Documentation:
Using AspectC++ for
Qt Application Development:

Ute Spinczyk and Olaf Spinczyk

Version 1.0, 28 April 2011

{us,os}@aspectc.org
http://www.aspectc.org

Contents

1 Introduction

Qt[1] is a state-of-the-art cross-platform application and UI framework. It provides C++ class libraries for user interfaces, graphics, networking, database access, and a lot more. Qt applications are typically written in C++ even though interfaces for other languages exist.

AspectC++[2] extends C++ by specific language constructs that enable *Aspect-Oriented Programming*.

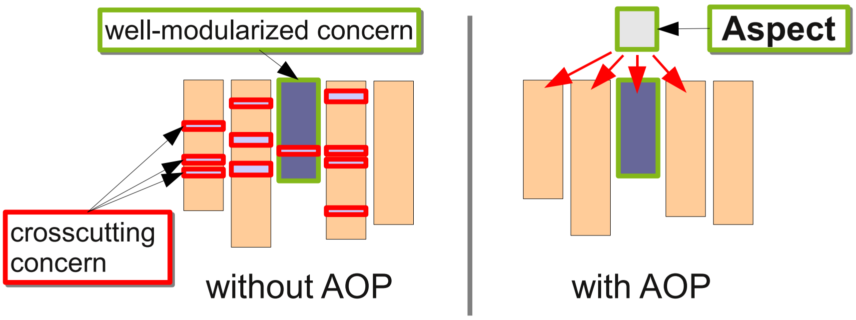

Figure 1: Implementation of crosscutting concerns without and with AOP

Figure 1 illustrates how a project can benefit from this paradigm: While most concerns of an implementation can be modularized using object-oriented programming alone, the so-called *crosscutting concerns* affect many parts. Typical examples are tracing and profiling code. This *tangling* of code, which implements different concerns, complicates the software maintenance, make re-use more difficult, and contradicts with the principle of *Separation of Concerns* in general. *Aspect Oriented Programming* on the other hand allows designers to isolate the crosscutting concerns. In AspectC++ crosscutting concerns can be implemented as *aspects*. So called *pointcuts* define where the aspects will take effect, allowing the programmer to inject or replace code in any module.

This manual explains how Qt and AspectC++ can be combined. The AspectC++ tools are developed as an open source project and are available under the GPL. They can be used for arbitrary development projects, regardless whether they are commercial or non-commercial.

Section 2 describes the necessary tools and installation steps. Following, Section 3 explains how moc-compiler, aspect weaver and C++ compiler act in concert. Two simple introductory examples are presented in Section 4. The remaining two sections discuss further details, namely sources of additional information on the AspectC++ languange (Section 5) and open issues, restrictions,

---

[1] visit `http://qt.nokia.com`

[2] visit `http://www.aspectc.org`

and problems that are relevant for Qt developers (Section 6).

2 Installation

The AspectC++ compiler is implemented as a code transformation tool that converts AspectC++ code into semantically equivalent C++ code. This step is performed by the "aspect weaver" tool `ac++`. Using `ac++` directly is a bit complicated, because the integrated AspectC++ has to be configured and the configuration depends on the C++ compiler, which is being used. However, if `g++` is used to compile the application project a comfortable front end for `ac++` called `ag++` can be used instead. The front end transparently calls `g++` in order to determine the parser configuration. Furthermore, when an AspectC++ program is being translated, `ag++` internally calls `ac++` for the code transformation from AspectC++ to C++ and then also runs `g++` for the translation into object code. Both tools can be downloaded from the AspectC++ project web site:

`http://www.aspectc.org`

Besides the latest "releases" there are also "daily builds" of Windows and Linux binaries and the source code[3]. The compressed `tar` archives can be unpacked with "`tar xjf <filename>`". Besides the two binaries the archive contains some example code, a `Makefile` for building and running it, and the AspectC++ manuals. The most convenient way to use the tools is to move them into a directory that is in your command search path, e.g. `~/bin` on many Linux systems. Make sure that both binaries are stored in the *same* directory.

Qt is available as a ready-to-use package in all major Linux distributions. The package name and format differs. Make sure to have installed the development package, which includes the Qt C++ header files, and not only the runtime libaries. Alternatively, Qt can be downloaded and installed from the project's homepage.

The Makefiles for Qt applications are typically built with the help of `qmake`, a tool that comes along with Qt and that converts a platform-independent project file into a platform-specific Makefile. The behaviour of `qmake` can be adapted by so called configuration features. The feature file `acxx.prf` will tell `qmake` to replace the C++ compiler by `ag++` and to take into account that all object files depend on all aspect headers. `acxx.prf` can be downloaded from the *Documentation*

---

[3] If you are interested in integrating AspectC++ or the underlying Puma parser and code manipulation frame into commercial products, contact the AspectC++ developer team and ask for a commercial license.

page at the AspectC++ web site. There you will also find the example code that is shown in Section 4. To use the AspectC++ configuration feature you should store the file `acxx.prf` anywhere on your file system and let the environment variable `QMAKEFEATURES` name the directory. Refer to the documentation of `qmake` to learn more about configuration features and the places `qmake` looks for those files.

3 Aspect Weaving

As depicted in figure 2 the compilation process for a Qt project that is enriched by aspects consists of several steps:

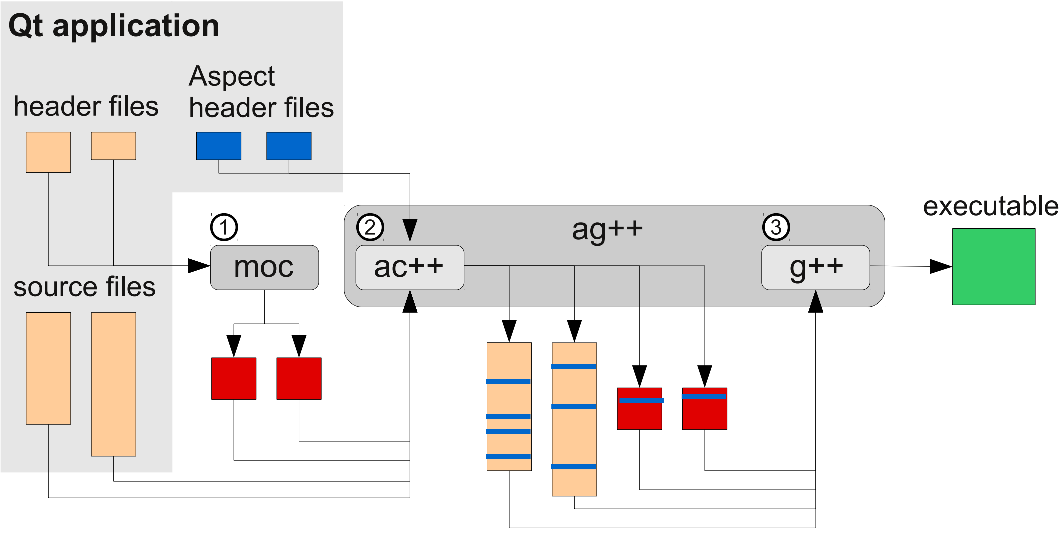

Figure 2: Compiling a Qt project with AspectC++

1. The Meta Object Compiler `moc` reads the header files of the Qt application and generates C++ source files that implement the meta-object methods for all Qt classes.
2. The AspectC++ Compiler `ac++` reads the aspect headers, preprocesses the C++ source code and determines where the given aspects must take effect. It then weaves the aspects into the code, generating new source files.
3. The C++ compiler compiles and links the woven files and creates an executable.

As already mentioned, the wrapper tool `ag++` provides a more convenient interface than `ac++` and is used to combine steps two and three.

4 Examples

4.1 “Hello Qt World”

Consider the well known “Hello world” program, implemented as a Qt application:

Listing 1 helloworld.cpp

```
#include <QApplication>
#include <QLabel>

int main (int argc, char **argv) {
    QApplication app (argc, argv);
    QLabel hello ("Hello, Qt world!", 0);
    hello.show ();
    return app.exec ();
}
```

The corresponding file `helloworld.pro` describes this simple Qt project in a platform independent way:

Listing 2 helloworld.pro without aspects

```
CONFIG   += qt
SOURCES += helloworld.cpp
```

With the information contained in the project file `qmake` can now generate a platform-specific `Makefile`, that specifies all the rules and definitions necessary to compile the project.

When executed the helloworld application will create a small window containing the label “Hello, Qt world!”.

Hello, Qt world!

AspectC++ provides the possibility to extend or modify a program without explicitly changing the source code. As an example the modifier aspect shown in Listing 3 will extend the helloworld application to create a second `QLabel` widget for messages from the aspect.

As described by the pointcut definition `p1()` the aspect takes effect, when the `exec()` method is called on the application object. It then creates, instantiates

Listing 3 modifier.ah

```
#ifndef __modifier_ah__
#define __modifier_ah__
#include <QLabel>
aspect modifier {
  pointcut p1 () = call("% QApplication::exec()");
  advice p1() : around()
  {
    QLabel label;
    label.setText("aspect active");
    label.show();
    tjp->proceed();
  }
};
#endif
```

and shows another `QLabel` object. Finally it calls `tjp->proceed()` to execute the original code of the pointcut, i.e. `app.exec()`.

To compile the program, the project file must be edited: It should select the `acxx` feature and name all aspect header files that belong to the project:

Listing 4 helloworld.pro with aspects

```
CONFIG  += qt acxx
SOURCES += helloworld.cpp
ASPECT_HEADERS += modifier.ah
```

If the `qmake` feature file `acxx.prf` is properly installed (refer to section 2) a call to `qmake` will then create a `Makefile` where the C++ compiler is replaced by the ag++ compiler and where the object files also depend on the aspect header files:

Listing 5 Makefile created by qmake

```
####### Compiler, tools and options
CC            = gcc
CXX           = ag++ -a modifier.ah --Xcompiler
...
helloworld.o: modifier.ah
...
```

When the helloworld application is now rebuilt and executed, two little windows will appear: The already known "Hello, Qt world!" label and a second "aspect active" label:

Hello, Qt world!

aspect active

In a similar way aspects can be used to trace the execution of Qt applications, to monitor and possibly change the values of some function’s arguments, to introduce additional code and much more. There is no need to change the original code. Many aspects can even be formulated without knowing any implementation details. The following subsection will present a generic aspect that can be applied on virtually any Qt application.

4.2 “Connection Graph”

One of the most outstanding features of Qt is its communication mechanism via *slots* and *signals*: Qt objects can notify other objects about an event by emitting signals. Sender and receiver need not know each other, because they will be coupled dynamically calling `QObject::connect()`. This method binds a particular event (the *signal*), for example `clicked()`, to a method (the *slot*) of the receiver object. It is allowed to connect many signals to the same slot or to connect many slots to the same signal, creating a possibly large communication network.

Listing 6 ConnectionViewer.ah

```
aspect ConnectionViewer {
  pointcut connect() =
     "% QObject::connect (QObject*, char*, QObject*, char*,
                          Qt::ConnectionType)";
  advice call (connect()) : before() {
    QObject** object1 = (QObject**) tjp->arg(0);
    char** signal = (char**) tjp->arg(1);
    QObject** object2 = (QObject**) tjp->arg(2);
    char** slot = (char**) tjp->arg(3);

    _programInfo.addConnection (*object1,*signal,*object2,*slot);
  }

  advice execution("% main(...)") : after() {
    _programInfo.print();
  }
}
```

Listing 6 shows the most interesting part of the ConnectionViewer aspect. The complete code can be downloaded from *Documentation* page of the AspectC++ web site.

The ConnectionViewer aspect can visualize the connections between slots and signals as a graph in the DOT language. The aspect consists of two advice. The first takes effect whenever `QObject::connect()` is executed. It analyzes the arguments of `connect()` to gather information about all slots and signals that are used by the application. With the help of `QObject::metaObject()` even the name of the involved classes can be determined. The second advice takes effect immediately before the application terminates. It uses the information gathered so far to print statements in the DOT language that describe the connection graph.

External programs can convert the output into various graphics formats. As an example figure 3 shows the connection graph of `qt-examples/dialogs/classwizard` as described by the ConnectionViewer aspect and converted to pdf by the `dot` program.[4]

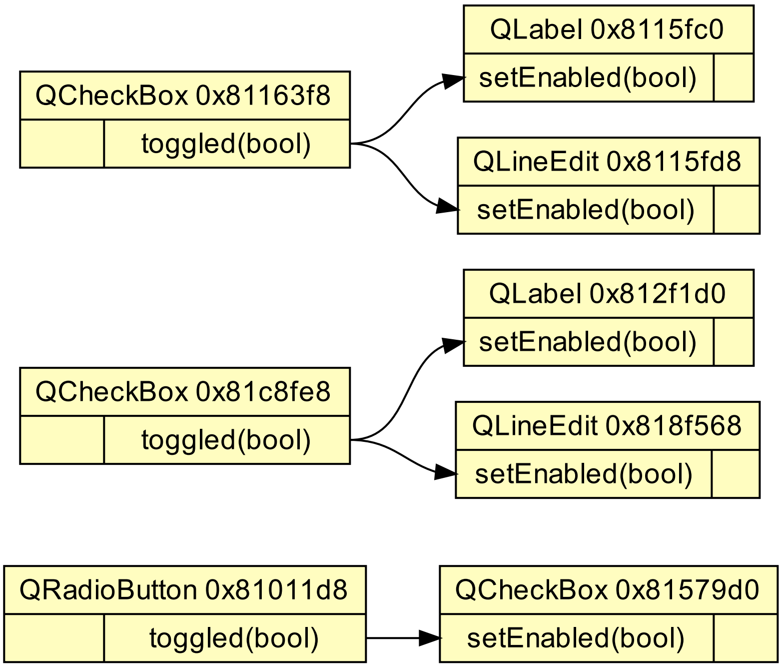

Figure 3: Connections created by classwizard

---

[4] dot - Graphviz version 2.20.2

5 Further Information

The AspectC++ language provides many more languages features than described in the manual. At the project's home page the following additional documentation is available and recommended for further reading:

AspectC++ Language Quick Reference Sheet: Gives a brief overview about the AspectC++ language elements.

AspectC++ Language Reference: A detailed AspectC++ language description.

AspectC++ Compiler Manual: Explains the usage and command line options of the AspectC++ compiler `ac++`.

Ag++ Manual: Explains the `ag++` wrapper command, which provides a very simple interface to `ac++` in environments with GNU `g++` back-end compiler.

6 Open Issues

Special effort has been spent to make sure that the AspectC++ tools `ac++` and `ag++` work well for Qt application code. For instance, we have created a test suite that consists of all Qt example programs. A test aspect weaves advice code for *all* potential call and execution joinpoints. Many bugs on the parser and aspect weaver level have been fixed in order to run this test without errors. Furthermore, we are testing the parser with all benchmarks, examples, and demos of the MeeGoTouch library. All together this is a test code base of significant size, which we are compiling regularly in our automated build and test system Akut. The AspectC++ developer team will make sure that future versions of `ac++` will also pass these tests[5]. However, there are a number of open issues that will be described in the following paragraphs:

Test platform Linux: At the moment our tests are only run on a MeeGo Linux system with g++ 4.5.0. Other Linux systems and other g++ versions have been tested as well, but not regularly. We have not spent any effort on Windows, MacOS, or any other platforms, yet.

Covered language features: The regular tests with Qt applications do not cover all AspectC++ language features. However, there are regression tests for this purpose, which cover most of them. Nevertheless, some language features don't work as you might expect. For more details on open issues with

---

[5] visit http://akut.aspectc.org to inspect the latest test results

`ac++` aspect weaver refer to the AspectC++ Compiler Manual, which can be downloaded from the AspectC++ homepage.

Co-operation with `moc`: The "meta object compiler" `moc` is a Qt-specific code transformation tool. It allows Qt developers to declare *slots* and *signals*. With the provided acxx Qt feature the aspect weaver runs *after* `moc`. This has the advantage that `ac++` "sees" all code that was generated by `moc`. On the other hand `ac++` can't use the provided language extension. For instance, it is not possible that aspects extend classes by *slots* or *signals*.

Handling of macro-generated code: `ac++` is not yet able to transform code that is generated by a C/C++ macro. Therefore users might see the following (or similar) warning: "transformation within macro". In this case advice matches a joinpoint located in macro-generated code. However, `ac++` cannot perform the transformation. It is recommended to check the macro and the location of its expansion. It is better to avoid matches like this by adapting the pointcut expessions accordingly.

A special problem occurs here, because of the Q_OBJECT macro. This expands the declaration of several `moc`-generated member functions. All these functions have to be avoided in the pointcut expressions of AspectC++/Qt-applications. The example code that comes with this manual shows how this can be achieved.