Meeting Minutes 28 September 2000

Author:
Warren Hedley (Bioengineering Institute, University of Auckland)
Contributors:
Scott Lett (Physiome Sciences Inc.)
Melanie Nelson (Physiome Sciences Inc.)
Poul Nielsen (Bioengineering Institute, University of Auckland)

1 Summary

A first for the CellML 2000 meeting minutes — they are being written in Princeton, New Jersey, as Warren is currently visiting Physiome, to explain his special brand of obfuscated documentation to the Physiome staff, who are on the front line of CellML implementation. These meeting minutes cover meetings between Melanie, Scott and Warren about component re-use, low-level XML re-use, math and extensions to CellML, as well as a teleconference back to half of the remaining CellML team in Auckland to confirm that the 28 September version of the CellML specification is indeed a fair summary of CellML thinking up to this point.

2 Component Re-use and Class Definition

I think the main problem in meetings between members of the Auckland and Physiome CellML development teams is terminology. Combine engineers, programmers and database people and you will quickly find that words like "class" and "object" can mean many very different things. Perhaps the most confusion has been caused by what has, up to this point, been known as "*component re-use*", which featured in the now legendary "CRIM" documents of April 2000. As of September 28, the phrase "*component re-use*" defined in that document (named for the sake of a shocking acronym) has been re-purposed for a more CellML oriented problem. What was formerly called "*component re-use*" will now be referred to as "*Low-level XML Re-use*" (look, no acronym!), and this is discussed in the next section. (Any takers on X-DRAM: XML Data Re-use and Modification?)

So before we go any further, let's come up with some standard definitions for our main overloaded terms.

- **Component**

  OK, from now on I'm going to try and use the word component to mean an actual component *instance* in a CellML model. A component might be something like "*the fast sodium channel encapsulated in the cellular membrane in the Luo-Rudy II model*." A component has nothing to do with a class.

- **Component Re-use**

  Where multiple components share the same functionality, it is convenient to define this functionality once and then re-use it as appropriate. In some cases we want to just duplicate a component's properties — to achieve this we would typically use the "*Low-level XML Re-use*" mechanism described in Section 3. In some cases we want to signal to the processing application that it might be able to optimise code generation based on the assumption that multiple components are similar — to achieve this, we rely on class definition.

- **Class Definition**

  First and foremost, "*class*" now has nothing to do with ontology. We are using *class* in an object-oriented programming like way. A CellML class definition will essentially provide a template for a CellML component, with variable declarations and/or mathematical equations, scripts and reactions, and/or metadata and documentation.

- **Extensibility**

  In object-oriented programming, class A can "*extend*" class B. In general this means that class A has additional functionality, in effect specializing class B. It is usually said that class A "*inherits*" functionality from class B.

The reader deserves a quick explanation of how class definition relates to ontology. The ontology associated with a CellML model will define a set of rules that define how CellML objects may interact. These CellML objects may be referred to by name or by *ontology-class*. In the case of components, ***ontology-class* is not the same as *class***. A component in a model may be based on a class definition called "`LR_fast_sodium_channel`" but belong to the ontology class of "`sub_membrane_components`". The rules defined in the ontology can consequently be completely independent of any class definitions in the CellML document, even though, classes and ontology classes may frequently coincide.

One of the important features that CellML's class definition mechanism must provide is *extensibility*, as defined in the short glossary above. A class definition must be able to extend another class definition, signalling the CellML processor that it contains all of the properties defined in the extended class, as well as any additional properties defined in the current definition. The union of the *inherited* properties and the new properties must still make up a valid set of CellML properties, as if they had been defined as one set.

Note that more advanced object-oriented concepts will not appear in CellML, as the associated implementation complexity far outweighs their usefulness. (I'm thinking of things like Java's `static` variables, and C++'s `virtual` functions.)

When a component is created in the model, it may choose to identify itself as a member of previously defined class. In this case it inherits all of the properties defined in the associated class definition, and it may also explicitly extend that definition by adding its own properties. It is important that CellML contains a mechanism that allows a component to modify some of the properties defined in the class definition — typically the default values of variables, or constants that occur in the math associated with a class.

The perceptive reader might well ask what the advantage is of class definition over a more generic low-level XML re-use system, if every component is free to define properties that aren't in the class definition, in effect creating its own class? CellML separates class definition from low-level XML re-use in its data model purely so that processing applications can take advantage of this mechanism to re-use components efficiently, in the event that components do not extend a class definition. This could be particularly useful for components with functionality equivalent to the mechanisms of CellML '99. To re-visit one of the examples from the CellML '99 website, let's consider the definition of a component representing a Hodgkin-Huxley-type gate, which might be re-used without modification (but with different parameters) at multiple points in a model. An example of the definition of this component is given in Figure 1. If you've never seen this before, the differential equation governing gate variable v, given rate constants a and b is:

$$\frac{dv}{dt} = a(1 - v) + bv \tag{1}$$

The well-read and observant reader may have noticed that the **`<define>`** element is also used in the low-level XML re-use scheme. This is because the functionality that we require in our class definition and referencing system is identical to that required for low-level XML re-use! In the code snippet in Figure 2, two components associate themselves with the class defined in Figure 1, and each sets a different default (i.e., starting) value for the variable `v`.

Note that extensibility can be simply implemented by putting the **`<copy>`** element that references a component inside a `define` element. The additions and/or modifications that are made inside the **`<copy>`** element determine the form of the extensions.

```
<define>
  <component name="hodgkin_huxley_gate">
    <variable name="v" public_interface="out" units="[,,]" />
    <variable name="t" public_interface="in" units="[,,]" />
    <variable name="a" public_interface="in" units="[,,]" />
    <variable name="b" public_interface="in" units="[,,]" />

    <!-- note that namespace below is not official -->
    <math xmlns="http://www.w3.org/1998/Math/MathML/2.0">
      <apply><eq />
        <apply><diff />
          <ci>v</ci><ci>t</ci>
        </apply>
        <apply><minus />
          <apply><times />
            <ci>alpha</ci>
            <apply><minus />
              <cn>1.0</cn><ci>v</ci>
            </apply>
          </apply>
          <apply><times />
            <ci>beta</ci><ci>v</ci>
          </apply>
        </apply>
      </apply>
    </math>
  </component>
</define>
```

FIGURE 1: A class definition that can be used by components that will represent the Hodgkin-Huxley-type equation in a model.

```
<copy
    element="component"
    name_ref="hodgkin_huxley_gate"
    new_name="gate_1">
  <insert_attribute xpath="variable[@name='v']/@value">0.0</insert_attribute>
</copy>

<copy
    element="component"
    name_ref="hodgkin_huxley_gate"
    new_name="gate_2">
  <insert_attribute xpath="variable[@name='v']/@value">1.0</insert_attribute>
</copy>
```

FIGURE 2: Two components are created that reference the class definition from Figure 1, each using low-level XML re-use techniques to set an initial default value of the variable v.

3 Low-level XML Re-use

As a result of the September 28 meeting where component re-use was discussed, the CRIM documents of April 2000 were re-visited and rechristened '*Low-level XML Re-use*'. This document is now available in the documentation section of the CellML developers website (it is too long to include here). In particular, the FieldML and AnatML examples were removed, and more substantial CellML examples were created demonstrating the key concepts.

4 Equations and Scripts

The Physiome development team were asked to review the "Equations and Scripts" section of the 28 September specification, (which is admittedly fairly short on detail), mainly to confirm that Warren wasn't coming up with anything completely ridiculous at this early point. Scott confirmed that restricting scripts to executing instantly with respect to integration of any DEs was a reasonable requirement, and was satisfied in all of their current models. Scott was also happy with the fact that equations need not be ordered within components or globally throughout the model. There are of course problems with cyclical dependencies between variables — resolving this problem is best left to software. Software would of course be allowed to stop processing if it detected this problem.

Given these two factors, it was proposed that it was reasonable to include something like "*for a model with explicit first order ODEs, different software should produce consistent results for a particular timestep, ordering the equations as appropriate*". Poul in Auckland later suggested that what we really meant was "*for a model consisting only of explicit first order ODEs, all the equations should be satisfied simultaneously*".

One problem with the September 28 specification that Scott pointed out involves the modification of variables. Scott suggested that state variables be differentiated from constants at the XML level, re-instating something like the `role` attribute on a variable declaration (from CellML '99). State variables may only be modified by differential equations in the the component to which they belong. We could then have regular variables that may be modified by non-differential equations, and constants that may not be modified at all. Some problems were already apparent with the first restriction however — in the Noble '98 model, for instance there are expressions like:

$$\frac{dCa}{dt} = \begin{cases} C_1 & Ca \geq C_2 \\ C_2 & Ca > C_2 \end{cases} \tag{2}$$

5 CellML extensions

Scott confirmed that Warren's idea of just using XML namespaces[1] to implement application-specific extensions to CellML was reasonable. The CellML specification will define a namespace with which all CellML elements and attributes should be associated. Equations marked up using MathML will be defined in the MathML namespace. Applications wishing to store their own proprietary data within a CellML document may declare their own namespaces. CellML processing applications need only respect elements and attributes in the CellML namespace. A simple example of the definition of a component making use of XML namespaces to define some application specific extensions is given in Figure 3.

It is hoped that CellML processing applications will respect the extension elements and attributes of other applications. If a model is created in application A which adds its own extension elements, and is subsequently edited in application B, it would be polite if application B was kind enough to include

[1] http://www.w3.org/TR/1999/REC-xml-names-19990114

```
<cellml:component
    app:render_corners="100, 100, 400, 400"
    name="membrane"
    xmlns:cellml="http://www.cellml.org/2000/cellML"
    xmlns:app="http://www.software.com/cellml_processor">
  <variable
      name="calcium_current"
      public_interface="out"
      app:colour="blue"
      xmlns="http://www.cellml.org/2000/cellML" />
  <mathml:math
      xmlns:mathml="http://www.w3.org/1998/Math/MathML">
    ...
  </mathml:math>
</cellml:component>
```

FIGURE 3: A simple example of the definition of a component making use XML namespaces to define some application specific extensions. On the **<component>** element, a prefix is used, and it is assumed that all non-prefixed attributes are in the CellML namespace. On the **<variable>** element, the default namespace is used, and this namespace will cascade down to all children of the variable element (there are none in the example). All namespace URIs in this document are imaginary.

application A's extension elements in its output, even if these extension elements are now invalid. The result of this is that applications will need to do some of their own validation of extension data in the event that CellML documents are being read in from a non-trusted location.

Just for the record, Warren had previously considered more complicated mechanisms like that described in Section 14 of the XSLT 1999 Recommendation[2]. This scheme also relies on namespaces to differentiate XSLT elements from extension elements, but defines functions with which a document can check if specific extensions are available (a conforming XSLT processor must implement these functions). XSLT is often thought of as a programming language so this functionality is reasonably important. CellML isn't, so this functionality can be dropped.

6 Auckland Teleconference

A teleconference was scheduled with the remainder of the Auckland team to confirm that they were happy with the September 28 edition of the CellML specification. In fact only Poul Nielsen made it. As tradition dictates, Poul and Warren have to argue stubbornly about something at every CellML meeting, and this was no exception.

Poul suggested that in the Components section of the September 28 spec[3], the term "*variables*" was in fact misleading, because not all the variables could vary (namely, those being mapped from other components). Warren's view on this was that every variable's value can be varied in some component, and when a variable is mapped from that component, you may really just be referencing the original varying variable (depending on the implementation). Hence variable is a reasonable name.

Poul also suggested separating a component into a "*declares*" and a "*body*" section, but was unable to suggest anything that belonged in the "*declares*" section other than variables. At one point Poul suggested

[2] http://www.w3.org/TR/1999/REC-xslt-19991116#extension

[3] http://www.cellml.org/private/really_old_specs/specification_20000928.html#sec_components

re-naming variables "*item*s".

Unless he brings it up again, we can safely assume that these ideas were the result of the stress of having to sit through two days of undergraduate project seminars. Staying awake for these is reasonably tough on the students, but for the lecturers it's just hell, as they are expected to actually grade every presentation, and maybe ask questions.

Poul did point out reasonably that the sections on encapsulation, geometry and grouping should be re-ordered, and it should be made clear that encapsulation and geometry can be thought of as sub-classes of the grouping mechanism. Warren had gone with the original order because of the importance of encapsulation in the general scheme of things. On the next specification re-write, the order of these sections will depend on any new concepts we introduce, where forward references are to be minimised.

7 Outstanding

This section is dedicated to all those people who keep reminding me of things that aren't mentioned in the September 28 edition of the CellML specification. Things like:

- Units.
- Stochastic Math.