CCP4i2 for Programmers

Liz Potterton

Introduction

This article will give an overview of the new CCP4i2 from the programmer's point of view and an outline of what is required to develop a new task interface. A good feature of the first Tcl-Tk based CCP4i was the ease with which developers with minimal knowledge of the Tcl language or the basic workings of CCP4i could create task interfaces. The same will be possible in the new Python based CCP4i2 and many features of the overall design of the new system will be familiar to developers who have worked with the old one.

The most important objective for the first release of the new GUI is to be easy-to-use for novice crystallographers; requiring minimal user input and providing simple job reports and clear hints on what to do next. To meet this objective a task developer does need to do some additional work in designing the best GUI and reports and providing more hints and error trapping.

As with the first CCP4i, a task in the new system requires two different sorts of script: one is a wrapper for the program and the other specifies the GUI for the task. The connection between the GUI and the scripts is a *def* file that specifies the data that appears in the GUI and is passed to the wrapper script. There was an equivalent file in the old system but the file format has changed to XML and there is a much stronger data typing in the new system.

Crystallographic Data Model and Python Data Classes

An important feature of the new system is a more rigorous data model for the crystallographic data. This is built up from basic Python classes such as *CInt* and *CList* (integer and list!) to representations of complex entities such as ensembles and rigid domains. Each class has qualifiers that provide information to enable better data validation (for example allowed value ranges) and defaults. In the core CCP4i2 there is a library of widget classes - one for each data class. So the GUI developer has a library of specialised widgets to create a task GUI and needs only to specify the layout and provide some explanatory text. The data and widget libraries are still being extended to cover new areas.

Another innovation is that the parameters must be clearly classified either as input data, output data or control parameters. The input data and output data should include the obvious input/output files but also any data specific to the crystal under study (e.g. NCS, heavy atoms etc). This classification makes it possible for the CCP4i2 system to automatically extract the input and output data for each job run and to save them to a database so the flow of data can be tracked.

The *def* file that specifies the data and parameters for a task must be organized into three containers for the three categories: *inputData*, *outputData* or *controlParameters*. Providing all the appropriate qualifiers will also help to improve the GUI. The *def* file, like all other utility files in the new system, is in XML format. There is a graphical utility, *defEd*, that developers can use to create and edit *def* files. This utility lists all the data classes (with some documentation) and also has an interface to enter the *qualifiers* for each selected class.

The *def* file specifies the parameters for each task and when a task is run (usually referred to as a *job*) the GUI creates an *input_params* file with the specific parameter values for the job.

Program Wrappers: *CPluginScript*

As with the first CCP4i each program run from the GUI must have a script wrapper that handles the input and output of that particular program. The Python base class for a wrapper is *CPluginScript* that provides functionality such as easy access to the contents of the *input_params* file and a mechanism to run programs as a separate process. It also has a method to return appropriate path names for files such as command files and log files and it is important to use this facility to ensure consistent file organization.

A task developer must sub-class *CPluginScript* but may need to do no more than provide a mechanism to write the command line and/or command file for the program. This can be done in either of two ways:

1. reimplementing *CPluginScript.makeComandAndScript()*
2. using command template file (equivalent to the *com* files for the first CCP4i)

Projects and the Database

As with the first CCP4i the user organizes work into projects. There is no fixed rule for what constitutes a project – one suggestion is that the result of a successful project is one structure for PDB submission! Projects can now be grouped into hierarchies so that it is possible to group together similar projects. Each project is associated with a project directory and within this directory each job has its own directory for input and output files and sub-jobs have sub-directories etc. The organization of directories and data files within a project is strictly controlled within CCP4i2. The destination of output files is determined automatically. So the user has no choice in naming output files but the GUI does provide tools to export files.
The database is used to keep track of projects and jobs but does not normally store crystallographic data – these remain in the PDB, MTZ and other files. The database is based on SQL and currently uses *sqlite* though it is intended that other SQL systems could be substituted in. The default arrangement is to have one database file per user with all of the users' projects stored in this

same file. Alternatives to this arrangement will be possible. There are mechanisms to allow colleagues access to your database.

From the developers perspective there is a Python module providing access to the database though implementing a task interface should not require direct database access as the database is automatically updated by the core CCP4i2 system.
But other programs will need access to the database in order to record aspects of the structure solution performed outside CCP4i2.

Job Reports

After each job the user can view a report file that should be a concise summary with information presented as graphically as possible. Reports can contain graphs, tables, pictures generated on the fly in CCP4mg, buttons to open Coot, CCP4mg or other programs and folders to show/hide more detailed information. The report is an HTML file but when it is displayed in the CCP4i2 browser it can contain specialized Qt widgets such as *Pimple* that is a Python/Qt/matplotlib-based replacement for *Loggraph,* implemented by Stuart McNicholas. We can provide other specialized widgets where necessary.

The mechanism to create job reports is an evolution from the *Baubles* system. The report is created automatically after the task has run based on a *report template file* that defines the layout of the report. The actual data that appears in the report is taken from XML files output by the program and/or the Python wrappers. Aside from needing to be XML there are no constraints on the content of the program/wrapper data files and current existing files can probably do the job. The *report template file* uses an xpath mechanism (www.w3schools.com/xpath) to specify the data to be extracted from the data files. There is a *report generator* utility that will process the *report template file* and read the necessary data files to create a report. The task programmer must provide the *report template file* and may need to add code to the program wrapper to analyse and output data for the report.