LEMS - User-defined types for model specification

Robert Cannon and Padraig Gleeson, October 2011

Contents

1 Compact model specification with user-defined types

A proposal for supporting models of synapses and other user-definable component types in NeuroML

Current version: These pages as a single PDF

LEMS - Low Entropy Model Specification

This proposal consists of:

- A An overview of how types and models are declared
- A set of XML elements for defining and using user-defined types
- A proof-of-concept interpreter for processing and running models built with these elements
- Some examples (left menu) illustrating how elements can be defined and used.
- A summary of the canonical form for a model that corresponds to the elements presented here
- Miscellaneous discussion of the objectives, design issues, benefits and weaknesses of this approach

The best place to start is probably the first example. Anyone familiar with modeling and model specification should be able to read the XML and make out what is going on.

After that, feel free to download the interpreter, run the examples, and try constructing your own.

A discussion of how LEMS is being used to specify NeuroML v2.0 ComponentTypes is available here.

What it does so far

You can define **`ComponentType`**s (e.g. a "HH channel" or "a bi-exponential synapse") which express the general properties of a particular type of thing that goes in a model. This includes saying what parameters they have, what child elements they are allowed, and how they behave (the equations).

You can then define **`Component`**s based on these types by supplying values for the parameters and adding any child elements that are required, so, for example, a bi-exponential synapse model with rise time 1ms and decay 5ms would be a component.

`ComponentType`s can extend other **`ComponentType`**s to add extra parameters, fix certain values, and otherwise modify their behavior. **`Component`**s can extend other **`Component`**s to reuse specified parameter values. There is also a loose notion of abstract types, so a component can accept children with a particular lineage without needing to know exactly what type they are. This can be used, for example, to define cells that accept synaptic connections provided they have a particular signature.

Each **`ComponentType`** can have a **`Behavior`** element that specifies how it behaves: what the state variables are, the equations that govern them, and what happens when events are sent or received. The interpreter takes a model consisting of type and component elements referenced from a network, builds an instance from them and runs it.

For those familiar with object oriented languages, the `ComponentType`/`Component` distinction is close to the normal Class/Instance distinction. When the model is run, the same pattern applies again, with the `Component`s acting as class definitions, with their "instances" actually containing the state variables in the running mode.

Background

The March 2010 NeuroML meeting identified a need to extend the capability within NeuroML for expressing a range of models of synapses. It was decided that the hitherto adopted approach of defining parameterized building blocks to construct models by combining blocks and setting parameters was unlikely to be flexible enough to cope with the needs for synapse models. This is not obvious a-priori, since, for example, the pre NeuroML 2.0 ion channel building blocks are fully adequate to describe the Behavior of a wide range existing channel models. But there appears to be no such commonality in models used for synapses, where the mechanisms used range from highly detailed biochemical models to much more abstract ones.

This work also has antecedents in Catacomb 3, which was essentially a GUI for a component definition system and model builder using a type system similar to that proposed here. Much of the XML processing code used in the interpreter was taken from PSICS which itself currently uses the "building block" approach to model specification. The need for user-defined types has been considered with respect to future PSICS development, and this proposal also reflects potential requirements for PSICS.

Example

Here is the XML for a simple integrate-and-fire cell definition:

```
<ComponentType name="refractiaf">
    <Parameter name="threshold" dimension="voltage" />
    <Parameter name="refractoryPeriod" dimension="time" />
    <Parameter name="capacitance" dimension="capacitance" />
    <Parameter name="vleak" dimension="voltage" />
    <Parameter name="gleak" dimension="conductance" />

    <Parameter name="current" dimension="current" />
    <Parameter name="vreset" dimension="voltage" />
    <Parameter name="deltaV" dimension="voltage" />
    <Parameter name="v0" dimension="voltage" />

    <EventPort name="out" direction="out" />
    <EventPort name="in" direction="in" />

    <Exposure name="v" dimension="voltage" />

<Behavior>
   <StateVariable name="v" exposure="v" dimension="voltage" />
   <OnStart>
    <StateAssignment variable="v" value="v0" />
   </OnStart>

   <Regime name="refr">
      <StateVariable name="tin" dimension="time" />
      <OnEntry>
         <StateAssignment variable="tin" value="t" />
         <StateAssignment variable="v" value="vreset" />
      </OnEntry>
      <OnCondition test="t .gt. tin + refractoryPeriod">
          <Transition regime="int" />
      </OnCondition>
   </Regime>

   <Regime name="int" initial="true">
```

```
      <TimeDerivative variable="v" value="(current + gleak * (vleak - v)) / capaci
tance" />
      <OnCondition test="v .gt. threshold">
         <EventOut port="out" />
         <Transition regime="refr" />
      </OnCondition>
      <OnEvent port="in">
        <StateAssignment variable="v" value="v + deltaV" />
      </OnEvent>

   </Regime>
</Behavior>

</ComponentType>
```

Once this definition is available, a particular model using this structure can be specified with the following XML:

```


    <refractiaf threshold="-40mV" refractoryPeriod="5ms" capacitance="1nF" vleak="-80mV" gleak="100pS" v
>
```

More complex models will have nested components and other types of parameters, but the basic principle of separating out the equations and parameters for reusable model components, such that the equations are only stated once, remains the same.

2 Model structure overview

Models are based on user-defined types (the term `ComponentType` is used in the XML) that contain parameter declarations, reference declarations and specification of what children an instance of a type can have. Typically they also contain a `Behavior` specification which can contain build-time and run-time declarations. Build-time declarations apply when a simulation is set up, for example to connect cells. Run-time declarations specify the state variables, equations and events that are involved.

An instance of a `ComponentType` is a model `Component` It specifies a particular set of parameters for a given `ComponentType`. It says nothing about state variables: in a simulation, typically many run-time instances will correspond to a single model component definition, and several model component definitions will use the same type. A run-time instance holds its own set of state variables as defined by the Type definition and a reference to its component for the parameter values specific to that particular model component. The update rules come from the type definition. As such, neither the `ComponentType` nor the `Component` is properly a "prototype" for the runtime instance.

Defining `ComponentType`s

`ComponentType`s are declared as, for example:

```
<ComponentType name="myCell">
    <Parameter name="threshold" dimension="voltage" />
 </ComponentType>
```

A `Component` based on such a type is expressed as:

```
<Component type="myCell" threshold="dimensional_quantity" />
```

The quoted value for 'threshold' here is a rich quantity with size and dimensions, typically consisting of a numerical value and a unit symbol. Assignments like this are the *only place* unit symbols can occur. Equations and expressions relate rich types, independent of any particular unit system.

An equivalent way of writing the above in shorthand notation (using an example of a string with size and dimension for threshold) is:

```
<myCell threshold="-30 mV" />
```

A type can contain elements for specifying the following aspects of the structure and parameters of a model component:

- `Parameter` - dimensional quantities that remain fixed within a model
- `Child` - a required single sub-component of a given type
- `Children` - variable number of sub-components of the given type
- `ComponentRef` - a reference to a top-level component definition.

- `Link` - a reference to a component definition relative to the referrer
- `Attachments` - for build-time connections
- `EventPort` - for run-time discrete event communication
- `Exposure` - quantities that can be accessed from other components
- `Requirement` - quantities that must be accessible to the component for it to make sense
- `DerivedParameter` - like parameters, but derived from some other quantity in the model

The "EventPort" and "Attachments" declarations don't have any corresponding elements in their model component specification. They only affect how the component can be used when a model is instantiated. EventPorts specify that a model can send or receive events, and should match up with declarations in its Behavior specification. An "Attachments" declaration specifies that a run-time instance can have dynamically generated attachments as, for example, when a new synapse run-time instance is added to a cell for each incoming connection.

Inheritance

A type can extend another type, adding new parameters, or supplying values (SetParam) for inherited parameters. As well as reducing duplication, the key application of this is with the Child and Children declarations, where a type can specify that it needs a child or children of a particular supertype, but doesn't care about which particular sub-type is used in a model. This applies, for example, where a cell requires synapses that compute a quantity with dimensions current, but doesn't need access to any other parts of the synapse Behavior.

Run-time Behavior

Run time Behavior are included within a Behavior block in a type specification. They include declaration of:

- state variables
- first order differential equations with respect to time of state variables
- derived quantities - things computed in terms of other local quantities or computed from other run-time instances

Run time Behavior can be grouped into Regimes, where only one regime is operative at a given time for a particular run-time instance. Regimes have access to all the variables in the parent instance and can define their own local variables.

Behavior can also contain event blocks:

- OnStart blocks contain any initialization declarations needed when a run-time state is instantiated
- OnEvent blocks specify what happens when an event is received on a specified port
- OnEntry blocks (only within regimes) specify things that should happen each time the system enters that regime.
- OnCondition blocks have a test condition and specify what should happen when it is met.

Blocks may contain state variable assignments, event sending directives and transition directives to indicate that the system should change from one regime to another.

Build-time Behavior

Build-time Behavior define the structure of a multi-component model. Currently there are:

- MultiInstantiate - for declaring that a component yields multiple run-time instances corresponding to a particular model component. Eg, for defining populations of cells.
- ForEach - for iterating over multiple instances in the run-time structure
- EventConnection - for connecting ports between run-time instances

Other

There are also Run, Show and Record Behavior for creating type definitions that define simulations and what should be recorded or displayed from such a simulation.

Observations

The numerous references to "run-time instances" above is problematic, since the structures do not dictate any particular way of building a simulator or running a model. In particular, there is no requirement that a component or Behavior declaration should give rise to any particular collection of state variables that could be interpreted as a run-time instance in the state of a simulator.

So, it is convenient to think of eventual state instances, and that is indeed how the reference interpreter works, but the model specification structure should avoid anything that is specific to this picture.

Type specification examples

Examples of type definitions using the various types of child element:

```
          <ComponentType name="synapse">
              <EventPort direction="in" />
           </ComponentType>
```

says that instances of components using this type can receive events.

```
              <ComponentType name="HHChannel">
                  <Children name="gates" type="HHGate" />
                 </ComponentType>
```

says that a HHChannel can have gates.

```
        <ComponentType name="HHGate">
               <Child name="Forward" type="HHRate" />
                <Child name="Reverse" type="HHRate" />
        </ComponentType>
```

says that a HHGate has two children called Forward and Reverse, each of type HHRate.

```
        <ComponentType name="synapseCell">
           <Attachments name="synapses" type="synapse" />
          </ComponentType>
```

says that instances of components based on the synapseCell type can have instances of component based on the synapse type attached to them at build time.

```
           <ComponentType name="Population">
               <ComponentRef name="component" type="Component" />
           </ComponentType>
```

says that components based on the Population type need a reference to a component of type Component (ie, anything) (which would then be used as the thing to be repeated in the population, but it doesn't say that here).

```
            <ComponentType name="EventConnectivity">
   <Link name="source" type="Population" />
           </ComponentType>
```

says that EventConnectivity components need a relative path to a local component of type Population which will be accessed via the name "source".

The model component declarations corresponding to the channel and gate types would be:

```
       <Component type="HHChanne">
           <Component type="HHGate">
               <Component type="some_hh_gate_type" role="Forward" />
               <Component type="some_hh_gate_type" role="Reverse" />
           </Component>
       </Component>
```

or, in the shorthand notation:

```
           <HHChannel>
               <HHGate>
                   <Forward type="some_hh_gate_type" />
                   <Reverse type="some_hh_gate_type" />
               </HHGate>
           </HHChannel>
```

For the population type it would be:

```
<Component id="myPopulation" type="population" component="myCellModel" />
```

And for the connections:

```
<Component type="EventConnectivity" source="myPopulation" />
```

3 All elements

This page summarizes all the element types currently supported by the proof-of-concept interpreter. The explanations are somewhat cursory so it is probably better to look at the examples to work out what is going on.

There is one block for each element type. All element names are in fixed width bold. Attribute names are in normal fixed width. Each block shows the attributes supported by an element and then the types of children allowed.

Model structure

Models can be spread over multiple files. The root element in each file is **`Lems`**.

`Lems`

`Include` *(any)*
`Dimension` *(any)*
`Unit` *(any)*
`ComponentType` *(any)*
`Component` *(any)*

For the purposes of the early examples, a couple of other, deprecated elements are also allowed: XSimulation and XNetwork

`Include`

`file` ***(string)*** the name or relative path of a file to be included

Files are included where the **`Include`** declaration occurs. The enclosing **`Lems`** block is stripped off and the rest of the content included as is.

Units and dimensions

`Dimension`

`name` ***(string)*** the name of the dimension. Other items refer to it by this name.
`m` ***(int)*** exponent for Mass
`l` ***(int)*** exponent for Length
`t` ***(int)*** exponent for Time
`i` ***(int)*** exponent for Current

The name attribute is required. The others default to zero if not present.

`Unit`

`symbol` ***(string)*** the symbol is used in the value of a dimensional quantity
`dimension` Reference to a component of type Dimension.
`powTen` ***(int)*** The power of ten that scales this unit from the SI unit of the same dimension

The symbol and dimension attributes are required. powTen defaults to zero if not present.

Defining ComponentType and Component elements

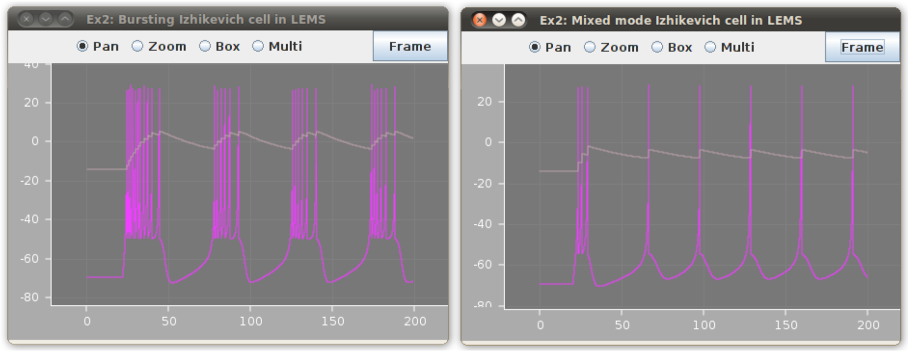
4 LEMS Reference Implementation

To illustrate the concepts of LEMS and to facilitate building, validating and execting models specified in the format, a Reference Implementation in Java has been created.

The implementation will process and run models defined using the proposed elements. It is mainly focused on model processing and validation. In particular, as well as handling type and component inheritance, it performs dimensionality checks on all equations to identify any inconsistencies.

It has a limited ability to run models, but only with the simplest possible (forward Euler) integration scheme. It doesn't do any symbolic manipulations and expressions are evaluated by visiting their parse trees.

After running a model, it opens a window to display the results and only exits when this window is closed. The image below shows traces of the V and U state variables from 2 instances of the Izhikevich cell model specified in LEMS and run using the interpreter.



Installation of LEMS Reference Implementation

Install Java J2SE 5 or higher. Available here. It's better to download the JDK (Java Development Kit), which includes the command line tools for compiling Java.

To try LEMS out, all you need is the executable jar file:

-

Just download this file and run:

```
              java -jar lems-x.x.x.jar model.xml
```

where model.xml is the file containing the main model.

Source code installation with examples

If you do not have Subversion installed, it may be simplest to download the latest code, unzip the file, and run the code as described below.

Alternatively, check out the latest version of the code (from the NeuroML SourceForge SVN repository) using:

```
            svn co https://neuroml.svn.sourceforge.net/svnroot/neuroml/
  LEMS            cd LEMS
            cd LEMS
```

To compile the code it's best to install Ant (see below), but the code can be compiled using:

```
          ./make.sh                               (Linux or Mac)
          make.bat                                (Windows)
```

The standard examples can be found in the examples directory. These can be run using:

```
          ./lems exam
ples/example1.xml           (Linux or Mac)          lems.bat examples
\example5.xml          (Windows)
 Mac)          lems.bat examples\example5.xml
 (Windows)
```

If Apache Ant is installed, the options for compiling/running examples with this include:

```
          ant                                 (build the main jar file)
          ant ex1                             (run the main examples ex1 to ex8)
          ant test                            (perform a number of tests including running all examples)

          ant -p                              (list all options)
```

A number of links to examples showing the core behaviour of LEMS is available in the left column.

5 Example 1: Dimensions, Units, ComponentTypes and Components

This page is structured as a walk-through of a single example file example1.xml explaining the various elements as they occur.

The whole model is wrapped in a block which, for now, is called "Lems" (Low Entropy Model Specification). Then we define the dimensions that will be used in this model. Typically these would be loaded from an external file along with various other stuff, not repeated in each model, but it is included here in the interests of having a single file for everything.

```
<Dimension name="voltage" m="1" l="2" t="3" i="-1" />
<Dimension name="time" t="1" />
<Dimension name="per_time" t="-1" />
<Dimension name="conductance" m="-1" l="-2" t="3" i="2" />
<Dimension name="capacitance" m="-1" l="-2" t="4" i="2" />
<Dimension name="current" i="1" />
```

Each dimension element just associates a dimension name with the exponents for mass, length, time and current.

At this stage, one can begin defining component types. This is done with the **`ComponentType`** element and child **`Parameter`** elements. A simple cell model with three parameters could be defined as:

```
<ComponentType name="cell1">
    <Parameter name="threshold" dimension="voltage" />
    <Parameter name="refractoryPeriod" dimension="time" />
    <Parameter name="capacitance" dimension="capacitance" />
</ComponentType>
```

Each of the **`Parameter`** elements defines a parameter that should be supplied when a component is defined based on this type. Before we can define a component though, we need some units to use in setting those values.

Defining a unit involves supplying the symbol, dimension and the power of ten by which it is scaled from the IS base unit. Note that units have a symbol, not a name. This is because they occur as a component of an assignment expression such as 'threshold="-45mV"' not as a reference such as 'dimension="voltage"'. In general, where one component refers to another, then the attribute value is the name of the thing being referred to, and the attribute name is the lower case version of the type of the thing being referred to. Thus when a dimension is declared with <Dimension name="voltage".../> then it is referred to from a Parameter as <Parameter dimension="voltage"/>. This holds for all references to components of a particular type.

Returning to the units, this model will use the following, which normally would also be loaded from an external file of standard settings.

```
<Unit symbol="mV" dimension="voltage" powTen="-3" />
<Unit symbol="ms" dimension="time" powTen="-3" />
<Unit symbol="pS" dimension="conductance" powTen="-12" />
<Unit symbol="nS" dimension="conductance" powTen="-9" />
<Unit symbol="uF" dimension="capacitance" powTen="-6" />
<Unit symbol="nF" dimension="capacitance" powTen="-9" />
<Unit symbol="pF" dimension="capacitance" powTen="-12" />
```

```
<Unit symbol="per_ms" dimension="per_time" powTen="3" />
<Unit symbol="pA" dimension="current" powTen="-12" />
```

Once the units are available it is possible to define a component. There are two equivalent ways of doing this: either by using the `Component` element and setting its type, or by using the type as a new XML element. The latter may be a little more readable, but for a simple component like this it doesn't make much difference. For more complicated components with nested children though, the second form is definitely clearer (eg see the HHChannel examples later).

```
<Component id="ctb" type="cell1" threshold="-30 mV" refractoryPeriod="2
 ms" capacitance="1uF" />
<!--  or  -->
<cell1 id="celltype_a" threshold="-30 mV" refractoryPeriod="2 ms" capacitance="3uF" />
```

In specifying a component, a value must be supplied for each of the parameters defined in the corresponding type. The value is composed of a number and a unit. It can't include expressions with multiple units for the values so, for example, to express an acceleration you couldn't write "3 m s^-2". Instead would need to define a unit element for the compound unit (and a dimension element for acceleration) and use that.

Specifying all the parameters for each component can lead to duplication. Suppose, for example, you want to build a range of cell models all based on cell1, but you don't want to change the threshold. You could define a new type without the threshold, but it is neater to still use the same type but specify that you are restricting attention to the set that all have a particular value for the threshold. This can be done by creating a new type that extends the cell1 type and includes a Fixed element to fix the threshold:

```
<ComponentType name="cell2" extends="cell1">
    <Fixed parameter="threshold" value="-45mV" />
</ComponentType>
```

The cell2 type can now be used by only setting the remaining two parameters.

As well as restricting types when you extend them, you can also add new parameters as shown in the next type. This also introduces an `EventPort`, to indicate that instances of components built from this type can receive events, and, finally, a `Behavior` block. This is where the Behavior of instances of the component can be specified. The phrase "instances of the component" is intentional. The type itself doesn't "behave": it is just a definition. A component built from the type doesn't "behave" either: it is just a set of parameter values linked back to the type. The thing that "behaves" is an instance in a runnable model that actually contains state variables. In general, many components may be based on one type, and one component may give rise to many instances in a running model.

Here is a basic capacitative cell with a leaking potential and a simple event handler.

```
<ComponentType name="cell3" extends="cell1">
    <Parameter name="leakConductance" dimension="conductance" />
    <Parameter name="leakReversal" dimension="voltage" />
    <Parameter name="deltaV" dimension="voltage" />

    <EventPort name="spikes-in" direction="in" />
    <Exposure name="v" dimension="voltage" />

    <Behavior>
       <StateVariable name="v" exposure="v" dimension="voltage" />
       <TimeDerivative variable="v" value="leakConductance * (leakReversal - v) /
 capacitance" />
```

```
        <OnEvent port="spikes-in">
            <StateAssignment variable="v" value="v + deltaV" />
        </OnEvent>
    </Behavior>
</ComponentType>
```

The **Behavior** involves a single state variable, a voltage called "v", and one equation, expressing how v drifts towards the leak reversal potential. The event block specifies what happens when an instance receives an event. In this case the state variable v is bumped up by deltaV. The value attribute in the TimeDerivative element is an expression involving the parameters and the state variables. It gives the right hand side of a first order differential equation dv/dt = (...). Expressions follow normal operator precedence rules with "^" for general powers and exp(x) for exponentials.

Below is another example of a Behavior, this time with an output port and a condition testing block that sends an event when the condition becomes true. The test attribute in the **OnCondition** element is a boolean valued expression. These use Fortran style operators (.gt. and .lt. for > and <) to avoid confusion with xml angle brackets.

```
<ComponentType name="spikeGenerator">
    <Parameter name="period" dimension="time" />
    <EventPort name="a" direction="out" />
    <Exposure name="tsince" dimension="time" />
    <Behavior>
        <StateVariable name="tsince" exposure="tsince" dimension="time" />
        <TimeDerivative variable="tsince" value="1" />
        <OnCondition test="tsince .gt. period">
            <StateAssignment variable="tsince" value="0" />
            <EventOut port="a" />
        </OnCondition>
    </Behavior>
</ComponentType>
```

The above model is one way of writing a regular event generator. It has a state variable that grows in sync with t until it reaches a threshold when the event fires and it is reset. The model below achieves the same effect without solving a differential equation. Instead, it asks for access to the global time variable ("t" is the one global variable that is always available) and uses that in the test condition. [aside - there's a slight problem here since t exists even if the model doesn't define a dimension called time].

```
<ComponentType name="spikeGenerator2" extends="spikeGenerator">
    <Behavior>
        <GlobalVariable name="t" dimension="time" />
        <StateVariable name="tlast" dimension="time" />
        <DerivedVariable name="tsince" exposure="tsince" value="t - tlast" />
        <OnCondition test="t - tlast .gt. period">
            <StateAssignment variable="tlast" value="t" />
            <EventOut port="a" />
        </OnCondition>
    </Behavior>
</ComponentType>
```

The examples so far have all been of very simple components which just had a single set of parameters. Real models however require rather more structure than this with components having children of various types and possibly multiple children of certain types. To illustrate this, the next example shows how the concept of an ion channel using Hodgkin-Huxley Behavior can be defined.

Starting from the bottom, we define the different types of rate equations that occur. These will supply terms in the equations for the derivatives of the gating particles. There are three different expressions used in the HH equations, but they can all be expressed with three parameters, rate, midpoint and scale. We first define a general rate class, and then extend it for the three cases.

The **`HHRate`** Behavior shows two new constructs. An **`Exposure`** declares that the component makes a quantity available to other components. A **`Requirement`** specifies that the component needs to know about a variable that it doesn't define itself. When it is used in a model, the specified variable must be available (and have the right dimension) in the parent component or one of its more remote ancestors.

Note that the general **`HHRate`** class defines an **`Exposure`** without a Behavior block to actually set its value. This is analogous to an abstract class in java: you can't actually make a component out of the **`HHRate`** element directly (the interpreter will complain) but any component using a **`HHRate`** will know it has an exposed variable called "r". The types that extend **`HHRate`** have to supply a value for "r" before they are fully defined and ready to be used.

So here is the basic HHRate and its three extensions:

```
<ComponentType name="HHRate">
    <Parameter name="rate" dimension="per_time" />
    <Parameter name="midpoint" dimension="voltage" />
    <Parameter name="scale" dimension="voltage" />
    <Exposure name="r" dimension="per_time" />

</ComponentType>

<ComponentType name="HHExpRate" extends="HHRate">
    <Behavior>
        <DerivedVariable name="r" exposure="r" value="rate * exp((v - midpoint)/scale)" /
>
    </Behavior>
</ComponentType>


<ComponentType name="HHSigmoidRate" extends="HHRate">
    <Behavior>
        <DerivedVariable name="r" exposure="r" value="rate / (1 + exp(0 - (v - midpoint)/
scale))" />
    </Behavior>
</ComponentType>

 <ComponentType name="HHExpLinearRate" extends="HHRate">
    <Behavior>
        <DerivedVariable name="x" value="(v - midpoint) / scale" />
        <DerivedVariable name="r" exposure="r" value="rate * x / (1 - exp(0 - x))" />
    </Behavior>
</ComponentType>
```

Now the rate elements are available, they can be used to define a component for a gate in a HH model. This introduces the **`Child`** element which says that components built using this type must include a subcomponent of the specified type. A HH gate needs subcomponents for the forward and reverse rates.

```
<ComponentType name="HHGate0">
    <Parameter name="power" dimension="none" />
    <Child name="Forward" type="HHRate" />
    <Child name="Reverse" type="HHRate" />
    <Exposure name="fcond" dimension="none" />
    <Requirement name="v" dimension="voltage" />
    <Behavior>
        <StateVariable name="q" dimension="none" />
        <DerivedVariable name="rf" select="Forward/r" />
        <DerivedVariable name="rr" select="Reverse/r" />
        <TimeDerivative variable="q" value="rf * (1 - q) - rr * q" />
        <DerivedVariable name="fcond" exposure="fcond" value="q^power" />
    </Behavior>
</ComponentType>
```

The above is a perfectly reasonable way to define a HH gate but unfortunately it needs smarter numerics than the simple forward Euler rule used in the proof of concept interpreter. Running this model with the Euler method leads to numerical instabilities. Happily, this problem can be circumvented without improving the numerics by changing the state variable. Instead of q which is defined on [0, 1] you can use x defined on (-infinity, infinity) which works much better with a naive integration scheme. This is what it looks like with x instead of q:

```
<ComponentType name="HHGate">
    <Parameter name="power" dimension="none" />
    <Child name="Forward" type="HHRate" />
    <Child name="Reverse" type="HHRate" />
    <Exposure name="fcond" dimension="none" />
    <Requirement name="v" dimension="voltage" />
    <Behavior>
        <StateVariable name="x" dimension="none" />
        <DerivedVariable name="ex" dimension="none" value="exp(x)" />
        <DerivedVariable name="q" dimension="none" value="ex / (1 + ex)" />
        <DerivedVariable name="rf" select="Forward/r" />
        <DerivedVariable name="rr" select="Reverse/r" />
        <TimeDerivative variable="x" value="(1 + ex)^2 / ex * (rf * (1 - q) - rr * q)" />
        <DerivedVariable name="fcond" exposure="fcond" value="q^power" />
    </Behavior>
</ComponentType>
```

Now the gate type has been defined, it can be used to say what a HH Channel actually is. In this picture, a channel just has a conductance and one or more gates:

```
<ComponentType name="HHChannel">
    <Parameter name="conductance" dimension="conductance" />
    <Children name="gates" type="HHGate" min="0" max="4" />
    <Requirement name="v" dimension="voltage" />
    <Exposure name="g" dimension="conductance" />

    <Behavior>

        <DerivedVariable name="gatefeff" select="gates[*]/fcond" reduce="multiply" />
        <DerivedVariable name="g" exposure="g" value="conductance * gatefeff" />
    </Behavior>
</ComponentType>
```

This introduces one new construct, the `Children` element, that allows for an indeterminate number of children of a given type. This means that the same type can be used for potassium channels with only one gate, sodium channels with two gates or indeed other channels with more gates. The first derived variable in the Behavior block uses a xpath-style selection function to process the indeterminate number of children. In this case it computes the produce of the `fcond` variables from the different gates.

With these definitions in place, it is now possible to define some channel models. The classic Hodgkin-Huxley sodium channel can be represented as:

```
<HHChannel id="na" conductance="20pS">
    <HHGate id="m" power="3">
        <Forward type="HHExpLinearRate" rate="1.per_ms" midpoint="-40mV" scale="10mV" />
        <Reverse type="HHExpRate" rate="4per_ms" midpoint="-65mV" scale="-18mV" />
    </HHGate>

    <HHGate id="h" power="1">
        <Forward type="HHExpRate" rate="0.07per_ms" midpoint="-65.mV" scale="-20.mV" />
        <Reverse type="HHSigmoidRate" rate="1per_ms" midpoint="-35mV" scale="10mV" />
    </HHGate>
</HHChannel>
```

The potassium channel uses exactly the same types, but has only one gate:

```
<HHChannel id="k" conductance="20pS">
    <HHGate id="n" power="4">
        <Forward type="HHExpLinearRate" rate="0.1per_ms" midpoint="-55mV" scale="10mV" />
        <Reverse type="HHExpRate" rate="0.125per_ms" midpoint="-65mV" scale="-80mV" />
    </HHGate>
</HHChannel>
```

These channel models are an example where the ability to use the type name as the XML tag makes the model much clearer: the alternative just with three levels of Component elements would look rather unhelpful.

Although the channel models have now been defined, they still need to be used in a cell before anything can be run. For this we'll just define a basic channel population type. There is one new construct here: the **`ComponentRef`** element which in this case says that a channel population needs a reference to a component of type **`HHChannel`**. This is much like a **`Child`** element, but instead of the component being defined then and there inside the channel population, there is just a reference to it.

The Behavior block for a cannel population just computes the total conductance and then the current, in this case using Ohm's law.

```
<ComponentType name="ChannelPopulation">
    <ComponentRef name="channel" type="HHChannel" />
    <Parameter name="number" dimension="none" />
    <Parameter name="erev" dimension="voltage" />
    <Requirement name="v" dimension="voltage" />
    <Exposure name="current" dimension="current" />
    <Behavior>
        <DerivedVariable name="channelg" select="channel/g" />
        <DerivedVariable name="geff" value="channelg * number" />
        <DerivedVariable name="current" exposure="current" value="geff * (erev - v)" />
    </Behavior>
</ComponentType>
```

To use these populations, they need inserting in a cell. The following type represents a simple cell with a number of populations and an option to inject a current so it does something interesting.

```
<ComponentType name="HHCell">
    <Parameter name="capacitance" dimension="capacitance" />
    <Children name="populations" type="ChannelPopulation" />
    <Parameter name="injection" dimension="current" />
    <Parameter name="v0" dimension="voltage" />
    <Exposure name="v" dimension="voltage" />
    <Behavior>
        <OnStart>
             <StateAssignment variable="v" value="v0" />
        </OnStart>
```

```
         <DerivedVariable name="totcurrent" select="populations[*]/
current" reduce="add" />
        <StateVariable name="v" dimension="voltage" />
        <TimeDerivative variable="v" value="(totcurrent + injection) / capacitance" />
    </Behavior>
</ComponentType>
```

This Behavior block introduces the **`OnStart`** element which is much like the **`OnEvent`** elements earlier, except the block applies only when the simulation starts. In this case it just sets the voltage to a value supplied as a parameter. The Behavior block uses another selector function "sum(..." to sum the currents delivered by the various populations.

Now all the definitions are in place to define a cell model with a couple of channel populations:

```
<HHCell id="hhcell_1" capacitance="1pF" injection="4pA" v0="-60mV">
    <ChannelPopulation channel="na" number="6000" erev="50mV" />
    <ChannelPopulation channel="k" number="1800" erev="-77mV" />
</HHCell>
```

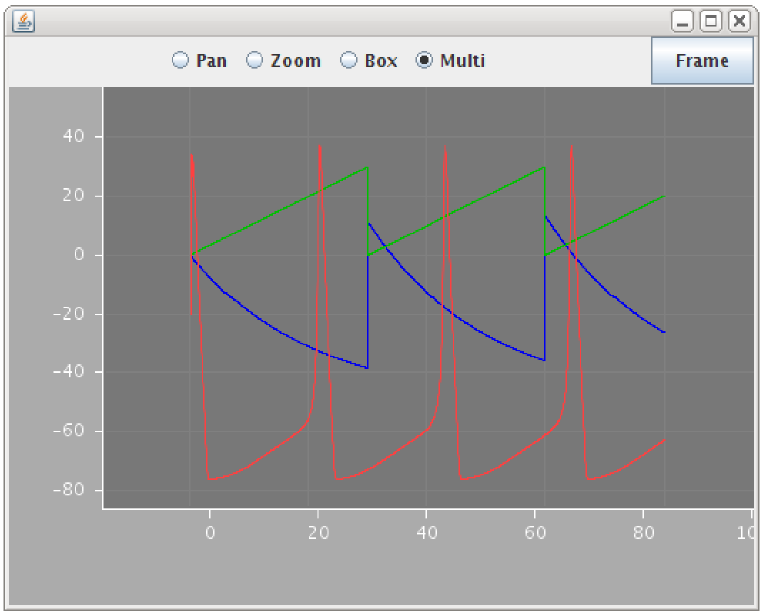
To go with this cell type, we can define some components using the other types defined earlier. Note how celltype_d is based on an existing component via the "extends" attribute and only replaces one parameter value.

```
<Component id="celltype_c" type="iaf3" leakConductance="3 pS" refractoryPeriod="3
 ms" threshold="45 mV" leakReversal="-50 mV" deltaV="5mV" capacitance="1uF" />

<Component id="celltype_d" extends="celltype_c" leakConductance="5 pS" />

<Component id="gen1" type="spikeGenerator" period="30ms" />

<Component id="gen2" type="spikeGenerator2" period="32ms" />

<Component id="cell3cpt" type="cell3" leakReversal="-50mV" deltaV="50mV" threshold="-30mV" leakConductan
>
```

Finally a simulation element says what component is to be run and for how long. It also contains an embedded display element so the results of the simulation can be visualized. These are also user-defined types: their definitions will be presented in example 6.

```
<Simulation length="80ms" step="0.05ms" target="hhcell_1">
    <Display unit="ms">
        <Line quantity="v" unit="mV" color="#0000f0" />
    </Display>
</Simulation>
```

That's it. When this model is run it produces the figure shown below (after rescaling a bit).



The next example shows a few more constructs that are needed for defining synapse models.

6 Example 2: tidying up example 1

This models is the same as in example 1, except that the definitions have been split out into several self-contained files. The files can be downloaded together as examples.zip.

The main file, included below, uses the `Include` element to include definitions from other files. Each file is only read once, even if several files include it. Because some of these files, such as the HH channel definitions, are intended to be used on their own, they include all the dimension definitions they need. These may also occur in other files with the same dimension names. This is fine as long as the dimensions being declared are the same. An error will be reported if a new definition is supplied that changes any of the values. The same applies for `Unit` definitions. For other element types names and ids must be unique. An id or name can't appear twice, even if the content of the elements is the same.

The main model

This defines a few components, then a network that uses them and a simulation to run it all. The HHCell component refers to channel types coming from the included hhmodels.xml file which in turn depends on hhcell.xml and hhchannel.xml.

Included files

The file hhchannel.xml contains complete definitions of a fairly general HH-style channel model with any number of gates based on the three standard types used in the original HH work.

As mentioned in example1, the numerics are too feeble to cope with this gate definition though, so a change of variables is employed instead:

The file hhcell.xml defines a simple cell model with some populations of HH channels.

A couple of spike generators.

And now the components themselves. These are the standard HH sodium and potassium channels (as used in Rallpack3).

Some miscellaneous iaf models.

Finally, a small collection of dimension definitions useful for things like the miscellaneous iaf cell definitions.

7 Example 3: Connection dependent synaptic components

In many models, a synapse is only created where a connection exists. This means that the model of the receiving cell should only declare that particular types of synapse can be added to it, not the actual synapse sub-components themselves.

Not much is needed beyond the elements described in example 1 except for some extensions to the component that declares the connectivity and a new child element in the component that the synapses are attached to. The full example is shown below. The synapse type includes an **`EventPort`** just like the previously defined cell type. The cell type however includes a new child element: **`Attachments`** defined as:

```
<Attachments name="synapses" type="synapse" />
```

This operates rather like the **`Children`** element except that when a component is defined using this type the sub-elements are not included in the component definition. Instead it indicates that instances of components of the particular type may be attached later when the model is actually run.

TODO - replacement of the hard-coded Receiver element with something generic that is accessible from a component type definition.

8 Kinetic schemes

The existing components provide everything necessary to define types that allow a model to specify a kinetic scheme (Markov model). The missing ingredient is the Behavior element to actually expresses how instances of the components develop through time.

First then, the following definitions can be used to express ion channel models where the the channel state is represented by an occupancy vector among a number of distinct states with rates for the transitions between states.

```
<ComponentType name="KSGate">
    <Parameter name="power" dimension="none" />
    <Parameter name="deltaV" dimension="voltage" />
    <Children name="states" type="KSState" />
    <Children name="transitions" type="KSTransition" />
</ComponentType>


<ComponentType name="KSState">
    <Parameter name="relativeConductance" dimension="none" />

    <Behavior>
        <StateVariable name="occupancy" dimension="none" />
        <DerivedVariable name="q" value="relativeConductance * occupancy" />
    </Behavior>
</ComponentType>

<ComponentType name="KSClosedState" extends="KSState">
     <Fixed parameter="relativeConductance" value="0" />
</ComponentType>


<ComponentType name="KSOpenState" extends="KSState">
    <Fixed parameter="relativeConductance" value="1" />
</ComponentType>


<ComponentType name="KSTransition">
    <Link name="from" type="KSState" />
    <Link name="to" type="KSState" />
    <Requirement name="v" dimension="voltage" />
    <Exposure name="rf" dimension="per_time" />
    <Exposure name="rr" dimension="per_time" />

</ComponentType>


<ComponentType name="KSChannel">
    <Parameter name="conductance" dimension="conductance" />
    <Children name="gates" type="KSGate" />
    <Exposure name="g" dimension="conductance" />
    <Behavior>
        <DerivedVariable name="fopen" dimension="none" select="gates[*]/
fopen" reduce="multiply" />
        <DerivedVariable name="g" exposure="g" dimension="conductance" value="fopen *
 conductance" />

    </Behavior>
</ComponentType>
```

This says that a gate can contain any number of states and transitions. A state has an occupancy variable, and a transition has links to two states giving the source and target states for the transition.

The transition element here is an abstract element because it doesn't provide a Behavior block but just specifies what quantities transitions should privide via the two exposures. One of the most useful forms of transition is a damped Boltzman equation which can be parameterizd as follows:

```
<ComponentType name="VHalfTransition" extends="KSTransition">
    <Parameter name="vHalf" dimension="voltage" />
    <Parameter name="z" dimension="none" />
    <Parameter name="gamma" dimension="none" />
    <Parameter name="tau" dimension="time" />
    <Parameter name="tauMin" dimension="time" />
    <Constant name="kte" dimension="voltage" value="25.3mV" />
    <Requirement name="v" dimension="voltage" />

    <Behavior>
        <DerivedVariable name="rf0" dimension="per_time" value="exp(z * gamma * (v -
 vHalf) / kte) / tau" />
        <DerivedVariable name="rr0" dimension="per_time" value="exp(-z * (1 - gamma) * (v
 - vHalf) / kte) / tau" />
        <DerivedVariable name="rf" exposure="rf" dimension="per_time" value="1 / (1/rf0 +
 tauMin)" />
        <DerivedVariable name="rr" exposure="rr" dimension="per_time" value="1 / (1/rr0 +
 tauMin)" />
    </Behavior>
</ComponentType>
```

Given these definitions, we can express a couple of simple channel models that use kinetic schemes. There is nothing special about these models. They are just examples used in PSICS that produce spikes (albeit rather unnatural looking ones) when used together.

```
<KSChannel id="na1" conductance="20pS">
    <KSGate power="1" deltaV="0.1mV">
        <KSClosedState id="c1" />
        <KSClosedState id="c2" />
        <KSOpenState id="o1" relativeConductance="1" />
        <KSClosedState id="c3" />
        <VHalfTransition from="c1" to="c2" vHalf="-35mV" z="2.5" gamma="0.8" tau="0.15ms" tauMin="0.001m
>
        <VHalfTransition from="c2" to="o1" vHalf="-35mV" z="2.5" gamma="0.8" tau="0.15ms" tauMin="0.001m
>
        <VHalfTransition from="o1" to="c3" vHalf="-70mV" z="1.1" gamma="0.90" tau="8.0ms" tauMin="0.01ms
>
    </KSGate>
</KSChannel>


<KSChannel id="k1" conductance="30pS">
    <KSGate power="1" deltaV="0.1mV">
        <KSClosedState id="c1" />
        <KSOpenState id="o1" />
        <VHalfTransition from="c1" to="o1" vHalf="0mV" z="1.5" gamma="0.75" tau="3.2ms" tauMin="0.3ms" /
>
    </KSGate>
</KSChannel>
```

This has all been done with the existing components. They allow types to be defined for expressing kinetic schemes, and models can be expressed that use these types, but there is nothing so far that says that the model actually is governed by a kinetic scheme. In particular, there is an "occupancy" state variable in each state element for which there is no governing equation and the rates generate "rf" and "rr" quantities that are not unused anywhere.

What is needed is a new element in the **Behavior** block to link these together and say that the rates apply to the changes of occupancy among the state elements. This is done by adding a **KineticScheme** element to the Behavior block for a gate as follows (this now shows the full definition of the KSGate element):

```
<ComponentType name="KSGate">
    <Parameter name="power" dimension="none" />
    <Parameter name="deltaV" dimension="voltage" />
    <Children name="states" type="KSState" />
    <Children name="transitions" type="KSTransition" />

    <Behavior>
        <KineticScheme name="ks">
            <Nodes children="states" variable="occupancy" />
            <Edges children="transitions" sourceNodeName="from" targetNodeName="to" forwardRate="rf" rev
>
            <Tabulable variable="v" increment="deltaV" />
        </KineticScheme>

        <DerivedVariable name="q" dimension="none" select="states[*]/q" reduce="add" />
        <DerivedVariable name="fopen" exposure="fopen" dimension="none" value="q^power" /
>
    </Behavior>
</ComponentType>
```

The new part here is the **KineticScheme** element and its children **Nodes**, **Edges** and **Tabulable**. The **Nodes** element says which elements in the parent container are goverened by the scheme, and which variable in those elements represents the relative occupancy.

The **Edges** element is a little more complicated. It has to say not only which elements define the transitions, but how the fields in the transitions map to things the scheme knows about. For a transition in a kinetic scheme, you need to know which state the transition comes from, which it goes to, and how fast it goes. It is possible (as here) that a single transition defines both directions, in which case it must also say which variable in the target objects provides the reverse transition rates. This is what the last four attributes of the **Edges** element do.

The **Tabulable** element is a temporary convenience for implementation purposes. In this case it says that the rates depend only on v and that the transition matrices can be cached an reused on a grid of spacing deltaV rather than recomputed every time. This is not used in the 0.2.1 version of the interpreter.

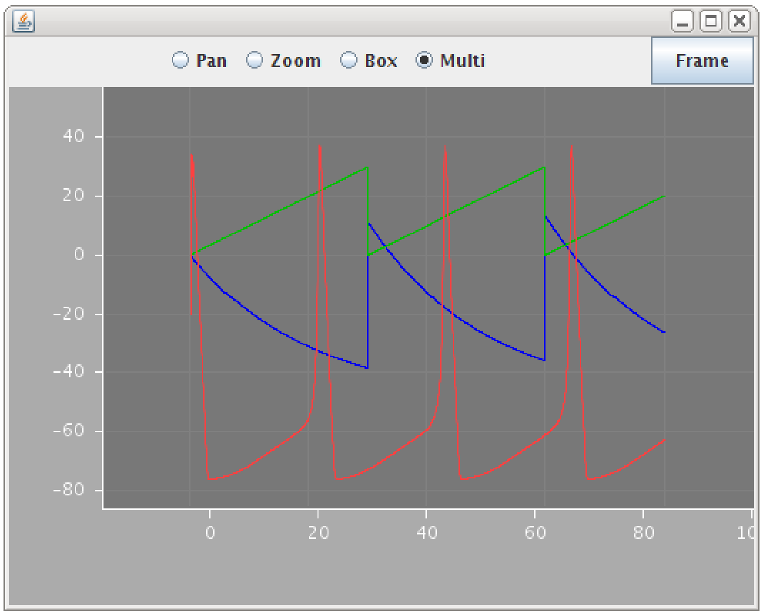
Note that the **KineticScheme** element doesn't say anything about what the outputs are. All it does is control the occupancy state variable in the state elements. The interpretation of these quantities is specified in the normal way with the two **DerivedVaraible** declarations. No special elements are needed in the scheme itself.

To actually use these models we need cell and population elements to link them all together. There is nothing new here - it all works just as for HH channels. The rest of the example4.xml file is:

```
<ComponentType name="ChannelPopulation">
    <ComponentRef name="channel" type="KSChannel" />
    <Parameter name="number" dimension="none" />
    <Parameter name="erev" dimension="voltage" />
    <Requirement name="v" dimension="voltage" />
    <Behavior>
        <DerivedVariable name="channelg" dimension="conductance" select="channel/g" />
        <DerivedVariable name="geff" value="channelg * number" />
        <DerivedVariable name="current" value="geff * (erev - v)" />
    </Behavior>
</ComponentType>
```

```
<ComponentType name="KSCell">
    <Parameter name="capacitance" dimension="capacitance" />
    <Children name="populations" type="ChannelPopulation" />
    <Parameter name="injection" dimension="current" />
    <Parameter name="v0" dimension="voltage" />
    <Behavior>
        <OnStart>
             <StateAssignment variable="v" value="v0" />
        </OnStart>

         <DerivedVariable name="totcurrent" dimension="current" select="sum(populations[*]/
current)" />
        <StateVariable name="v" dimension="voltage" />
        <TimeDerivative variable="v" value="(totcurrent + injection) / capacitance" />
    </Behavior>
</ComponentType>


<KSCell id="kscell_1" capacitance="1pF" injection="1pA" v0="-60mV">
    <ChannelPopulation channel="na1" number="600" erev="50mV" />
    <ChannelPopulation channel="k1" number="180" erev="-77mV" />
</KSCell>


<Network id="net1">
    <XPopulation id="kspop" component="kscell_1" size="1" />
</Network>


<Simulation length="80ms" step="0.07ms" target="net1">
    <Display timeScale="ms">
        <Line quantity="kspop[0]/v" scale="mV" color="#ff4040" />
    </Display>
</Simulation>
```

When run, this produces:



There are clearly some initialization issues but the basic Behavior is the same as the PSICS version of this model.

9 References and paths

The `ChannelPopulation` type used earlier repeats a common model specification error in that it makes the reversal potential of a population of channels a parameter of the population (often it is made a parameter of the channel specification, which is equally bad):

```
<ComponentType name="ChannelPopulation">
    <ComponentRef name="channel" type="KSChannel" />
    <Parameter name="number" dimension="none" />
    <Parameter name="erev" dimension="voltage" />
</ComponentType>
```

In fact, of course, the reversal potential is not a property of a channel population, or of a channel. It depends on the environment the channel is put in and the ions it is permeable to. But, it is needed in the Behavior specification for the population so just putting it in as a parameter solves the immediate problem. In the process, however, it introduces the potential for easily creating contradictory models, by, for example setting different reversals for populations of the same type of channel.

A much better approach is to let the channel just say what it is permeable to. Some other element in the model can define the membrane reversal potentials for different channels, and the channel population object should then look up the relevant value for the permeant ion of its channel. This provides a cleaner expression of what is there, removes redundancy and lowers the entropy of the model specification.

The following three types are sufficient to provide a simple framework to centralize the definitions of species and reversal potentials on one place:

```
<ComponentType name="Species">
    <Text name="name" />
    <Parameter name="charge" dimension="none" />
</ComponentType>

<ComponentType name="Environment">
    <Children name="membranePotentials" type="MembranePotential" />
</ComponentType>

<ComponentType name="MembranePotential">
    <ComponentRef name="species" type="Species" />
    <Parameter name="reversal" dimension="voltage" />
</ComponentType>
```

Once these are available, they can be used to define some species, and to create an environment component that sets their reversal potentials:

```
<Species id="Na" name="Sodium" charge="1" />
<Species id="K" name="Potassium" charge="1" />
<Species id="Ca" name="Calcium" charge="1" />


<Environment id="env1">
    <MembranePotential species="Na" reversal="50mV" />
    <MembranePotential species="K" reversal="-80mV" />
</Environment>
```

The next step is to add a species reference to the channel type, so that channel definitions can say what species they are permeant to.

```
<ComponentType name="KSChannel">
    <Parameter name="conductance" dimension="conductance" />
    <ComponentRef name="species" type="Species" />
    <Children name="gates" type="KSGate" />
    <Behavior>
        <DerivedVariable name="fopen" dimension="none" select="gates[*]/
fopen" reduce="multiply" />
        <DerivedVariable name="g" dimension="conductance" value="fopen * conductance" />

    </Behavior>
</ComponentType>
```

Finally the channel population type needs modifying to add a derived parameter that addresses the reversal potential from the membrane properties:

```
<ComponentType name="ChannelPopulation">
    <ComponentRef name="channel" type="KSChannel" />
    <Parameter name="number" dimension="none" />
    <Requirement name="v" dimension="voltage" />
    <DerivedParameter name="erev" dimension="voltage" select="//MenbranePotential[species
 = channel/species]/reversal" />
    <Behavior>
        <DerivedVariable name="channelg" dimension="conductance" select="channel/g" />
        <DerivedVariable name="geff" value="channelg * number" />
        <DerivedVariable name="current" value="geff * (erev - v)" />
    </Behavior>
</ComponentType>
```

This introduces a new construct, the `DerivedParameter` specification that defines a local parameter "erev" to hold the quantity from the specified path:

```
 <DerivedParameter name="erev" dimension="voltage" select="//MenbranePotential[species =
 channel/species]/reversal" />
```

The path here uses XPath like syntax operating on the component tree in the model. In this case, it finds all the elements of thpe MembranePotential in the model. The predicate selects the one for which the species is the same as the species referred to from the channel used for this population. Finally, it takes the "reversal" parameter from the membrane potential component. This is made locally available as the parameter "erev".

The Behavior of this model is exactly the same as example 4. The full model including both the type definitions and the components is included below.

10 User defined types for simulation and display

Up until now, the examples have used a set of simple **`Simulation`**, **`Display`** and **`Line`** constructs without explaining how they are defined. This shows what is needed in the **`Behavior`** block to let the user defined types to specify that they actually define a runnable simulation or settings that can be used to display results.

This means that the user can select their own names for the different parameters required for a simulation, and, more importantly, simulation and display attributes can be added to existing type definitions to make multi-faceted type definitions that can both be run on their own or as part of a larger simulation.

Example 6 shows two new elements that can be used in the **`Behavior`** block, **`Run`** and **`Show`** as illustrated in the following user-defined type that defines a simulation:

```
<ComponentType name="Simulation">
    <Parameter name="length" dimension="time" />
    <Parameter name="step" dimension="time" />
    <ComponentRef name="target" type="HHCell" />
    <Children name="displays" type="Display" />
    <Behavior>
        <StateVariable name="t" dimension="time" />
        <Run component="target" variable="t" increment="step" total="length" />
        <Show src="displays" />
    </Behavior>
</ComponentType>
```

The 'component' attribute of the **`Run`** element specifies which parameter of the type contains the reference to the component that should actually be run. The 'step' and 'increment' attributes specify the parameters that hold the timestep and total runtime. The 'variable' attribute is for future use - at present, the independent variable is always 't'.

A **`Run`** element can be added to the Behavior block in any type definition to make it independently runnable.

Running a simulation without any output is rarely much use, so there are two futher elements that can be included in the **`Behavior`** block: **`Show`** and **`Record`**. The 'src' attribute of the **`Show`** element points to the components that should be shown. These in turn can contain other **`Show`** elements but eventually everything pointed to by a **`Show`** element should contain one or more **`Record`** elements. These specify what will actually be sent as output. They have the path to the variable as the 'quantity' attribute, its scale as the 'scale' attribute and the line color for plotting.

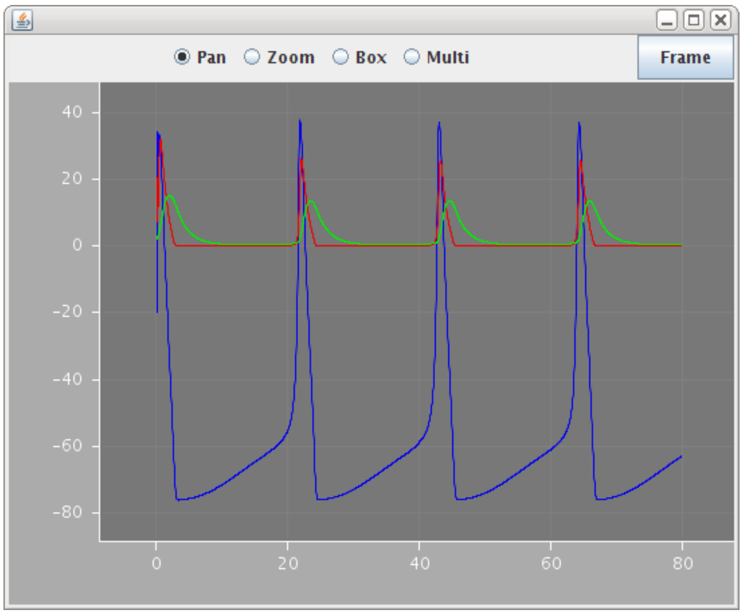
The following two types show one way that these can be combined to allow the user to express a display object containing one or more lines.

```
<ComponentType name="Display">
    <Parameter name="timeScale" Dimension="time" />
    <Children name="lines" type="Line" />
    <Behavior>
         <Show src="lines" scale="timeScale" />
    </Behavior>
</ComponentType>

<ComponentType name="Line">
    <Parameter name="scale" Dimension="*" />
    <Text name="color" />
    <Path name="quantity" />
    <Behavior>
        <Record quantity="quantity" scale="scale" color="color" />
    </Behavior>
</ComponentType>
```

Once these have been defined, a component can be constructed that uses them as follows:

```
<Simulation id="sim1" length="80ms" step="0.05ms" target="hhcell_1">
    <Display timeScale="1ms">
        <Line id="V" quantity="v" scale="1mV" color="#0000f0" />

        <Line id="Na_q" quantity="NaPop/geff" scale="1nS" color="#f00000" />
        <Line id="K_q" quantity="KPop/geff" scale="1nS" color="#00f000" />

    </Display>
</Simulation>
```

When run, this produces the output shown below:



Note how the scale attributes are set to 1mV and 1nS for the different lines so that they show up on the same axes.

11 User defined types for networks and populations

This example shows how the standard component type structures can be used to declare components for simple networks. The following three definitions allow networks to be constructed containing fixed size populations of a particular component type.

```
<ComponentType name="Network">
    <Children name="populations" type="Population" />
    <Children name="connectivities" type="EventConnectivity" />
</ComponentType>


<ComponentType name="Population">
    <ComponentRef name="component" type="Component" />
    <Parameter name="size" dimension="none" />
 </ComponentType>



<ComponentType name="EventConnectivity">
    <Link name="source" type="Population" />
    <Link name="target" type="Population" />
    <Child name="Connections" type="ConnectionPattern" />
</ComponentType>
```

The harder part is to provide elements in the **`Behavior`** blocks to express what should be done with components based on these types. The Network element doesn't pose any problems because the default behavior on instantiation will do the right thing: it will instantiate each of the child populations and EventConnectivity elements.

But the population element needs to say that its instantiation involves making 'size' instances of the component referred to by the 'component' reference, where 'size' is the value supplied for the size parameter in a component specification. This can be done by including a **`Build`** element inside the **`Behavior`** block:

```
<ComponentType name="Population">
    <ComponentRef name="component" type="Component" />
    <Parameter name="size" dimension="none" />
    <Behavior>
        <Build>
            <MultiInstantiate number="size" component="component" />
        </Build>
    </Behavior>
</ComponentType>
```

The **`MultiInstantiate`** specification says that there should be 'size' instances of the component referred to in the 'component' parameter created when the model is built. This overrides the default behavior. [TODO: what is the **`Build`** element content corresponding to the default behavior?].

This serves to create some rather simple populations. More complex specifications, such as putting one instance at each point of a grid satisfying a particular constraint could be handled via first declaring elements to form the grid, and then using selectors that pick the points in the population element to actually put the cells at [its not clear to me how much more would be required to make this work, other than implementing proper xpath-like selectors].

The following three types define a general connectivity structure with an abstract **`ConnectionPattern`** type, and a specific instance for All-All connectivity.

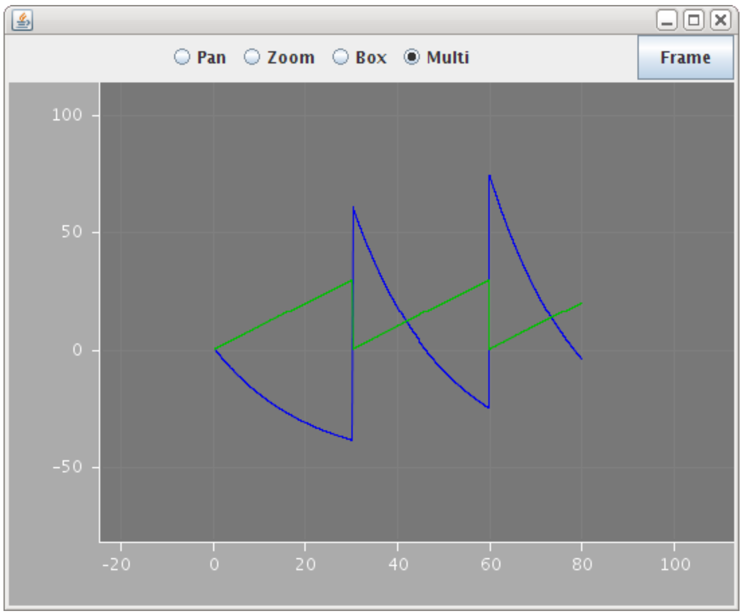
```
<ComponentType name="EventConnectivity">
    <Link name="source" type="Population" />
    <Link name="target" type="Population" />
    <Child name="Connections" type="ConnectionPattern" />
</ComponentType>

<ComponentType name="ConnectionPattern">
</ComponentType>


<ComponentType name="AllAll" extends="ConnectionPattern">
    <Behavior>
        <Build>
            <ForEach instances="../source" as="a">
                 <ForEach instances="../target" as="b">
                      <EventConnection from="a" to="b" />
                 </ForEach>
            </ForEach>
        </Build>
    </Behavior>
</ComponentType>
```

The **`Build`** element in the **`AllAll`** pattern uses a new **`ForEach`** construct and the **`EventConnectin`** element from before. The **`ForEach`** element operates selects each instance matching its 'instances' attribute, and applies the enclosing directives, much in the same way as **`for-each`** in XSL. The proof of concept interpreter also has **`Choose`**, **`When`** and **`Otherwise`** elements that operate much like their XSL equivalents, although these are not used in this example.

With these definitions in place, a network simulation can be defined with the following:

```
<Network id="net1">
    <Population id="p1" component="gen1" size="2" />
    <Population id="p3" component="iaf3cpt" size="3" />

    <EventConnectivity id="p1-p3" source="p1" target="p3">
        <Connections type="AllAll" />
    </EventConnectivity>
</Network>


<Simulation id="sim1" length="80ms" step="0.05ms" target="net1">
    <Display timeScale="1ms">
        <Line id="gen_v" quantity="p3[0]/v" scale="1mV" color="#0000f0" />
        <Line id="gen_tsince" quantity="p1[0]/tsince" scale="1ms" color="#00c000" />
    </Display>
</Simulation>
```

The output when the model is run is shown below, followed by the full listing.


Pan
Zoom
Box
Multi
Frame
100
50
0
-50
-20
0
20
40
60
80
100

12 Regimes in Behavior definitions

This example introduces the `Regime`, `Transition` and `OnEntry` elements within a Behavior block. Rather than having a single state instance, the entity can be on one of the defined regimes at any given time. The `Transition` element occurring inside a condition block serves to move it from one regime to another. The `OnEntry` block inside a regime can contain initialization directives that apply each time the entity enters that regime.

```
<ComponentType name="refractiaf">
    <Parameter name="threshold" dimension="voltage" />
    <Parameter name="refractoryPeriod" dimension="time" />
    <Parameter name="capacitance" dimension="capacitance" />
    <Parameter name="vleak" dimension="voltage" />
    <Parameter name="gleak" dimension="conductance" />

    <Parameter name="current" dimension="current" />
    <Parameter name="vreset" dimension="voltage" />
    <Parameter name="deltaV" dimension="voltage" />
    <Parameter name="v0" dimension="voltage" />

    <EventPort name="out" direction="out" />
    <EventPort name="in" direction="in" />

<Behavior>
   <StateVariable name="v" dimension="voltage" />
   <OnStart>
    <StateAssignment variable="v" value="v0" />
   </OnStart>

   <Regime name="refr">
      <StateVariable name="tin" dimension="time" />
      <OnEntry>
         <StateAssignment variable="tin" value="t" />
         <StateAssignment variable="v" value="vreset" />
      </OnEntry>
      <OnCondition test="t .gt. tin + refractoryPeriod">
          <Transition regime="int" />
      </OnCondition>
   </Regime>

   <Regime name="int" initial="true">
      <TimeDerivative variable="v" value="(current + gleak * (vleak - v)) /
 capacitance" />
      <OnCondition test="v .gt. threshold">
         <EventOut port="out" />
         <Transition regime="refr" />
      </OnCondition>
      <OnEvent port="in">
        <StateAssignment variable="v" value="v + deltaV" />
      </OnEvent>

   </Regime>
</Behavior>

</ComponentType>
```

Full listing:

13 Status, problems and future requirements

This is only a proof-of-concept implementation of the rudiments of a specification language. However, it is extensive enough to define a range of models including a variety of synapses and simple networks. This should help inform where further capabilities are needed. Some such areas are described below.

Status

This proof of concept prioritizes various things that other specification languages treat as optional extras to be included in some distant version. In particular:

- There is a proper model for dimensions and units. Units are not just strings: they refer to a structured entity which specifies the dimensionality in terms of Mass, Length, Time and Current (you can't express luminous intensity as yet).
- All equations are checked to make sure they are dimensionally consistent.
- It supports extension and specialization among ComponentTypes (eg "this ComponentType has all the properties of that one and a few extras" or "it has all the properties of that one except some are set to particular values so the user only needs to set the rest").
- It supports component prototyping ("this model element is like that one but these bits are different").
- There is a clear distinction between ComponentTypes (categories of thing), components (a thing with a particular set of parameter values) and and the model that is run (any number of instances corresponding to each set of parameter values).

Notably, it does not have any support at present for **user-defined functions.** Somewhat surprisingly, it is possible that these may not be needed. See for example the implementation of the conventional HH model. A priori I would have expected any compact expression of this model to require user defined functions, but it doesn't use any functions and is still relatively compact. One could, of course, introduce a generic functions to express the functional form of, for example, the sigmoid used in the example, but it is not clear that this would make it more compact, readable or of lower entropy. At some stage, an external reference to a generic case is actually of higher entropy than a concise local expression with no other dependencies.

Technical issues

A number of technical issues exist with the specification and the interpreter. Some should be straightforward to resolve. Others may take more work.

- `DerivedVariables` currently require a dimension even though this should be deducible from their target
- Paths in derived variables use simple expressions, but only the simplest forms are supported by the interpreter. It needs a smarter XPath like grammar and support for this in the interpreter*.
- You can't define and use functions yet
- The numerics are trivial: it could do with smarter numerics and code generation. Janino would be a good choice (as used with Catacomb) to dynamically compile component Behaviores.
- Error reporting is somewhat cryptic and full of stack traces.

* With respect to accessing variables on the running simulation, it may be that a good solution would be to (virtually) expand the component definitions to a full XML tree and use genuine XPath expressions over that tree. This could be more difficult than it sounds because of constructs like that in example3 which dynamically instantiate

component instances (inserting synapses in this case) that are not defined in the component hierarchy itself. This may still work OK because the container for the instances is still there in the xpath, but a path to its contents won't resolve until after a model is built. On the other hand, since the focus here is on synapse modeling, not on large networks but it is probably reasonable to map the instantiated model to xml and use a standard XPath processor on that.

On another, related, point. Excluding visualization, parsing and file utilities, the current interpreter is about 4000 lines of Java (7000 with the parsers). I'd guess that, thanks largely to XPath, a model could be mapped to an XML representation of the runnable instantiation via XSLT with a similar, possibly smaller, amount of XLST. I'm not sure if this would be useful, but it would be very interesting to know. In general it seems that a good rule of thumb is that a specification such as this shouldn't include anything that can't be processed relatively easily in XSLT. If a need for such a thing arises, then it could suggest that the concept should be expanded into a "more declarative" form until it can be handled by straightforward XSLT.

Other problems

A system allowing user-defined types can go wrong in a number of ways. It could fail to work at all. It could prove too hard to use for anyone to bother. It could be formally powerful but too complicated for anyone (else?) to write an interpreter. It could yield model representations that are too messy to appeal to users (high entropy models). It could make it too complicated to do simple things that users expect to be simple. It could force people to think in an unfamiliar way to the extent that they choose to do something else. It could end up as just another programming language.

The last point is a particular concern. After all, a programming language is a pretty powerful user-defined type system: the thing that differentiates it from a model specification language is precisely the restrictions in the latter. If you keep taking restrictions away, at some stage it ceases to achieve other objectives.

Of these, the most likely pitfalls here seem to be that it could require users to think in an unfamiliar way and it could become too complicated for anyone to write an interpreter. Both of these issues relate to the three-layer structure involving types, components and instances (for comparison, SBML just has a 1 layer structure: models are the same as state instances). As far as I can see, three layers are the minimum for a low entropy model description capable of expressing the type of models that need to be expressed but I'm sure others disagree. The main counter-contender seems to be the NeuroSpaces approach with a smooth (rather than layered) hierarchy which makes a seamless transition from type to instance within a single layer by using prototyping throughout (rather than two class/instance divisions as here).

Miscellaneous observations

Layers

The need for user defined types in NeuroML parallels in some ways the goals of NineML. However, NineML is layered by design. LEMS is slightly layered, but not very: one could compare the structure available for defining types to the NineML abstraction layer, and the structures for using them to the user layer. However, looking at the list of elements (ignoring the deprecated bits), 95% is to do with defining types, and only one paragraph describes how they are used. If someone only wanted to use types, they would also need some of the information in the types section, such as the syntax of path expressions, so there would be a little more than one paragraph to a "user layer" specification, but it would still be an extremely short document. On the other hand, every example defines a new bunch of types: if *these* were included as part of the user level specification, then it could rapidly become very large. But a more natural place for these seems to be some catalog of type definitions rather than a specification document. There is scope for selecting a preferred set of types (eg for HH or Kinetic Scheme channels) so you don't get a proliferation of similar but incompatible models, but the best mechanism for doing this is not clear.

Single element type in the user-layer

Given the simplicity of the model specification layer, (defining components rather than types), where everything is a component or a parameter value, it could be argued that there should be some segregation into different component types (eg things that produce spikes, things that define connectivity etc). For convenience, I started with custom types for simulation and display elements but rapidly got rid of them. They are slightly easier to implement as custom types, but insidious problems keep cropping up. Eg, the runtime in a simulation specification element should have dimensions time, but specific dimensions, such as time are only defined in component-space, not type-space so you can't actually say that a hard-coded component has a parameter with a particular dimensionality. Issues like this strongly suggest that everything in a model should just be a "component" corresponding to a particular user defined type - no special cases. For expressive convenience, though, models don't have to be written "<Component type="MySynapse" .../>, but simply as <MySynapse .../>. This proves to be simple to implement and makes the models much more readable. But it would also cause confusion if there were any elements allowed that *weren't* the names of user-defined types - another reason to make everything a component.

Specific types for networks and populations not needed

As well as starting with a hard-coded simulation element, the early examples also have hard-coded network and population elements. I had previously thought that this would be hard to avoid but it turns out not to be as bad as expected. The **`Build`** element with **`MultiInstantiate`** and **`ForEach`** children proved sufficient to replace the (albeit trivial) hard-coded population and connectivity elements with user defined types. Whether this extends to more subtle ways of specifying networks remains to be seen, but I suspect that by adding a few more constructs like **`ForEach`** (Choose, When, Otherwise, If etc) and using the selection rules effectively one could do most of what would be needed. This relates to the debate as to whether the NineML user layer needs network-specific constructs, and would tend to suggest that it doesn't.

Joys of dimension checking

Even from limited experience making up the toy models, the automated dimension checking for equations and transparent unit handling is invaluable. It cuts out a whole family of time-consuming potential errors. It is just a shame there isn't support for this kind of thing in IDEs yet...

What to do if you get beyond point process models

Everything here is to do with point models without any spatial extent. While it would be easy to define types to represent, say, a cell morphology, I have no clue how to attach a Behavior to them in a meaningful way. For a morphology, one could imagine associating a membrane potential state variable with each section, and computing resistances to the neighboring sections, but that would be some kind of crime against numerical analysis since it would actually be representing a different model (one where all the capacitance was at the midpoints of the segments). A correct approach would involve introducing Behavior elements for scalar fields and geometrical volumes but this could well increase the complexity to the point where it becomes useless to try writing an interpreter. Perhaps an intermediate route via magic tags to say for example "this structure can be approximated by a 1D scalar field satisfying equation ... etc" might work.

Retrofitting existing component types

After the false start with hard-coded elements for specifying networks and display settings, it proved surprisingly easy to retrofit user defined types to these elements. In general, this was not even restricted to creating equivalent functionality, but it could use exactly the same xml. If this applies in general, then it might be possible to retrofit more extensive building-block languages such as NeuroML or PSICS with user defined types. For the point models this could let a generic simulator run them. It wouldn't help with more complex, spatially extended, models but it would at least make it rather easier to read and process them. In a sense, the type definitions would just be acting like a rather restricted domain-specific schema.

Correspondence to XSL

If one ignores "apply-templates", the structures used for building populations and connections are more than a little reminiscent of XSL. In a way this is hardly surprising, given that they are both about processing nodes from a tree. It also suggests that maybe using XSL and XPath directly might work, and would avoid gradually introducing equivalents to half the element types in XSL as they prove to be needed.

Correspondence to CSS

Somewhat surprisingly, there isn't any. None of the examples seemed to fit better with a css-like pattern than with an XPath like one. Perhaps this is because you generally need to be sure which nodes you will hit instead of the rather heterogeneous matching that css is best for.

Orientation

Model description languages differ markedly in where their focus lies and how they value (or disregard) particular features. Such features include how important it is for model specifications to be:

- Concise
- Minimally redundant
- Low entropy (see below)
- Machine readable
- human readable
- Writable from existing simulators
- Writable by hand
- Mappable onto existing simulators
- Language independent
- Declarative
- Mathematically oriented (dimensionless variables and equations)
- Physically/biologically oriented (everything dimensional)

Low entropy

The term "entropy" is used as a loose analogy. The idea is that a model as conceived by a modeler or as described in a paper is highly structured. The quantities occurring in it are physical quantities (voltages and times rather than just numbers) and the structures are concise, hierarchical and minimally redundant. This is a low entropy representation. As the model gets converted into something that can be run on a computer, most of the structure is removed. Dimensional quantities get divided by units to provide dimensionless numbers and mechanistic concepts get converted to equations. It goes through a state of being a bunch of state variables and equations and eventually ends up numerical code implementing state update rules. This is the high entropy end. Models that are only available as compiled executables are the extreme high entropy end. Those that are only available as c-code are a close second which can only be converted to low entropy forms by extensive manual curation.

You can automate the process of turning a low-entropy representation into a runnable model, but in general you can't automatically get back to a low entropy representation from a higher entropy one. Simulators vary in how well they represent and preserve low entropy models, but, particularly older simulators tend to increase the entropy from the start and the only internal model representation used is often of rather higher entropy than the model representation created by the modeler. For example a modeler might be forced to render their model dimensionless before getting it into the simulator. The units they used to do this would probably be there in comments in the source files, but they are not part of the internal state of the simulator so it is unable to write out a low entropy model.

This proposal is all about expressing and protecting low entropy representations. These are the most valuable representation of a model because they can readily be turned into a variety of higher entropy representations as used by different simulators. Note that this low entropy focus may not be suitable to a *model exchange* language such as NineML which is intended to be writable by existing simulators. For that a medium entropy representation is probably required.

> For those familiar with software engineering, the entropy discussion is essentially a > variant of the DRY > (Don't Repeat Yourself, or 'DIE': Duplication Is Evil) principle in software design. >

Mathematical v. Physical/biological

There are two issues here. One is whether a bundle of state variables and equations is enough to make a model. Mathematically, of course, this is all there is to many models, but scientifically, such a representation is of relatively high entropy since the structure, hierarchy and relations have been lost. The second question is about quantities that go into a model. Are they numbers or are they physical quantities? The distinction is that for a mathematical model one might say (as eg CellML does) "v is a number which represents a voltage measured in milliviolts" Ie, its what you get when you take a voltage and divide it by another voltage which is 1mV. For a physical model, you'd just say "v is a voltage" and leave it at that. Then v is a rich quantity with magnitude and dimensions.

If your modeling system takes the first approach, it can force the user to render quantities dimensionless themselves or can provide some support for unit conversions. But in either case, the quantities in the equations that the modeler enters are just numbers.

In the second approach, the quantities occurring in equations are dimensional quantities. This is the norm in written model descriptions but it is generally not the norm in model description software. Being focused on turning things into executable code (involving bare numbers) the latter tends to dispense with dimensionality as soon as possible. This seems unfortunate because premature non-dimensionalisation opens up all sorts of cans of worms for the modeler and indeed for the software developer that simply don't need to be opened up. Sticking with dimensional quantities throughout the model description phase makes most of these problems vanish.

Incidentally, this is related to the xml construct often seen in model specifications where a quantity has both a value and a unit as in '<mass value="3" unit="kg"/>'. This suggests that somehow the mass of the item, (the 'value' of its mass) was '3' and that the mass itself has a unit, rather than its mass (or the 'value' thereof) being '3kg' as in normal usage. In fact, of course, neither the '3' or the 'kg' are attributes of the mass. Exactly the same quantity could be expressed with '3000' and 'g' or '3E-6' and 'Ton'. Neither is meaningful in on their own, so ideally this looks to call for an XML datatype.

Objectives

In the light of the above discussion, this proposal prioritizes:

- Low entropy models

- Human readability
- Human writability
- Physical/biological (as opposed to mathematical or computational) model specification
- Conciseness and minimal redundancy

A consequence of these priorities is that it is probably going to be very hard to automatically export models in this format from existing simulators unless they already have a low-entropy internal representation. In general, it will involve rewriting them by hand.

Canonical form

This format is not intended as a canonical form for a model, although there is a clear need for such a model specification format. Rather, it is better to think of it as lightweight XML user interface to a nascent canonical form. Canonical forms are inevitably hard to work with directly (eg, a canonical form for model specification should probably use MathML which requires an intermediate tool to read or write) and the present form makes a much simpler structure within which to develop and explore model specification capabilities. Once a suitable structures have been arrived at, the corresponding canonical form can be specified.

It is intended that a relatively simple XML mapping should map losslessly between this format and the canonical form with a couple of constraints. The present format allows multiple ways to express the same model. For example, the same quantity can be expressed with different magnitude units and elements can be written in a number of different ways. Such variants of the same model should all map to the same canonical form. For the mapping to be invertible, the additional information (such as that the original value of a current was given in nA for example) will have to be stored in metadata in the canonical form of the model.

More on canonical forms.

Comparison with other systems

There are strong parallels in VHDL. The hierarchical components proposal for SBML level 3 looks to be heading towards the same end point from a different direction. There are also comparisons to be made with the facilities for modular model representation in CellML 1.1. Like SBML, it is arriving from the other end, with a substantial body of models expressed in a standalone, medium entropy form and a curation process to abstract out modules that can then be referenced from several models.

There are also close parallels with NineML, which may, ideally, provide a standardized format for losslessly writing and re-reading models expressed in LEMS.

Comparison with 'building-block' languages

The great thing about a building-block language is that a model can be reliably and relatively easily mapped onto efficient code to execute it. This is not the case with the present proposal or with other general systems where the user can define their own equations. One way round this is to develop smarter symbolic algebra capabilities so that efficient numerical implementations can be generated from the equations. This could work in some cases. It is hard to see it working in all cases. Another way round it is for an implementation to spot structures it recognizes and map them to efficient hard-coded implementations, but to keep the capability to run new, unrecognized, structures (albeit more slowly). However, I'm not aware of any implementations that actually do this yet, so it remains to bee seen whether it is a workable approach.

If a system in which modelers develop new models and simulators gradually add hard-code support for the most popular ones is to be made to work, then there must be a strong incentive for reusing an existing type (that can then be recognized by the simulator) rather than re-expressing the whole model from scratch. This is probably more of an issues for simulators and model sharing infrastructure than for the language itself, although, of course, the language itself must provide capabilities to do this cleanly.

XML format

It is designed to be read and written by hand (as well as machines). Nevertheless, the parsers used for the XML and the expressions are among the more straightforward parts of the implementation. The most common alternative is to to pre-parse the expressions and make the XML store the parse tree. This would make it easier to process in a simulator (no need for an expression parser). Similarly, dimensional quantities could be split into a number attribute and a unit attribute which would also make it slightly easier to process. However, these benefits are slight, and would only accrue to the small number of developers writing simulators. In the meantime, every modeler would have to write quantities out the long way and find a way to generate the xml parse trees for expressions. At the very least, an interpreter should provide tools for the user to do this, so the user can write input as a normal expression. But if the interpreter can do this, why standardize on the unreadable post-parsed form rather than the concise form written by the user? It has been suggested that this is necessary to avoid ambiguities in what a model means, but I suspect that is more of a hypothetical danger than a real problem, given all the other thins that can go wrong with a model specification.

Finally, for those who want it, it is straightforward to convert models in this format to pre-digested element-only XML with parse trees instead of expressions. Indeed, the interpreter will generate this format if called with the '-p' qualifier in 'java -jar lems-x.x.x.jar -p model.xml'.

14 Possible extensions

Repeatable components

In a sense, the `Dimension` and `Unit` elements operate as model-independent global quantities since they may be restated in different files without problems as long as the definitions are identical.

There is probably a case for a user-definable type with similar properties. The canonical application would be in defining the ionic or molecular species present in a simulation. For example, in specifying that a channel is permeable to Na+ ions, or blocked by TTX, it is necessary to match up the occurrences of these quantities in the channel definition with those in the environment of the cell model in which the channel is used.

This can be done with the `ExternalQuantity` element as in example 5, but that currently requires all references to a global component to point to the *same* component since a component cannot be repeated. This prevents there being fully independent self contained channel definitions as they would need to share the same species file. It should be easily resolved by allowing a flag on type definitions to say that duplicate components are allowed for particular types as long as the definitions are identical.

Scalar fields

Conceptually, you need a number of things:

- Something like a `FieldVariable` declaration in a `Behavior` analogous to a `StateVaraible` but spatially extended.
- Some notion of geometry
- A mechanism for extracting a value for a `FieldVariable` at a point for use in a point process model. this could involve sampling, interpolation or integration over some kind of kernel
- A mechanism for associating point models with positions in a scalar field

Of these, the geometry and positioning is the hard part. Whether an implementation could actually do something useful with such a model is another question entirely.