CellML API

Matt Halstead

Table of Contents

Table of Contents

CellML API

Date: 2005−02−20
Author: Matt Halstead
Contact: matt.halstead@auckland.ac.nz

The minimum set of interfaces for the CellML 1.1 API

The goal is to satisfy only those interfaces we think are necessary. It does not limit the interfaces. Anyone is free to implement other interfaces and mix them in.

While there is some attempt to provide some validation comments, this needs a lot more attention.

Basic Building Blocks

CellML Attribute String

This holds attribute values for the XML attributes of elements in serialised CellML. Character data in XML can be any Unicode character; the sequence of characters must conform to the rules of XML character data which define escaping of reserved characters etc(http://www.w3.org/TR/REC−xml/#syntax).

```
CellMLAttributeString

    Unicode value
```

URI

URIs need to conform to RFC2396 (http://rfc.net/rfc2396.html)

```
URI

    Boolean validate()
```

CellML Object

A CellMLObject represents CellML/XML elements. The validation method is something that all of these objects need to define, so consider this one virtual. Since there are no interfaces defined for attributes of CellML/XML elements, the validation methods of the CellML Object that own then needs to check constraints on these.

See Handling Rdf for a discussion on how we want to handle it in the API.

```
CellMLObject

    # the cellml version this model conforms to
    CellMLAttributeString cellml_version

    # A cmeta id may be defined on any CellML element
    CellMLAttributeString cmeta_id

    # rdf metadata associated with object.  Note: the object must have
    # a cemta:id for any RDF to be able to refer to it.
    RDF rdf

    # extension objects
    CollectionOfExtensionObjects extension_objects

    # Validate. Some validation only makes sense once entire model is loaded
    Boolean validate()

    # the generic return type of parent_objects is given for those
    # static typing kind of languages. While it is very useful for
    # implementation, I am not so clear on external use cases for
    # this method.  CellMLObject parent_object()

    # return the model object this CellML object belongs to
    Model model_object()
```

Named CellML Object

Some CellML constructs require a name which is unique within a particular scope. This uniqueness should be checked in the validation of those constructs. We need to resolve the issue of uniqueness across imported components(see Unique names).

```
NamedCellMLObject : CellMLObject

    CellMLAttributeString name
```

Extension Object

An extension object is a user defined representation of data in an extension namespace. In CellML/XML, a user is free to add other data elements that do not conform to CellML names or rules, so long as they are defined in another namespace. It is up to the application developer to handle the creation of extension objects, which usually means interacting with the XML parser to trigger the correct methods.

```
ExtensionObject
```

CellML Constructs

Represents a CellML Model. For a discussion on the visibility of imported structures see Resolving imported structures.

Validation consists of calling validation on:

- units
- components
- connections
- groups
- imports

Model

```
Model : NamedCellMLObject

    # interface to the manager for loading and keeping track of models.
    ModelManager model_manager

    CollectionOfGroups groups
    CollectionOfImports imports
    NamespaceMap namespaces

    # models should really set an 'xml:base' attribute in the
    # model element as we need to be able to unambiguously and
    # uniquely reference a model independent of its location.  If
    # this is not set, the base_uri could be set to the URI used
    # to obtain it.
    URI base_uri


    # model units.  If any imports have been fully instantiated,
    # then it includes those units objects as well.
    CollectionOfUnits units()

    # returns names of all units visible at the model level.
    # I.e. this does not include units declared in components, but
    # will include units declared in imports.  This interface has
    # the benefit of always including imported units, whereas the
    # units() interface does not.
    CollectionOfCellMLAttributeString units_names()

    # returns all components with interface 'Component' in the
    # model.  If imports have been fully instantiated then it
    # includes these too.
    CollectionOfComponents components()

    # returns names of all components declared in the model,
    # including those declared in the import structures.
    CollectionOfCellMLAttributeString component_names()

    # model connections.  If any imports have been fully
    # instantiated, then it includes connections relevant to those
    # also.
    CollectionOfConnections connections()
    # the following also searches user-defined relationships.
    CollectionOfGroups find_groups_with_relation_ref_name(CellMLAttributeString name)

    #This fully instantiates the import definitions
    fully_instantiate_imports()

    # return a flattened model, i.e. all imported component trees are
    # promoted to model level and import objects are removed.
    Model generate_flattened_model()
```

Component

Represents the component object

Validation checks that

- all variables are valid
- all units are valid
- all math is valid – (what does that mean?)

```
Component : NamedCellMLObject


    CollectionOfVariables collection_of_variables
    CollectionOfUnits collection_of_units
    CollectionOfMath collection_of_math

    # returns a collection of connections that this component participates in
    CollectionOfConnections connections()

    # both of the following methods return NULL if the component does not
    # participate in an encapsulation hierarchy
    Component encapsulation_parent()
    CollectionOfComponents encapsulation_children()

    # both of the following methods return NULL if the component does not
    # participate in a containment hierarchy
    Component containment_parent()
    CollectionOfComponents containment_children()
```

Imports

This section describes the interfaces to imported structures. For an explanation of full instantiation, see Representation of imported objects.

Import

```
Import : CellMLObject

    # The value of the href attribute must be a URI reference as defined in
    # [IETF RFC 2396] - see http://www.w3.org/TR/xlink/#link-locators
    URI xlink_href
    CollectionOfImportComponents components
    CollectionOfImportUnits units
    # the following represents the connections maintained in the import.
    CollectionOfConnections imported_connections

    # fully instantiate the import.
    full_instantiate()
    Boolean is_fully_instantiated()
```

Import Component

This is an extension of interface Component for use in Import constructs. This hooks the methods of Component and where methods require it, checks that the imported component has been instantiated.

The name attribute of the component declaration is available through the name interface of Component.

Validation checks that

- the referenced component actually exists in the model referenced by the import.

```
ImportComponent : Component

    CellMLAttributeString component_ref

    Boolean is_fully_instantiated()
    fully_instantiate()
```

Import Units

This is an extension of interface Units for use in Import constructs. This hooks the methods of Units and where methods require it, checks that the imported units has been instantiated.

The name attribute of the units declaration is available through the name interface of Units.

Validation checks that

- the referenced units actually exists in the model referenced by the import.

```
ImportUnits : Units

    CellMLAttributeString units_ref

    Boolean is_fully_instantiated()
    fully_instantiate()
```

Unit definitions

Units

```
Units : NamedCellMLObject

    # the following is set to true if this is a new type of base units.
    Boolean is_base_units

    CollectionOfUnit units
```

Unit

```
Unit : CellMLObject

    int prefix
    float multiplier
    float offset
    float exponent
    CellMLAttributeString units
```

Maths

The Math elements have two faces:

1. they are objects in CellML that hold math equations, but which the rest of the CellML objects do not reference, or need to interpret.
2. they are mathematical definitions that use CellML objects, such as units of variables referenced, and require a set of interfaces so that developers who need to access the math can so. MathElement is a placeholder for such a set of interfaces. They are not defined as part of the CellML API specification.

One note from the CellML 1.1 Specification w.r.t the validation of math in the CellML context. On MathML elements, the mathml:id attribute must be used. A cmeta:id attribute must specifically not be added to MathML elements because a given element may only contain one attribute of type ID.

In the following, MathElement is a used defined interface to the math objects declared in the MathML.

```
Math : MathElement
```

Variables

Variable Interface

Represents the value of public_interface or private_interface Validation checks it is one of (in,out,none)

```
VariableInterface

    CellMLAttributeString value # or enumerated type

    # the following test the states of the interface
    Boolean is_in()
    Boolean is_out()
    Boolean is_none()

    validate()
```

Variable

Represents the variable object.

Validation checks that

- interfaces have allowed values

- interfaces are connected in correct ways according to interface rules.
- units name value refers to a visible units object in the model
- units in connections have common bases.
- initial_value is numeric or the name of a variable in the current component.
- at the model level we may, after construction, validate that all 'in' variables propagate back to a single source and that an initial_value, if set for this variable chain, is set at the source location and nowhere else in the model.

```
Variable : NamedCellMLObject

    # we leave initial_value as a string since it can be a
    # numerical value or the name of variable from which to get
    # its value.
    CellMLAttributeString initial_value

    CellMLAttributeString unit_name
    VariableInterface public_interface
    VariableInterface private_interface

    # get a collection of variables connected to
    CollectionOfVariables connected_variables()

    # trace back to the source for this variable by going back through
    # the chain of variable connections leading to it.
    Variable find_source_variable()
```

Component References

Component Ref

```
ComponentRef : CellMLObject

    # the name of the component being referenced
    CellMLAttributeString component_name

    # child component ref objects
    CollectionOfGroupComponentRefs component_refs

    # get parent component ref, returns NULL if this is a top level
    # component_ref
    GroupComponentRef parent_component_ref()

    # get parent group
    Group parent_group()
```

Relationship Ref

Validation checks that if a non standard relationship is set, that the namespace of this object is not the cellml one.

```
RelationshipRef : CellMLObject

    CellMLAttributeString name # optional, so validation does not need to check for it.
    CellMLAttributeString relationship
    URI raltionship_namespace
```

Group

```
Group

    CollectionOfRelationshipRefs relationship_refs
    CollectionOfComponentRefs component_refs

    Component get_parent_component_name(component name)
    Component get_children_component_names(component name)

    # the following test to see if the standard cellml relationships
    # of encapsulation and containment are defined for the group
    Boolean is_cellml_encapsulation()
    Boolean is_cellml_containment()
```

Connection

```
Connection : CellMLObject

    MapComponents map_components
    CollectionOfMapVariables map_variables
```

Map Components

```
MapComponents : CellMLObject

    CellMLAttributeString component_1_name
    CellMLAttributeString component_2_name

    # the following get references to the components component_1
    # and component_2 refer to.  If imports have not been fully
    # instantiated these should return an error.
    # to an imported component.
    Component get_component_1()
    Component get_component_2()
```

Map Variables

```
MapVariables : CellMLObject

    CellMLAttributeString variable_1_name
    CellMLAttributeString variable_2_name

    # the following get references to the variables
    # variable_1_name and variable_2_name refer to.  If the model
    # has not been instantiated these should return an error when
    # referring to a variable of an imported component
    Variable get_variable_1()
    Variable get_variable_2()
```

Reactions

Reaction

```
Reaction : CellMLObject

    CollectionOfReactantVariableRefObjects reactants
    CollectionOfProductVariableRefObjects products
    RateVariableRef rate
```

Variable Ref

```
VariableRef : CellMLObject

    variable
```

Reactant Variable Ref

```
ReactantVariableRef : VariableRef

    variable
    ReactantRole role
```

Product Variable Ref

```
ProductVariableRef : VariableRef

    variable
    ProductRole role
```

Rate Variable Ref

```
RateVariableRef : VariableRef

    variable
    RateRole role
```

Role

```
Role : CellMLObject

    direction
    delta variable
    stoichiometry
```

Reactant Role

```
ReactantRole : Role

    role = "reactant"
```

Product Role

```
ProductRole : Role

    role = "product"
```

Rate Role

```
RateRole : Role

    role = "rate"
    CollectionOfMath maths
```

RDF metadata

The metadata is stored in RDF elements in the rdf namespace. RDF is a user defined interface to RDF models. See also Handling Rdf.

```
RDF

    # need to check the namespace is a valid RDF namespace.
```

Collections

Collection of CellML Attribute Strings

```
CollectionOfCellMLAttributeStrings
```

Collection Of CellML Objects

```
CollectionOfCellMLObjects

    #cardinality of collection s
    length()

    #test x for membership in s
    contains(CellMLObject x)

    # add x to collection
    add(CellMLObject x)

    # remove x from collection
    remove(CellMLObject x)

    # replace x with y
    replace(CellMLObject x,CellMLObject y)

    clear()
    CollectionOfCellMLObjectsIterator begin()
    CollectionOfCellMLObjectsIterator end()


CollectionOfCellMLObjectsIterator

    # return next item in collection. Different implementations
    # will have different methods for handling the stop/end
    # condition.
    next()
```

Collection Of Named CellML Objects

```
CollectionOfNamedCellMLObjects : CollectionOfCellMLObjects

    """
    Callable as a dictionary like structure
    """
    # get named object
    get(CellMLAttributeString name)

    # add x to collection
    add(NamedCellMLObject x)

    # replace x with y
    replace(NamedCellMLObject x,NamedCellMLObject y)

    # remove x from collection
    remove(NamedCellMLObject x)

    # remove object named x from collection
    remove_by_name(CellMLAttributeString name)
```

Collection Of Extension Objects

```
CollectionOfExtensionObjects
```

Collection Of Models

```
CollectionOfModels : CollectionOfNamedCellMLObjects
```

Collection Of Components

```
CollectionOfComponents : CollectionOfNamedCellMLObjects
```

Collection Of Imports

```
CollectionOfImports : CollectionOfCellMLObjects
```

Collection Of Variables

```
CollectionOfVariables : CollectionOfNamedCellMLObjects
```

Collection Of Units

```
CollectionOfUnits : CollectionOfNamedCellMLObjects
```

Collection Of Unit

Collections of Unit occur in Units objects.

```
CollectionOfUnit : CollectionOfCellMLObjects
```

Collection Of Math

```
CollectionOfMath : CollectionOfCellMLObjects
```

Collection Of Connections

```
CollectionOfConnections : CollectionOfCellMLObjects

    # retrieve a subset connections that uses a component with component
    # name.
    CollectionOfConnections connection by component name
```

Collection Of Groups

```
CollectionOfGroups : CollectionOfCellMLObjects
```

```
    # returns a subset of the collection that match a relationship_ref
    CollectionOfGroups groups of relationship_ref(CellMLAttributeString relationship_ref)

    CollectionOfGroups groups_of_relationship_ref_containment
    CollectionOfGroups groups_of_relationship_ref_encapsulation
```

Collection Of Relationship Refs

```
CollectionOfRelationshipRefs : CollectionOfCellMLObjects

   # the following is different to the same feature in named collections
   # in that we can't assume all members have a name.  It will return NULL
   # if a relationship_ref object is not present with the given name.

   RelationshipRef get_relationship_ref_by_name(CellMLAttributeString name)
```

Collection Of ComponentRefs

Don't think we need the following

```
CollectionOfComponentRefs : CollectionOfCellMLObjects
```

Collection Of MapVariables

```
CollectionOfMapVariables : CollectionOfCellMLObjects
```

Collection Of VariableRef Objects

```
CollectionOfVariableRefObjects : CollectionOfCellMLObjects
```

Collection Of Role Objects

```
CollectionOfRoleObjects : CollectionOfCellMLObjects
```

Model Management API

This is a separate API from the CellML API, but is useful to describe here.

Model Manager

The Model Manager describes the interfaces required to load CellML from serialised CellML/XML and to create CellML model representations in memory.

```
ModelManager

    ModelMap models

    # create an empty model
    Model create_empty_model()
```

Model Map

A map to maintain a collection of models that have been loaded into memory.

```
ModelMap

    # one problem is that there is no guarantee of unique model
    # names.  Do we raise an exception or instead use collections
    # for the return of methods based on name?

    add_model(CellMLAttributeString model_name,URI model_uri,Model model)
    remove_model_by_name_and_uri(CellMLAttributeString model_name,URI model_uri)
    remove_model_by_name(CellMLAttributeString model_name)
    remove_model_by_uri(URI model_uri)
    remove_model(Model model)
    Model get_model_by_name(CellMLAttributeString model_name)
    Model get_model_by_uri(URI model_uri)
    Model get_model_by_name_and_uri(CellMLAttributeString model_name,URI model_uri)

    CollectionOfModels models
```

Serialiser

The serialisation of CellML object model to CellML/XML. If no namespace entry is found for a namespace map, then a prefix will be automatically generated.

```
Serialiser

    serialise(Model model,NameSpaceMap name_space_map)
    Model create_model_from uri(uri of xml document)
    Model create_model_from bytestream(bytestream of xml)
```

Namespaces

Namespace Prefix

The string format of this needs to conform to the XML definition of namespace prefixes (http://www.w3.org/TR/REC–xml–names/#NT–NCName). Defining an interface for this is perhaps overkill and could be left up to a more general validator to check these.

```
NamespacePrefix

    Boolean validate()
```

Namespace Map

Holds prefix : URI pairs. At the moment we don't handle contexts, i.e. this is a general map that can be used by a serialiser; it would be up to something else to handle specific contexts for various namespaces, e.g. xmlns="http://www.w3.org/1998/Math/MathML" is only declared in the math elements and not the root element. This is a matter of preferred serialisation, and it is not a rule that document structure, including context specific namespace declaration, is maintained between de–serialising and serialising models. However, a particular implementation may offer this.

```
NamespaceMap

    # set_namespace creates a new item if there is not one present that
    # already matches the prefix.  namespace_prefixes are unique,
    # namespace_uris are not.
    set_namespace(namespace_prefix,namespace_uri)

    remove_namespace_by_prefix(NamespacePrefix namespace_prefix)
    remove_namespaces_by_uri(URI namespace_uri)
    URI get_namespace_uri(NamespacePrefix namespace_prefix)
    NamespacePrefix get_namespace_prefix(URI namespace_uri)
    CollectionOfNamespacePrefixes prefixes()
    CollectionOfNamespaceUris namespace_uris()
    clear()
```

Factories

Factory are used to create the CellML objects. A factory is required for each element that can be defined in the CellML namespace of CellML/XML serialisation. In addition to these, there should be factories that handle the math and RDF elements which have their own namespaces. It is up to the implementation to decide how to load the XML and call the factories.

CellML Attribute Map

This is a map to hold the names and values of attributes found in CellML elements in the CellML/XML. The most common namespace_uri will be the CellML one.

```
CellMLAttributeMap

    get_attribute_by_name(CellMLAttributeString attribute_name,URI namespace_uri)
```

CellML Object Factory

Generic factory handler

```
CellMLObjectFactory

    # create_cellml_object is the generic method called to create the
    # object type required.  This is overridden by each factory.
    # Obviously the return type needs to be one that conforms to the
    # correct interface for the cellml object being created.
    CellMLObject create_cellml_object(CellMLAttributeMap attribute_map)
```

Specific factories

```
ModelFactory : CellMLObjectFactory

    Model create_cellml_object(CellMLAttributeMap attribute_map)

ComponentFactory

    Component create_cellml_object(CellMLAttributeMap attribute_map)

ImportComponentFactory

    ImportComponent create_cellml_object(CellMLAttributeMap attribute_map)

UnitsFactory

    Units create_cellml_object(CellMLAttributeMap attribute_map)

UnitFactory

    Unit create_cellml_object(CellMLAttributeMap attribute_map)
```

etc...

Exceptions

We need to think about exceptions more. It would seem more appropriate to use error objects that are returned from various functions such as validation. This is because :

1. Many of these may require alternative action and continuation of execution proceeding from the point of where a function was called that caused the error.
2. Some environments handle exceptions as return objects anyway.

Utilities

This space will be filled out soon. It is composed of utilities such as

- decomposing models into single component models that are reimported
- decomposing a model that has initial_values set into a generic model and a simulation instance model
- the handling of modifications to imported structures.
- etc...

General Notes

Resolving imported structures

Unique names

While specific components are given local names in an import declaration, components hidden in their subtrees are not. This leaves us with a potential for a name clash if these hidden components are to be accessible by name. We have two options :

- these components are not visible to methods available for the model doing the import, i.e. there will not be name clashes, and we cannot refer to them by name.
- these components are visible, but have a generated local name in the model doing the import.

The choice affects the scope of some methods defined on Model objects.

Representation of imported objects

Instantiation

Imported objects are declared using the import structures. These declarations are simple references to the relevant structures in another model. In the simplest representation we need only provide interfaces to these declarations. But it is also useful to build full representations of these structures so that the information they contain can be used by a system operating on the model that imported them. We refer to these representations as fully instantiated. It is important to understand that an imported component is not imported within the context of an instantiated representation of the model from which it came from. Imports are not connecting one model to another, but instead are using another model to serve as a template for structures that can be used in the importing model.

Implicit structures

We have already introduced the problem of hidden imported components in the section on Unique names. These components, hidden in the subtrees of declared components, are structures that become implicit members of a model with imports. The connections between these components are also implicit structures that should be made available through the API, since these form the only method for navigating through a subtree.

Units structures can also form trees through the units attribute of their unit children. It is useful to be able to provide access to these objects also.

The interfaces in Imports and Model need to reflect this thinking.

Modifying imported structures

The declaration of imported structures can be modified, and this may or may not have implications on the version or variant of a model (see here for a proposal on this as part of a model repository).

Components, connections, and units of fully instantiated imports are read only and cannot be modified. If a user wants to modify these, then they are essentially modifying a structure that is declared in another model. We could simply promote the object into the model level of the current model, but this would break the re−use philosophy of imports by making it simple for users to create many unrelated variations of an existing structure. It would be better to force the creation of a new model that represents altered structures and that relate to the original model these were imported from in a useful manner. A useful manner means one of the following:

- directly imports that object and modifies it through standard CellML declarations; this may require further decomposition of structures to open up parts that need changing. Note: it is important to understand that the structure of CellML structures cannot be directly modified through declarative syntax. If the structure is modified directly, for example, a variable is removed from a component, then this constitutes a change that can only be labelled through variant attributes(see next point).
- forms a variant and adds the appropriate attribute that this model is a variant of the original model it was imported from.

Handling Rdf

Every cellml object can have rdf defined. Rdf needs to be thought about some more − we have this problem of RDF can be defined anywhere in the model, but that it really only defines information w.r.t particular named objects that use the cmeta:id This RDF can go anywhere, should there be some rules for how the structure is maintained or not? We cannot guarantee the same RDF serialisation if the RDF objects are held in an internal RDF representation. If we went the way of only holding those objects relating to this object by RDF reference, then we need a collection of RDF here, since there may be multiple RDF elements that describe attributes of them. My recommendation would to be collect together all the RDF statements about the cmeta:id that is attributed to this object and keep hold of that collection. The RDF/XML in the CellML/XML should be structured this way if the person is concerned about some readable order to their RDF. We might want to give options of all the RDF being serialised next to their objects or somewhere else. But that is up to the serialiser.

Glossary

Instantiated object

> This represents the creation of an instance of an object from a CellML declaration. The most common form of an CelLML declaration is the CellML/XML syntax.

CellML/XML

> This is the XML representation of a CellML model.

Todo

- we have extension objects for elements, but what about attributes? We do have an explicit representation of namespace for one attribute that has significance to a model if it is user defined (see Relationship Ref), but this is different from arbitrary attributes in an extension namespace.
- RDF object notes need to be resolved – see Cellml Object