CellML Specification 1.1

Draft — 6 November 2002

6 Grouping

6.1 Introduction

It is often useful to organise groups of components within a model into a hierarchical structure. This structure might reflect a logical organisation of components within the group or their physical configuration. CellML provides a single mechanism for the specification of both of these types of hierarchy that is based on a grouping scheme. This grouping scheme is general enough that it can be used within CellML documents to specify non-hierarchical grouping relationships between components.

In CellML, a *hierarchy* is a tree of components linked by parent-child relationships, where all of these relationships are of the same type. A hierarchy has a single root component and at least one child component. A model may contain numerous hierarchies of the same type. A component must only appear once within a set of hierarchies of a given type, but may appear in multiple hierarchies if each of these hierarchies is of a different type. CellML defines two types of relationship for use within the grouping scheme: *encapsulation* and *containment*.

The definition of a logical hierarchy of components in a network is known as "*encapsulation*". Encapsulation allows the modeller to hide part of a model by using a single component as an interface to a hidden submodel. The *parent* component hides the details of one or more *child* components from the rest of the model. Encapsulation provides a powerful mechanism for simplifying the structure of the model by preventing connections between specified sets of components. Components in the main model must not be connected to child components in the encapsulated submodel — all variables must be mapped through the encapsulating parent component. A component in the submodel must only be connected to its parent component, other components in the same submodel, and components that it encapsulates. A modeller wishing to re-use an encapsulated submodel may treat the submodel as a "black box", and deal exclusively with the interface presented by the encapsulating component.

The definition of physical hierarchies within a model is known as "*containment*". A model author can specify that one or more *child* components are physically inside of a *parent* component without describing the geometric aspects of the relationship in detail. This information would typically be used by CellML processing software to provide a physical representation of a model. A model may contain multiple types of containment hierarchy, which are differentiated based on names that the modeller assigns to these hierarchies.

Model authors are also free to extend the grouping scheme with user-defined types of relationships between components. These relationships need not be hierarchical in nature. However, CellML processing software is only required to recognise encapsulation and containment relationships.

Encapsulation and containment hierarchies do not add any mathematical information to the model. Model authors must not define their own grouping relationships that are intended to imply mathematical information.

Models may define multiple hierarchies of multiple types. CellML processing software is free to treat all hierarchies of the same type as separate hierarchies. Alternatively, it may combine all hierarchies of the same type into a single hierarchy by assuming that the root components of all explicitly defined hierarchies are children of a single *anonymous* component. This anonymous component is not explicitly defined within the CellML document and has no properties.

6.2 Basic Structure

6.2.1 Definition of groups

Logical and physical hierarchies are both declared using the `<group>` element. This element must be a child of a `<model>` element. A `<group>` element can be used to define multiple hierarchies and associate multiple relationship types with each hierarchy. The definition of a hierarchy or set of hierarchies of the same type can be split up over multiple `<group>` elements, as long as all the children of a given parent component in a hierarchy appear in a single `<group>` element.

A `<group>` element must contain one or more `<relationship_ref>` elements, each of which must define a `relationship` attribute, the value of which references one type of relationship. CellML processing software is required to recognise two types of relationship: encapsulation and containment, which are indicated by `relationship` attribute values of `"encapsulation"` and `"containment"`, respectively. Model authors can define new types of relationship by specifying a `relationship` attribute that is not in the CellML namespace on the `<relationship_ref>` element. All of the relationships referenced by the `<relationship_ref>` elements within a given `<group>` element are associated with all the parent-child pairs defined within that `<group>` element.

A `<group>` element must also contain one or more `<component_ref>` elements. Each `<component_ref>` element must define a `component` attribute, the value of which references a component within the current model. A parent-child link is created between components by nesting a `<component_ref>` element that references the child component inside a `<component_ref>` element that references the parent component. Multiple levels of nesting may be used within a single `<group>` element to define a hierarchy.

All `<component_ref>` elements defined immediately inside the `<group>` element must contain further `<component_ref>` elements when defining an encapsulation or containment hierarchy. This ensures that valid hierarchical structures are defined. Top-level `<component_ref>` elements need not contain further `<component_ref>` elements in `<group>` elements that reference only user-defined relationships. This allows the definition of non-hierarchical relationships.

A single hierarchy may be defined in multiple `<group>` elements. This occurs when a component is referenced in two groups that reference the same relationship type. However, all of the children of a given parent component must be defined within a single `<group>` element. Therefore, any given component can only be referenced once as a parent and once as a child for a given relationship type across the entire model. This simplifies the construction and validation of hierarchies.

A `<relationship_ref>` element may define a `name` attribute in addition to the required `relationship` attribute. The value of the `name` attribute on `<relationship_ref>` elements can be used to refine a given relationship type. This allows, for instance, the creation of several overlapping containment hierarchies within the same model, each with a different name. See Section 6.2.4 for more information on this.

Geometric containment relationship information is formally independent of logical encapsulation information, but CellML processing software is free to check for inconsistencies between the two relationships — it would generally be an error for an encapsulating component to be physically inside one of its encapsulated child components.

6.2.2 The *encapsulation* relationship type

Encapsulation allows the modeller to split a model into layers of complexity. A single component can be used to encapsulate a complex partial model, and thereby provide a unified interface for all information passing between that submodel and the rest of the model. This allows a modeller to refine the encapsulated submodel without having to make any changes to the rest of the model.

A model may contain any number of encapsulation hierarchies, as long as these do not overlap. If more than one hierarchy is explicitly defined, it may be useful to combine these into a single hierarchy by assigning all unencapsulated components an *anonymous* parent component. This anonymous component could make it easier to check that the hierarchies do not overlap and do not define any circular relationships between components.

The components in a model can be divided into four sets with respect to any given component (the *current* component). The set of all components immediately encapsulated by the current component is the *encapsulated set*. The *parent* component is the component that encapsulates the current component. Other components encapsulated by the same parent make up the *sibling set*. All other components, which are not available to make connections with the current component, make up the *hidden set*. If the current component is not encapsulated, then it has no parent and the sibling set consists of all other unencapsulated components in the model.

These sets are best demonstrated by example. Given the network shown in Figure 8, Table 4 lists the parent components and the components in the *encapsulated*, *sibling*, and *hidden* sets for a selected set of components picked as the *current* component.

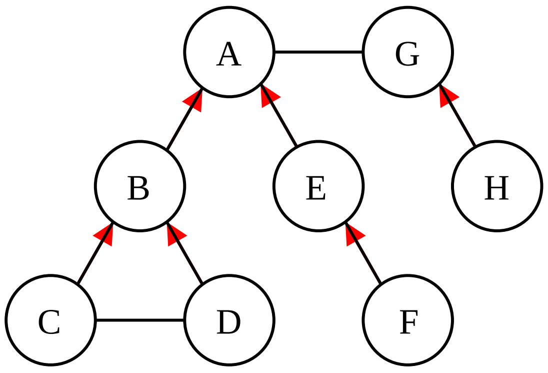

FIGURE 8: This simple model provides the basis for the demonstration of the concepts of *encapsulated sets*, *parents*, *sibling sets*, and *hidden sets*, as described in the text. The model consists of eight components each represented by a circle. The lines between the components represent connections, and a red arrowhead on one of these lines indicates that the component at the tail of the arrow is encapsulated by the component at the head of the arrow.

Current Component	**Encapsulated Set**	**Parent**	**Sibling Set**	**Hidden Set**
A	B, E	*anonymous*	G	C, D, F, H
B	C, D	A	E	F, G, H
C	*none*	B	D	A, E, F, G, H
E	F	A	B	C, D, G, H
G	H	*anonymous*	A	B, C, D, E, F

TABLE 4: This table lists the *parent* components, and the components in the *encapsulated, sibling,* and *hidden sets* for a selected few components from the example model in Figure 8. Components A and G are root components of separate hierarchies. It may be useful, however, to assign them an anonymous parent component that enables the formation of a single encapsulation hierarchy for the entire model.

Every variable must define its availability for use in other components. This is done with the `public_interfac` and **`private_interface`** attributes on the **`<variable>`** element. The interface exposed to the parent component and components in the sibling set is defined by the **`public_interface`** attribute. The **`private_interface`** attribute defines 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. The separation of interfaces allows the modeller to incrementally add complexity to an encapsulated submodel without changing the interface that the encapsulating component presents to the rest of the model.

The mappings allowed between variables declared in each component are defined by the public and private interfaces of each variable and the prohibition on connecting an encapsulated component to components other than its parent component, members of its sibling set, and any components it in turn encapsulates. Variables with a **`public_interface`** attribute value of `"in"` must be mapped to a single variable in a component in the sibling set with a **`public_interface`** attribute value of `"out"` or to a single variable in the parent of the current component with a **`private_interface`** attribute value of `"out"`. Similarly, variables with a **`public_interface`** value of `"out"` may be mapped to variables in components in the sibling set with a **`public_interface`** attribute value of `"in"` or to variables in the parent component with a **`private_interface`** value of `"in"`. Note that defining a **`public_interface`** attribute value of `"out"` on a variable makes it legal to map the variable to other variables, but does not require that such a mapping occur. If a variable has a **`public_interface`** attribute value of `"none"`, it is neither input from or exposed to its parent or components in the sibling set.

Variables with a **`private_interface`** attribute value of `"in"` must be mapped to a single variable in a single component in the encapsulated set with a **`public_interface`** attribute value of `"out"`. Variables with a **`private_interface`** attribute value of `"out"` may be mapped to variables in components in the encapsulated set with a **`public_interface`** attribute value of `"in"`. If a variable has a **`private_interface`** attribute value of `"none"`, it is neither input from or exposed to the components in the encapsulated set.

If either the **`public_interface`** attribute or the **`private_interface`** attribute of a variable have a value of `"in"`, that variable is declared elsewhere and its value must not be mathematically modified in the current component. Otherwise, the variable belongs to the current component. If both the **`public_interface`** and **`private_interface`** attributes of a variable have a value of `"none"`, the variable can only be used in the current component and is invisible to all other components in the model.

The two interface attributes of a variable are completely independent with one exception: it is invalid for a variable to have both **`public_interface`** and **`private_interface`** attributes with a value of `"in"`. An interface with a value of `"in"` reflects an unmet need in the current component that must be satisfied — this need can be met in either the public or private interface, but not both.

6.2.3 The *containment* relationship type

The containment relationship allows the modeller to specify that a particular component is physically inside another. This might be used by software to create a physical representation of the model. Containment relationships can be specified either in combination with or independent of encapsulation relationships. Containment relationships do not influence any aspect of model definition or behaviour.

6.2.4 Named containment relationship types

CellML allows the definition of multiple overlapping containment hierarchies in a given model. This functionality allows the modeller to define several different ways of organising a model, each of which might highlight a different aspect of the model's physical structure.

Multiple containment hierarchies are created by defining **`name`** attributes on the **`<relationship_ref>`** elements that have **`relationship`** attribute values of `"containment"`. In effect, the introduction of a

name attribute defines a new relationship type that has the same semantics as the unnamed containment relationship type. All containment hierarchies that share the same name are subject to the same rules that apply to any set of hierarchies that share the same relationship type. That is, each component must be referenced at most once as a parent or child for a given relationship type, and circular hierarchies must not be defined. Note that **<group>** elements that contain **<relationship_ref>** elements with a **relationship** attribute value of "containment" and that do not define a **name** attribute belong to a single relationship type that is separate from any named containment relationship types.

6.2.5 User-defined relationship types

Modellers are free to use the grouping syntax of CellML to organise model components in ways not defined by the CellML specification. To do this, the model author defines a new relationship type, the name of which is used as the value of the **relationship** attribute on the **<relationship_ref>** element. The **relationship** attribute must be placed in an extension namespace, because future versions of the CellML specification may define additional relationship types, the names of which could otherwise conflict with user-defined relationship types.

User-defined relationship types can be used in the definition of hierarchical relationships and can also be used to define more generic *grouping* relationships. For example, a modeller may define a relationship type called **adjacency**, that indicates that any components referenced inside the group are physically adjacent to each other.

Modellers are free to use the **name** attribute on the **<relationship_ref>** element to specify multiple hierarchies for user-defined relationship types, as is possible for the containment relationship type.

This specification does not provide a mechanism by which modellers may specify the meaning of a user-defined type of relationship. This definition must be provided by the processing software declaring the new relationship type.

6.3 Examples

Figure 9 demonstrates the use of the **<group>** element to define an encapsulation relationship. This example is taken from the two reaction pathway with encapsulation example[1] from the examples section of the CellML website. It shows how a component representing an overall reaction (**total_reaction**) can encapsulate components representing intermediate reactions (**first_reaction** and **second_reaction**) and their by-products (**C** and **D**).

```
<group>
  <relationship_ref relationship="encapsulation" />
  <component_ref component="total_reaction">
    <component_ref component="first_reaction" />
    <component_ref component="second_reaction" />
    <component_ref component="C" />
    <component_ref component="D" />
  </component_ref>
</group>
```

FIGURE 9: Example demonstrating the use of the **<group>** element to defi ne a logical encapsulation relationship. See text for more details.

[1]http://www.cellml.org/examples/examples/signal_transduction_models/basic_reaction_models/two_reaction_model_with_encapsulation

Figure 10 demonstrates the use of the `<group>` element to define encapsulation and containment relationships, the construction of two named containment relationship types, and the specification of a custom relationship type (`adjacency`) in an extension namespace.

```
<group>
  <relationship_ref name="membrane" relationship="containment" />
  <component_ref component="cell">
    <component_ref component="cell_membrane" />
  </component_ref>
</group>

<group>
  <relationship_ref relationship="encapsulation" />
  <relationship_ref name="membrane" relationship="containment" />
  <component_ref component="cell_membrane">
    <component_ref component="sodium_channel" />
    <component_ref component="calcium_channel" />
  </component_ref>
</group>

<group>
  <relationship_ref name="intracellular" relationship="containment" />
  <component_ref component="cell">
    <component_ref component="network_sarcoplasmic_reticulum" />
    <component_ref component="junctional_sarcoplasmic_reticulum" />
  </component_ref>
</group>

<group>
  <relationship_ref
      app:relationship="adjacency"
      xmlns:app="http://www.software.com/cellml_processor" />
  <component_ref component="network_sarcoplasmic_reticulum" />
  <component_ref component="junctional_sarcoplasmic_reticulum" />
</group>
```

FIGURE 10: Examples demonstrating the use of the `<group>` element. See text for more details.

The first `<group>` element states that the `cell_membrane` component is physically inside the `cell` component and that this particular containment relationship type is called `membrane`. The next `<group>` element states that the `sodium_channel` and `calcium_channel` components are both physically inside and logically encapsulated by the `cell_membrane` component. This containment relationship completes the `membrane` containment hierarchy. The encapsulation relationship prevents the sodium and calcium channel components from being connected to any components other than the `cell_membrane` component, each other, and any components they in turn encapsulate.

The third `<group>` element states that the two components representing parts of the sarcoplasmic reticulum are physically inside the cell, and that this relationship type is called `intracellular`. Finally, the fourth `<group>` element introduces the user-defined relationship `adjacency` and states that the two sarcoplasmic reticulum components share this relationship. This relationship type is declared by putting the `relationship` attribute in an extension namespace and assigning it a value of `"adjacency"`. Note that this relationship is not hierarchical in nature, and CellML processing software is free to ignore the information provided by this group.

6.4 Rules for CellML Documents

6.4.1 The `<group>` element

1. **Allowed use of the `<group>` element**

   - A `<model>` element may contain any number of `<group>` elements.
   - A `<group>` element must contain only the following elements, which may appear in any order:
     - `<relationship_ref>` and `<component_ref>` elements in the CellML namespace,
     - `<RDF>` elements in the RDF namespace.

     [Recommended practice is to define the CellML namespace child elements in a `<group>` element in the order stated above.]
   - A `<group>` element must contain at least one `<relationship_ref>` element.
   - A `<group>` element must contain at least one `<component_ref>` element.

6.4.2 The `<relationship_ref>` element

1. **Allowed use of the `<relationship_ref>` element**

   - A `<relationship_ref>` element must contain only the following elements:
     - `<RDF>` elements in the RDF namespace.
   - Each `<relationship_ref>` element must define a `relationship` attribute in either the CellML namespace or an extension namespace. It may also define a `name` attribute.

     [A `relationship` attribute declaring a user-defined relationship type is placed in an extension namespace. This restriction has been included to prevent conflicts with future versions of the CellML specification, which may define additional types of relationships in the CellML namespace.]

2. **Allowed values of the `relationship` attribute**

   - The value of a `relationship` attribute in the CellML namespace must be `"containment"` or `"encapsulation"`.

3. **Allowed values of the `name` attribute**

   - The value of the `name` attribute must be a valid CellML identifier as discussed in Section 2.2.1[2].

     [Note that unlike most other `name` attributes, the value of the `name` attribute on a `<relationship_re` element is not expected to be unique across the current model. Instead, `<group>` elements that include `<relationship_ref>` elements that share the same `name` attribute value form the parts of a single hierarchy.]

4. **Proper use of the `name` attribute**

   - A `name` attribute must not be defined on a `<relationship_ref>` element with a `relationship` attribute value of `"encapsulation"`.

     [A model must define at most one unnamed encapsulation hierarchy.]

[2] http://www.cellml.org/public/specification/20021106/cellml_specification.html#sec_fundamentals_identifiers

5. **Proper use of the `relationship` and `name` attributes**

   [The following rules together prevent the model author from referencing the same hierarchy more than once within a given `<group>` element.]

   - A `<group>` element must not contain two or more `<relationship_ref>` elements that define a `relationship` attribute in a common namespace with the same value and that have the same `name` attribute value.
   - A `<group>` element must not contain two or more `<relationship_ref>` elements that define a `relationship` attribute in a common namespace with the same value and do not define `name` attributes.

6.4.3 The `<component_ref>` element

1. **Allowed use of the `<component_ref>` element**

   - A `<component_ref>` element must contain only the following elements, which may appear in any order:
     - `<component_ref>` elements in the CellML namespace,
     - `<RDF>` elements in the RDF namespace.
   - A `<component_ref>` element must define a `component` attribute.

2. **Proper use of the `<component_ref>` element**

   - A `<component_ref>` element that is defined immediately within a `<group>` element that contains a `<relationship_ref>` element with a `relationship` attribute value of `"encapsulation"` or `"containment"` must contain at least one child `<component_ref>` element.

     [Containment and encapsulation relationships must be hierarchical.]
   - In a given hierarchy, only one of the `<component_ref>` elements that reference a given component may contain further `<component_ref>` elements.

     [This rule prevents a given component from being a parent more than once in a given hierarchy. A hierarchy is a set of components linked by a common type of parent-child relationship. The definition of a hierarchy may be split over multiple `<group>` elements, but the definition of a set of parent-child links must not be. It would be much more difficult to assemble a hierarchy from a CellML document if a set of parent-child links could be defined in multiple `<group>` elements.]
   - In a given hierarchy, only one of the `<component_ref>` elements that reference a given component may be contained inside another `<component_ref>` element.

     [Complements the previous rule. This one prevents a given component from being a child more than once in a given hierarchy.]
   - In a given hierarchy, a child component must not directly or indirectly contain its parent among its children.

     [A hierarchy must not be circular.]

3. **Allowed values of the `component` attribute**

   - The value of the `component` attribute must equal the value of the `name` attribute of a `<component>` element contained within the current `<model>` element.

6.5 Rules for Processor Behaviour

6.5.1 Treatment of relationship types in a single model

A given relationship type must have the same semantics across a model and at all levels in every hierarchy associated with that relationship type. The semantics of the encapsulation and containment relationship types are defined in Section 6.2.2 and Section 6.2.3, respectively.

Within a given `<model>` element, any hierarchies defined in `<group>` elements that contain `<relationshi` elements with identical `relationship` and `name` attribute values belong to the same relationship type, and must be treated as such. Any hierarchies defined in `<group>` elements that contain `<relationship_ref>` elements with identical `relationship` attribute values and undefined `name` attributes belong to the same relationship type, and must be treated as such.

6.5.2 Groups must not imply mathematical information

Modellers must not use CellML groups to add mathematical information to the model. Modellers must not define their own types of relationships that imply mathematical behaviour. This ensures that the mathematical behaviour of a model can be properly reproduced by all CellML processing software.

6.5.3 Groups must not imply metadata information

Modellers must not use CellML groups to associate properties or classification information with sets of components. The metadata functionality is the proper method for making such associations. This increases the chance of that information being used by a range of CellML processing software.