The CellML API

The CellML API (application programming interface) provides a programmatic interface to CellML models.

Specifying the API

- The API is specified using OMG IDL (interface definition language).
- This provides a machine-readable representation of the *attributes* and *operations* that can be performed on the CellML API.
- As with all IDL specified interfaces, the CellML API is entirely object orientated.

Language neutrality

- OMG IDL is *language neutral*. It does not specify which programming language the API is implemented in, or called from.
- *Language mappings* describe what the API looks like in a particular programming language.
- *Bridges* connect an implementation in one language to a caller in another.

The Physiome C++ Mapping

- I have produced an implementation of the CellML API in C++.
- The language mapping used is called the Physiome C++ Mapping (PCM).
- The existing CORBA C++ mapping was not used because it was considered too cumbersome to use when CORBA is not used.

Generating headers and bridges

- Because IDL is machine readable, it is possible to automatically generate language specific interfaces and bridges.
- Using omniidl, an extensible IDL compiler, I have produced two-way PCM √ CORBA and PCM √ XPCOM bridges, as well as header files for PCM and XPCOM objects.
- The CORBA bridge can be used to access the API via CORBA from a range of languages.

The base interface

- The base interface, which all objects implement, is called XPCOM::IObject (because it has operations resembling those on COM objects).
- It supports reference counting, as well as 'query interface' and fetching an identifier which uniquely identifies the object (for comparison purposes).

CellMLElement

- All CellML elements implement an interface called CellMLElement. This interface has a range of operations and attributes to:
- Insert and remove CellML and extension elements.
- Get/set the cmeta ID attribute.
- Allows arbitrary user-data (not extension elements) to be attached.

Sets

- Sets are lists of elements in a CellML model.
- Sets can produce 'iterator' objects, which can be used to iterate through the objects in the set.
- Sets are always live: changes to the model will immediately change what the iterators see.

Model

- Represents a CellML model.
- Different (1.0 vs 1.1) versions of the model can be fetched. This just changes the namespace, it doesn't 'flatten' the model.
- A serialised XML representation of the model can be retrieved.
- All RDF/XML in the model can be fetched as a collated document, either as a string or as a DOM document.

Model

- Models act like a factory object, and allow all other elements to be created for subsequent insertion.
- It is possible to access the group, connection, and import sets in a given model.
- It is also possible to get a set of groups with a given relationship ref.

Model

- There are three types of component / units sets:
- localComponents: Components which are described in this model document only.
- modelComponents: Components either described in this model document, or imported into it.
- allComponents: All components, including those which are not directly imported from imported models.

Imports

- CellMLImport represents an import element.
- Imports can be *instantiated*, meaning that the imported model is loaded into the API implementation.
- For access and reference counting purposes, imported model elements are treated as if they were a child of the import element.

Maths

- The MathList interface allows iteration over maths, as well as insertion of new maths.
- MathML apply elements and their descendants are accessed and modified using the Content MathML DOM API.
- My implementation of the API includes a complete implementation of the DOM and content MathML DOM APIs.

Events in CellML

- The CellML Event model is closely based of DOM Events.
- Can register an event listener with a CellMLEventTarget (any CellMLElement), and be notified when changes / insertions / deletions are made.
- Can also use DOM Events to monitor specific parts of MathML or extension elements if required.

Bootstrapping

- The CellMLBootstrap interface allows access to model loading / parsing facilities, as well as DOM facilities.
- As IDL is entirely object-orientated, it provides no mechanism to get the initial bootstrap object. This is done on a mapping-by-mapping basis (e.g. in C++ there is a special function to get a CellMLBootstrap object).

My implementation

- Written in C++ using the PCM.
- Built on top of the W3C DOM (all operations on models go through my DOM implementation).
- Includes comprehensive unit tests.
- Portable. Has been built on Linux (32 & 64 bit, big & little endian), AIX, OS X (x86 and PPC), and Win32 (MSVC & MingW32). Should be easy to get going anywhere with a standard C++ compiler.

Optional extensions

- The CellML API provides a number of optional extensions, which I will describe next.
- All optional extensions are cleanly separated from the API, in the sense that they use only API functions to access the model.

The CellML Context

- The CellML Context allows tools to maintain a hierarchy of open models.
- Tools and services can also register themselves with the CellML Context, so other tools can find them, allowing tool-to-tool communication.
- Interfaces can be 'annotated' by tools, to facilitate tool inter-operation.

CellML Code Generation Service

- The CCGS is an optional extension to the CellML API, which allows procedural code to be generated.
- The CCGS firstly computes an internal datastructure representing a sequence of 'procedural steps', then translates that sequence into a procedural language.
- It should be possible to add support for other languages by changing the second part of the process.

CellML Integration Service

- The CellML Integration Service allows models to be integrated, and run.
- The CIS uses the CCGS to generate C code, and compiles and loads this code dynamically.
- CIS supports integrators from SUNDIALS CVODE (US Department of Energy) and from the GNU Scientific Library.

Comments and discussion