Meeting Minutes 18 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

This document is divided into two parts. The first part is in fact a summary prepared specifically for the September 18 meeting. The second part describes the actual decisions made during the meeting.

2 Encapsulation Problems

During the September 18 meeting some very basic problems with the CellML data model must be urgently resolved. At the very minimum, this involves finding a solution to the encapsulation/variable-role problem. We thought we'd sorted out encapsulation in the meeting of 13 September where the variable role of `in-out` was introduced to pass variables from the encapsulator's peers down to the encapsulated components. For simplicity in the following discussion, let's define two terms. The "*complex layer*" is the set of components that are encapsulated by the current component. The "*simple layer*" is the set of nodes that the current component may be connected to, excluding the components in the "*complex layer*".

Unfortunately, we weren't anywhere close to finished with encapsulation. As Melanie pointed out, the whole point of encapsulation is to hide the network that makes up the complex layer from the simple layer. Ideally, the variables that the current component exposes to the simple layer may be a subset of the variables it exposes to the complex layer. This is a fairly obvious requirement, and one that had been forgotten in the excitement of laying down rules for encapsulation over the last few meetings.

There are a number of possible solutions to this. The simplest is to allow mapping to variables with role `none` in the encapsulator from the components in the complex layer. This kind of clashes with the whole idea of mapping `in` to `out`, and would prevent the current component from keeping some variables private.

A more attractive solution is to have three new roles: `protected`, `in-protected-out`, and `out-protected-in`. The first allows the modeller to declare variables in the current component which can only be mapped to variables in encapsulated components. The second and third function like `in-out` but remove the ambiguity of the direction — the first term refers to the external interface and the second refers to the internal (protected) interface. (Note that this paragraph has been edited since the meeting, as no-one really understood the original wording.)

Perhaps a better solution is to effectively split the role attribute into input- and output-specific properties. A variable is either defined locally, in which case it may be modified in the current component, or it is an input from another component, in which case it may not be modified. Thus a variable has either an `input` property, or it doesn't. (Or as David suggested, the variable could have a role of `dependent` or `independent`.) We now have three possible output roles, `public` corresponding to `out`, `private` corresponding to `none`, and `protected` for the new encapsulation-related functionality. A property like this is commonly called `scope`, and its values are familiar to any object-oriented programmer.

The last solution comes from Physiome and is based on the currently implemented method in ISC. Currently a component would contain a list of all variables available in the current component, a list of all of the variables that are inputs to the current component, and a list of all components that are outputs. It would be a simple enough matter to split up the list of outputs into a list of `public` and a list of `protected` outputs. The problem with this method as a whole is its verbosity and the potential for inconsistencies in its

serialization. It would certainly be more convenient to have all of the information about a variable in one place.

From the implementation angle it is interesting to note that an internal Java class (probably the nearest thing in Java to an encapsulated component) can actually reference private variables in its parent class. It is subclasses (i.e., extensions to a Java class) that can only reference protected but not private variables. Maybe we need to check if there is a case for having private variables in an encapsulator that aren't available to the complex layer.

It is tempting to define encapsulated components inside the encapsulator. A possible problem with this is deciding where to define the connections between the encapsulator and the components in the complex layer — this has already been a problem when trying to express subspace hierarchies in CellML '99.

3 Grouping Problems

OK. So we've biffed variable inheritance in favour of `in-out` or something similar. This leaves us with geometry and encapsulation to sort out. Assuming we get the problem discussed in the previous section sorted out, the encapsulation mechanism is pretty much sorted. That leaves geometry.

The meeting of September 14 (unfortunately, not properly documented) highlighted some of the problems with grouping for the purposes of geometry. The issues are:

- Should groups require a root component?
- Will groups be re-used (i.e., in different hierarchies)?
- Do groups have to be uniquely named? (which of course raise the issue: should components be uniquely named?). Is this actually a serialization issue?
- Do groups need to be able to reference groups?

In the following sections, some possible solutions and their problems are considered.

3.1 Grouping By Connection

"Grouping By Connection" refers to original idea from the earlier versions of the CellML 2000 data model. Geometry and encapsulation information is stored as a property of a directed connection — connections to the same "parent" component imply a grouping of the "child" components. This scheme is very simple, and removes the need to identify groups at all. Unfortunately any serialization of this scheme involves the separation of the grouping in an arbitrary manner, making the grouping non-obvious without a complete scan of the connections in the model. Because all connections and components are serialized in parallel, all components in the model must have a unique name.

3.2 Grouping With Root Components

Grouping by connection links a child component to a parent component. An alternative formulation would be to introduce a new object `group` which contains a reference to the parent component and then a list of the child components. Essentially the functionality is identical to a grouping-by-connection mechanism, but the relationship between the child components and their parent is limited to one place.

It is possible that requiring root components isn't so evil after all. In the September 13 minutes we cited AnatML as having no use for the root components used for grouping. In fact this is not the case — components are necessary at each stage in the geometric hierarchy to store coordinate system transformations.

3.3 Unique Everything

OK - the meeting's over, and I've got something else to write about. For the record, in this scheme, groups must be uniquely named, do not need a root component, and may include other groups as children. You start running into problems however when you actually want to add encapsulation or geometric relationship information to this scheme. Essentially we would need some kind of connection that runs between a component and a group and says "the contents of this group are encapsulated/physically inside this component." Kinda' back to stage one.

4 The Actual Meeting Minutes

The meeting of September 18 was somewhat unusual in that we actually made some progress. Although we didn't solve any of the grouping problems discussed in the previous section, we did reach some kind of agreement finally on the variable role problem that has been bothering us for a while now.

Instead of trying to mix all of the role ideas into one attribute, or splitting the role attribute into input and output sub-sections, why not separate the role into its private and public interfaces? The public interface of a variable is the interface that is exposed to the simple layer. The private interface of a variable is the interface that is exposed to the complex layer, and to any encapsulated children of the components in the complex layer. Each interface can have values of `in`, `out` and `none`, where `none` is equivalent to the absence of an interface value. The nice thing about this method is there is no `in-out`, and adding complexity to a component should never affect the public interface of a component.

With one exception the two interfaces are completely independent. For instance a variable with an public interface of `out`, could have an private interface of `in`, `out` or `none` where the private interface can completely alter the variable's behaviour. With an private interface of `in`, the variable must be passed up from one of the encapsulated components — it may be used in the current component but is essentially just passing through. With an private interface of `out`, the value is an independent variable in the current component and may be modified there — it is exported to both the simple and complex layers. With an private interface of `none`, the variable is independent, and may be passed to the simple layer but not the complex layer. The one exception to the independence of the interfaces is when both private and public interfaces are of type `in`. Clearly this is inconsistent, so must be invalid. Two of the combinations of interface values are probably also not needed, as marked in the table in Table 1.

Private / Public	in	out	none
in	n	y	y
out	y	y	y
none	?	?	y

TABLE 1: A matrix with the allowable values of private interface on the rows and the allowable values of public interface on the columns. A "y" marks a possible combination, a "n" marks an illegal combination, and a "?" marks a possible but unlikely combination. Note that an interface value of `none` is equivalent to the absence of an interface.