dynamics evolve as:

$$\frac{d(cobaExcit\_g)}{dt} = -\frac{cobaExcit\_g}{cobaExcit\_tau}$$

The neuron has two EventPorts, *iaf_spikeoutput* is a send port, which sends events when the neuron fires, and *cobaExcit_spikeinput* is a recv port, which tells the attached synapse that it should 'fire'. The neuron has 4 transitions, 2 *OnEvent* transitions and 2 *OnCondition* transitions. Two of the Transitions are triggered by *cobaExcit_spikeinput* events, which cause the conductance of the synapse to increase by an amount q, These happen in both *Regimes*. The other *OnConditions*:

- One is triggered the voltage being above threshold, which moves the component from *RegularRegime* to *RefractoryRegime*, sets V to the reset-voltage also emits a spike
- The other is triggered by enough time having passed for the component to come out of the *RefractoryRegime* and move back to the *RegularRegime*

The corresponding *Regime* Graph is shown in Figure 5.

The resulting description for the *Abstraction Layer* is:

```
NineML:
  '@namespace': http://nineml.net/9ML/1.0
  ComponentClass:
  - name: IafCoba
```

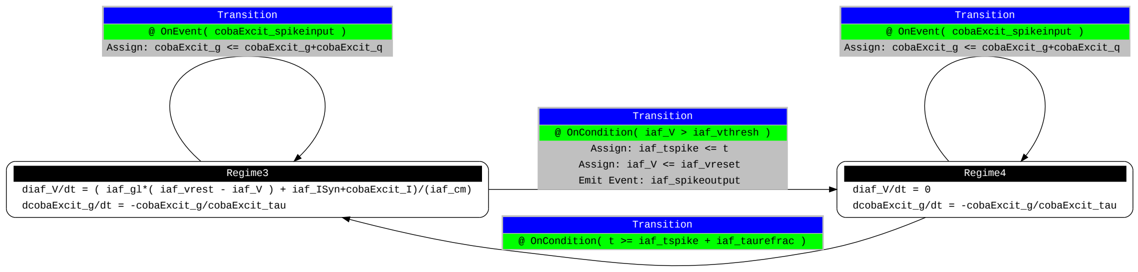
Transition
@ OnEvent(cobaExcit_spikeinput)
Assign: cobaExcit_g <= cobaExcit_g+cobaExcit_q
Transition
@ OnEvent(cobaExcit_spikeinput)
Assign: cobaExcit_g <= cobaExcit_g+cobaExcit_q
Transition
@ OnCondition(iaf_V > iaf_vthresh)
Assign: iaf_tspike <= t
Assign: iaf_V <= iaf_vreset
Emit Event: iaf_spikeoutput
Regime3
diaf_V/dt = (iaf_gl*(iaf_vrest - iaf_V) + iaf_ISyn+cobaExcit_I)/(iaf_cm)
dcobaExcit_g/dt = -cobaExcit_g/cobaExcit_tau
Regime4
diaf_V/dt = 0
dcobaExcit_g/dt = -cobaExcit_g/cobaExcit_tau
Transition
@ OnCondition(t >= iaf_tspike + iaf_taurefrac)

Fig. 7.3: RegimeGraph for the XML model in this section

```
Parameter:
- {name: cobaExcit_q, dimension: conductanceDensity}
- {name: cobaExcit_tau, dimension: time}
- {name: cobaExcit_vrev, dimension: voltage}
- {name: iaf_cm, dimension: capacitance}
- {name: iaf_gl, dimension: conductanceDensity}
- {name: iaf_taurefrac, dimension: time}
- {name: iaf_vreset, dimension: voltage}
- {name: iaf_vrest, dimension: voltage}
- {name: iaf_vthresh, dimension: voltage}
EventReceivePort:
- {name: cobaExcit_spikeinput}
AnalogReducePort:
- {name: iaf_ISyn, dimension: current, operator: +}
EventSendPort:
- {name: iaf_spikeoutput}
AnalogSendPort:
- {name: cobaExcit_I, dimension: current}
- {name: iaf_V, dimension: voltage}
Dynamics:
  StateVariable:
  - {name: cobaExcit_g, dimension: conductanceDensity}
  - {name: iaf_V, dimension: voltage}
  - {name: iaf_tspike, dimension: time}
  Regime:
  - name: RefractoryRegime
    TimeDerivative:
    - {variable: cobaExcit_g, MathInline: -cobaExcit_g/cobaExcit_tau}
    OnEvent:
    - port: cobaExcit_spikeinput
      target_regime: RefractoryRegime
      StateAssignment:
      - {variable: cobaExcit_g, MathInline: cobaExcit_g + cobaExcit_q}
    OnCondition:
    - Trigger: {MathInline: t > iaf_taurefrac + iaf_tspike}
      target_regime: RegularRegime
  - name: RegularRegime
    TimeDerivative:
    - {variable: cobaExcit_g, MathInline: -cobaExcit_g/cobaExcit_tau}
    - {variable: iaf_V, MathInline: (cobaExcit_I + iaf_ISyn + iaf_gl*(-iaf_V +
        iaf_vrest))/iaf_cm}
    OnEvent:
    - port: cobaExcit_spikeinput
      target_regime: RegularRegime
```

```
            StateAssignment:
            - {variable: cobaExcit_g, MathInline: cobaExcit_g + cobaExcit_q}
          OnCondition:
          - Trigger: {MathInline: iaf_V > iaf_vthresh}
            target_regime: RefractoryRegime
            StateAssignment:
            - {variable: iaf_V, MathInline: iaf_vreset}
            - {variable: iaf_tspike, MathInline: t}
            OutputEvent:
            - {port: iaf_spikeoutput}
        Alias:
        - {MathInline: cobaExcit_g*(cobaExcit_vrev - iaf_V), name: cobaExcit_I}
  Dimension:
  - {name: capacitance, m: -1, l: -2, t: 4, i: 2}
  - {name: conductanceDensity, m: -1, l: -2, t: 3, i: 2}
  - {name: time, t: 1}
  - {name: voltage, m: 1, l: 2, t: -3, i: -1}
```

The *User Layer* description for the above example is:

```
NineML:
  '@namespace': http://nineml.net/9ML/1.0
  Component:
    name: IaFCobaProperties
    Definition: {'@body': IafCoba}
    Property:
    - {name: cobaExcit_q, SingleValue: 1.0, units: uF_per_cm2}
    - {name: cobaExcit_tau, SingleValue: 2.0, units: ms}
    - {name: cobaExcit_vrev, SingleValue: 0.0, units: mV}
    - {name: iaf_cm, SingleValue: 0.02, units: nF}
    - {name: iaf_gl, SingleValue: 0.1, units: mS}
    - {name: iaf_taurefrac, SingleValue: 3.0, units: ms}
    - {name: iaf_vreset, SingleValue: -70.0, units: mV}
    - {name: iaf_vrest, SingleValue: -60.0, units: mV}
    - {name: iaf_vthresh, SingleValue: 20.0, units: mV}
  Unit:
  - {symbol: mS, dimension: conductanceDensity, power: -3}
  - {symbol: mV, dimension: voltage, power: -3}
  - {symbol: ms, dimension: time, power: -3}
  - {symbol: nF, dimension: capacitance, power: -9}
  Dimension:
  - {name: capacitance, m: -1, l: -2, t: 4, i: 2}
  - {name: conductanceDensity, m: -1, l: -2, t: 3, i: 2}
  - {name: time, t: 1}
  - {name: voltage, m: 1, l: 2, t: -3, i: -1}
```

The simulation results is presented in Figure 6.

7.2 Network Models

7.2.1 COBA IAF Network example

This example is an implementation of *Benchmark 1* from *[Brette2009]*, which consists of a network of an excitatory and inhibitory IAF populations randomly connected with COBA synapses *[Vogels2005]*. The excitatory and inhibitory *Population* elements are created with 3,200 and 800 cells respectively. Both populations are then concatenated into a

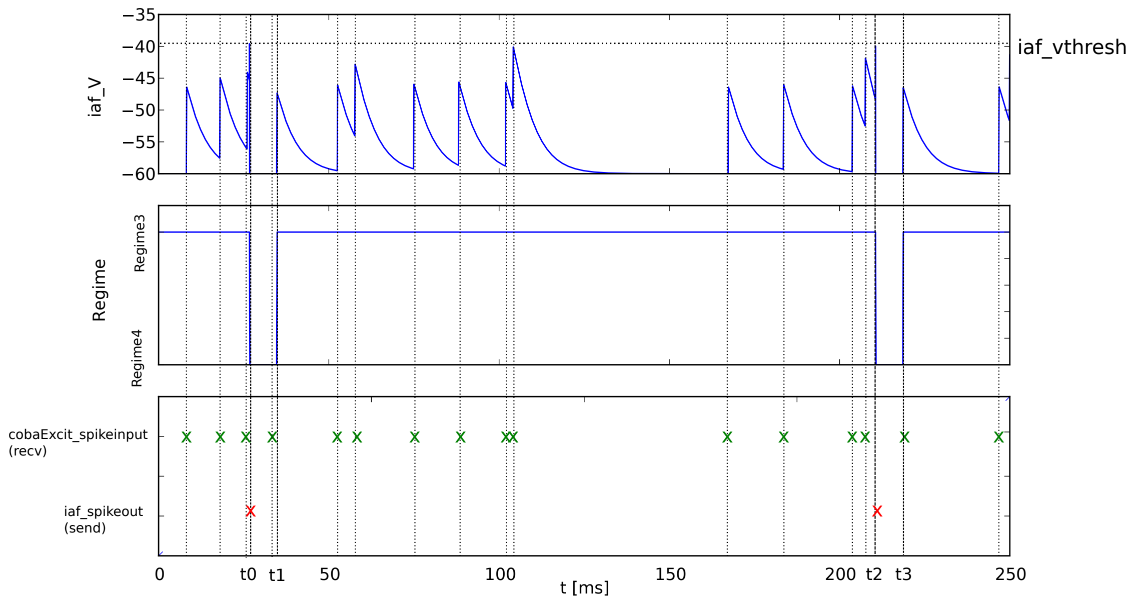

Fig. 7.4: Result of simulating of the XML model in this section. *cobaExcit_spikeinput* is fed events from an external Poisson generator in this simulation

single *Selection* element, “AllNeurons”, which is used to randomly connect both populations to every other neuron in the network with a 2% probability.

The abstraction layer description of the IAF input neuron ComponentClass is:

```
NineML:
  '@namespace': http://nineml.net/9ML/1.0
  ComponentClass:
  - name: IaF
  Parameter:
  - {name: iaf_cm, dimension: capacitance}
  - {name: iaf_gl, dimension: conductanceDensity}
  - {name: iaf_taurefrac, dimension: time}
  - {name: iaf_vreset, dimension: voltage}
  - {name: iaf_vrest, dimension: voltage}
  - {name: iaf_vthresh, dimension: voltage}
  AnalogReducePort:
  - {name: iaf_ISyn, dimension: current, operator: +}
  EventSendPort:
  - {name: iaf_spikeoutput}
  AnalogSendPort:
  - {name: iaf_V, dimension: voltage}
  Dynamics:
    StateVariable:
    - {name: iaf_V, dimension: voltage}
    - {name: iaf_tspike, dimension: time}
    Regime:
    - name: RefractoryRegime
      OnCondition:
      - Trigger: {MathInline: t > iaf_taurefrac + iaf_tspike}
        target_regime: RegularRegime
    - name: RegularRegime
      TimeDerivative:
      - {variable: iaf_V, MathInline: (iaf_ISyn + iaf_gl*(-iaf_V + iaf_vrest))/iaf_cm}
      OnCondition:
      - Trigger: {MathInline: iaf_V > iaf_vthresh}
        target_regime: RefractoryRegime
        StateAssignment:
        - {variable: iaf_V, MathInline: iaf_vreset}
        - {variable: iaf_tspike, MathInline: t}
        OutputEvent:
        - {port: iaf_spikeoutput}
```

The description of the COBA ComponentClass is:

```
NineML:
  '@namespace': http://nineml.net/9ML/1.0
  ComponentClass:
  - name: CoBa
    Parameter:
    - {name: coba_q, dimension: conductanceDensity}
    - {name: coba_tau, dimension: time}
    - {name: coba_vrev, dimension: voltage}
    EventReceivePort:
    - {name: coba_spikeinput}
    AnalogReceivePort:
    - {name: iaf_V, dimension: voltage}
    AnalogSendPort:
    - {name: coba_I, dimension: current}
    Dynamics:
      StateVariable:
      - {name: coba_g, dimension: conductanceDensity}
      Regime:
      - name: RegularRegime
        TimeDerivative:
        - {variable: coba_g, MathInline: -coba_g/coba_tau}
        OnEvent:
        - port: coba_spikeinput
          target_regime: RegularRegime
          StateAssignment:
          - {variable: coba_g, MathInline: coba_g + coba_q}
      Alias:
      - {MathInline: coba_g*(coba_vrev - iaf_V), name: coba_I}
```

The connection probability component class:

```
NineML:
  '@namespace': http://nineml.net/9ML/1.0
  - name: Probabilistic
    Parameter:
    - {name: probability, dimension: dimensionless}
    ConnectionRule: {standard_library: 'http://nineml.net/9ML/1.0/connectionrules/
↪Probabilistic'}
```

Note: More complex connection rules are planned for NineML v2.0

The cell *Component* are parameterized and connected together in the User Layer via *Population*, *Selection* and *Pro-*

jection elements:

```
NineML:
  '@namespace': http://nineml.net/9ML/1.0
  Component:
  - name: ExcConnectProb
    Definition: {'@body': Probabilistic}
    Property:
    - {name: probability, SingleValue: 0.02, units: unitless}
  - name: IaFProperties
    Definition: {'@body': IaF}
    Property:
    - {name: iaf_cm, SingleValue: 0.2, units: nF}
    - {name: iaf_gl, SingleValue: 0.05, units: mS}
    - {name: iaf_taurefrac, SingleValue: 5.0, units: ms}
    - {name: iaf_vreset, SingleValue: -60.0, units: mV}
    - {name: iaf_vrest, SingleValue: -60.0, units: mV}
    - {name: iaf_vthresh, SingleValue: -50.0, units: mV}
  - name: IaFSynapseExcitatory
    Definition: {'@body': CoBa}
    Property:
    - {name: coba_q, SingleValue: 0.004, units: uF_per_cm2}
    - {name: coba_tau, SingleValue: 5.0, units: ms}
    - {name: coba_vrev, SingleValue: 0.0, units: mV}
  - name: IaFSynapseInhibitory
    Definition: {'@body': CoBa}
    Property:
    - {name: coba_q, SingleValue: 0.051, units: uF_per_cm2}
    - {name: coba_tau, SingleValue: 5.0, units: ms}
    - {name: coba_vrev, SingleValue: -80.0, units: mV}
  - name: InhConnectProb
    Definition: {'@body': Probabilistic}
    Property:
    - {name: probability, SingleValue: 0.02, units: unitless}
  Population:
  - name: Excitatory
    Cell:
      Reference: {'@body': IaFProperties}
    Size: 3200
  - name: Inhibitory
    Cell:
      Reference: {'@body': IaFProperties}
    Size: 800
  Selection:
  - name: AllNeurons
    Concatenate:
      Item:
      - index: 0
        Reference: {'@body': Excitatory}
      - index: 1
        Reference: {'@body': Inhibitory}
  Projection:
  - name: Excitation
    Source:
      Reference: {'@body': Excitatory}
    Destination:
      Reference: {'@body': AllNeurons}
      FromResponse:
      - {send_port: coba_I, receive_port: iaf_ISyn}
    Connectivity:
      Reference: {'@body': ExcConnectProb}
    Response:
      Reference: {'@body': IaFSynapseExcitatory}
      FromSource:
      - {send_port: iaf_spikeoutput, receive_port: coba_spikeinput}
    Delay: {SingleValue: 1.5, units: ms}
  - name: Inhibition
    Source:
      Reference: {'@body': Inhibitory}
    Destination:
      Reference: {'@body': AllNeurons}
      FromResponse:
      - {send_port: coba_I, receive_port: iaf_ISyn}
    Connectivity:
      Reference: {'@body': InhConnectProb}
    Response:
      Reference: {'@body': IaFSynapseInhibitory}
      FromSource:
      - {send_port: iaf_spikeoutput, receive_port: coba_spikeinput}
    Delay: {SingleValue: 1.5, units: ms}
  Unit:
  - {symbol: mS, dimension: conductanceDensity, power: -3}
  - {symbol: mV, dimension: voltage, power: -3}
  - {symbol: nF, dimension: capacitance, power: -9}
  - {symbol: unitless, dimension: dimensionless, power: 0}
  Dimension:
  - {name: capacitance, m: -1, l: -2, t: 4, i: 2}
  - {name: conductanceDensity, m: -1, l: -2, t: 3, i: 2}
  - {name: dimensionless}
  - {name: time, t: 1}
  - {name: voltage, m: 1, l: 2, t: -3, i: -1}
```

CHAPTER 8

Acknowledgments

We would like to thank the former INCF NineML Task Force members for their contributions to the text and the concepts presented in this document. In particular: A. Gorchetchnikov, M. Hull, Y. Le Franc, P. Gleeson, E. Muller, R. Cannon, Birgit Kriener, Subhasis Ray and S. Hill.

Former NineML INCF Task Force members

- Robert Cannon
- Robert Clewley
- Alex Cope
- Hugo Cornelis
- Andrew P. Davison
- Erik De Schutter
- Mikael Djurfeldt
- Damien Drix
- Hans Ekkehard Plesser
- Padraig Gleeson
- Anatoli Gorchetchnikov
- Valentin Haenel
- Sean Hill
- Michael Hull
- Birgit Kriener
- Yann Le Franc
- Chung-Chua Lo
- Abigail Morrison

- Eilif Muller
- Dragan Nikolic
- Ivan Raikov
- Subhasis Ray
- Raphael Ritz
- Malin Sandström
- Lars Schwabe

Bibliography

[Davison2008] Davison, A.~P., Br"{u}derle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., and Yger, P. (2008). PyNN: A Common Interface for Neuronal Network Simulators. *Frontiers in neuroinformatics*, 2(January):11.

[Gleeson2010] Gleeson, P., Crook, S., Cannon, R.~C., Hines, M.~L., Billings, G.~O., Farinella, M., Morse, T.~M., Davison, A.~P., Ray, S., Bhalla, U.~S., Barnes, S.~R., Dimitrova, Y.~D., and Silver, R.~A. (2010). Neuroml: A language for describing data driven models of neurons and networks with a high degree of biological detail. *PLoS Comput Biol*, 6(6).

[Goddard2001] Goddard, N. and Hucka, M. (2001). Towards NeuroML: model description methods for collaborative modelling in neuroscience. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 356(1412):1209–28.

[Cannon2014] Cannon, R.~C., Gleeson, P., Crook, S., Ganapathy, G., Marin, B., Piasini, E., and Silver, R.~A. (2014). LEMS: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2. *Frontiers in neuroinformatics*, 8(September):79.

[Abbott1999] Abbott, L.~F. (1999). Lapicque's introduction of the integrate-and-fire model neuron (1907)}. *Brain Research Bulletin*, 50(99):303–304.

[Brette2009] Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., James, M., Diesmann, M., Morrison, A., Goodman, P.~H., Jr, F. C.~H., Zirpe, M., Natschl"{a}ger, T., Pecevski, D., Ermentrout, B., Djurfeldt, M., Lansner, A., Rochel, O., Vieville, T., Muller, E., Davison, A.~P., El, S., and Destexhe, A. (2009). Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of computational neuroscience*, 23(3):349–398.

[Izhikevich2003] Izhikevich, E.~M. and Izhikevich, E.~M. (2003). Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–72.

[Vogels2005] Vogels, T.~P. and Abbott, L.~F. (2005). Signal Propagation and Logic Gating in Networks of Integrate-and-Fire Neurons. *The Journal of Neuroscience*, 25(46):10786 –10795.