International Neuroinformatics Coordinating Facility

Network Interchange for Neuroscience Modeling Language (NineML)

Specification

NineML Standardization Committee

Version: 1.0

Editors:

- Alex Cope
- Andrew P. Davison
- Erik De Schutter
- Ivan Raikov
- Paul Richmond
- Thomas G. Close

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.

Date: March 2, 2015

Contents

1 Introduction 5
- 1.1 Scope . . . 5
- 1.2 Design considerations . . . 5
- 1.3 Identifiers . . . 6
- 1.4 Extensions . . . 6

I Abstraction Layer 7

2 Component Classes and Parameters 8
- 2.1 ComponentClass . . . 8
  - 2.1.1 Name attribute . . . 9
- 2.2 Parameter . . . 9
  - 2.2.1 Name attribute . . . 9
  - 2.2.2 Dimension attribute . . . 9

3 Units and Dimensions 10
- 3.1 Dimension . . . 10
  - 3.1.1 Name attribute . . . 10
  - 3.1.2 M attribute . . . 10
  - 3.1.3 L attribute . . . 10
  - 3.1.4 T attribute . . . 10
  - 3.1.5 I attribute . . . 11
  - 3.1.6 N attribute . . . 11
  - 3.1.7 K attribute . . . 11
  - 3.1.8 J attribute . . . 11
- 3.2 Unit . . . 11
  - 3.2.1 Symbol attribute . . . 11
  - 3.2.2 Dimension attribute . . . 11
  - 3.2.3 Power attribute . . . 11
  - 3.2.4 Offset attribute . . . 11

4 Mathematical Expressions 12
- 4.1 MathInline . . . 12
- 4.2 Alias . . . 13
  - 4.2.1 Name attribute . . . 14
- 4.3 Constant . . . 14
  - 4.3.1 Name attribute . . . 14
  - 4.3.2 Units attribute . . . 14
  - 4.3.3 Text format . . . 14

5 Ports 15
- 5.1 AnalogSendPort . . . 15
  - 5.1.1 Name attribute . . . 15
  - 5.1.2 Dimension attribute . . . 15
- 5.2 AnalogReceivePort . . . 15
  - 5.2.1 Name attribute . . . 16
  - 5.2.2 Dimension attribute . . . 16
- 5.3 AnalogReducePort . . . 16
  - 5.3.1 Name attribute . . . 16
  - 5.3.2 Dimension attribute . . . 16
  - 5.3.3 Operator attribute . . . 16
- 5.4 EventSendPort . . . 17
  - 5.4.1 Name attribute . . . 17
- 5.5 EventReceivePort . . . 17
  - 5.5.1 Name attribute . . . 17

6 Dynamic Regimes 18
- 6.1 Dynamics . . . 19
- 6.2 StateVariable . . . 19
  - 6.2.1 Name attribute . . . 19
  - 6.2.2 Dimension attribute . . . 19
- 6.3 Regime . . . 20
  - 6.3.1 Name attribute . . . 20
- 6.4 TimeDerivative . . . 20
  - 6.4.1 Variable attribute . . . 20

7 Transitions 21
7.1 OnCondition . . . 21
7.1.1 TargetRegime attribute . . . 21
7.2 OnEvent . . . 21
7.2.1 Port attribute . . . 22
7.2.2 TargetRegime attribute . . . 22
7.3 Trigger . . . 22
7.4 StateAssignment . . . 22
7.4.1 Variable attribute . . . 22
7.5 OutputEvent . . . 23
7.5.1 Port attribute . . . 23
8 Random Distributions 24
8.1 RandomDistribution . . . 24
8.1.1 StandardLibrary attribute . . . 25
9 Network Connectivity 26
9.1 ConnectionRule . . . 26
9.1.1 StandardLibrary attribute . . . 26

II User Layer 28
10 Components and Properties 29
10.1 Component . . . 29
10.1.1 Name attribute . . . 29
10.2 Definition . . . 29
10.2.1 Url attribute . . . 29
10.2.2 Text format . . . 29
10.3 Prototype . . . 30
10.3.1 Url attribute . . . 30
10.3.2 Text format . . . 30
10.4 Property . . . 30
10.4.1 Name attribute . . . 30
10.4.2 Units attribute . . . 30
10.5 Reference . . . 31
10.5.1 Url attribute . . . 31
10.5.2 Text format . . . 31
11 Values 32
11.1 SingleValue . . . 32
11.1.1 Text format . . . 32
11.2 ArrayValue . . . 32
11.3 ArrayValueRow . . . 32
11.3.1 Index attribute . . . 33
11.3.2 Text format . . . 33
11.4 ExternalArrayValue . . . 33
11.4.1 Url attribute . . . 33
11.4.2 MimeType attribute . . . 33
11.4.3 ColumnName attribute . . . 33
11.5 RandomValue . . . 34
11.5.1 Port attribute . . . 34
12 Populations 35
12.1 Population . . . 35
12.1.1 Name attribute . . . 35
12.2 Cell . . . 35
12.3 Number . . . 35
12.3.1 Text format . . . 35
13 Projections 36
13.1 Projection . . . 36
13.1.1 Name attribute . . . 36
13.2 Connectivity . . . 36
13.3 Source . . . 37
13.4 Destination . . . 37
13.5 Response . . . 37
13.6 Plasticity . . . 38

13.7 FromSource . . . 38
13.7.1 Sender attribute . . . 38
13.7.2 Receiver attribute . . . 38
13.8 FromDestination . . . 39
13.8.1 Sender attribute . . . 39
13.8.2 Receiver attribute . . . 39
13.9 FromPlasticity . . . 39
13.9.1 Sender attribute . . . 39
13.9.2 Receiver attribute . . . 39
13.10 FromResponse . . . 40
13.10.1 Sender attribute . . . 40
13.10.2 Receiver attribute . . . 40
13.11 Delay . . . 40
13.11.1 Units attribute . . . 40
14 Selections: combining populations and subsets **41**
14.1 Selection . . . 41
14.2 Concatenate . . . 41
14.3 Item . . . 41
14.3.1 Index attribute . . . 41

III Annotations **42**
15 Annotations **43**
15.1 Annotations . . . 43

Appendix **44**
A Examples **45**
A.1 Izhikevich Model . . . 45
A.2 Leaky Integrate and Fire model . . . 48
B Transition resolution **53**
B.1 Serial implementation of transition resolution . . . 53
B.1.1 Algorithm . . . 53
B.1.2 Notes . . . 54
B.2 Parallelising of event resolution . . . 54
C Acknowledgments **56**
C.1 Former NineML INCF Task Force members . . . 56
References **57**

1 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 to facilitate the exchange neuronal network models between researchers and simulator platforms.

This description language is based on a common object model describing the different elements of network models. This work, initiated and supported by the International Neuroinformatics Coordinating Facility as part of the Multiscale Modeling Program, involve computational neuroscientists, simulator developers and developers of simulator-independent languages (NeuroML, PyNN). The name of the proposed language is NineML (Network Interchange for Neuroscience Modeling Language).

1.1 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.

As of version 1.0, 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

1.2 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 Section 15). 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 the NineML XML schema, or use code generation to produce a model representation for a target simulation environment. It is important to note that the NineML XML schema is isomorphic to the NineML object model.

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 either one). 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.

1.3 Identifiers

Elements are identified by *names*, which are unique in the scope they are enclosed by (either within a component class or in the global 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-sensitive) as one of the built-in symbols or functions (see **MathInline**).

1.4 Extensions

"Extensions" increase the descriptive power of NineML by providing syntax for compact, high-level descriptions of models that can also be expressed in "Core" NineML (i.e. syntactic sugar). Envisaged examples include concise forms for kinetic equations and multi-component/compartment dynamic models.

Core NineML is everything defined by this document, except those parts specifically described as Extensions. To be classified as a NineML Extension, a modelling language must meet the following requirements.

- Models written in the Extension format (or part thereof) must be collapsible to Core NineML without change in behaviour.
- The flattened (to Core NineML) descriptions must be convertible back to their original form (the additional information required to perform the reverse conversion can be stored in **Annotations**).
- Stand-alone tools that perform the two-way conversion between the extended and core forms (preferably written in commonly used languages such as XSLT, Python, Java, etc...) must be downloadable from a publicly available URL (a section of the NineML website, http://nineml.net, is available for this purpose).
- Non-standard extensions must be defined within a unique namespace
- Detailed specifications, preferably in the same format as this document (see the 'ninemlspec.cls' latex class), must be made publicly available along with the conversion tools.

One of the rationales behind the division between core and extended NineML is to minimise the set of features that a tool needs to support to be NineML compliant, i.e. as long as the core is supported all extensions should also be supported via flattening to core NineML. However, tool builders may want to take advantage of the additional structure provided by the extensions to optimise the implementation of the model or provide more intuitive interfaces to the user.

> **Note:** To facilitate the reverse conversion from core to extended formats, NineML compliant tools must preserve all annotations during reading, writing and transformation, with the exception of when the return conversion is no longer possible due to the applied transformation or the annotations are explicitly added/deleted by the user.

Part I

Abstraction Layer

2 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 Figure 1).

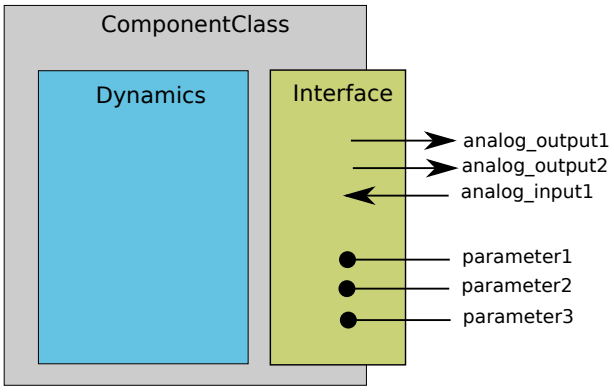

Figure 1: *ComponentClass Overview*

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.

2.1 ComponentClass

ComponentClass Structure		
Attribute name	*Type/Format*	*Required*
name	*identifier*	yes
Element type	*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

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.

2.1.1 Name attribute

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

2.2 Parameter

On reading through the old specs it is pretty clear that requiring dimensions for parameters and properties, including dimensionless ones, was deemed pretty important. I have also realised that it makes them easier to handle as you don't have to test for the case when they are not there.

Parameter Structure

Attribute name	*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, Parameters are set at the start of the simulation, and remain constant throughout.

2.2.1 Name attribute

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

2.2.2 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 global scope. For a dimensionless parameters a **Dimension** with all attributes of power 0 can be used.

3 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) (Cannon et al., 2014).

3.1 Dimension

Dimension Structure

Attribute name	*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 $l\,t^{-2}$ and voltage is $m l^2 t^3 i^{-1}$. **Dimension** objects must be declared in the top-level scope of the NineML document where they are referenced.

3.1.1 Name attribute

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

3.1.2 M attribute

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

3.1.3 L attribute

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

3.1.4 T attribute

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

3.1.5 I attribute

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

3.1.6 N attribute

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

3.1.7 K attribute

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

3.1.8 J attribute

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

3.2 Unit

Unit Structure

Attribute name	*Type/Format*	*Required*
symbol	*identifier*	yes
dimension	**Dimension**@name	yes
power	`integer`	no
offset	`real number`	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.

3.2.1 Symbol attribute

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

3.2.2 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 global scope.

3.2.3 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.

3.2.4 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.

4 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.

4.1 MathInline

MathInline Structure	
Description of text	*Required*
Inline mathematical 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`.

I have added exponent operator since it is commonly used in neutron models and is quite convenient but it is not ANSI C89 so I am not sure whether you guys think this is a good idea

The following operators are supported in inline maths expressions,

- Arithmetic operators
  - Addition +
  - Subtraction -
  - Division /
  - Multiplication *
  - Exponent ^

  The '+', '-', '/' and '*' operators have the same interpretation and precedence levels as in the ANSI C89 standard. '^', which is not part of the ANSI C89 standard, is the exponent operator e.g.

  ```
  a^3 = a*a*a
  ```

  and has the highest level of precedence.
- Inequality operators
  - Greater than >
  - Lesser than <
- Logical operators
  - 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)`
- `ceil(x)`
- `floor(x)`

*I removed the "random" namespace since it was proving difficult to parse and it is made redundant by adding the **RandomDistribution** elements to standard?*

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.

4.2 Alias

Alias Structure		
Attribute name	*Type/Format*	*Required*
name	*identifier*	yes
Element type	*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 $i = g * (E - V)$.

4.2.1 Name attribute

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

Was tossing up whether 'Constant' was a better name for this or not. Was thinking that this could also be used to remove 'pi' from the list of built-in symbols and maybe provide some 'standardLibrary' attributes.

4.3 Constant

Constant Structure		
Attribute name	*Type/Format*	*Required*
name	*identifier*	yes
units	**Unit** @name	yes
Text format		*Required*
`real number`		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. $6.0221412927 \times 10^{23}\mathrm{mol}^{-1}$).

4.3.1 Name attribute

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

4.3.2 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 global scope.

4.3.3 Text format

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

5 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 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.

5.1 AnalogSendPort

AnalogSendPort Structure		
Attribute name	*Type/Format*	*Required*
name	[**StateVariable** | **Alias**]@name	yes
dimension	**Dimension**@name	yes

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

5.1.1 Name attribute

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

5.1.2 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 global scope.

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

5.2 AnalogReceivePort

AnalogReceivePort Structure		
Attribute name	*Type/Format*	*Required*
name	*identifier*	yes
dimension	**Dimension**@name	yes

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

5.2.1 Name attribute

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

5.2.2 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 global scope.

5.3 AnalogReducePort

AnalogReducePort Structure

Attribute name	*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.

5.3.1 Name attribute

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

5.3.2 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 global scope.

5.3.3 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="synaptic_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.

5.4 EventSendPort

EventSendPort Structure		
Attribute name	*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.

5.4.1 Name attribute

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

5.5 EventReceivePort

EventReceivePort Structure		
Attribute name	*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**.

5.5.1 Name attribute

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

6 Dynamic Regimes

Component classes that contain a **Dynamics** block define dynamic models such as neurons, post-synaptic responses or the plasticity of synaptic weights. In these components, state variables are evolved by sets of ordinary differential equations (ODE), which describe the behaviour of the model. The state of the model can transition between distinct regimes, which define different sets of differential equations, via 'transitions'. Multiple regimes can be used to model qualitatively distinct behaviour by neuron models, such as sub-threshold, spiking and refractory periods of an abstract neuron model for example.

Figure 2 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.

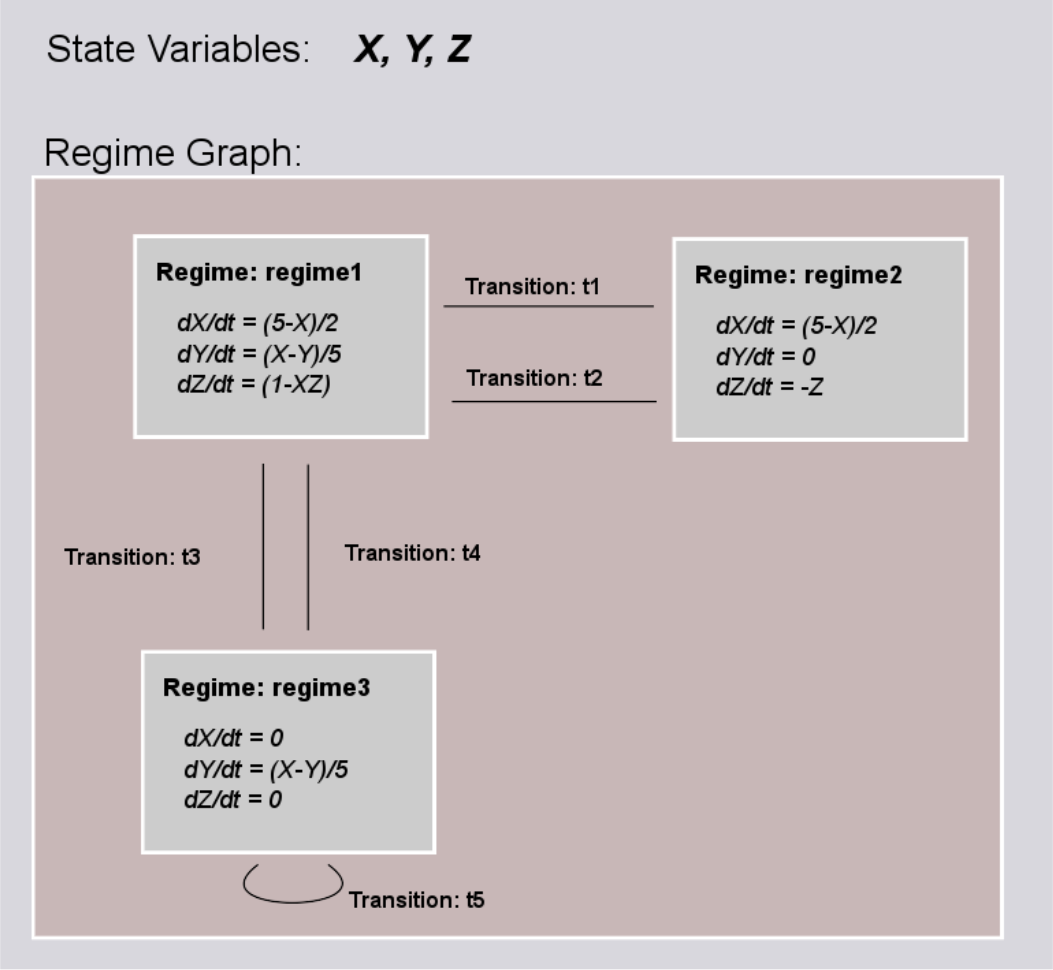

Figure 2: *The dynamics block for an example component.*

6.1 Dynamics

Dynamics Structure		
Element type	*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. Figure 2) 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.

6.2 StateVariable

StateVariable Structure		
Attribute name	*Type/Format*	*Required*
name	*identifier*	yes
dimension	**Dimension**@name	yes

The internal state of a component is defined by a set of state variables – variables that can change either continuously or discontinuously as a function of time.

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. $X = X + 1$.

6.2.1 Name attribute

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

6.2.2 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 global scope.

6.3 Regime

Regime Structure		
Attribute name	*Type/Format*	*Required*
name	*identifier*	yes
Element type	*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. **Regime** is must have non-vanishing temporal extent. Once construction of the **Regime** is complete, it should have defined the following properties:

6.3.1 Name attribute

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

6.4 TimeDerivative

TimeDerivative Structure		
Attribute name	*Type/Format*	*Required*
variable	**StateVariable**@name	yes
Element type	*Multiplicity*	*Required*
MathInline	singleton	yes

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

$$\frac{\mathrm{d}variable}{\mathrm{d}t} = expression \tag{1}$$

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.

6.4.1 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**.

7 Transitions

Movements between dynamic regimes occurs via transitions, and have vanishing temporal extents (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.

*Should **OnEvent** transitions be able to emit new events or would this be unnecessarily complex to handle?*

During either type of transition three instantaneous actions will/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**).

Multiple state assignments can be sent and multiple events can be sent within a single transition block (for more on the resolution of transitions see Appendix Section B).

7.1 OnCondition

OnCondition Structure		
Attribute name	*Type/Format*	*Required*
targetRegime	**Regime**@name	no
Element type	*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.

7.1.1 TargetRegime attribute

An **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 trigger condition is met. If the *targetRegime* attribute is omitted the regime will transition to itself.

7.2 OnEvent

OnEvent Structure		
Attribute name	*Type/Format*	*Required*
targetRegime	**Regime**@name	no
port	**EventReceivePort**@name	yes
Element type	*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.

7.2.1 Port attribute

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

7.2.2 TargetRegime 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.

7.3 Trigger

Trigger Structure		
Element type	*Multiplicity*	*Required*
MathInline	singleton	yes

Triggers define when an **OnCondition** transition should be 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 '<'). 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.

7.4 StateAssignment

StateAssignment Structure		
Attribute name	*Type/Format*	*Required*
variable	**StateVariable**@name	yes
Element type	*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.

7.4.1 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.

7.5 OutputEvent

OutputEvent Structure		
Attribute name	*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.

7.5.1 Port attribute

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

8 Random Distributions

8.1 RandomDistribution

RandomDistribution Structure		
Attribute name	*Type/Format*	*Required*
standardLibrary	URL	yes

As of version 1.0, the only random distributions available to the user are those defined in the standard library. 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
- InverseGammaDistribution
- LaplaceDistribution
- LogisticDistribution
- LogNormalDistribution
- MixtureModel
- MultinomialDistribution
- NegativeBinomialDistribution
- NormalDistribution
- NormalInverseGammaDistribution
- ParetoDistribution
- PoissonDistribution

- StudentTDistribution
- UniformDistribution
- WeibullDistribution

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

8.1.1 StandardLibrary attribute

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

9 Network Connectivity

9.1 ConnectionRule

ConnectionRule Structure		
Attribute name	*Type/Format*	*Required*
standardLibrary	URL	yes

In Version 1.0 of the NineML standard, connection rules must be one of 6 standard library types (in future versions, built-in connectivity rules are to be replaced with arbitrary connection rule generators):

- AllToAll
- OneToOne
- ProbabilisticConnectivity
- ExplicitConnectionList
- RandomFanOut
- RandomFanIn

9.1.1 StandardLibrary attribute

The *standardLibrary* attribute is required and should point to a URL in the http://nineml.net/standardlibraries/connectionrules/ directory.

http://nineml.net/standardlibraries/connectionrules/AllToAll

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

http://nineml.net/standardlibraries/connectionrules/OneToOne

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.

http://nineml.net/standardlibraries/connectionrules/ProbabilisticConnectivity

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. In the case of the **ArrayValue** or **ExternalArrayValue** properties, the list represents probabilities of connections existing between the first cell in the source population with every cell in the destination population in order of their index, followed by the second cell in the source with with every cell in the destination population in order of their index and so forth for each cell in the source population.

http://nineml.net/standardlibraries/connectionrules/ExplicitConnectionList

Cells 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.

http://nineml.net/standardlibraries/connectionrules/RandomFanOut

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 property supplied to the *probability* parameter should be a **SingleValue**.

http://nineml.net/standardlibraries/connectionrules/RandomFanIn

Cells in the destination population are connected to cells in the source population with a probability defined by a parameter, which should be named *probability*. The property supplied to the *probability* parameter should be a **SingleValue**.

Part II

User Layer

10 Components and Properties

10.1 Component

Component Structure		
Attribute name	*Type/Format*	*Required*
name	*identifier*	yes
Element type	*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.

10.1.1 Name attribute

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

10.2 Definition

Definition Structure		
Attribute name	*Type/Format*	*Required*
url	URL	no
Text 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.

10.2.1 Url attribute

If the **ComponentClass** referenced by the definition element is defined outside the current document, the *url* attribute specifies a URL for the file which contains the **ComponentClass** definition. If it is omitted the **ComponentClass** is assumed to be in the current document.

10.2.2 Text format

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

10.3 Prototype

Prototype Structure		
Attribute name	*Type/Format*	*Required*
url	URL	no
Text 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.

10.3.1 Url attribute

If the prototype **Component** is defined outside the current file, the *URL* attribute specifies a URL for the file which contains the prototype **Component**.

10.3.2 Text format

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

*I have removed the **Quantity** s in favour of adding units to the **Property** elements as it seemed superfluous and incongruous with the AL (i.e. **Parameter** s define the dimension). I will create a GitHub issue about this where we can discuss this and potentially decide to change it back if you like*

10.4 Property

Property Structure		
Attribute name	*Type/Format*	*Required*
name	**Parameter**@name	yes
units	**Unit**@symbol	yes
Element type	*Multiplicity*	*Required*
SingleValue | **ArrayValue** | **ExternalArrayValue** | **RandomValue**	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.

10.4.1 Name attribute

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

10.4.2 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 global scope. For a dimensionless units a **Unit** >**Dimension** with no SI dimensions can be used. The SI dimensions of the **Unit** >**Dimension** should match the SI dimensions of the corresponding **Parameter** >**Dimension**.

10.5 Reference

Reference Structure		
Attribute name	*Type/Format*	*Required*
url	URL	no
Text format		*Required*
*@name		yes

Reference elements are used to locate User Layer elements in the global 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 global 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).

10.5.1 Url attribute

The *url* attribute specifies a URL for the file which contains the User Layer element to be referenced.

10.5.2 Text format

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

11 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.
- **RandomValue**, a return value from a **RandomDistribution** component.

11.1 SingleValue

SingleValue Structure	
Text format	*Required*
`real number`	yes

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

11.1.1 Text format

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

11.2 ArrayValue

ArrayValue Structure		
Element type	*Multiplicity*	*Required*
ArrayValueRow	set	no

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

11.3 ArrayValueRow

ArrayValueRow Structure		
Attribute name	*Type/Format*	*Required*
index	integer	yes
Text format		*Required*
`real number`		yes

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

11.3.1 Index attribute

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

> **Note:** The order of **ArrayValueRow** 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 (see Section 1.2).

11.3.2 Text format

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

11.4 ExternalArrayValue

ExternalArrayValue Structure		
Attribute name	*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 in XML (which is slow to parse) 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.

11.4.1 Url attribute

The *url* attribute specifies the URL of the external data file.

11.4.2 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.

11.4.3 ColumnName attribute

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

What do you think of this? This allows a single component to return two (related) values at different places at the model (e.g. weight and delay in distance dependent rules). It also seemed a bit neater

11.5 RandomValue

RandomValue Structure		
Attribute name	*Type/Format*	*Required*
port	**AnalogSendPort**@name	yes
Element type	*Multiplicity*	*Required*
Component>**RandomDistribution** | **Reference**→**Component**>**RandomDistribution**	singleton	yes

The array of values represented by a **RandomValue** element is populated from the output port of a component. As of version 1.0, only **RandomDistribution** components are supported within **RandomValue** but function and generator components are planned for later versions. The size of the generated array is determined by the size of the container (i.e. **Population** or **Projection**) the **RandomValue** is nested within.

11.5.1 Port attribute

Each **RandomValue** requires a *port* attribute, which selects which port of the **Component** to read.

12 Populations

12.1 Population

Population Structure		
Attribute name	*Type/Format*	*Required*
name	*identifier*	yes
Element type	*Multiplicity*	*Required*
Number	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 **Number** 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 Section 11).

12.1.1 Name attribute

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

12.2 Cell

Cell Structure		
Element type	*Multiplicity*	*Required*
Component>**Dynamics** | **Reference**→**Component**>**Dynamics**	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.

12.3 Number

Number Structure	
Text format	*Required*
`integer`	yes

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

12.3.1 Text format

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

13 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 and connected to and from explicitly defined ports of the cell components in the source and projection populations if the connection rule determines there is a connection between a particular source and destination cell pair.

I just drafted this so it will need checking

SingleValue and **RandomValue** elements used in properties of a projection (in the **Connectivity**, **Response**, **Plasticity** and **Delay** elements) take the size of the number of connections made. While **ArrayValue** and **ExternalArrayValue** elements can either be the size of the number of connections made or the number of possible unique connections (i.e. $N_{\text{source}} * N_{\text{dest}}$). In either case, the expected array data is ordered by the indices

$$i_{\text{value}} = i_{\text{source}} * N_{\text{dest}} + i_{\text{dest}} \tag{2}$$

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. For the case of the explicit arrays being the size of the number connections made, data for i_{value} indices that do not correspond to connected pairs are omitted.

13.1 Projection

Projection Structure		
Attribute name	*Type/Format*	*Required*
name	*identifier*	yes
Element type	*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.

13.1.1 Name attribute

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

13.2 Connectivity

Connectivity Structure		
Element type	*Multiplicity*	*Required*
Component>ConnectionRule	singleton	yes

The **Connectivity** element contains a **Component** >**ConnectionRule**, which defines how the source and destination populations are connected together.

13.3 Source

Source Structure		
Element type	*Multiplicity*	*Required*
Reference→**Population** | **Reference**→**Selection**	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** >**ConnectionRule** determines that the source and destination cells should be connected.

13.4 Destination

Destination Structure		
Element type	*Multiplicity*	*Required*
Reference→**Population** | **Reference**→**Selection**	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** >**ConnectionRule** determines that the source and destination cells should be connected.

13.5 Response

Response Structure		
Element type	*Multiplicity*	*Required*
Component>**Dynamics** | **Reference**→**Component**>**Dynamics**	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** >**Dynamics** 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** >**ConnectionRule** determines that the source and destination cells should be connected.

13.6 Plasticity

Plasticity Structure		
Element type	*Multiplicity*	*Required*
Component>**Dynamics** | **Reference**→**Component**>**Dynamics**	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 Section 13.5). 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** >**ConnectionRule** determines that the source and destination cells should be connected.

13.7 FromSource

FromSource Structure		
Attribute name	*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.

13.7.1 Sender attribute

Each **FromSource** element requires a *sender* attribute. This should refer to the name of a **AnalogSendPort** or **EventSendPort** in the **Cell**>**Component**>**ComponentClass** 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.

13.7.2 Receiver attribute

Each **FromSource** element requires a *receiver* attribute. This should refer to the name of a **AnalogReceivePort**, **EventReceivePort** or **AnalogReducePort** in the **Component**>**ComponentClass** 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.

13.8 FromDestination

FromDestination Structure		
Attribute name	*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.

13.8.1 Sender attribute

Each **FromDestination** element requires a *sender* attribute. This should refer to the name of a **AnalogSendPort** or **EventSendPort** in the **Cell** >**Component**>**ComponentClass** 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.

13.8.2 Receiver attribute

Each **FromDestination** element requires a *receiver* attribute. This should refer to the name of a **AnalogReceivePort**, **EventReceivePort** or **AnalogReducePort** in the **Component**>**ComponentClass** 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.

13.9 FromPlasticity

FromPlasticity Structure		
Attribute name	*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.

13.9.1 Sender attribute

Each **FromPlasticity** element requires a *sender* attribute. This should refer to the name of a **AnalogSendPort** or **EventSendPort** in the **Cell**->**Component**>**ComponentClass** 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.

13.9.2 Receiver attribute

Each **FromPlasticity** element requires a *receiver* attribute. This should refer to the name of a **AnalogReceivePort**, **EventReceivePort** or **AnalogReducePort** in the **Component**>**ComponentClass** 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.

13.10 FromResponse

FromResponse Structure		
Attribute name	*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.

13.10.1 Sender attribute

Each **FromResponse** element requires a *sender* attribute. This should refer to the name of a **AnalogSendPort** or **EventSendPort** in the **Cell**->**Component**>**ComponentClass** 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.

13.10.2 Receiver attribute

Each **FromResponse** element requires a *receiver* attribute. This should refer to the name of a **AnalogReceivePort**, **EventReceivePort** or **AnalogReducePort** in the **Component**>**ComponentClass** 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.

13.11 Delay

Delay Structure		
Attribute name	*Type/Format*	*Required*
units	**Unit**@symbol	yes
Element type	*Multiplicity*	*Required*
SingleValue | **ArrayValue** | **ExternalArrayValue** | **RandomValue**	singleton	yes

In version 1.0, the **Delay** element specifies the delay between the pre-synaptic cell port and both the **Plasticity**>**Component** and **Response**>**Component**. 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.

13.11.1 Units attribute

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

14 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.

14.1 Selection

Selection Structure		
Element type	*Multiplicity*	*Required*
Concatenate	singleton	yes

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

14.2 Concatenate

Concatenate Structure		
Element type	*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. $i = 0, \ldots, N-1$).

14.3 Item

Item Structure		
Attribute name	*Type/Format*	*Required*
index	`integer`	yes
Element type	Multiplicity	*Required*
Reference→[**Population** | **Selection**]	singleton	yes

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

14.3.1 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

Part III

Annotations

15 Annotations

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 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.

15.1 Annotations

Annotations Structure

Element type	*Multiplicity*	*Required*
*	set	no

The **Annotations** element is the top-level of the annotations attached to a NineML element. They can be included within any NineML element (User Layer and Abstraction Layer) and any valid XML is allowed within them.

Appendix

A Examples

A.1 Izhikevich Model

In this first example, we are describing how to represent the Izhikevich model in NineML. 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:

$$\frac{dV}{dt} = 0.04 * V * V + 5 * V + 140.0 - U + i_{\text{synapse}} + i_{\text{injected}} \tag{3}$$

$$\frac{dU}{dt} = a * (b * V - U) \tag{4}$$

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:

$$U \leftarrow U + d \tag{5}$$

$$V \leftarrow c \tag{6}$$

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 Figure 3

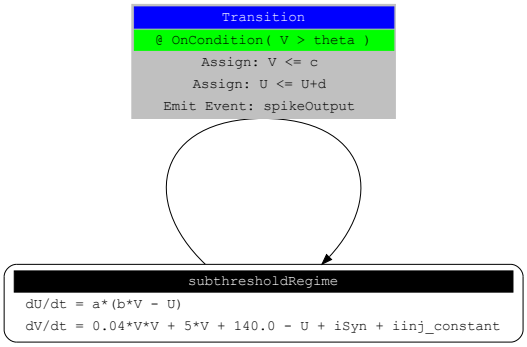

Figure 3: `RegimeGraph` *for the XML model in this section.*

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 Figure 4.

In Figure Figure 4, 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 XML description for this model is the following:

```
<?xml version="1.0" encoding='UTF-8'?>
<NineML xmlns="http://nineml.net/9ML/1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://nineml.net/9ML/1.0/NineML_v1.0.xsd">
  <ComponentClass name="IzhikevichCell">
    <Parameter name="a" dimension="dimensionless"/>
    <Parameter name="c" dimension="voltage"/>
    <Parameter name="b" dimension="per_time"/>
    <Parameter name="d" dimension="voltage_per_time"/>
    <Parameter name="theta" dimension="voltage"/>
    <Parameter name="i_injected" dimension="current"/>
    <AnalogReducePort name="i_synapse" operator="+" dimension="current"/>
    <AnalogSendPort name="U" dimension="dimensionless"/>
    <AnalogSendPort name="V" dimension="voltage"/>
    <EventPort name="spikeOutput" mode="send"/>
    <Dynamics>
        <StateVariable name="V" dimension="voltage"/>
        <StateVariable name="U" dimension="voltage_per_time"/>
        <Alias name="rv">
            <MathInline>V*U</MathInline>
        </Alias>
        <Regime name="subthresholdRegime">
          <TimeDerivative variable="U">
            <MathInline>a*(b*V - U)</MathInline>
          </TimeDerivative>
          <TimeDerivative variable="V">
            <MathInline>0.04*V*V + 5*V + 140.0 - U + i_synapse + i_injected</MathInline>
          </TimeDerivative>
          <OnCondition>
            <Trigger>
              <MathInline>V &gt; theta </MathInline>
            </Trigger>
            <StateAssignment variable="V" >
              <MathInline>c</MathInline>
            </StateAssignment>
            <StateAssignment variable="U" >
              <MathInline>U+d</MathInline>
            </StateAssignment>
            <OutputEvent port="spikeOutput" />
          </OnCondition>
        </Regime>
    </Dynamics>
  </ComponentClass>
  <Dimension name="per_time" t="-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"/>
  <Dimension name="current" i="1"/>
  <Dimension name="dimensionless"/>
</NineML>
```

User Layer description for the above example:

```
<?xml version='1.0' encoding='UTF-8'?>
<NineML xmlns="http://nineml.net/9ML/1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://nineml.net/9ML/1.0/NineML_v1.0.xsd">
  <Component name="IzhikevichNeuron">
    <Definition url="http://nineml.net/catalog/izhikevichCell.9ml"
      >IzhikevichCell</Definition>
    <Property name="V" units="mV">
      <SingleValue>-60</SingleValue>
    </Property>
    <Property name="U" units="mV_per_ms">
      <SingleValue>0</SingleValue>
    </Property>
    <Property name="theta" units="mV">
      <SingleValue>50</SingleValue>
    </Property>
    <Property name="a" units="none">
      <SingleValue>0.02</SingleValue>
    </Property>
    <Property name="b" units="per_ms">
      <SingleValue>0.2</SingleValue>
    </Property>
    <Property name="c" units="mV">
      <SingleValue>-65</SingleValue>
    </Property>
    <Property name="d" units="mV_per_ms">
      <SingleValue>8</SingleValue>
    </Property>
  </Component>
  <Dimension name="per_time" t="-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"/>
  <Dimension name="current" i="1"/>
  <Dimension name="dimensionless"/>
  <Unit symbol="mV" dimension="voltage" power="-3" />
  <Unit symbol="per_ms" dimension="per_time" power="-3" />
  <Unit symbol="mV_per_ms" dimension="voltage_per_time" />
  <Unit symbol="none" dimension="dimensionless" />
</NineML>
```

Here, we show the simulation results of this XML representation.

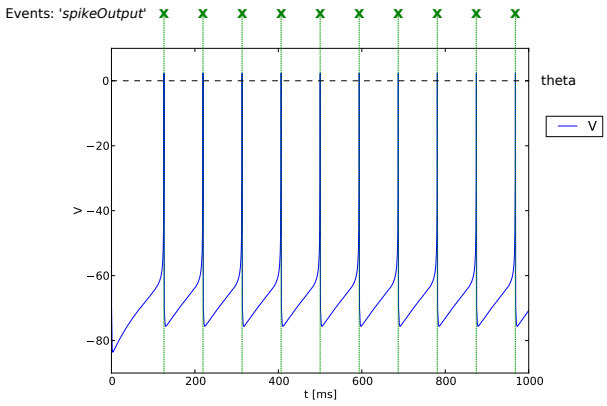

Figure 4: *Result of simulating of the XML model in this section*

A.2 Leaky Integrate and Fire model

In this example, we build a representation of a integrate-and-fire neuron, with an attached input synapse. 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:

$$\frac{d(iaf\_V)}{dt} = \frac{iaf\_gl * (iaf\_vrest - iaf\_V) + iaf\_ISyn + cobaExcit\_I}{iaf\_cm} \tag{7}$$

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

$$\frac{d(iaf\_V)}{dt} = 0 \tag{8}$$

In both Regimes, the synapses dynamics evolve as:

$$\frac{d(cobaExcit\_g)}{dt} = -\frac{cobaExcit\_g}{cobaExcit\_tau} \tag{9}$$

The neuron has 2 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 **OnCondition**s:

- 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.

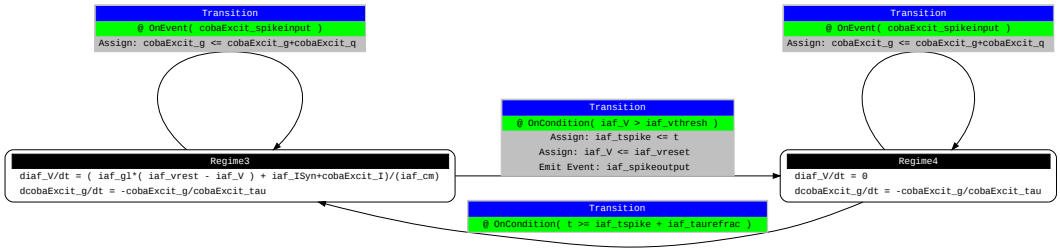

Figure 5: *RegimeGraph for the XML model in this section*

The resulting XML description for the Abstraction Layer is :

```
<?xml version='1.0' encoding='UTF-8'?>
<NineML xmlns="http://nineml.net/9ML/1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://nineml.net/9ML/1.0/NineML_v1.0.xsd">
  <ComponentClass name="Iaf1Coba">
    <AnalogSendPort dimension="voltage" name="iaf_V" />
    <AnalogReducePort dimension="current" operator="+" name="iaf_ISyn" />
    <AnalogSendPort dimension="current" name="cobaExcit_I" />
    <EventSendPort name="iaf_spikeoutput"/>
    <EventReceivePort name="cobaExcit_spikeinput"/>
    <Parameter dimension="area" name="iaf_cm"/>
    <Parameter dimension="time" name="iaf_taurefrac"/>
    <Parameter dimension="conductance" name="iaf_gl"/>
    <Parameter dimension="voltage" name="iaf_vreset"/>
    <Parameter dimension="voltage" name="iaf_vrest"/>
    <Parameter dimension="voltage" name="iaf_vthresh"/>
    <Parameter dimension="time" name="cobaExcit_tau"/>
    <Parameter dimension="conductance" name="cobaExcit_q"/>
    <Parameter dimension="voltage" name="cobaExcit_vrev"/>
    <Dynamics>
      <StateVariable dimension="voltage" name="iaf_V"/>
      <StateVariable dimension="time" name="iaf_tspike"/>
      <StateVariable dimension="conductance" name="cobaExcit_g"/>
      <Regime name="RefractoryRegime">
        <TimeDerivative variable="iaf_V">
          <MathInline>0</MathInline>
        </TimeDerivative>
        <TimeDerivative variable="cobaExcit_g">
          <MathInline>-cobaExcit_g/cobaExcit_tau</MathInline>
        </TimeDerivative>
        <OnEvent target_regime="RefractoryRegime" src_port="cobaExcit_spikeinput">
          <StateAssignment variable="cobaExcit_g">
            <MathInline>cobaExcit_g+cobaExcit_q</MathInline>
          </StateAssignment>
        </OnEvent>
        <OnCondition target_regime="RegularRegime">
          <Trigger>
            <MathInline>t &gt;= iaf_tspike + iaf_taurefrac</MathInline>
          </Trigger>
        </OnCondition>
      </Regime>
      <Regime name="RegularRegime">
        <TimeDerivative variable="iaf_V">
          <MathInline>( iaf_gl*( iaf_vrest - iaf_V ) + iaf_ISyn+cobaExcit_I)/(iaf_cm)</MathInline>
        </TimeDerivative>
        <TimeDerivative variable="cobaExcit_g">
          <MathInline>-cobaExcit_g/cobaExcit_tau</MathInline>
        </TimeDerivative>
        <OnEvent target_regime="RegularRegime" src_port="cobaExcit_spikeinput">
          <StateAssignment variable="cobaExcit_g">
            <MathInline>cobaExcit_g+cobaExcit_q</MathInline>
          </StateAssignment>
        </OnEvent>
        <OnCondition target_regime="RefractoryRegime">
```

```
          <StateAssignment variable="iaf_tspike">
            <MathInline>t</MathInline>
          </StateAssignment>
          <StateAssignment variable="iaf_V">
            <MathInline>iaf_vreset</MathInline>
          </StateAssignment>
          <OutputEvent port="iaf_spikeoutput"/>
          <Trigger>
            <MathInline>iaf_V &gt; iaf_vthresh</MathInline>
          </Trigger>
        </OnCondition>
      </Regime>
      <Alias name="cobaExcit_I">
        <MathInline>cobaExcit_g*(cobaExcit_vrev-iaf_V)</MathInline>
      </Alias>
    </Dynamics>
  </ComponentClass>
  <Dimension name="time" t="1"/>
  <Dimension name="voltage" m="1" l="2" t="-3" i="-1"/>
  <Dimension name="conductance" m="-1" t="3" l="-2" i="2"/>
  <Dimension name="area" l="2"/>
</NineML>
```

User Layer description for the above example:

```
<?xml version='1.0' encoding='UTF-8'?>
<NineML xmlns="http://nineml.net/9ML/1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://nineml.net/9ML/1.0/NineML_v1.0.xsd">
  <Component name="IaFNeuron">
    <Definition url="http://nineml.net/catalog/neurons/iafICoba.9ml"
      >Iaf1Coba</Definition>
    <Property name="iaf_V" units="mV">
      <SingleValue>-60</SingleValue>
    </Property>
    <Property name="iaf_tspike" units="ms">
      <SingleValue>-1</SingleValue>
    </Property>
    <Property name="cobaExcit_g" units="mS">
      <SingleValue>0</SingleValue>
    </Property>
    <Property name="iaf_cm" units="cm_square">
      <SingleValue>0.02</SingleValue>
    </Property>
    <Property name="iaf_taurefrac" units="ms">
      <SingleValue>3</SingleValue>
    </Property>
    <Property name="iaf_gl" units="mS">
      <SingleValue>0.1</SingleValue>
    </Property>
    <Property name="iaf_vreset" units="mV">
      <SingleValue>-70</SingleValue>
    </Property>
    <Property name="iaf_vrest" units="mV">
      <SingleValue>-60</SingleValue>
    </Property>
```

```
    <Property name="iaf_vthresh" units="mV">
      <SingleValue>20</SingleValue>
    </Property>
    <Property name="cobaExcit_tau" units="ms">
      <SingleValue>2</SingleValue>
    </Property>
    <Property name="cobaExcit_q" units="ms">
      <SingleValue>1</SingleValue>
    </Property>
    <Property name="cobaExcit_vrev" units="mV">
      <SingleValue>0</SingleValue>
    </Property>
  </Component>
  <Dimension name="time" t="1"/>
  <Dimension name="voltage" m="1" l="2" t="-3" i="-1"/>
  <Dimension name="conductance" m="-1" t="3" l="-2" i="2"/>
  <Dimension name="area" l="2"/>
  <Unit symbol="mV" dimension="voltage" power="-3"/>
  <Unit symbol="ms" dimension="time" power="-3"/>
  <Unit symbol="cm_square" dimension="area" power="-4"/>
  <Unit symbol="mS" dimension="conductance" power="-3"/>
</NineML>
```

The simulation results is presented in Figure 6.

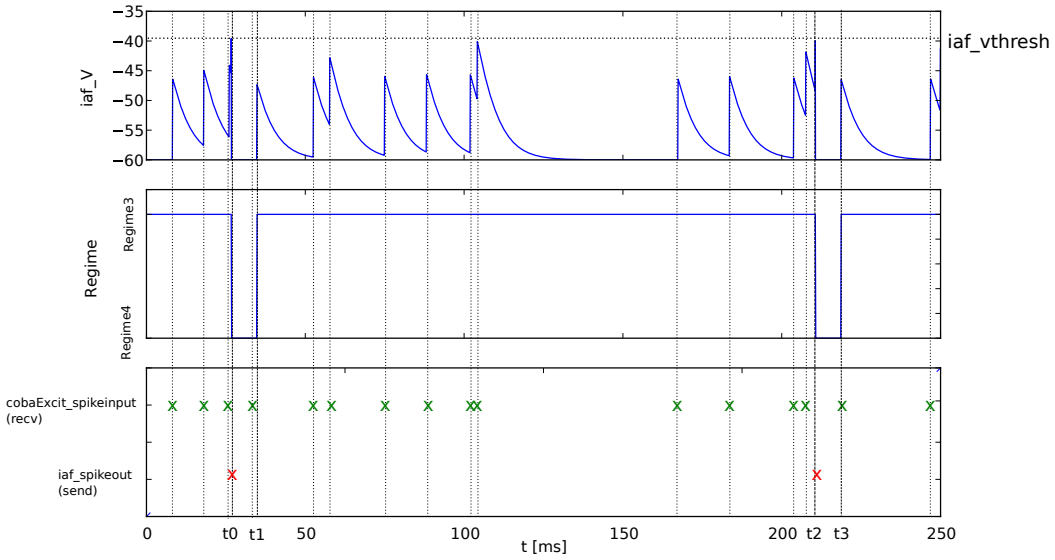

Figure 6: *Result of simulating of the XML model in this section.* cobaExcit_spikeinput *is fed events from an external Poisson generator in this simulation*

B Transition resolution

Do we really want to include this here? Transitions (with the exception of those which just emit spikes) should really be wound back to the exact time point in my opinion because handling multiple simultaneous (in terms of time steps) transitions doesn't seem well defined to me.

This section outlines pseudo code which defines the order of transition-triggering, state assignment execution, event emission, transmission and resolution in a system of connected components. Implementations do not need to implement this algorithm but should produce the same behaviours.

A `TransitionResolutionBlock` represents an instant in time. It begins before any transitions occur and ends after each component has moved into its new **Regime**, all **StateAssignments** have been executed and all Events generated and resolved in the system.

B.1 Serial implementation of transition resolution

We have a system of N components *{C_1,C_2,...,C_N}*, at time, *t*, where each component, *C_n*, is in **Regime** R_n^t.

From **Regime** R_n^t, there are:

- OnEvent transitions $OnEv_n^t = \{...\}$
- OnCondition transitions $OnCond_n^t = \{...\}$

Component *C_n*has:

- **send** EventPorts $EvSend = \{EvSend_{n,1,}, EvSend_{n,2,}, ...\}$
- **recv** EventPorts $EvRecv = \{EvRecv_{n,1,}, EvRecv_{n,2,}, ...\}$

EventPort connections are stored in a a map, *EvPortConnections*, which maps EvSend to a list of EvRecv ports. i.e.,

{EvSend → [EvRecv,EvRecv,..,EvRecv], EvSend → [EvRecv,EvRecv,EvRecv,...,EvRecv]}.

Each component has 3 associated data structures

- RegimeChangeList (RCL_n) (This list will contain target-regimes of triggered transitions)
- ActiveUnresolvedTransitionsQueue ($AUTQ_n$) (This queue will contain transitions which will occur, but their effects have not be evaluated yet)
- EventQueue (EQ_n) (This list contains events delivered to this component from other components via EventPort-connections)

B.1.1 Algorithm

1. Enter `TransitionResolutionBlock`
2. For each component, *C_n*: clear RCL_n, $AUTQ_n$and EQ_n.
3. For each component, *C_n*: for each *oncond* in $OnCond_n^t$: if *oncond.trigger* evaluates to true, add *oncond* to $AUTQ_n$.
4. For each component, *C_n*: for each *tr* in $AUTQ_n$:
   - remove *tr* from $AUTQ_n$
   - add the target_regime to RCL_n
   - for each *action* in *tr.do*:

- if *action* is an OutputEvent: test if the OutputEvent port is a key in *EvPortConnections*. If so, add the OutputEvent to the EventQueue (*EQ_target*) corresponding to each *EvRecv* in the *EvPortConnections* map.
- if *action* is a StateAssignment, execute that state-assignment immediately.

5. For each component *C_n*: for each event, *EvRecv* in EQ_n: test whether there is a transition, *tr* triggered by this event, i.e an OnEvent in $OnEv_n^t$ from R_n^t ; if so; then add it to $AUTQ_n$.
6. While any component has a non-empty *AUTQ*: Goto (4).
7. For each component, *C_n*, check that all the target-regimes in the $RCL_n$are the same regime. (If not raise a RuntimeError). Each component moves into this target-regime, or remains in the same regime if $RCL_n$is empty.
8. Leave `TransitionResolutionBlock`

B.1.2 Notes

Not sure about this either. Since a transition can also mean transitioning to a new regime it would be very bad if two conditions were met at the same time and it was ambiguous which was applied first. Should we not require an order of execution if there are multiple ***OnConditions*** *in the same regime. Could be quite difficult for the user to handle this otherwise*

1. 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 up the user to ensure this does not happen.

2. This Resolution System allows *cascading* of Events, which in theory could be recursive through components, depending on connectivity. The implementation allows for this; and it is the users responsibility to ensure that there are not such issues. The implementation may decided to terminate Step (6) after a given number (say 1000) of iterations to prevent infinite loops.

B.2 Parallelising of event resolution

I am not sure this is a good idea, we could just not allow ***OutputEvent*** *in* ***OnEvent*** *transitions*

This algorithm can be parallelised as following. We create a thread for each Component, which can independently execute Steps (3 to 6). The threads need to be synchronized after steps (4) and (5) as shown in Figure Figure 7.

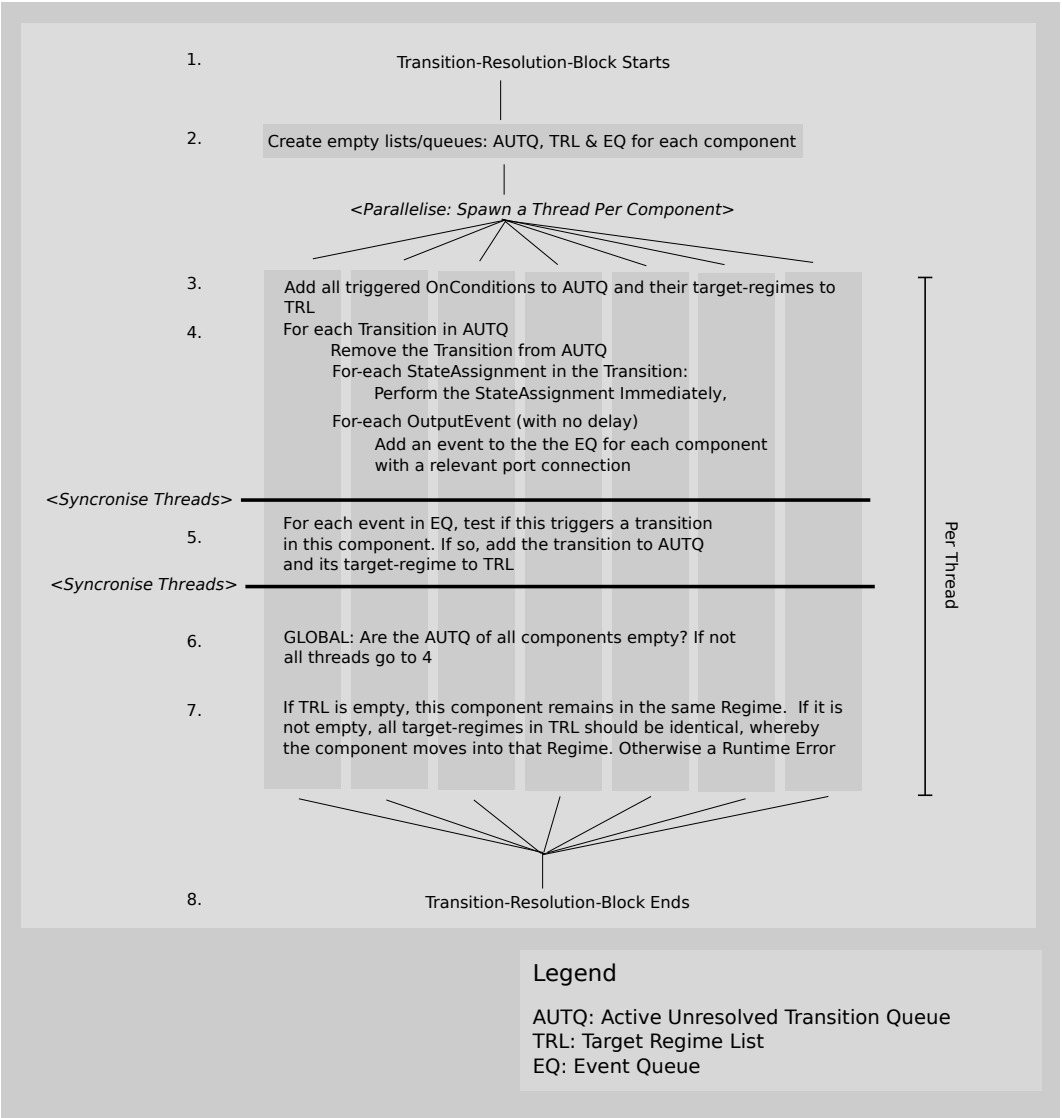

Figure 7: *Parallelising of Event Resolution.*

C Acknowledgments

C.1 Former NineML INCF Task Force members

- Abigail Morrison
- Alex Cope
- Anatoli Gorchetchnikov
- Andrew P. Davison
- Birgit Kriener
- Chung-Chuan Lo
- Damien Drix
- Dragan Nikolic
- Eilif Muller
- Erik De Schutter
- Hans Ekkehard Plesser
- Hugo Cornelis
- Ivan Raikov
- Lars Schwabe
- Malin Sandström
- Michael Hull
- Mikael Djurfeldt
- Padraig Gleeson
- Raphael Ritz
- Robert Cannon
- Robert Clewley
- Sean Hill
- Subhasis Ray
- Valentin Haenel
- Yann Le Franc

References

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.