libCellML Documentation

Release latest

Mar 22, 2021

Contents

CHAPTER 1

Contents:

1.1 Roadmap

Updated: 29 May 2019.

The current roadmap had evolved from that present in the 0.1.0 release of libCellML.

Each milestone may consist of several 'releases' and future requirements may impact the design and implementation of earlier releases of libCellML. Major changes in the API will be accepted up to the release of libCellML version 1.0.0.

Contents

1.1.1 High level objectives

libCellML is focused on CellML 2.0 and beyond.

- The implementation of libCellML should be driven by the requirements for supporting CellML 2.0 and beyond.

  - libCellML should be designed to support the CellML specification with the flexibility for extra restrictions, constraints, or additions coming from future proposals for changing the specification.
- libCellML should be able to parse models in CellML 2.0 and newer versions of the specification.
- libCellML will only serialise models to the current version of the specification.

1.1.2 Environment

This section will specify the environment for the development of libCellML.

- GitHub to host the primary libCellML source repository and issue tracker under the CellML organisation (current and former editorial board members).
- Development language: C++ with SWIG bindings.
- Build: CMake for generating cross-platform build rules.
- Test: using Buildbot on the ABI BaTS to run continuous integration testing.
- Test: unit testing to use gtest.
- Documentation: written in reStructuredText.
- Documentation: API and source code examples will be documented using C++-style Doxygen comments.

Requirements

- Documentation: made available on readthedocs.io. Read the Docs uses Sphinx for generating documentation.
- Documentation: is amenable for inclusion in external documentation efforts
- Development: Agile, test driven development where:
  - Functionality is more important than API stability in early releases.
  - Release early and often.
- Development: code review prior to acceptance into the primary repository using the pull request feature on GitHub.
- Development: objectives are added and broken down into incremental tasks.
- Development: a single task should be no more than two weeks.

We should avoid using non-standard system libraries unless there is a compelling reason. Once features are available, the API can be fine tuned in consultation with the CellML community.

1.1.3 Milestone 1: Python bindings, validation, code generation, and documentation

1. Python bindings.
   1. Wrap the libCellML API using Swig to be able to generate Python bindings for the library.
   2. Package the bindings so they can be easily installed across Windows, Linux, and macOS.
2. Load a CellML 2.0 model and validate it.
   1. Validate models against the rules defined in the current draft of the CellML 2.0 specification (currently an active document, outstanding issues regarding the new `reset` construct are likely to require updates to the validation implementation).

3. Code generation.
    1. Focus on non-DAE models.
    2. Generate code for any simulatable non-DAE CellML model in PMR (i.e., translated from CellML 1.0/1.1).
    3. Guide the code generated for a given model (e.g., a variable to be controlled from an external data source, SED-ML defined changes, etc.).
4. Documentation available.
    1. API.
    2. Tutorials/documented code examples.
    3. Integrating libCellML into common IDEs (Visual Studio, Qt Creator, and PyCharm).
    4. Provide documentation on the installation and use of the Python bindings across Windows, Linux, and macOS.

1.1.4 Milestone 2: units, JavaScript, and resets

1. Units.
    1. Checking units within mathematical expressions.
    2. Debugging assistance for model authors regarding units.
2. JavaScript.
    1. Use Emscripten to create a JavaScript API for libCellML.
    2. Provide a suitable packaged version of the JavaScript API for integration in common JavaScript environments (e.g., Node, Webpack).
    3. Document the installation and use of the JavaScript API.
3. Resets.
    1. Extend libCellML implementation to fully support resets.

1.1.5 Milestone 3: DAE models

1. DAE models.
    1. Code generation support for models with DAEs.

1.1.6 Milestone 4: advanced capabilities

1. High-order model manipulation (recall the discussion with Andrew McCulloch at the 8th CellML workshop).
    1. Again, it is outside the scope of libCellML, but helping tool developers provide these kinds of services is very important.

1.2 Current Thinking for libCellML

This document simply outlines some of the current rationale that has an influence on how the codebase is developed.

- Temporarily dealing with external documents that are stored on the local file system (relative to the current CellML model document).
  - Allows testing of the most common model import scenario (local files addressed with a relative URL).
  - Absolves libcellml from fetching files and communicating across the Internet.
  - Expect to provide another layer that would perform this role as a libCellML I/O library, which would allow the removal of this temporary inclusion.
  - No avenue to retrieve remote external references.
- Serialise and deserialise from a string.
- Present a useful interface not one tied to the XML serialisation structure.
- Validation is quite separate (you are free to make invalid CellML models).
- Public API treats MathML as strings only.
  - Internal to the code generation we are currently creating our own MathML object model based on an abstract syntax tree.
  - Minimal implementation to support the immediate requirement of code generation.
  - Expect to provide another layer that would handle MathML as a separate thing, potentially linking back to the advanced functionality envisions for symbolic analysis of the model.
  - Internal to the validator, the MathML strings are parsed into a DOM for use in schema validation against the MathML schema.

1.3 Use-cases for libCellML

1. **Create:** create a model from scratch and serialise it to XML (in each case the test is that the serialised model matches manually validated XML documents)
   i. an empty model
   ii. a model with a valid name
   iii. a model with an invalid name
   iv. a model with a single component
      a. a component with a valid name
      b. a component with an invalid name
   v. a model with two or more components
   vi. a model with three components and an encapsulation hierarchy
      a. one component encapsulating two children
      b. one component encapsulating a single child which in turn encapsulates a single child
      c. an invalid cyclical encapsulation hierarchy
   vii. manipulation of a model with multi-level component encapsulation hierarchy
      a. remove a top-level component
      b. remove an encapsulated child component
      c. change the name of a top-level component

d. change the name of an encapsulated child component

e. replace one component with a new component

f. take a component (remove the component and return it to the user)

g. determine if a component with a given name exists in a model or component

h. determine the number of components encapsulated by a model or component

viii. a model with imported components

a. import a component from a model

b. import two components from the same model as separate components

c. import a component into a hierarchy

d. import a component from a non-existent URL

ix. a model with units

a. a single base units with valid name

b. a single base units with an invalid name

c. a units which defines micro-Ampere * Kelvin / milli-siemens

d. the units from *1.ix.a* and *1.ix.c* and multiplies them

e. create a new base units e.g. 'pH'

x. a model with imported units

a. import a units from a model

1. with a valid name

2. with an invalid name

b. import a units from a non-existent URL

c. import a units from a model and scale it, prefix it, offset it, exponentise it

xi. a model with variables

a. model from *1.iv.a* and define a variable with a valid name and units dimensionless

1. with a valid variable initial value of 0.0

2. with a private interface

b. model from *1.iv.a* and define a variable with an invalid name and units dimensionless

c. model from *1.iv.a* and define a variable with a valid name and invalid units name.

d. a model with a single component containing two variables.

1. with valid variable initial values of 1.0 and -1.0, respectively.

2. one with an initial value of 1.0 and the other with an initial value of the first variable.

3. with one public and one public_and_private interface, respectively.

xii. a model with connections

a. model from *1.vi.a*, each child containing a single variable

1. with a private interface in the parent and public interface in the child components and connect the variable in both children to the parent.

2. with a public interface in all components and connect the variables in the children to the parent

xiii. a model with maths and variables

a. model from *1.xi.d.1* and define valid maths

xiv. a model with maths, variables and connections

a. model with two components, each containing two variables, maths, and one connection

2. **Modify:** modify models from 1.

i. add {components, units, maths, variables, connections}

ii. remove {components, units, maths, variables, connections}

iii. update {components, units, maths, model attributes, variables, connections}

3. **Load:** load each of the models from 1 and 2 (new models can be added for this part if required).

i. a model with imported components

a. a single component

b. a component with a hierarchy

c. a component from a non-existent URL

4. **Validate:** create, load, and modify models and then validate them (the test is that the models are correctly identified as valid or invalid, and for the case when they are invalid the correct reason is given, covering each rule in the specification).

5. Import CellML 1.0/1.1 models.

6. Export CellML 1.1 (and by extension CellML 1.0).

1.4 libCellML Object Model

1.4.1 Introduction

The object model described by this document is a very high level conceptual design. The focus is on a design to support the initial use cases from the use case document *Use-cases for libCellML*. This document is organic and is expected to change in accordance with community decisions/discussion.

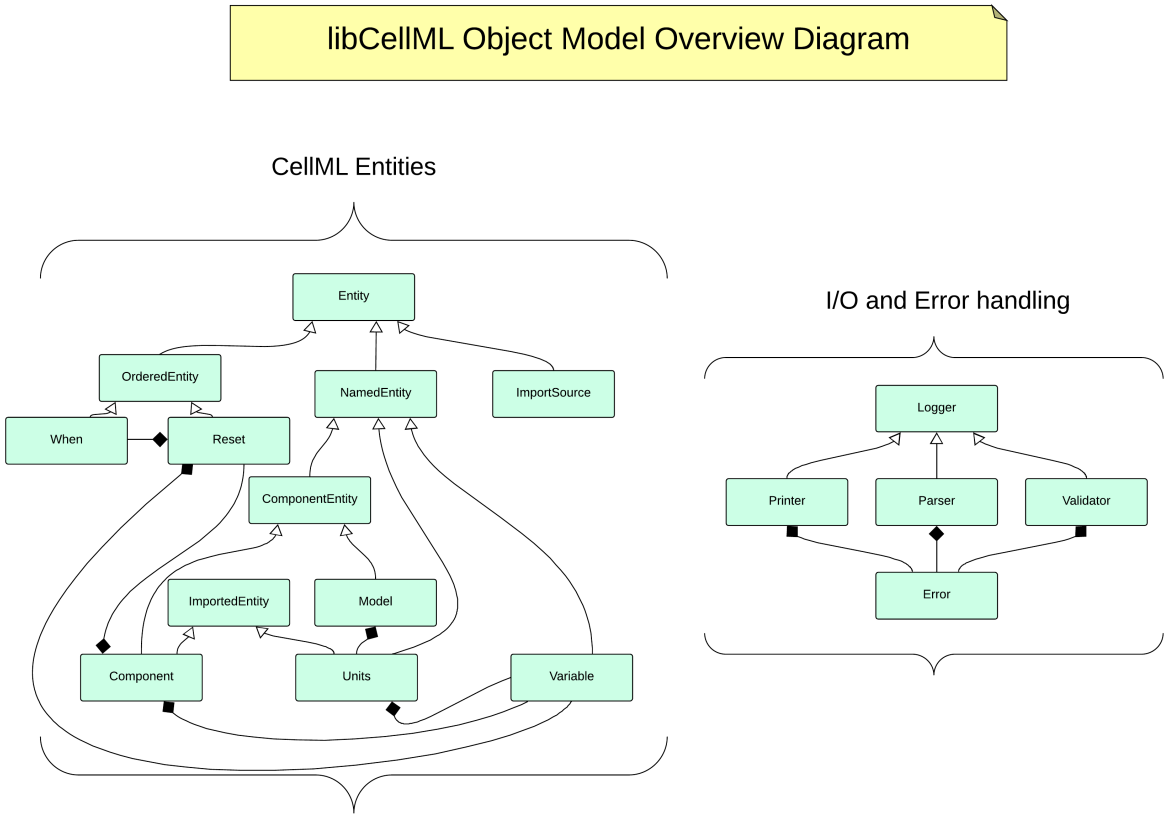
1.4.2 Overview of Object Model

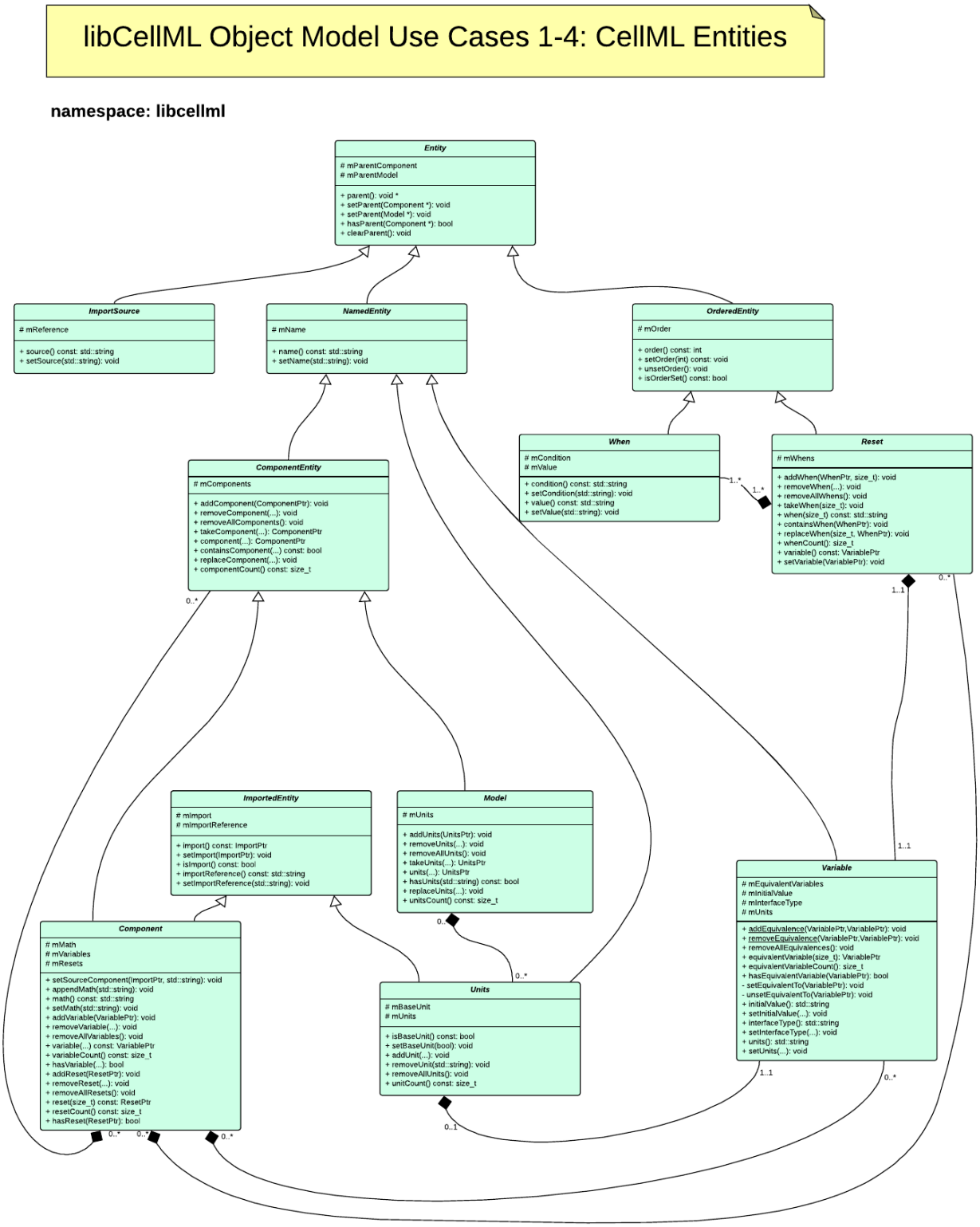
1.4.3 Object Model for Use Cases 1 - 4

1.5 API Documentation

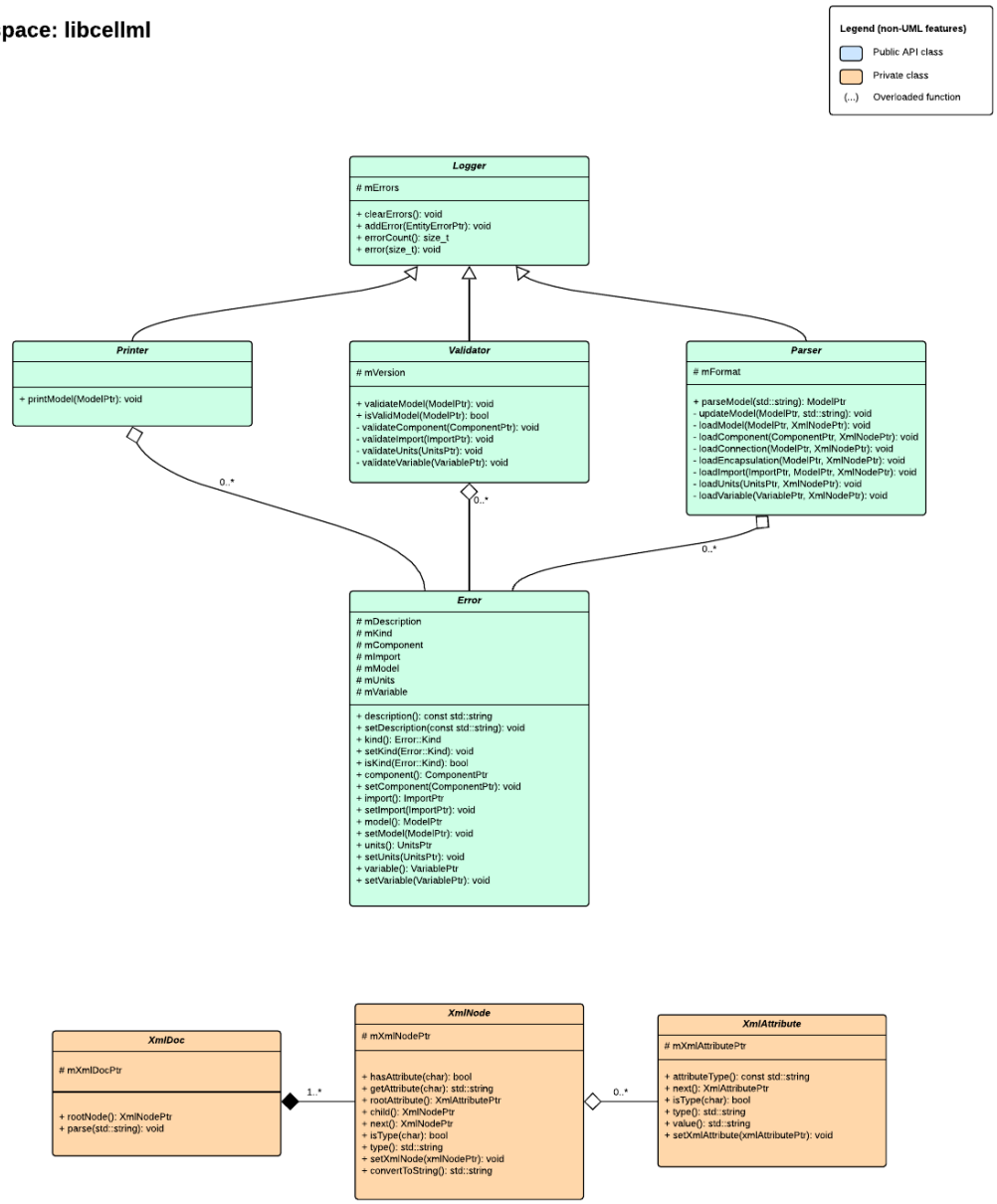
The API is documented through Doxygen, the generated files are available here.

Note: The Doxygen API documentation pages are **not** currently available on readthedocs.


libCellML Object Model Overview Diagram
CellML Entities
Entity
OrderedEntity
NamedEntity
ImportSource
When
Reset
ComponentEntity
ImportedEntity
Model
Component
Units
Variable
I/O and Error handling
Logger
Printer
Parser
Validator
Error


libCellML Object Model Use Cases 1-4: CellML Entities
namespace: libcellml
Entity
mParentComponent
mParentModel
+ parent(): void *
+ setParent(Component *): void
+ setParent(Model *): void
+ hasParent(Component *): bool
+ clearParent(): void
ImportSource
mReference
+ source() const: std::string
+ setSource(std::string): void
NamedEntity
mName
+ name() const: std::string
+ setName(std::string): void
OrderedEntity
mOrder
+ order() const: int
+ setOrder(int) const: void
+ unsetOrder(): void
+ isOrderSet() const: bool
When
mCondition
mValue
+ condition() const: std::string
+ setCondition(std::string): void
+ value() const: std::string
+ setValue(std::string): void
Reset
mWhens
+ addWhen(WhenPtr, size_t): void
+ removeWhen(...): void
+ removeAllWhens(): void
+ takeWhen(size_t): void
+ when(size_t) const: std::string
+ containsWhen(WhenPtr): void
+ replaceWhen(size_t, WhenPtr): void
+ whenCount(): size_t
+ variable() const: VariablePtr
+ setVariable(VariablePtr): void
ComponentEntity
mComponents
+ addComponent(ComponentPtr): void
+ removeComponent(...): void
+ removeAllComponents(): void
+ takeComponent(...): ComponentPtr
+ component(...): ComponentPtr
+ containsComponent(...) const: bool
+ replaceComponent(...): void
+ componentCount() const: size_t
ImportedEntity
mImport
mImportReference
+ import() const: ImportPtr
+ setImport(ImportPtr): void
+ isImport() const: bool
+ importReference() const: std::string
+ setImportReference(std::string): void
Model
mUnits
+ addUnits(UnitsPtr): void
+ removeUnits(...): void
+ removeAllUnits(): void
+ takeUnits(...): UnitsPtr
+ units(...): UnitsPtr
+ hasUnits(std::string) const: bool
+ replaceUnits(...): void
+ unitsCount() const: size_t
Variable
mEquivalentVariables
mInitialValue
mInterfaceType
mUnits
+ addEquivalence(VariablePtr,VariablePtr): void
+ removeEquivalence(VariablePtr,VariablePtr): void
+ removeAllEquivalences(): void
+ equivalentVariable(size_t): VariablePtr
+ equivalentVariableCount(): size_t
+ hasEquivalentVariable(VariablePtr): bool
- setEquivalentTo(VariablePtr): void
- unsetEquivalentTo(VariablePtr): void
+ initialValue(): std::string
+ setInitialValue(...): void
+ interfaceType(): std::string
+ setInterfaceType(...): void
+ units(): std::string
+ setUnits(...): void
Component
mMath
mVariables
mResets
+ setSourceComponent(ImportPtr, std::string): void
+ appendMath(std::string): void
+ math() const: std::string
+ setMath(std::string): void
+ addVariable(VariablePtr): void
+ removeVariable(...): void
+ removeAllVariables(): void
+ variable(...) const: VariablePtr
+ variableCount() const: size_t
+ hasVariable(...): bool
+ addReset(ResetPtr): void
+ removeReset(...): void
+ removeAllResets(): void
+ reset(size_t) const: ResetPtr
+ resetCount() const: size_t
+ hasReset(ResetPtr): bool
Units
mBaseUnit
mUnits
+ isBaseUnit() const: bool
+ setBaseUnit(bool): void
+ addUnit(...): void
+ removeUnit(std::string): void
+ removeAllUnits(): void
+ unitCount() const: size_t
0..*
1..1
1..*
0..1

libCellML Object Model Use Cases 1-4: I/O and Error Handling



1.6 Coverage Statistics

The output from the coverage testing using gcov is available here.

Note: The coverage testing pages are **not** currently available on Read the Docs. The coverage test sometimes (this behaviour has been observed on macOS using Clang) reports single lines containing only a closing curly brace as not covered. This is currently being treated as a false positive. This can be seen in the 'Missing' column of the test report where only single lines are reported. For the case discussed here, the reported line should only contain a single closing curly brace. In this situation, we will accept the missed coverage report.

1.7 Development Setup

This section describes how someone wanting to contribute to the libCellML project should set up their *working copy* for developing libCellML.

Contents

1.7.1 Overview

The libCellML codebase is hosted on GitHub and therefore Git is used to track changes. Before you begin, you will need to have a few pre-requisites satisfied:

1. GitHub user account (for the rest of this document we will call our user *andre*).
2. Git.
3. CMake.
4. Toolchain for building software (dependent on the operating system).
5. LibXml2.

Some optional tools can also be used to speed up compilation:

1. Ninja.
2. clcache (on Windows) / ccache (on Linux and macOS).

1.7.2 Pre-requisite acquisition

In this section, we cover the retrieval and installation of pre-requisites.

Git

Creating a GitHub user account is straightforward and can be done here. Installing a Git client is particular to each operating system and some pointers are offered below:

- Windows Git is available from a variety of vendors.

We commonly use Git for windows, but other popular Git implementations are:

- GitHub Desktop
- GitKracken
- Git SCM
- Ubuntu (and other Linux distributions) Git can be installed using the package manager with the command `sudo apt install git`.
- macOS Git is pre-installed and available from the command line.

CMake

CMake is the cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and to generate native makefiles and workspaces that can be used in the compiler environment of your choice.

Again, installation of CMake is particular to each operating system. For Ubuntu (and other Linux distributions), CMake can be installed using the package manager with the command `sudo apt install cmake`. For Windows and macOS, CMake provides installation binaries. Choose the binary appropriate for your operating system and follow the installation instructions.

Toolchain

The toolchain specifies the compiler that we will use to build libCellML. Toolchains are highly dependent on the operating system. When we test libCellML, we currently use Visual Studio on Windows, GCC on Ubuntu, and Clang on macOS. We recommend using these compilers on these systems, but feel free to use a different toolchain. We sometimes use the Intel C++ compiler to build libCellML, but we do not (at the time of writing) test with it.

The following sub-sections provide guidance on how to install the recommended toolchain on the major operating systems that libCellML supports.

Windows

Visual Studio is available to download from here. We currently test with Visual Studio 2015 (version 14), but later versions are known to work. The *Community* edition is more than sufficient for the needs of libCellML. To minimize the size of the installation, you may install only the C++ compiler. This component (and its requirements) is sufficient for building libCellML.

LibXml2

LibXml2 is already installed on macOS, so no further action is required on that platform. On Windows, we must install LibXml2 using the recommended implementation available from here while on Ubuntu LibXml2 can be installed using `sudo apt install libxml2-dev`.

Python bindings

Optional Python bindings are provided using SWIG. To compile the bindings, a SWIG installation is required, as well as a Python 2 or Python 3 installation (including the development packages on Linux systems, e.g. `python-dev`). Creation of Python bindings can be enabled/disabled at configuration time.

1.7.3 Optional tools

Ninja

Ninja is a replacement for *make*. It can be downloaded from here. Alternatively, on Ubuntu (and other Linux distributions), it can be installed using the package manager with the command `sudo apt install ninja-build`. On macOS, it can be installed using Homebrew with the command `brew install ninja`.

clcache / ccache

clcache (on Windows) and ccache (on Linux and macOS) are compiler caches. They cache compilations, which means that the first time they are used, compilation will be slower than normal. However, subsequent compilations will be significantly faster.

clcache can be downloaded and installed from here. Note that it will only work with paths that do *not* contain spaces. So, if you installed the recommended implementation of LibXml2, you will need to move it to a location that does not contain spaces and update your *PATH* accordingly (or uninstall LibXml2 and reinstall it in a *PATH* that does not contain spaces).

On Ubuntu (and other Linux distributions), ccache can be installed using the package manager with the command `sudo apt install ccache`. Alternatively, you can get the latest version from here, and build it and install it yourself:

```
./configure --prefix=/usr
make -j
sudo make install
```

On macOS, ccache can be installed using Homebrew with the command `brew install ccache`.

1.7.4 Setting up the codebase

The remainder of this document assumes that the above pre-requisites have been met. It covers setup from the command line. If you are using a GUI like GitHub Desktop then you will need to adjust the commands for the GUI you are using.

The goal here is to get a working copy of source code, tests, and documentation onto your computer so that you can begin development. To make this happen, you will need to fork the *prime libCellML repository*, make a clone onto your computer, and set up the Git remotes. In `fig_devSetup_githubRepos`, you can see a pictorial representation of what we are aiming to achieve.

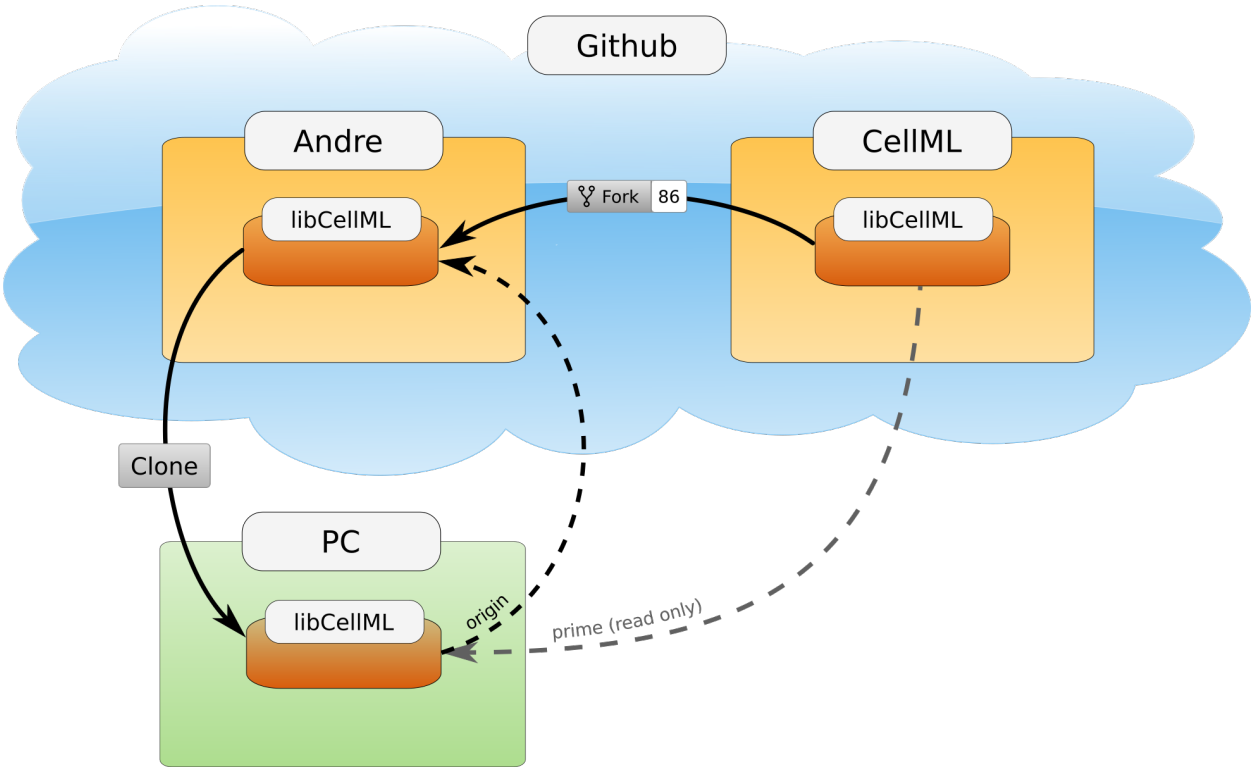


Fig. 1: Setup of repositories for development.

The four steps to getting set up are detailed below.

Forking your own copy

Login to GitHub using your credentials and go to https://github.com/cellml/libcellml.

Use the fork button to create a libcellml repository under your own account, see `fig_devSetup_githubFork` for locating this button.

Clone

You now need to clone the libCellML repository to your computer. You do this by going to your fork (in this example user *andre*'s fork) at https://github.com/andre/libcellml.

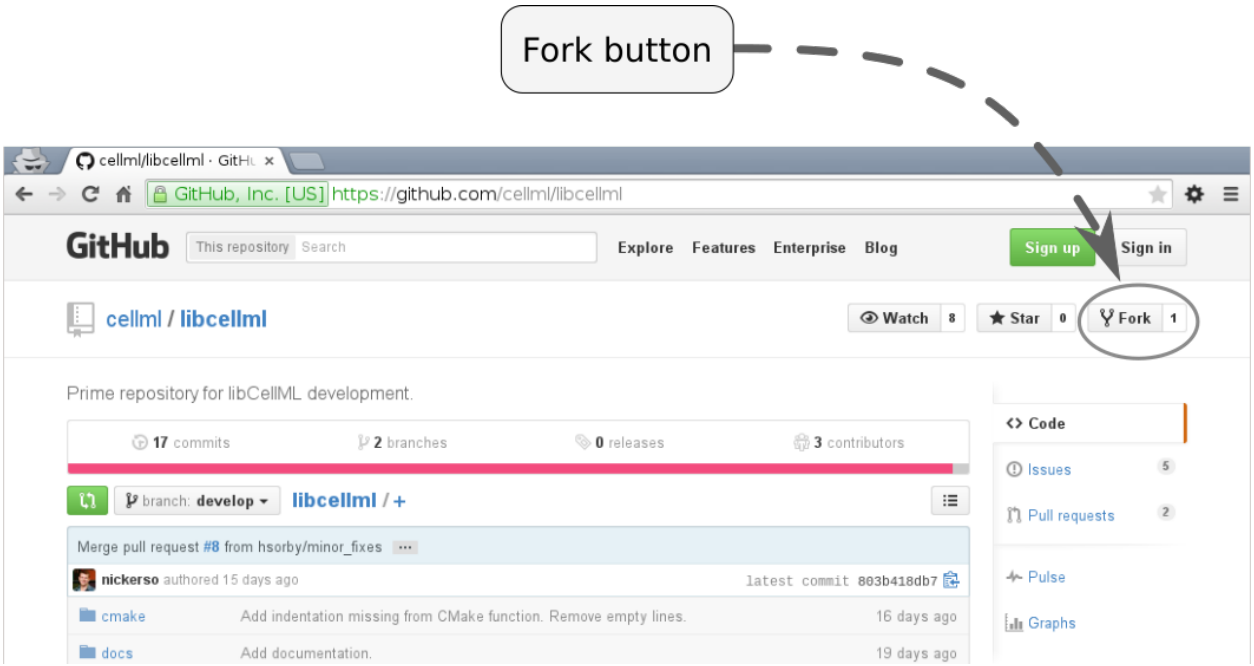


Fig. 2: Fork button for libCellML repository.

Warning: Do not try and clone this location substitute your GitHub username for *andre*. In all likelihood, it does not exist.

On the right hand side of the webpage, on your fork of the repository, you can get the link for cloning the repository to your computer, in our example:

```
https://github.com/andre/libcellml.git
```

Now clone the repository:

```
cd <somewhere/you/keep/development/code>
git clone https://github.com/andre/libcellml.git
```

Note: again, do not clone this location substitute your GitHub username for *andre*.

Set Git remotes

You now need to setup a read-only remote connection to the *prime libCellML repository*. Given that you are still in the directory where you cloned the libCellML repository from, do the following:

```
cd libcellml
git remote add prime https://github.com/cellml/libcellml.git
git config remote.prime.pushurl "You really did not want to do that!"
```

You have now added a new remote named `prime` and set origin as the default fetch and push location to point at repositories under your control on GitHub. Here, `prime` is a reference to the main definitive repository where releases are made from for the libCellML project. You have also set the `prime` repository as read-only by setting an invalid push URL.

1.7.5 Finally

You are all done and ready to start development, read *Building* on how to build libCellML. Then, read *Contribution* to get your changes into libCellML's prime repository.

1.8 Building libCellML

This document covers building libCellML from source. It is assumed that you already have the codebase downloaded and ready for building. The variable `LIBCELLML_SRC` shall be used to refer to the directory containing the `LICENSE` file for libCellML.

1.8.1 Build Directory

It is best to build libCellML outside of the source tree. To this end, create a build directory that is not the `LIBCELLML_SRC` directory. A sibling directory of `LIBCELLML_SRC` is a good choice, named something like `build` or `libcellml-build`. The variable `LIBCELLML_BUILD` shall be used to refer to the build directory.

1.8.2 Configuration

The libCellML library uses the CMake build configuration tool to configure the library. Version 3.2 or greater of CMake is required to configure libCellML.

The configuration options for the library are detailed in the following table. The command line options can be set with the `-D` flag, like so `-DBUILD_TYPE=Release`. Please note that in CMake GUI Configuration applications, the config variable is prefixed with `LIBCELLML_`

Options

Config	Default	Description
BUILD_SHARED	ON	Build shared libraries (so, dylib, DLLs).
BUILD_TYPE	Release	The type of build Release, Debug, etc.
COMPILER_CACHE	ON	Enable compiler cache (if available).
COVERAGE	ON	Enable coverage testing (if available).
INSTALL_PREFIX	/usr/lib	Install path prefix (platform specific).
MEMCHECK	ON	Enable memcheck testing (if available).
TWAE *	ON	Treat warnings as errors.
UNIT_TESTS	ON	Enable tests.

* In CMake GUI Configuration applications this option is given in full `LIBCELLML_TREAT_WARNINGS_AS_ERRORS`

From the command line (bash shell), libCellML can be configured to create an optimised shared object library like so:

```
cd $LIBCELLML_BUILD
cmake -DBUILD_TYPE=Release $LIBCELLML_SRC
```

Windows

When configuring libCellML on Windows, we may need to set the location of the LibXml2 library, which is dependent on the computer's environment settings. We can set the location of the LibXml2 library when we configure libCellML. When we configure libCellML, the location of LibXml2 can be specified through the command line by adding the parameter:

```
-DLibXml2_DIR="C:\Program Files\libxml2 2.9.6\lib\cmake"
```

to the configuration command. This assumes that the recommended LibXml2 binaries have been installed to the default location `C:\Program Files\libxml2 2.9.6`. Please note that this method will only work with the recommended LibXml2 binaries, LibXml2 binaries from other sources will not work in this way.

Windows CMake-GUI

When we use the CMake-GUI application on Windows, we first set the location of the source files and the location for the generated build files. `fig_devBuilding_windowsCMakeGUISourceBuildDirs` shows the source files directory and the build directory set for user *andre*.

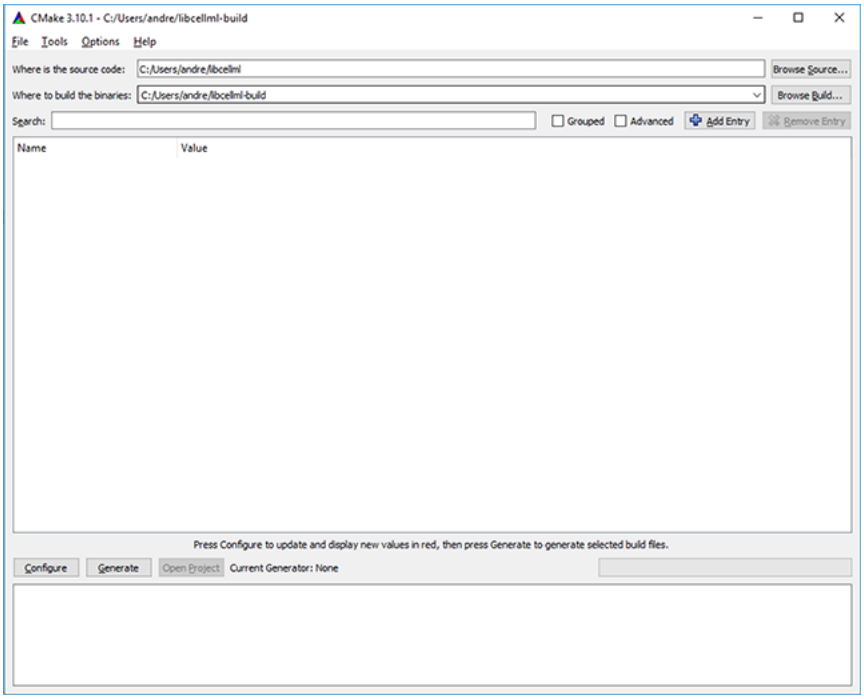


Fig. 3: CMake-GUI with source and build directores set for user *andre*.

When we press the *Configure* button, CMake performs an initial configuration. This initial configuration is likely to encounter an error because CMake is not able to find LibXml2. We can see in `fig_devBuilding_windowsCMakeConfigurationError` that this has happened for user *andre*.

We can resolve this error easily if we set the value of the `LibXml2_DIR` variable to the location of the LibXml2 `cmake` directory. `fig_devBuilding_windowsCMakeLibXml2DIRNotFound` shows the `LibXml2_DIR` variable with the value of `LibXml2_DIR-NOTFOUND`.

Setting the value of `LibXml2_DIR` to `C:\Program Files\libxml2 2.9.6\lib\cmake` and configuring again will result in a successful configuration (`fig_devBuilding_windowsCMakeLibXml2DirSet` shows a successfully configured `LibXml2_DIR` variable) from which build files may be generated using the *Generate* button.

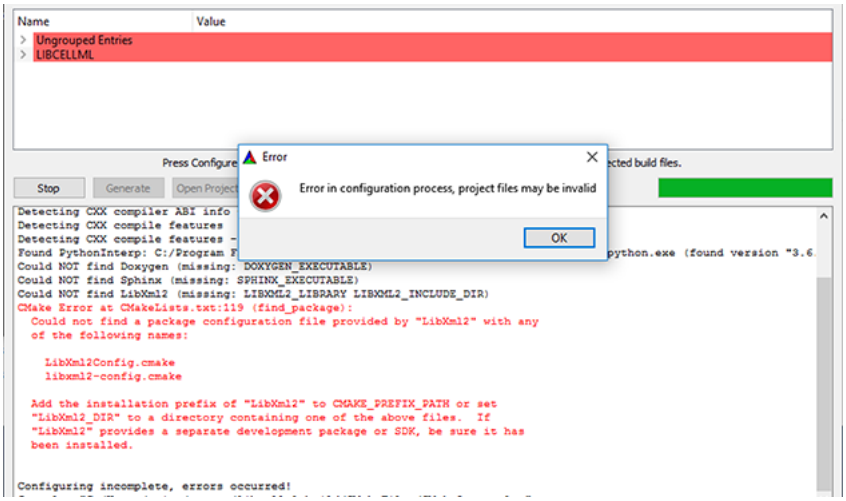


Fig. 4: CMake-GUI showing configuration error after initial configuration attempt.

Fig. 5: `LibXml2_DIR` variable with a value of `LibXml2_DIR-NOTFOUND`.

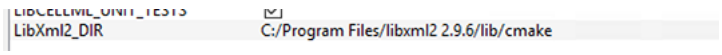


Fig. 6: `LibXml2_DIR` variable with a value of `C:\Program Files\libxml2 2.9.6\lib\cmake`.

If LibXml2 was not installed to `C:\Program Files\libxml2 2.9.6`, you will need to adjust the path to match your situation.

1.8.3 Build

Once the build scripts have been generated by CMake, invoke the build with the appropriate command. For `Makefile`-based configurations, the command is simply:

```
make
```

If testing is enabled, run the tests using the test target:

```
make test
```

or using the ctest application:

```
ctest
```

For a more verbose output, run:

```
ctest -V
```

1.9 Submitting Code for Testing

If you wish to test some new code without having to create a pull request you can. Buildbot has the capacity to build a libCellML compliant repository through the use of a ForceScheduler. To make use of this facility, you will need to authenticate with the Buildbot system. Access to this facility is granted on request to David Nickerson. The Buildbot system uses MD5 encrypted passwords as created by htpasswd. With your request to David include the output of this command:

```
htpasswd -n <your-chosen-username>
```

The website http://www.htaccesstools.com/htpasswd-generator/ can be used if you do not have access to the htpasswd application.

When your request has been dealt with, you will be able to login to Buildbot and submit code for testing.

1.10 Contributing

This document covers the process to follow for getting your changes into the *prime repository*. While there are many types of contribution, this section focuses on contributions made through GitHub and Git, or in other words assets that are managed using the version control system. It is assumed that *Setup* and *Building* have already been read and followed.

Contents

1.10.1 Overview

For any body of work intended for the *prime repository* start with a GitHub issue. The issue can be used to discuss the topic and clarify any problems related to it. Once progress has been made towards addressing the issue, a pull request is created that references the issue.

Reviewers provide feedback on the changes by adding comments to the pull request or associated commits. The Buildbot `build/test` procedure will run each time changes are pushed to the pull request's branch, and the results are displayed in the pull request view.

Once all the changes and reviews are complete, one of the *prime repository* owners will merge the pull request into the prime repository, onto the `develop` branch.

Note that a bug is just a type of issue, and that resolving the bug should have both the implementation to fix the bug and a test that triggers the bug.

`Figure %s` gives a graphical overview of the developer contribution process. For more details, see the text below.

1.10.2 GitHub Issue

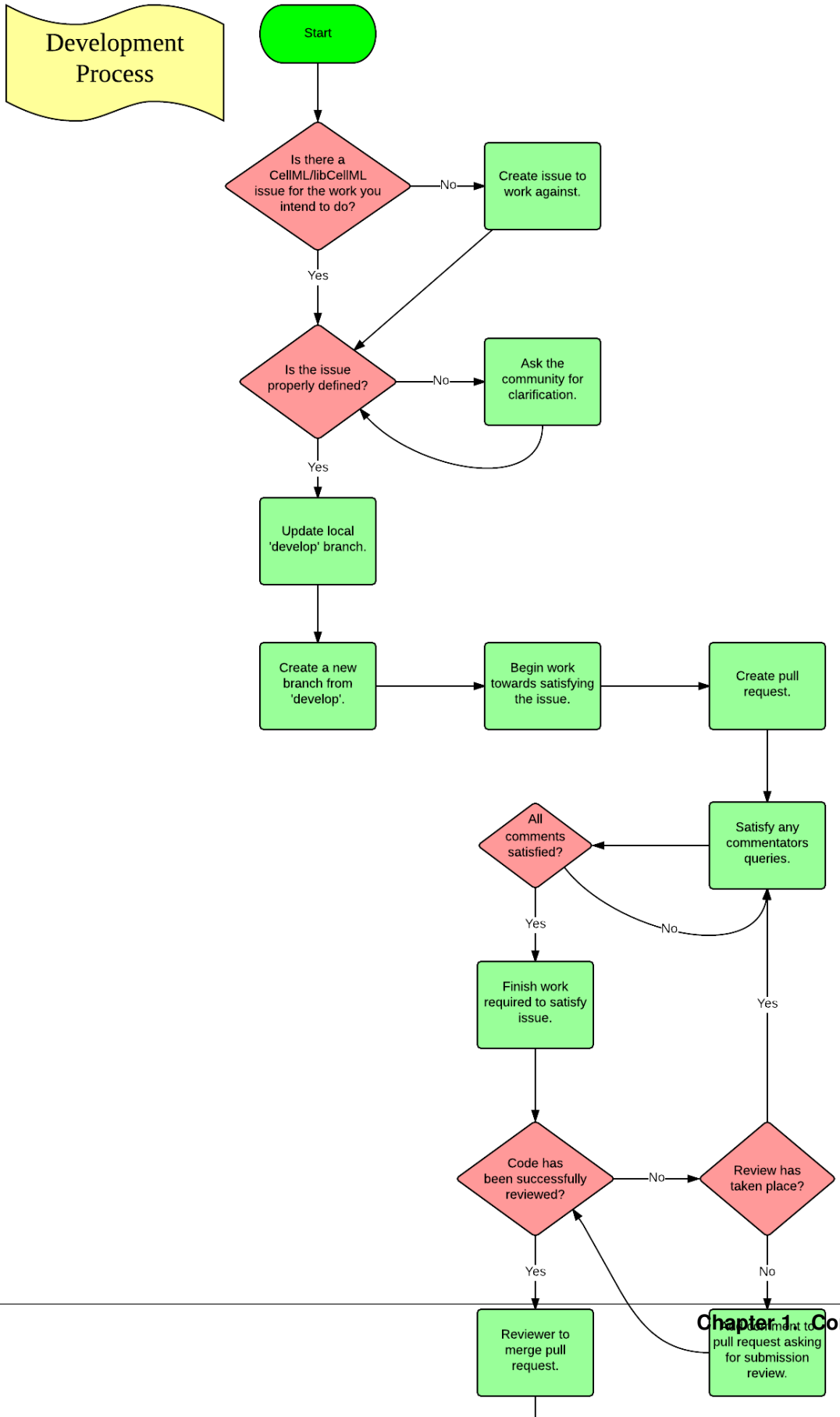
If an issue does not exist for the required work (e.g. implementation of a feature, fixing of a bug), then create a new one. The issue is the place to discuss the particulars related to the issue, discussions on determining the scope of the issue or clarification of any points that are unclear.

Labels

A GitHub issue may be assigned labels by the project administrators to help identify its status at a glance. General labels currently used for libCellML are:

- **Bug**: the issue identifies a malfunction in the current codebase.
- **Feature**: the issue constitutes a request or plan for a new feature.
- **Needs tests**: the issue requires test(s) to be complete. This may refer to a bug report, contributed code, comments, etc. in the issue.
- **Needs documentation**: the issue requires documentation to be complete. This may refer to a bug report, contributed code, comments, etc. in the issue.
- **Needs reviewing**: the issue requires further review from project participants to be complete. This may refer to a bug report, contributed code, comments, etc. in the issue.

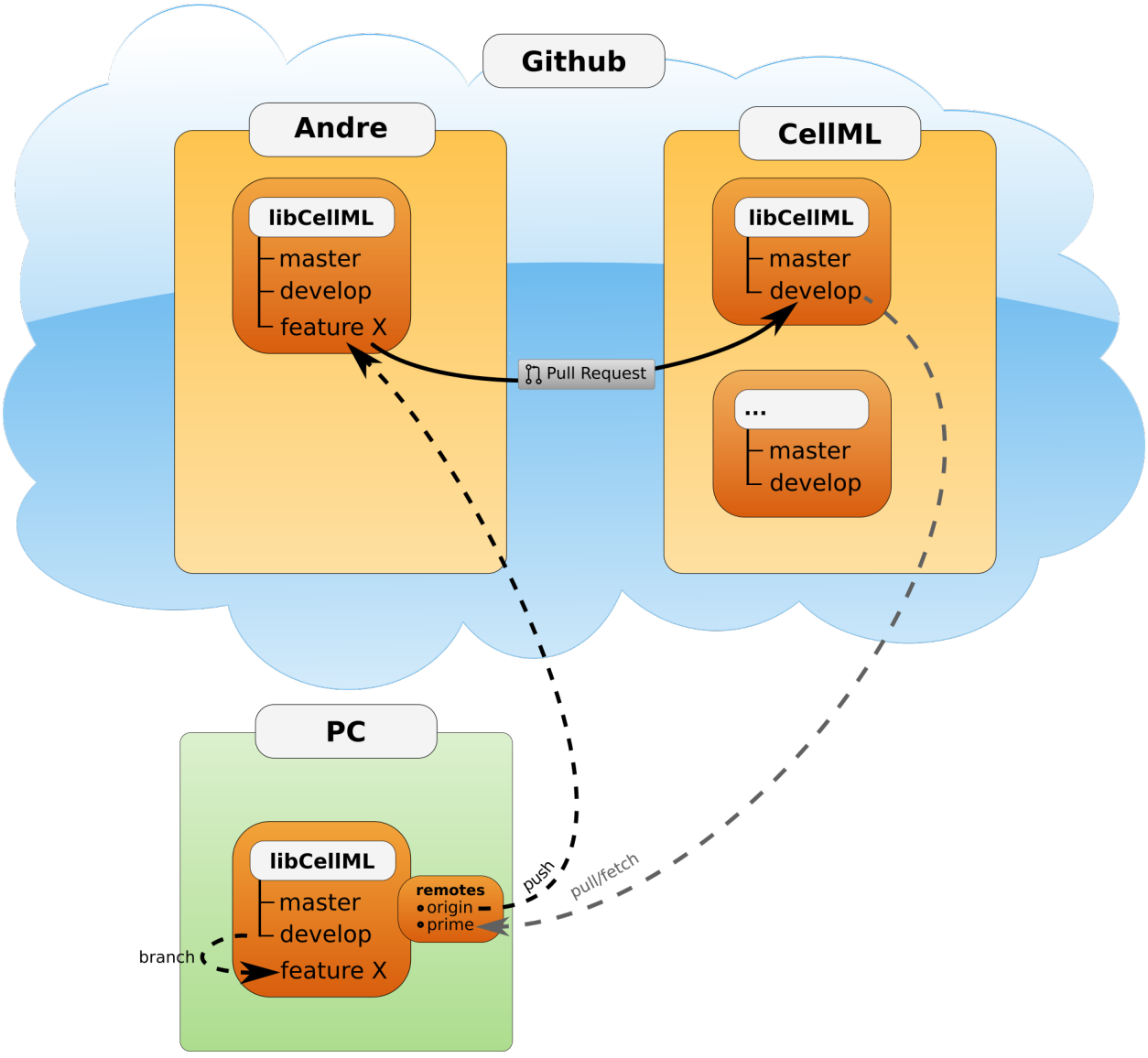
In addition, a **Platform** label may be used to identify the issue as specific to a given platform (Windows/Linux/macOS). **Milestone** labels may be used to project when a feature is expected to be complete and/or indicate the priority of a


Development Process
Start
Is there a CellML/libCellML issue for the work you intend to do?
No
Create issue to work against.
Yes
Is the issue properly defined?
No
Ask the community for clarification.
Yes
Update local 'develop' branch.
Create a new branch from 'develop'.
Begin work towards satisfying the issue.
Create pull request.
Satisfy any commentators queries.
All comments satisfied?
No
Yes
Finish work required to satisfy issue.
Yes
Code has been successfully reviewed?
No
Review has taken place?
Yes
No
Reviewer to merge pull request.
pull request asking for submission review.

given issue. Higher priority issues will take precedence and therefore be assigned a more immediate (lower) milestone number.

1.10.3 Topic Branch

A topic or feature branch is a branch that is local to you (and anyone you collaborate with), it is a branch that will not be available from the *prime repository*. All development work should be carried out on a topic branch, for example any major feature that you work on or minor bug fix. Before creating a local topic branch, pull the latest changes from the *prime repository* `develop` branch.



Following this process will make it easier to have multiple topic branches at once and keep them in-sync with the *prime repository* `develop` branch, which will in turn make it easier to manage multiple pull requests.

The following Git command line commands show an example of how to create a topic branch for fixing a (hypothetical) bug described in *issue #123*:

```
git fetch prime develop
git checkout develop # Not required if already on develop branch
git merge prime/develop
git checkout -b issue123
```

1.10.4 Test Driven Development

Test driven development entails writing a test that covers the intended functionality (this may require a suite of tests to be written) and no more. The tests will require some skeleton implementation so that the test(s) can compile but by definition not pass, at least not pass all the tests. The purpose of this is two-fold:

1. write the test(s) first, set out the intended design that can be shared through a pull request; and
2. implement the skeleton that will include the documentation clearly describing the intended purpose.

Following this contribution process allows others to comment and make corrections before time is spent on the functional code.

It may be necessary to refactor the current design to enable the easiest possible way to add the new feature. This is a good thing as the quality of the design improves and this makes it easier to work with in the future. Refactoring means improving the code without adding features, and the tests provide validation that the refactored code performs as well as before.

For simple or obvious bugs, which have fallen through the testing gaps, just the implementation is fine.

1.10.5 GitHub Pull Request

Once some changes have been made and local commits committed, push your changes to your GitHub libCellML repository (refer to `Figure %s`). From there, create a pull request from your topic branch to the *prime repository* `develop` branch. When creating the pull request, make sure to add in the comment `Addresses issue #123` (of course, replace the number 123 with the actual number of the issue you are addressing), or something to that effect. This will create a link between the issue and the pull request enabling other people to see that you are working on this issue and comment on your work.

The following Git command line commands show an example of how to add all files, commit the changes and push them to a GitHub repository for the first time:

```
git add .
git commit -m "Descriptive message about the changes made."
git push -u origin issue123
```

The `git add` and `git commit` commands should be obvious, the `git push` command sets the local branch `issue123` to be linked with the remote branch `issue123` in the origin (the default shorthand for your libCellML repository on GitHub) repository. This branch will be created in the origin repository if it does not already exist.

To create a pull request from one GitHub repository to another, follow the instructions here.

1.10.6 Satisfy Comments

It is important to respond to all feedback appropriately, the review process will check to make sure that all comments have been dealt with. Feel free to respond to comments as appropriate, e.g. through code changes, posting a direct reply, etc.

1.10.7 Review

It may happen that submitted work is not reviewed immediately or the work is finished before any comments have been made. If this is the case add a comment to the pull request asking for the submission to be reviewed. An email will be sent out to the repository owners who will respond and review the submission, please remember that everyone is busy and it may not happen right away.

1.10.8 Completion

To complete the process, it is required to have two owners of the *prime repository* comment on the pull request that they are satisfied that the work on the issue is complete and also that the feedback has been addressed, in essence that they are "happy" to merge the submission. For small submissions, it is sufficient for the second owner to show satisfaction by performing the merge. For larger submissions one of the owners will post a comment on the issue notifying subscribers that they intend to merge the pull request. If no further objections are raised, the pull request will be merged and closed.

A little reminder for the repository owners to check that the *Review Process* has been followed/(is going to be followed) when merging the pull request.

1.11 Review Process

1.11.1 Check for the Green Tick

Before accepting a tranche of work into the libCellML prime repository check that Buildbot has tested and passed the code. The status of the code is shown in the last commit of a pushed group of commits in the pull request. The last commit will have a red cross for a failed build or a green tick for a passed build. Obviously, make sure that the last commit has a green tick before merging.

1.11.2 Read the Documentation

The documentation for the project is built as part of the testing process. The details link at the bottom of the pull request web page will take you to the Buildbot build of the library. This page shows the results of the unit tests for each target operating system, the results of the coverage test, the results of the memory check test, and the results of the documentation build.

The Documentation Builder link (entry 2 in step 5) will take you to the build for the documentation. On this page, you can see the steps taken to build the documentation. In the last step of the build (step 7), there is a link 'dox' (entry 2) that will take you to the built documentation.

The documentation should be reviewed in its final format particularly those parts of the documentation that (should) have changed due to the current pull request. The API documentation (generated by Doxygen) can be reached from the *API Documentation* page. The coverage statistics for the library (generated from gcovr) can be reached through the *Coverage Statistics* page.

1.11.3 Comments Resolved

All comments on the pull request and associated issue should be responded to and satisfied. It is the reviewers responsibility to check that this has happened before merging the pull request.

1.11.4 Coding Standard

Currently, there is no fully defined libCellML coding standard set, but the Google C++ Style Guide can be considered a baseline for the standard of code that is expected for libCellML. See the *Coding Standards* document for deviations from this guideline.

1.11.5 Merging

When merging a pull request, the reviewer should add a comment so that the corresponding issue is closed. This can be done by adding a directive to the merge commit, like so:

```
closes #123
```

where the numeral corresponds to the issue that needs to be closed. You can use other directives that will achieve the same outcome, here is a list of all directives that will work on GitHub.

1.12 Coding Standard

The coding standard for libCellML follows that specified in the Google C++ Style Guide, but it does not have to be followed to the letter, other people have not so positive opinions about the quality of the google style guide. The coding standard in use for libCellML is in evidence in the code itself, so new code should be consistent with what is already there.

In essence, we seek code that looks good, is easy to read and has great documentation. We do not want to spend time discussing the minutiae of the coding style.

The following is a list of exceptions/deviations from the google style guide that have been agreed upon for libCellML software development purposes. Think of it like case-law.

- Lower camel case class method names.
- Indent core code 4 spaces at a time (no tabs); for CMake files use 2-space indentation.

1.12.1 Doxygen Comments

- Code-words (e.g. `true`/`false`, `std::string`) should be styled as typewriter text with a preceding "@c".
- Doxygen comments should be sentence-style: beginning with capitalisation (except code-words) and ending with punctuation. However, they do not need to form grammatically correct sentences.

1.12.2 Test Naming

- Names should respect the lower camel case convention.
- Names should be explicit enough to identify the specific code features they cover.

1.12.3 Variable Naming

The following rules for naming of variables should be followed.

- Class member variables: `mMyClassMemberVariable`.
- Function parameter variables: `myFunctionParameterVariable`.

- Local variables: `myLocalVariable`.

1.13 Contributors

All contributors to the libCellML software project agree to the terms and conditions of the Apache v2.0 license.

1.13.1 List of Funding Organisations

- University of Auckland
- VPR

1.13.2 List of Contributors

The following is a list of contributors (in surname alphabetical order) who have contributed lines of source code to the libCellML project on or before 2019-09-23.

- Ted Ahmadi
- Robert Blake
- Michael Clerx
- Jonathan Cooper
- Alan Garny
- David Ladd
- Massimiliano Leoni
- Kyle Medley
- Keri Moyle
- David Nickerson
- Hugh Sorby

For an up-to-date list of contributors see https://github.com/cellml/libcellml/graphs/contributors.

1.14 Glossary

Prime repository

Prime libCellML repository The repository at https://github.com/cellml/libcellml is the definitive repository for the software and used for creating software releases. We will refer to this repository as the **prime** repository.

1.15 Options

Config	Default	Description
BUILD_SHARED	ON	Build shared libraries (so, dylib, DLLs).
BUILD_TYPE	Release	The type of build Release, Debug, etc.
COMPILER_CACHE	ON	Enable compiler cache (if available).
COVERAGE	ON	Enable coverage testing (if available).
INSTALL_PREFIX	/usr/lib	Install path prefix (platform specific).
MEMCHECK	ON	Enable memcheck testing (if available).
TWAE *	ON	Treat warnings as errors.
UNIT_TESTS	ON	Enable tests.

* In CMake GUI Configuration applications this option is given in full `LIBCELLML_TREAT_WARNINGS_AS_ERRORS`

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

CHAPTER 3

Supported by:

Index

P