

The design and construction of a Conditions Database Interface.

Marcel Cutts

September 2010

Abstract

The Conditions Database is method of storing and accessing with the data that characterizes detectors, a requirement for modern particle physics reconstruction and analysis techniques. Initially developed at CERN, the Conditions Database has grown to be incorporated in a large number of projects globally, including the International Linear Collider group's software toolkit. This report aims to outline the progress made on the construction of an upgraded graphical user interface for Conditions Database. The design process and considerations will be explained and commented upon, and the method of implementation described and justified. The current progress will be summarised, and finally, an evaluation of current work and an outlook onto future tasks is given.

Contents

1	Introduction	2
2	Design	2
2.1	Framework choice	2
2.2	Software Structure	3
3	Implementation	3
3.1	Main Window	3
3.2	Menu Bar	4
3.3	Object Control	4
3.4	Conditions Object	4
3.5	Grid	4
3.6	Tagging System	5
4	Progress and current state	5
5	Outlook and Evaluation	5

1 Introduction

The Conditions Database (CondDB) is a C++ application protocol interface (API) for databases, which began development at the European Center for Nuclear Research (CERN) during 2001. A similar concept was previously utilised at the Stanford Linear Accelerator Center (SLAC) which inspired the CondDB project to begin alongside the construction of the Large Hadron Collider (LHC). CondDB can be thought of as a layer that can be “wrapped around” all kinds of common database types, such as MySQL, Oracle or Objectivity to make them appear identical. This has several key advantages, for both researchers and the information technology (IT) departments that must maintain them. Firstly, it creates a scenario where IT departments need to only maintain one type of database, and any necessary changes can quickly be wrapped around supported databases, in contrast to having to update several database types using a variety of methods. For researchers, it means data objects from multiple different experiments can be accessed without requiring to learn a new set of database languages and syntaxes, increasing the ease of collaboration between groups. The CondDB system has since been adopted by a variety of particle physics groups globally, including the International Linear Collider (ILC) group, who have incorporated it into the Linear Collider Conditions Data (LCCD) toolkit.

The purpose of these databases is to store “conditions data”, which characterizes the state of a detector at a given time period. This data could include such factors as environmental parameters, detector calibration data and detector geometry, all of which are stored in a conditions object. This data is necessary to read and store as during reconstruction or analysis of detector data, it is important to know the state of the detector to understand what the acquired event data represents.

Currently, two applications exist which offer the ability to interact with a CondDB, a command-line interface (CLI) program and a graphical “Tcl” script. While the command-line program is often used and considered powerful, a graphical user interface (GUI) offers several advantages which the CLI cannot match. The core advantage of a GUI is the ability to visualise large sets of data, for example in the form of a graph, which is valuable when attempting to find trends in the data. Additionally, it is also often easier to group and sort data when it can be displayed simultaneously. Several user-based advantages exist through using a GUI, such as ease of use through a more intuitive medium. While the command line tool is powerful, it does require the user to understand the program. A user that isn’t sufficiently competent could make significant unwanted changes to the database.

As mentioned above, a Tcl based GUI for the CondDB system exists, however, it contains several key flaws. The primary complaint found in practical use is

the instability of the program. Crashing is often reported by users, and no “crash recovery” function exists, the user needs to restart the work that he or she had been conducting up to that point. This is partially as a result of the coding, but also of the nature of the coding language itself. “Tcl” is a scripting language intended for rapid prototyping, which can be used to create GUIs when combined with the “Tk” graphical framework, and the scope of the program has outgrown the capability of these two components. This causes the program to be difficult to extend, modify or “fix”. Additionally, a common user complaint is of a lack of error checking on the part of the program. For example, the current Tcl GUI can does not check the existence of tags or folders before applying new tags, leading to null referenced objects existing in the database. This can cause long term annoyances for all users of the CondDB.

This report aims to outline the progress in the design and construction of an improved and stand-alone application to which seeks to replace the Tcl based database viewer. Design choices will be explained, and the implementation methods will be commented upon and finally, future prospects and additional expansion options will be explored.

2 Design

The various stages of the design work and research will be described below. These activities were performed before the code was written to ensure the construction process would result in clean and manageable code. When writing software without strong planning and design, it is common for factors or requirements to be overlooked resulting in an inelegant solution to a problem. This creates code that is more difficult to understand by both the original developer, and later others if they wish to extend or fix the code themselves. Furthermore, strong planning generally results in faster completion times, as the writing process can be focused on the required tasks more accurately.

2.1 Framework choice

The initial step in the creation of a GUI application is the choice of framework to utilise. The framework is the mechanism in which code is converted from a text format into an actual graphical entity that a user can interact with, and many choices exist. Research was conducted to examine which to utilise for the creation of the CondDB GUI with respect to several requirements. Due to the varied user-base, it is important that the application be portable and “cross-platform”, conveying that it can be used on a broad spectrum of computer operating systems. This ensures the application is portable and can be sent to global colleagues without concern that the end user may not have the correct computer profile to

use it. This requirement excludes a large number of very common frameworks which are native to a single operating system, such as the “Microsoft Foundation Classes” (MFC) for windows, or the Apple centric “Cocoa”. To improve on the Tcl based code, it was also preferred that the framework should be coded in C++, a more commonly known and used language among particle physicists. This is to ensure that if fixes or extensions are required, a large portion of users should be able to develop them. Additionally, a C++ application framework will be compiled, rather than interpreted like the Tcl script, normally resulting in significant performance increases within the program.

With these exclusion factors considered, a few candidates remained to be chosen from, however the decision was made to utilise the “Qt” framework. This was due to a variety of factors, such as Qt’s large widget library, but primarily due to it’s maturity and good documentation. Additionally, the Qt framework is also free in both cost and development constraints, being licensed under the GPL. The advantage of this is that no future developer should be impeded from further development of the software via cost or licensing laws, making it as accessible as possible.

2.2 Software Structure

It is common to design the structural overview of the software in broad terms before beginning to write code. This generally ensures that the code will be written in a cleaner, more concise way, which is an important factor in maintainability of the application over the upcoming years. In order to encourage extensibility, a focus on modular coding was used when designing the overview. This implies that there is a strong separation of many of the core portions of the application, with the intent that if the user would want to implement additional code, the user would only have to update a small portion of the program. This saves the user time in development and ensures that future users of this addition could also understand and expand upon the work. A tree diagram illustrating this concept can be seen in the Appendix (A1).

The key idea is that Object Control class, the segment that controls the receiving, storage and sending of conditions objects should remain consistent, and therefore other functions of the program can be “children” of this core functionality. For example, a user may want the application to be compatible with a new variety of conditions object, and so a “child class” of the object control is object creation, named Conditions Objects. The user only has to alter the code in the Conditions Object class with the parameters that the new conditions object may require, but the objects will still need to be stored by Object Control for use by the program in a consistent manner.

While Object Control is the “core” part of the pro-

gram that handles the portions of the program that complete the processes the program was designed for, it itself is a child of “Main Window” class, which is a separation for convenience of future GUI programmers. The Main Window class dictates the general layout of the program, such as where to place and what size to use for the graphing portion of the program, or what buttons exist on the menu bar, but has no actual functionality of its own. This separation of function and broad graphical elements should ensure that developers who wish to extend the GUI do not have to understand or worry about the core functionality of object control, while inverse is true for developers wishing only to extend the functionality without having to contemplate the GUI.

This modular approach is hoped to ensure the program remains understandable and expendable throughout a number of iterations by distributed developers.

3 Implementation

This section of the report will focus on the challenges and techniques used in the writing of the software. Major classes, their purpose and functionality will be outlined, giving a broad understanding of the various purposes of each code block.

As the author had not previously developed applications using the Qt framework, the graphical elements were implemented first learn and understand the Qt structure with the intention to fill an empty GUI with standard C++ based functionality on a later date. This order was chosen as developing the core functionality without knowing how it could integrate with the GUI could result in a situation where the two simply were not compatible, but through learning and understanding the GUI portion first, the functionality could be planned and implemented cleanly with the prior knowledge of C++.

3.1 Main Window

The purpose of the Main Window class, as mentioned above in the “Software Structure” section of the report, is to control the general layout of the program’s graphical components, often referred to as “widgets”. It initialises the window itself, and the other core independent component, the menu bar

The Main Window establishes the outline parts of the database, it generates the outer window and the menu bar used to control the application. It also instantiates the Connection Manager class that is used to communicate with the database and send and receive objects. This connections class is a child of the Main Window rather than a child of the object control so the user can have the ability of connecting to a database without having to load the entire Object Control class. At the time of writing the Main Window class also instantiates an

Object Control class, but this could be altered easily to only occur on incoming objects.

3.2 Menu Bar

The menu bar is part of the Main Window class, and contains functions that may be useful to be user at all times. These include the ability to connect and disconnect from a database, exit the application and clearing all current objects from the screen. This is implemented through a series of “QAction” items, which can be combined with the Signal and Slot mechanism within Qt to act as a function call to various member functions of established classes.

3.3 Object Control

Much of the core conditions object functionality resides in the Object Control class. This class’s purpose is largely to store and manage conditions objects once they have been loaded. The conditions objects themselves have a custom class, named Conditions Objects, which contains all the parameters required to create a conditions object, which are then filled with data from the database. The conditions objects themselves are stored by the Object Control class using a “QList” template container class, and accessed via an index system. This provides the foundation of the sorting functions of the class. The “sort by tag” function, for example, will loop through the QList containing the conditions objects, and for each object check the tags it contains. If a valid tag match is found, the conditions object will be displayed upon the grid.

3.4 Conditions Object

A child class of Object Control, one of these is instantiated whenever a new conditions object is loaded from the database. It has a large number of member variables which can be set by the information from the database, or by the user via “setter” functions, which are public functions that can be used to change member variables. The “shape” of the conditions object, when displayed on the grid is determined by the length of the event, and its initial starting position on the time axis. The length of the event is measured from the database in terms of nanoseconds since the beginning of the year, nineteen-seventy (1.1.1970). This is converted into a four co-ordinate set in the QRectF class, in which four “double” values define a rectangle, where the first two define the initial x and y co-ordinate respectively, and the final two the length and height respectively. This is illustrated in Figure 1 below.

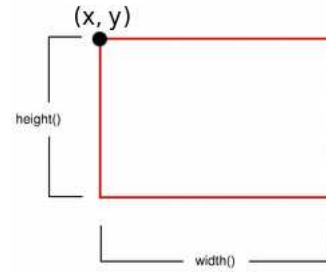


Figure 1: QRectF dimension systems

Tags for the object are again contained within a QList template class. This ensures the object can have a large number of tags, saved as QStrings, without any problem. These can also be set by the user via “setTag” setter public function.

3.5 Grid

The grid is a “graphing area” of the program, which is where the Conditions Objects are displayed. The x-axis represents the interval of validity of the Conditions Objects represented as time, and the y-axis corresponds to the “version” or “layer”, as it is often the case that there are multiple Conditions Objects at any given interval. The grid itself is coded in a QGraphicsScene subclass, instead of a more conventional QPainter class. While the “QGraphics” based classes were only added to Qt recently, they have a greater focus on object interaction with the user, and being able to display multiple independent objects presented it as a more convenient choice than that of QPainter, which has a greater focus on simply drawing and displaying objects. The QGraphics class set operates on the principle of having a “scene” and a “view”, a scene being the location where graphical elements can be placed, and the view is the selection and how the graphical elements are observed. This simplifies actions such as zooming, as a consistent scene can be created, in which only the view change. This is in contrast to the earlier “QPainter” method, in which the scene would have to be redrawn after each action.

When no conditions objects are present in Object Control, the grid simply displays an empty text statement, however when an object is loaded, a background grid structure is drawn. The background is a large set of blue intersections divided into time intervals and separated into layers. An example of this, with two condition objects represented by gray bars, can be seen in Figure 2.

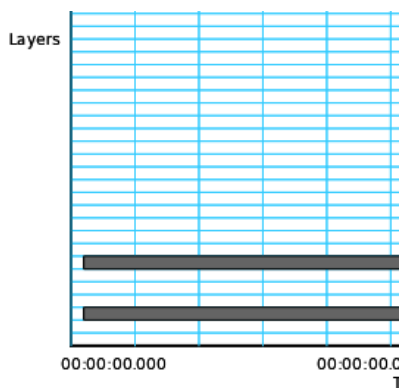


Figure 2: Snapshot of a portion of the grid.

A problem can arise when dealing with Conditions Objects, as the their interval of validity can vary dramatically, and the user could attempt to load conditions objects which have a large difference in their intervals of validity. There are two choices when dealing with such scenarios. Firstly the grid could expand, but this could result in the user having to scroll an unreasonable distance to reach the wanted data. Alternatively, the grid could be a constant size, with conditions objects being resized respectively against each other. After some user testing, it was found to be easier to find data on a fixed size grid, as the QGraphics zoom ability is strong and allowed the user to find conditions objects that are ordinarily dwarfed by a much larger conditions object. As a result, a system was implemented where the largest conditions object would span the entirety of the grid, with each smaller conditions objects resized relative to the resize factor of the largest. This is found through a simple resize factor show in Equation 1:

$$Resize = \frac{Width\ of\ largest\ object}{Width\ of\ grid}$$

Equation 1: Resize formula

The width of the grid is by default set to 5000 pixels, but this can be changed by the user to suit their monitor resolution. The width of the largest object can be found by applying the “width” member function to the largest conditions object. It is also important to consider not only the width of the conditions object itself, but also its starting coordinate, as this will dictate how much “span” the object will require across the grid.

3.6 Tagging System

The tagging system displays all the tags of the currently loaded Conditions Object classes. This is achieved through retrieving each Condition Object’s tag set when the item is loaded into Object Control and sending it to the Tagging System. The tagging system adds this to a QStringList, a convenience class based on QList with a focus on string-only storage, and loads it onto a widget.

Qt operatives a model/view architecture to present data to the user, consequently a list view was chosen o present this QStringList to the user. When the user clicks upon any of the strings within this list, the previously mentioned tag sorting system in Object Control comes into effect, with the application sorting based upon the selected string.

4 Progress and current state

With the current graphical elements described above in place, the software has made considerable progress from pure concept. It’s ability to load and use dummy Condition Objects illustrates that the various components are communicating correctly, and the underlying functionality of the software is working. Currently, the only critical missing functions are of the database interaction, which is well documented by members of the ILC group.

A significant challenge was encountered when an attempt to integrate database abilities into the application was made. The CondDB in use in the ILC is constructed using a software build process named CMake. The role of CMake is to seek out dependencies, and organise them, finally producing a build file, such as a “Makefile” on UNIX-like operating systems. CMake achieves this through a selection of files known as “CMakeLists”, which give the developer a the ability to provide a series of arguments and commands to the CMake progress. This gives developers a powerful method of controlling their dependencies and builds. Qt applications on the other hand, have a separate build tool, named QMake, which cannot use CMakeLists and operates in a different manner. QMake is simpler to use, and performs the functions of a meta-object compiler which links the various sources of code which have Signal and Slot elements together. This resulted in a problem, as the application would require both CondDB components, to read, write and manage data on the database, and Qt components, to enable to graphical elements.

It is currently thought to not be possible to compile the Qt and CondDB components separately, instead attempts were made to integrate the QMake build process into CMake. This decision was reached as the rest of the CondDB software bundle was already built using CMake, and it would be much preferred to keep a single build process within the software bundle. Eventually, a method was devised in which the QMake process could be wrapped into CMake allowing integration of the two build processes.

5 Outlook and Evaluation

The remaining primary portion of the code awaiting implementation is the Connections Manager class. This will use standard methods, outlined in LCCD documen-

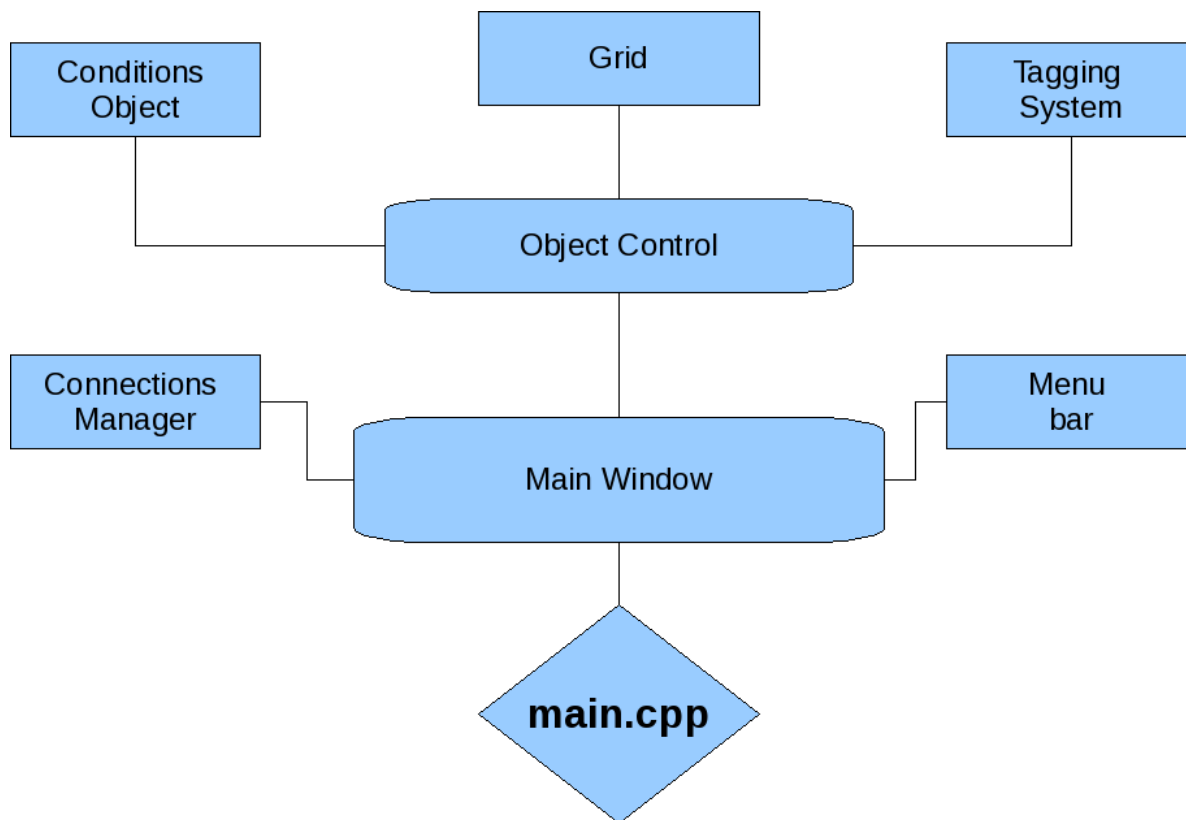
tation, to call and search the CondDB. Once this is complete, application will require minor alterations to ensure it loads objects from the database, and no longer forced dummy objects created by the user. Once this is complete, testing should be conducted to ensure the stability and user friendliness of the software. Testing could include attempting to connect to several databases to ensure the Connections Manager is functional universally, loading many greatly varying kinds of Conditions Objects and heavy tag sorting. For user testing, users could be asked whether the software is intuitive and easy to understand, and a user manual could be produced to help this.

A key improvement which could be made in future application development projects would be to consider the integration aspects of software earlier, leading to early

recognition of problems such as those encountered between QMake and CMake. An additional benefit of this project, however, was to encounter both minor and major hurdles in GUI development, as the lessons learnt ensure that future projects can proceed more elegantly from concept to conclusion.

In summary, the core Qt portions of the application are complete and require only integration with the database. The fundamental requirements of this project, to be an extensible, and portable CondDB GUI are being fulfilled with noticeable improvements in stability even within the current pre-release state of the software. As a consequence, the application will soon be available for initial testing amongst local users, before being made freely available online.

Appendix:



A1. A tree diagram of the application structure