Meeting Minutes 19 September 2000

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

1 Summary

Again, this document is not just limited in scope to meeting minutes, but also contains what might potentially be parts of the CellML specification, and some documentation created in preparation for upcoming meetings. Two meetings occurred on Tuesday the 19th of September, the first was a teleconference with Melanie at Physiome, and the second was attended only by the Auckland team mid-afternoon.

2 Physiome Teleconference

2.1 Summary

Prior to the September 19 teleconference, Melanie had submitted a list of priorities for CellML development. The Auckland team was proud to have knocked off the first one (variable role/scope/privileges) with its work from the previous day (see Section 4 of the minutes), a project which actually still required some further work. Grouping, which was the second priority, was suggested as a possible mechanism for handling the specification of reactions in pathway models, and some convergence to the "grouping with root components" scheme presented in Section 3.2 of the September 18 minutes was noted.

2.2 Role / Scope / Privileges

David was still dissatisfied with the solution proposed to the variable role problem in Section 4 of the September 18 minutes. It turned out that this was mainly due to a misunderstanding as to the meaning of the value `none` in the public and private interfaces (note that this has since been clarified in the September 18 minutes.) Essentially, if we explicitly specify that a value of `none` is equivalent to the absence of an interface, and vice versa, we have agreement.

For the record, David's solution was to split up the interfaces completely. At the top level, a component has a public interface which is a list of variables with role `in` or `out`. Next, the component defines its protected interface (the interface to its encapsulated components and their children), which is a list of variables with role `in` or `out`. Note that some of the variables in the private interface may also be in the public interface, in which case only a reference is needed. Note that it would still be illegal for a variable to have role `in` in both interfaces. Finally a component has its private interface, which consists of a list of variables that are only available for use inside the current component. This scheme provides only a subset of the functionality provided by the separation of private and public interfaces from the September 18 meeting minutes.

Of course, it is debatable whether we actually that extra functionality — is anyone ever going to want to have a variable in the public interface that isn't in the private interface? Probably not. But there is no reason not to provide this functionality if it can be done in an obvious and self-consistent manner.

2.3 Adjacentness

One of the more disturbing outcomes of the morning teleconference was the possible demise of the relationship formerly known as *is-next-to*. It appears that there is very little support for the inclusion of this kind of information in the CellML data model, particularly as it wouldn't fit into the grouping scheme discussed in the previous section, and would probably have to be kept as a property of a connection. This kind of topic always brings a lot of emotion to the surface as it is really edging well into the realm of rendering information, not model information, but I still think that it's possible that this is on the model information side of the line.

3 Variable Interface Specification

3.1 Summary

When this section was originally written, it was thought that variable interface specification had been largely agreed on during the morning teleconference. In fact this was not the case, although as usual, the argument revolved around a lack of understanding of the documentation (the original of this section as it happened.) The BS-generation department at Auckland had been busy all morning working on the most obfuscated, least obvious description of what had been agreed on. Luckily this has been fleshed out and elaborated in the current version, with diagrams added. (This doesn't necessarily make the document any clearer, but it does make it longer!)

Note that David persisted in pushing the implementation of variable interface specification described in Section 2.2. Poul was also originally not convinced. As neither could actually point out any advantages over Warren's implementation (described below in Section 3.4), they were eventually forced to concede that the increased functionality and conciseness of Warren's implementation was vastly superior.

3.2 Some Terms

Before attempting to define the interface system for variables in CellML let us define some terms that allow us to easily describe the important sets of components in a network, with respect to a particular component, which will be referred to as the *current* component. The set of all components immediately encapsulated by the current component will be referred to as the *encapsulated set*. The *parent-and-sibling set* refers to the remainder of the components in the model that may be connected to the current component. If the current component is encapsulated inside another component, the *parent-and-sibling set* consists of the parent of the current component along with any other components with the same parent. If the current component is not encapsulated, then the *parent-and-sibling set* consists of all other components that are not encapsulated in the model.

As the above paragraph had caused some problems in its first presentation, some examples will be given with respect to the encapsulation network shown in Figure 1. Table 1 lists the components in the *encapsulated* and *parent-and-sibling* sets with a selected set of components picked as the *current* component.

3.3 Variable Interface Requirements

The variable interface scheme to be used in CellML must be able to satisfy the requirements in the following list.

1. The current component must be able to expose variables to the components in the encapsulated set that are not available to components in the parent-and-sibling set. (It may also be useful to handle the reverse requirement.)

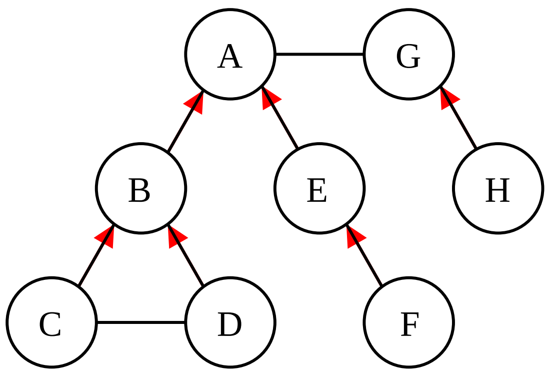

FIGURE 1: A simple model where much of the model is described as encapsulated complexity. Recall that a red arrow on a connection points to the parent in an encapsulation relationship. This model is used to demonstrate the concepts of the *encapsulated* and *parent-and-sibling* sets.

Current Component	**Encapsulated Set**	**Parent-And-Sibling Set**
A	B, E	G
B	C, D	A, E
C	D	B
E	F	A, B
G	H	A

TABLE 1: This table lists the components in the *encapsulated* and *parent-and-sibling* sets for a selected few components from the example network given in Figure 1.

2. The current component must be able to declare variables for use internally that are not available to components in either the encapsulated or parent-and-sibling sets.
3. It must be possible to add complexity (i.e., encapsulated components) to the current component without affecting the interface it presents to the components in the parent-and-sibling set.
4. It must be clear which variables may be modified in the current component. (That is, modifications to the value of a variable whose value is obtained from another component are not allowed.)

3.4 Implementation in the Data Model / XML

A variable has two properties: a public and a private interface. The public interface of a variable describes the interface exposed to components in the parent-and-sibling set, whereas the private interface describes the interface exposed to components in the encapsulated set. Each interface has three possible values: `in`, `out` and `none`, where `none` indicates the absence of an interface.

Considering the public interface first. Variables with a public interface value of `in` must be mapped to a single variable in the sibling set with a public interface value of `out`, or a single variable in the parent of the current component (if it exists) with a private interface value of `out`. Similarly, variables with a public interface value of `out` may (note that this isn't a requirement) be mapped to variables in any components in the parent-and-sibling set with a public interface value of `in`. If a variable has no public interface, it is neither input from or exposed to the components in the parent-and-sibling set.

The private interface is the interface exposed to all components in the encapsulated set. Variables with a private interface value of `in` must be mapped to a single variable from a single component in the encapsulated set with a public interface value of `out`. Variables with a private interface value of `out` may be mapped to any variables from components in the encapsulated set with a public interface value of `in`. A variable may also have no private interface, in which case it is neither input from or exposed to the components in the encapsulated set.

Note that if a variable has no private or public interface, it may only be used in the current component, and is visible to no other components in the model. In order to work out which variables may be modified in the current component, we must check if either the public and private interface has a value of `in`. If so, the variable is declared elsewhere and may not be modified in the current component. If not, the variable belongs to the current component.

The two interface properties of a variable are completely independent with one exception: it is invalid for a variable to have both public and private interfaces with value `in`. An interface with value `in` reflects an un-met need in the current component that must be satisfied — clearly this need can only be met in either the public or private interface but not both.

Just for the purposes of illustration, a possible XML serialization of the variable interface scheme proposed here is given in Figure 2.

4 Grouping

4.1 Summary

It was quickly agreed on during the September 19 teleconference that grouping was by far the biggest and hardest problem left on the agenda. A new proposal from the Physiome team for the description of qualitative information in pathway models led to the promotion of "grouping with root components" (see Section 3.2 of the September 18 minutes) as a very likely candidate for solving this as well as the encapsulation and geometry hierarchy problems.

```
<component
    name="cellular_membrane"
    display_name="cellular membrane">
  <variable
      name="I_Na"
      public_interface="out"
      private_interface="in"
      display_name="sodium current">
    <documentation_text>The sodium current is calculated in an
      encapsulated component and exported out to the rest of
      the model.</documentation_text>
  </variable>
  <variable
      name="conc_Na_i"
      public_interface="in"
      private_interface="out"
      display_name="intracellular sodium concentration">
    <documentation_text>The intracellular sodium concentration
      is declared in the intracellular subspace, but is required
      in the calculation of the sodium current in an encapsulated
      component.</documentation_text>
  </variable>
</component>
```

FIGURE 2: A possible XML serialization of the variable interface scheme described in this section. Please note this is an illustration only, and may yet change significantly.

4.2 Discussion

"Grouping with root components" is probably a bit of a misnomer considering the functionality that we might add to this concept — a better name might be "*grouping with special components*" — but we're just going to settle for "*grouping*" from now on. In our new scheme, a grouping object is used to imply some kind of relationship between a "major" component and a group of "minor" components (note that it's legal to have a relationship with minors in CellML ;-). In the meeting minutes of September 13, it was suggested that the relationship in question could include encapsulation and/or geometry information. To solve our reaction identification problem, we could use a grouping to indicate that the "major" component was the reaction and the "minor" components were the participants.

Note that the grouping and corresponding relationship information would be completely unrelated to the connections under this scheme. Connections would undoubtedly exist between most of the components in any group, and in particular between the major and minor components. (Although I had previously complained that having, for instance, geometric relationships on connections made it hard to find the siblings of any component, one could well complain with this system that it is difficult to check if there is any kind of encapsulation or geometric relationship along a connection. The latter is definitely the lesser of the two evils.)

Encapsulation, geometric and reaction relationships make up the three distinct classes of grouping defined in the CellML specification. A set of rules is defined that limit the interactions between the components that take part in each of these relationships as well as the variables belonging to those components. For instance, the restrictions on components in an encapsulation relationship were defined in Section 2.2 of the September 13 meeting minutes. The restrictions on geometry and reaction groupings have not yet been defined, but these will be addressed in the next version of the specification. There is no reason that for the user not to define other classes of grouping, for which they can define their own set of rules, possibly in an ontology associated with a model. Note that CellML processing software would be free to ignore user-defined grouping classes.

A complete geometry or encapsulation hierarchy is made up of a system of one-level hierarchies. Each level in the hierarchy requires a root component, even when this component may contribute no information to the model. As the modeller may wish to define multiple geometric hierarchies over a single base network of components, a mechanism for associating the geometric relationships at each level of a hierarchy is needed. This is most simply implemented by associating a unique name with the geometric groupings across a hierarchy. Note that a particular grouping could easily be re-used in multiple geometric hierarchies. Although we don't require this functionality for encapsulation and reaction groupings, it could also be made available to the modeller for their own user-defined classes of grouping.

To make the grouping concept even more powerful, user-defined classes of grouping need not even have a major component. This could be useful when the sole purpose of a grouping is to assign a name to an association between components. A good example of this (when you consider how *Adjacentness* was dismissed earlier in this document) might be a grouping class called `is-next-to`, which is used to tell a processor that one component is adjacent to another. The user could define a rule that states that such a group may only contain two minor components and it may also be desirable to define rules relating this class of grouping to the geometry class of grouping.

For a large anatomical network with multiple hierarchies described across the actual functional part of the network, the large number of components defined purely for the purposes of grouping may well lead to naming clashes (currently all of the components in a model must be uniquely named). It has been contended however that these problems would rarely occur as individual hierarchies would generally be stored separately so the names of components would typically be unique within the context in which they were processed, even if they were not unique across an entire network.

4.3 Sample Implementation in Data Model and XML

From the discussion above we can see that a group object:

- may have a reference to a major component. Note that any group which includes encapsulation, geometry or reaction relationship information must have a major component.
- must have references to one or more minor components.
- must declare at least one relationship. Some relationships such as encapsulation relate the major component to the minor component while other relationships like `is-next-to` relate the minor components to each other.

Note that each relationship object consists of a class and a name. The relationship class may be one of the CellML-defined types `is-encapsulated-by`, `is-in` or `is-a-participant-in` (corresponding to the encapsulation, geometry and reaction relationships) or some user-defined class. An encapsulation relationship need not be given a name, as there can only be one encapsulation hierarchy in a model. Similarly, a reaction object does not need a name, as it is strictly a grouping, and does not form any kind of hierarchical structure. However geometry relationships and user-defined relationships may be given a name, to associate groupings of the same class but different hierarchy together.

An example XML serialization of the grouping mechanism is given in Figure 3. The first two groupings demonstrate the possible use of the encapsulation, geometry and reaction information in a CellML-like context. The next three demonstrate how the naming of relationships allows the formation of multiple geometry hierarchies. In that example the `lower_leg_bones` group (where the name comes from the major component) is re-used in both the `leg_bones` and the `lower_leg` groups. It is still possible to find our way up the `skeleton` hierarchy however because that name is associated with the geometric relationship at each stage in the `skeleton` hierarchy. Finally, a fairly trivial example of a user-defined relationship class is given.

```
<group>
  <relationship class="is-encapsulated-by" />
  <relationship class="is-in" />
  <major_component_ref name_ref="cellular_membrane" />
  <minor_component_ref name_ref="sodium_channel" />
  <minor_component_ref name_ref="calcium_channel" />
</group>

<group>
  <relationship class="is-a-participant-in" />
  <major_component_ref name_ref="abc_pathway" />
  <minor_component_ref name_ref="a" />
  <minor_component_ref name_ref="b" />
  <minor_component_ref name_ref="c" />
</group>

<group>
  <relationship class="is-in" name="skeleton" />
  <major_component_ref name_ref="lower_leg_bones" />
  <minor_component_ref name_ref="tibia" />
  <minor_component_ref name_ref="fibula" />
</group>

<group>
  <relationship class="is-in" name="skeleton" />
  <major_component_ref name_ref="leg_bones" />
  <minor_component_ref name_ref="lower_leg_bones" />
  <minor_component_ref name_ref="upper_leg_bones" />
</group>

<group>
  <relationship class="is-in" name="anatomy" />
  <major_component_ref name_ref="lower_leg" />
  <minor_component_ref name_ref="lower_leg_bones" />
  <minor_component_ref name_ref="lower_leg_muscles" />
</group>

<group>
  <relationship class="are-close-together" />
  <minor_component_ref name_ref="tibia" />
  <minor_component_ref name_ref="fibula" />
</group>
```

FIGURE 3: An example XML serialization of the grouping mechanism discussed in the previous section. Please note this is an illustration only, and may yet change signifi cantly.