PRQA
Programming Research

COMPONENT MANUAL

PRQA-FRAMEWORK 2.0.1

February, 2016

IMPORTANT NOTICE

CONTACTING PROGRAMMING RESEARCH LTD

For technical support, contact your nearest Programming Research Ltd authorized distributor or you can contact Programming Research's head office:

by telephone on	+44 (0) 1932 888 080
by fax on	+44 (0) 1932 888 081
or by webpage:	www.programmingresearch.com/services/contact-support/

Contents

List of Figures

List of Tables

1 Introduction

PRQA Framework is a Source Code Analysis Management framework for managing source code analysis projects and analyzers.

PRQA Framework supports analyzers for a variety of coding languages: currently C, C++, C# and Java. There are minor differences per language in PRQA Framework where the nature of the languages differs, but the general functionality is common across languages.

1.1 Interfaces

PRQA Framework provides four interfaces through which all features are presented to the user:

QA·GUI
A Graphical User Interface to PRQA Framework.

QA·CLI
A Command Line Interface to PRQA Framework.

QA·Visual Studio
A Microsoft Visual Studio Integration that provides an interface to PRQA Framework.

QA·Eclipse
An Eclipse Integration that provides an interface to PRQA Framework.

All of the above interfaces provide the same set of functionality, save for where it makes no sense to do so.

All interfaces are available from the PRQA Framework package and are available for all of the supported platforms [1].

1.2 Desktop and Server-side Working

The variety of interfaces provided by PRQA Framework means that two main usage patterns are possible:

- Use on the desktop by a developer
- Use as part of an automated build-test-distribute workflow

The recommendation is that both of these should be used.

[1]With the exception of QA·Visual Studio, which is only available from the Microsoft Windows PRQA Framework package.

Desktop usage allows rapid correction of problems by the developers, as the code is being written. It is good practice for developers to run an analysis before checking any code into the version control system, just as unit tests would normally be run. In most cases, the analysis can be run and results viewed directly from the developer's IDE. QAGUI can be used if a non-supported IDE is being used for development.

Server-side usage would normally be run as part of a Continuous Integration workflow, or an overnight build process. Complete automation is possible, as QACLI commands can be called from scripts, which in turn can be called from CI tools. There is also a Jenkins plug-in available. Ideally, the results of the server-side processing should be uploaded into QA·Verify for further analysis and distribution, but PRQA Framework also has reports that can be run directly from QACLI.

Server-side processing is important because it operates on code extracted from the version control system - the same code that will be used to build the application. In addition, the processing options are known and can be controlled. Together, these mean that the results are reliable and auditable.

These two usage patterns work well together. The Desktop usage helps to minimise the number of issues that reach the server-side processing. As the server-side processing is automated, there is little cost in turning on some deeper analysis options such as Dataflow. Having these results available from the server means that Desktop processing can skip these options, making it quicker to run.

2 Getting Started

PRQA Framework is a Source Code Analysis Management framework for managing source code analysis projects and analyzers.

This chapter is a guide to getting started with PRQA Framework and introduces basic concepts and usage.

2.1 Introduction

The more information the analyzer knows about the compilation environment of a file, the more accurate and consistent results will be.

As an example, take the following piece of C code:

```
// file.h
#define MACRO 1
unsigned int i;
int foo(void);

// file.c
int foo(void)
  {
    return i = MACRO + CLIMACRO;
  }
```

which is compiled with the following command:

```
gcc -o file.o -DCLIMACRO=2 -I. file.c
```

Looking at *file.c*, we can see that there are many identifiers that are not resolved within that file itself: foo, i, MACRO and CLIMACRO. This poses a question for any *would-be* designer of a static analyzer or compiler:

> What to do when an identifier is not resolved within the source file it was declared or used in.

All designers of static analyzers encounter this question. How that question is answered determines the robustness of analysis results. Programming Research's primary analysis components do not carry on, assuming that identifiers and language constructs correctly resolve to something safe and legal. Conversely, they resolve them, and learn what they resolve to.

For this reason, the analysis components need to know the information that your compiler knows about your source files when they are compiled. This includes compiler specifics, such as:

- the compiler's built-in extensions to the language,
- the location of compiler include directories or references,
- the sizes of the various types known to your compiler,

and also build environment information, such as:

- the search paths that are used when compiling your source files to locate include files or references,
- environment variables that are present when compiling your source files,
- macros that are passed to your compiler on the command line while compiling.

This amounts to a great deal of data, data that may change from project to project, sub-project to sub-project or, indeed, source file to source file. Fortunately, PRQA Framework is equipped with features that automate the extraction of this data.

As the information that accompanies a source file when being compiled is intrinsic to the resolution of all, not some, types and identifiers, PRQA Framework has not been designed as a tool that leads the user to simply point to a source file and expect coherent analysis results. The apparent complication of the setup process is the price to pay in order to have insightful and complete diagnostic data as an outcome from the analysis results.

Section Create a PRQA Framework Project will guide you through creating your first QA· Framework project. Section Extraction of Configuration Data describes the various data extraction features that are available so that you may choose one that suits your environment. Section Analyzing Your Project instructs on the various ways in which you can analyze your new project.

2.2 Create a PRQA Framework Project

The steps to creating your first PRQA Framework project are as follows:

- choose a name for your project, more in section Naming Your Project
- choose the compulsory PRQA Framework configuration files you wish to use for your project, more in section Choose Configuration Files
- choose the file extensions of your sources, more in section Set the File Extensions for Your Sources
- choose the optimum PRQA Framework root directory for sources, more in section Set the Root Directory of Your Sources

Once complete, a file named 'prqaproject.xml' and a directory named 'prqa', containing your chosen configuration files, are created in the project directory. This project does not, as yet, have any sources associated with it. It is termed a ’headless PRQA Framework project’ and can be used as a project template to be used for more projects or to distribute to other machines where PRQA Framework is installed; just copy the ’prqaproject.xml’ and the ’prqa’ directory to any directory that you wish to create a project for.

2.2.1 Naming Your Project

PRQA Framework is designed to integrate with code projects that already exist. For this reason, the name of a PRQA Framework project is the name of a directory. If you choose a name for a project for which there is no directory that matches, PRQA Framework will attempt to create a new directory of that name for your project.

It is intended that you choose the root directory of a cohesive buildable entity for your QA· Framework project. Generally, a code base consists of many smaller sub-projects / components / modules that are buildable entities in themselves. Not all sub-projects may require the exact same configuration and, just like organising your code base into smaller cohesive sub-projects helps to manage the code base, the same is true for your analysis projects. It is suggested that you organise your PRQA Framework projects as the root directories of your sub-projects, components or modules.

Naming a new project from QA·GUI:

- **Project** menu : **New Project** : *Project Name:*

Naming a new project from QA·CLI:

-
  ```
  qacli admin --qaf-project <path> --qaf-project-config
                 --cct <file> --acf <file> --rcf <file>
  ```

If your are using either QA·Visual Studio or QA·Eclipse, your project name will be automatically set to the location for your project or solution.

To create a project in QA·Visual Studio, access the follwing menu item:

- **PRQA Framework** menu : **Admin** : *Create/Sync Project*

The same can be done in QA·Eclipse by accessing the right-click menu on your project of choice and clicking on the following menu item:

- **PRQA Framework** sub-menu : **Convert to PRQA Framework Project**

2.2.2 Choose Configuration Files

There are certain compulsory configuration files that must be chosen when creating a QA· Framework project. These are:

- at least one Compiler Compatibility Template (CCT)
- one Analysis Configuration File (ACF)
- one Rule Configuration File (RCF)

PRQA Framework is shipped with support for numerous compilers. Choose one that is an exact match for the compiler with which you compile your source code. If you use both a C and C++ compiler to build your sources, choose a CCT for each one that match your compilers. If you do not see a match for your compiler(s) in the provided list, please contact support and ask for support for your compiler.

Leave the default choices for the ACF and RCF for now. The ACF specifies a toolchain of analysis components for each source language and the RCF specifies coding standard rules and the messages that enforce each rule.

If your are using QA·Visual Studio, default configuration files will be chosen when your project is created. You can add, change and/or edit these configurations after project creation if you wish, through the PRQA Framework Project Properties panel. This panel is launched automatically after project creation.

2.2.3 Set the File Extensions for Your Sources

PRQA Framework determines how to process a source file based on the language of the source file, e.g. C or C++. It does this using the file extension of the source file. For each project, a set of file extensions is mapped to a source language. By default, for the C language these are:

```
.c .C
```

and for the C++ language:

```
.cc .CC .cpp .CPP .cxx .CXX
```

For C# it is .cs, and for Java it is .java.

You can change these setting for any project from QA·Project Properties' *General* tab. QA· Project Properties can be launched from QA·GUI as follows:

- **Project** menu : **Open Project Properties**

from the command line as follows:

```
qapp <path-to-project-root-directory>
```

from QA·Visual Studio as follows:

- **PRQA Framework** menu : **Admin** : *Project Properties*

and from QA·Eclipse through the right-click menu on your chosen project:

- **PRQA Framework** sub-menu : **Open PRQA Framework Project Properties**

2.2.4 Set the Root Directory of Your Sources

As stated before, code bases, in general, consist of many projects, sub-projects, components and/or modules. Often, the build of each entity is dependent on information from other entities, such as include directories.

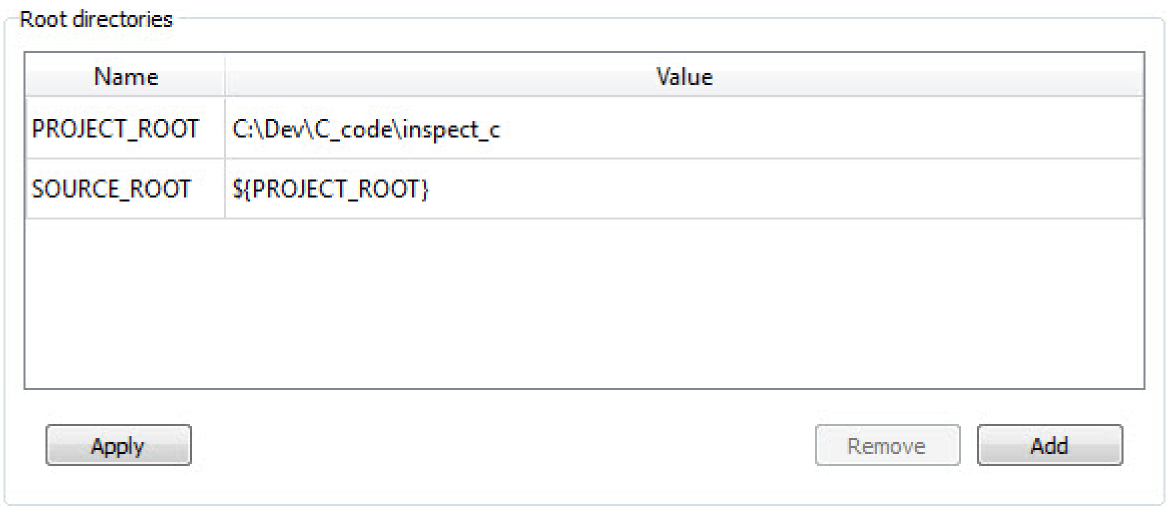
PRQA Framework projects are designed to be portable. For example, they are capable of being checked in to version control from one machine, checked out on another and analyzed. However, in order for this to work, dependencies, such as third-party and project include directories, must be capable of being located wherever the project has been checked-out or moved to.

Some software houses maintain build dependent directories absolutely. This necessitates that they are installed in the same absolute location on every machine that needs to build the project. Others maintain build dependent directories relative to the current project. Many software houses have a mixture of both absolute and relative build dependent directories.

PRQA Framework allows you to set multiple root directories for source files. Most often just a single root directory is enough, but you may need more than one root directory if your project uses third-party code, for example.

By default, the source root directory is set to be the same as the project root directory, specified when the project was created. To change the source root directory, or to add additional directories, navigate to:

Project Properties : **General** : *Root directories*


Root directories

Name	Value
PROJECT_ROOT	C:\Dev\C_code\inspect_c
SOURCE_ROOT	${PROJECT_ROOT}

Figure 2.1: Root directories

The PROJECT_ROOT setting holds the location of the PRQA Framework project. This cannot be changed. The SOURCE_ROOT setting is initially set to the PROJECT_ROOT value, but can be changed to a location more suited to your own file structure. To change the location, double-click on the cell in the Value column, and edit the text.

If you need to have more than one source root directory, click on the Add button. In the dialog that appears, set a Name to describe the location, and either type in the Directory location, or use the file selection dialog. The location can refer to other root directories by wrapping the name in ${ }, as shown in the screenshot. A path can then be appended, for example: ${PROJECT_ROOT}\src\calcs to refer to a location 2 levels below the project root.

As files are added to the project, PRQA Framework will determine the 'best match' of the file location to a source root. If there is no match, the absolute path to the file will be used.

The power of project roots can be seen if a project definition is copied between users. Often, user A will already set into the project the list of source files before copying the project definition across to user B. It may be that the two users have the same location for third-party code, but a different location for the source within the project itself. If so, then the user B can just adjust the source roots to suit the locations on the local machine. The project is now ready for use. There is no need for the user B to re-synch the project, the files added by the first user will be found in their local locations. Further, the changes made by user B are not set into the normal project definition file prqaproject.xml. Rather, they are stored in a user-specific file prqa\config\prqa-framework-app.xml. The significance is that the original prqaproject.xml is still valid for all users of the project, and can be checked in to source control without worrying about per-user differences. (Newly-added roots will be set into prqaproject.xml so they are available to all users.)

As new source roots are added and Apply is pressed, PRQA Framework will check whether any existing source files are better matched to the new root rather than an existing root. If so, the file will be put under the new root. That is, the path shown will be relative to the new root rather than the old. This technique can be useful in shortening some relative paths: by setting a new root at a lower level in the directory hierarchy, files switching to the new root will have a shorter relative path.

For example, sample project sample_cgicc_diff has three separate modules under its SOURCE_ROOT. File CgiUtils has relative path SOURCE_ROOT\src\cgicc\CgiUtils.cpp. If a new root is added as:

```
CGICC_ROOT = ${SOURCE_ROOT}\src\cgicc
```

Then once Project Properties is closed, the files are reallocated. All the files that were under SOURCE_ROOT\src\cgicc are now under root CGICC_ROOT.

Note that shortening the relative paths cannot be done by just changing an existing root location. In the example above, CgiUtils.cpp has the relative path SOURCE_ROOT\src\cgicc, which might convert to an absolute path c:\dev\sample_cgicc_diff\src\cgicc. If SOURCE_ROOT itself is changed to c:\dev\sample_cgicc_diff\src, the relative path is unchanged, giving an incorrect absolute path of c:\dev\ sample_cgicc_diff\src\src\cgicc. This retention of existing relative paths is necessary to find files in the situation where user B needs to adjust the path provided by user A.

2.3 Extraction of Configuration Data

2.3.1 For Plug-in Users

If you are using the QA·Visual Studio or the QA·Eclipse plugins, data extraction will be automatically done when you create your PRQA Framework project. This is accomplished in QA·Visual Studio in the following way:

- **PRQA Framework** menu : **Admin** : *Create/Sync Project*

and, in QA·Eclipse by accessing the right-click menu on your project of choice and clicking on the following menu item:

- **PRQA Framework** sub-menu : **Convert to PRQA Framework Project**

If you are using either of the QA·Visual Studio or QA·Eclipse interfaces, then skip to subsection Analyzing Your Project.

2.3.2 For C and C++ Users

QA·Framework has various features that help in the extraction of configuration data. The method that you will use will depend on the nature of your build environment. The methods are as follows:

Build Process Monitoring, which monitors your build process for the required information and populates a specified project with it,

Compiler Wrapping, which wraps your compile command during the build process so that PRQA Framework can extract necessary information form each command,

Manual Extraction, which allows the user to add a file and its compile dependencies to a specified PRQA Framework project,

Scripted Extraction, which is a method based on the manual method, except that it can be re-run automatically whenever necessary.

Each of these methods will be described in the following sections.

The most reliable way to extract data from your build environment is to use the Build ProcessMonitoring feature. The feature is designed to work with a vast number of compilers and build automation systems on the market. However, there are cases where the feature will not work with a specific compiler or build system, in which case one of the other methods can be used.

The various methods for data extraction available reflect the extremely varied build systems with which PRQA Framework is designed to work. Using any particular method does not restrict the way in which you can use a PRQA Framework project as any valid project can be used with any of the PRQA Framework user interfaces.

2.3.3 For C# and Java Users

C# users who do not use the Visual Studio plug-in, and Java users who do not use the Eclipse plug-in, should use the Manual Extraction method described below.

2.3.4 Build Process Monitoring

Build process monitoring works by recognising compiler processes that are spawned by a build process. It needs to be able to recognise the compiler to do this. The PRQA Framework Build Process Monitoring feature has support for a wide range of compilers and their variants and this is the preferred method to populate your project. It assumes that you are able to build your project on the command line. Build Process Monitoring has the added bonus that it may analyze your files as they are added to the project.

As an example, let's say that the "*make all*" command builds a project and that the Makefile is located in the '/home/prqa/workspace/' directory. To populate your project, change directory to the project directory and issue the following command:

```
qacli analyze -P /path/to/MyQAFProject -f -b "make all"
```

PRQA Framework will start your build process and attempt to extract the necessary information. Each file that is extracted will be analyzed and added to the PRQA Framework project.

If your build command is complex, it may contain characters that will be misinterpreted by the shell. This can be overcome by creating a small script that runs your build, as in the following example:

```
#!/bin/bash
cd /home/prqa/workspace/    # necessary if run from QA GUI
export SOME_MACRO=10
make clean                  # ensure to clean first
make target1 CXX=g++-4.7 CC=gcc-4.7
```

A Windows batch file similar to the above script is as follows:

```
c:
cd C:\prqa\workspace
set SOME_MACRO=10
nmake clean
nmake target1 CXX=cl.exe CC=cl.exe
```

Save the script, giving it a name, make the script executable and issue the following command[2]:

```
qacli analyze -P /path/to/MyQAFProject -f -b MyScript.sh
```

Any executable shell script or batch file can be passed to the *-b* option. Naturally, the project must be cleaned first before building, otherwise nothing may get compiled and there will be nothing to monitor.

To use the Build Process Monitoring feature from QA·GUI, it is necessary to use a script, such as the one above. Operating in a graphical user shell is quite different from operating on a command line, in that a graphical user shell or windowing environment, such as the Microsoft Windows desktop environment, has no concept of a current working directory. For this reason, it is necessary to either specify the working directory explicitely or the script needs to change the directory you wish to build your project. To access the feature:

- **Project** menu : **Synchronize**

[2]POSIX paths are valid on Windows. In any case, QA·Framework understands both Windows and POSIX paths. However, paths that contain spaces must be surrounded by quotes.

Source code projects are rarely static. Files and dependencies are regularly added and removed. The Build Process Monitoring feature allows you to re-synchronize your project PRQA Framework project whenever you need to. Simply use the feature with an existing QA· Framework project to do this.

2.3.5 Compiler Wrapping

An alternative to process monitoring for data extraction is to use the compiler wrapping feature. This works for any build systems that use an environment variable to specify the compiler to be used. Examples are as follows:

```
CC='gcc-4.7'
CXX='cl.exe'
```

When invoking your build, do so with the required compiler variables substituted for the qacli command for adding a file and it's dependencies to a PRQA Framework project. The following command is an example for a GNU Make build the uses *CXX* to specify the C++ compiler and *CC* to specify the C compiler. *qacli* command:

```
make CXX='qacli admin -P /path/to/QAFproj -z g++-4.7 --add-file --' /
CC='qacli admin -P /path/to/QAFproj -z gcc-4.7 --add-file --'
```

For each file that is compiled during the build, what would normally be passed to the compiler will now get passed to *qacli*. This allows *qacli* to extract the compile dependent information, (the path to the file that is compiled, the include paths that are necessary to locate included files and definitions), from the compile command.

The *qacli 'add-file'* feature will return a non-zero value if it failes to add the file. This may not be desireable in all circumstances as a build will normally exit if a tool returns a non-zero value. To prevent the build from exiting, for make-based systems, a dash can be added in front of the command:

```
make CXX='-qacli admin -P /path/to/QAFproj -z g++-4.7 --add-file --' /
CC='-qacli admin -P /path/to/QAFproj -z gcc-4.7 --add-file --'
```

The *-z g++-4.7* tells PRQA Framework to compile each file after extracting it and its compile dependent information. This is necessary for build systems that build artefacts that are needed later in the build process. If your compile command contains spaces (e.g. g++ -Wall) or other characters that may confuse a shell, then place the compile command in a script or batch file and make it executable. For example, a file named compiler_cpp.sh could wrap the compile command as follows:

```
#!/bin/bash
g++ -Wall "$@"
```

and, considering that a similar file for the C compiler was created, the make command from the example above would now look like:

```
make CXX='qacli admin -P /path/to/QAFproj -z compiler_cpp.sh --add-file --' /
CC='qacli admin -P /path/to/QAFproj -z compiler_c.sh --add-file --'
```

The Compiler Wrapping method can be used in any build system that allows the substitution of the compile command.

As with Build Process Monitoring, ensure to clean the project before building.

2.3.6 Manual Extraction

The *–add-file* option to the *qacli admin* command is very versatile. For example, the following command will add the file *file.c* to a PRQA Framework project along with its compile dependent information, the include path and the definition[3]:

```
qacli admin -P C:\path\to\QAFproj --add-file -- /IC:\workspace\inc
/DMYMACRO=1 C:\workspace\src\file.c
```

The *–add-file* option will parse all data that follows it and extract any compile dependent information within that data.

Another way of using the *–add-file* feature is to use the operating systems tools to automatically add all files in a given directory structure. On Unix-like systems, this is quite simple:

```
find . -iname "*.c" | xargs -n qacli admin -P <project-location>
--add-file
```

will add all files with a *.c* extension in the directory tree rooted at the current directory. The equivalent on Windows operating systems is as follows:

```
for /f \%f in ('dir /s /b *.c') do qacli admin -P <project-location>
--add-file -- \%f
```

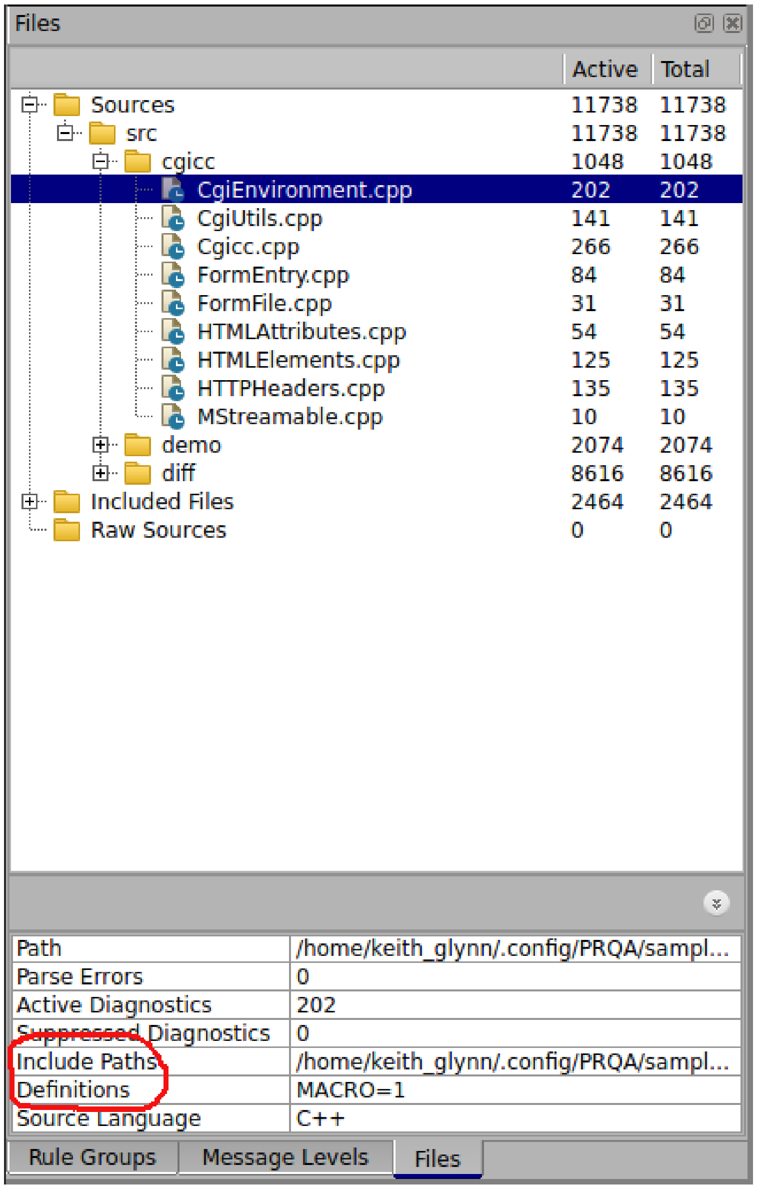
This approach will not add any of the compile dependent data for those files. This can be done manually by opening the QA Project Properties user interface for the PRQA Framework project in question and selecting the analysis component you wish to add the information[4]. Include paths and definitions can be added using the following options, respectively:

- **Analysis** tab : select-analysis-component : -I
- **Analysis** tab : select-analysis-component : -D

[3] A space is optional between the /I and the subsequent path. The feature will work with forms of include and define specifiers and formats for most compilers.

[4] For the C language, select the the C Source Language toolchain and *qac* analysis component. Likewise for the C++ language, choosing the *qacpp* analysis component

This will add include paths and definitions for all files for a specific source language. Adding include paths and definitions for a specific source file can be done through QA·GUI by selecting the source file in the file/folder tree and editing the properties, as shown in figure Adding/Changing Compile Dependencies for a File:



	Active	Total
Sources	11738	11738
src	11738	11738
cgicc	1048	1048
CgiEnvironment.cpp	202	202
CgiUtils.cpp	141	141
Cgicc.cpp	266	266
FormEntry.cpp	84	84
FormFile.cpp	31	31
HTMLAttributes.cpp	54	54
HTMLElements.cpp	125	125
HTTPHeaders.cpp	135	135
MStreamable.cpp	10	10
demo	2074	2074
diff	8616	8616
Included Files	2464	2464
Raw Sources	0	0

Path	/home/keith_glynn/.config/PRQA/sampl...
Parse Errors	0
Active Diagnostics	202
Suppressed Diagnostics	0
Include Paths	/home/keith_glynn/.config/PRQA/sampl...
Definitions	MACRO=1
Source Language	C++

Figure 2.2: Adding/Changing Compile Dependencies for a File

2.3.7 Scripted Extraction

The information can be sometimes be extracted from a project file using a script and subsequently added to a PRQA Framework project using the *add-file* option to *qacli*. An example of how to do this with a Freescale Code Warrior project file[5] with a simple python script[6] can be found in the following directory:

- PRQA Framework Install Directory>/common/doc/samples/data_extraction/

2.4 Analyzing Your Project

All three graphical user interfaces, QA·Visual Studio, QA·Eclipse and QA·GUI, have an 'Analyze' menu or sub-menu. There are two types of analysis available, file-based analysis and Cross-Module Analysis (CMA). These two types of analysis are only related through the fact that file-based analysis results must be available for project-based analysis to work[7]. CMA Analysis is described in Cross-Module Analysis.

File-based analysis runs on each file independently. As such, it can be run on one file, a group of files or all files in the project. Each file is analyzed with all analysis components in the toolchain for the specific source language of the file[8].

QA·GUI

File-based analysis can be invoked on a specific file, files or the entire project using the following sub-menu:

- **Analyze** menu : **File-Based Analysis**

QA·Eclipse

File-based analysis can be invoked on a specific file, files or the entire project using the following sub-menu:

- **PRQA Framework** menu : **Analyze** sub-menu : **File-Based Analysis**

[5]The Freescale Code Warrior project format is not text-based, but XML representation of the file can be exported. This is what is used for the example.

[6]The example is written in Python, however, any scripting language can be used.

[7]CMA does not imply file-based analysis.

[8]PRQA Framework has the functionality of the toolchain containing analysis components for each source language. See section Analysis Configuration for more information on analysis configuration.

QA·Visual Studio

File-based analysis can be invoked on a specific file, files or the entire project using the following sub-menu:

- **PRQA Framework** menu : **Analyze** sub-menu : **File-Based Analysis**

To perform file-based analysis on one file or a subset of files, right-click on the selected file(s) and run File-based analysis.

QA·CLI

To clean a project of analysis results and run file-based analysis, issue the following command:

- `qacli analyze -cf -P <project-path>`

To do all of those actions for a subset of files within the project, use the *-F* option to provide a file containing a list of files:

- `qacli analyze -cf -P <project-path> -F <file-with-list>`

3 Overview

PRQA Framework is designed to present the user with an Analysis System that handles several programming languages, using selectable analysis components. In the case of C and C++, it supports mixed language projects that use both languages.

The default setup uses the underlying Primary Analysis components such as the QA·C and QA·C++ parsers, but its strength lies in the flexibility of configuring an analysis chain, using other Secondary Analysis components. The Rule Configuration system enables the user to easily customize user messages and rules to suit company standards.

3.1 GUI Features

An introduction to PRQA Framework is provided in the 'GUI Quick Start' document, supplied with the installation package. These sections describe the functionality that is achieved directly from the GUI:

- Open a Project
- Select files for editing
- View in-line diagnostic messages
- View all messages found in selected file(s)
- View rules violated in selected file(s)
- Open message help content
- Open rule help content

3.1.1 Selecting a Project

A newly created project appears at the top of the 'Recent Projects' list. When you hover over an entry in the list, a tool-tip appears showing the path to the project directory.

A recent project may be re-opened by double-clicking the name in the Recent Projects list, or by right-clicking the projects and selecting the 'Reopen this project' option.

The currently active project is indicated in 'bold'.

When a project is opened, or re-opened, select the 'Files' tab in the Files panel in order to select files. The message Levels, rule group and Analysis Results/Diagnostics panels are not populated until files are selected.

3.1.2 Selecting Files

The Files panel is used to select files for editing and presenting diagnostic messages. It shows the summary diagnostic results for each file in the project. There are three different formats for presenting results:

- Files View
- Message Levels view
- Rule Groups view

Each view also has a summary table below it, which may be hidden.

3.1.2.1 Files View

The 'Files' panel shows the source files and header files for the project, with a corresponding icon, which indicates whether the analysis results are up to date or not (indicated by a green icon with a tick, or a blue icon with a clock). If the file contains a parsing error, this is shown with a red warning.

The panel shows the total number of diagnostic messages found in each file, represented by two counts - 'Active' and 'Total'. In this tab, the 'Active' count differs from the 'Total' count for any of the following reasons:

- If there are any suppressed messages
- If a baseline has been specified (baseline suppression)

The 'Active' and 'Total' column counts show the totals of diagnostic messages found in each file.

The 'Files' panel is used for selecting files within PRQA Framework for editing, analyzing or results display. When a file is selected, by a single left-click, the source code is presented in the Editor. Diagnostic results are displayed in the Analysis Results/Diagnostics window.

A user can see results from more than one file by the common selection methods:

- CTRL-Left_Click: select or deselect individual file (or directory).
- SHIFT-Left_Click: select a range of files/directories.
- Left_Click: select just one file/directory (and deselect all others).

Selecting a directory will automatically select all files within that directory and subdirectories.

Deselecting a directory will deselect all files within that directory and subdirectories.

It is the group of selected files that dictates the content of the Message Levels and Rule Group tabs. In order to migrate quickly to 'parsing errors':

1. Select the files of interest - this may be the top-level directories if the files are unknown.
2. In the Message Levels tab, collapse the 'Warnings' category - this brings the Errors category into view.
3. Right-click on a particular error message and select 'Show Only'.
4. The affected files are listed in the Analysis results/Diagnostics panel.

A file that is open in the editor will have its diagnostics marked in the editor, with icons at the left of the code window. When an icon is selected, a separate text box presents all associated diagnostic messages.

The Editor may be used to make quick changes to the source code for immediate re-analysis. The internal toolbar contains the usual editor features of cut, copy, paste, save and search. Searches can be case-sensitive, and key combinations CTRL-F and CTRL-B can be used for forwards and backwards searching. The tabs present a context menu, on right-click, for file operations:

- Close
- Close all
- Close all but this
- Save
- Save all
- Open containing folder
- Copy file path

Use key combinations CTRL-Tab and CTRL-SHIFT-Tab to cycle forwards and backwards through the open files. CTRL-W closes the current tab.

There is a context menu associated with a file, selected by a right-click. The options presented are:

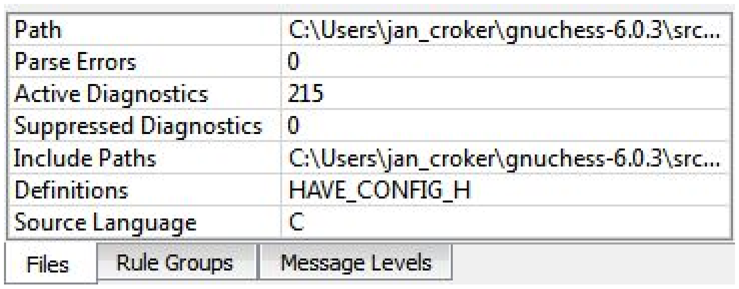
- Analyze selected files
- Generate Report
- Open file
- Remove file
- Clean Selected

- Add File to Project
- Add Multiple Files to Project

Some messages generated by whole-project tools such as RCMA and MTR do not have a single location. Instead, there is a top-level message, and sub-messages representing multiple locations. These top-level messages are put under a tree node called ’cma’. When the cma node is selected, the messages are shown in the Viewing Analysis Results pane.

The ’Raw Sources’ section shows any manually added raw source files, i.e. pre-processed source file (.i). These are designed to be used for temporary analysis only, and are not maintained as part of the project.

The Summary table in the Files panel shows the following summary diagnostics for a single selected file:



Path	C:\Users\jan_croker\gnuchess-6.0.3\src...
Parse Errors	0
Active Diagnostics	215
Suppressed Diagnostics	0
Include Paths	C:\Users\jan_croker\gnuchess-6.0.3\src...
Definitions	HAVE_CONFIG_H
Source Language	C

Figure 3.1: Files View

Note that it contains fields for the Include Paths and Definitions used by the file. These two entries are editable, opening up a text window to add or delete Include paths or Definitions.

3.1.2.2 Message Levels View

The Message Levels view is activated by selecting the ’Message Levels’ tab at the bottom of the Files panel. It presents the messages for the selected files in a tree view, sorted by group, listed in numerical order. The counts at the top of the panel are those for the selected files only. Messages are grouped under the Analysis Component that produces them, e.g. ’qac’, ’qacpp’, ’pal’, or a Secondary Analysis component, e.g. m3cm.

The ’Active’ and ’Total’ column counts show the totals of each diagnostic message found in the selected files. When a message is selected, the summary table at the bottom of the panel shows the Rule group mapping.

The context menu for any message in this view enables the user to filter on a particular diagnostic by selecting ’Show only’, or to ’Hide’ messages in the Analysis Results panel.

By this means, for example, a user could choose to show/hide all instances of a single message for the entire project, or for a given subset of files.

When a message is selected, the Summary table shows the rules enforced by the message.

3.1.2.3 Rule Groups View

The Rule Groups View is activated by selecting the 'Rule Groups' tab at the bottom of the Files panel. It opens with the name of the Rule Configuration group(s) to which the messages belong. For a C and C++ project, separate groups for each language can be seen.

When a message group is expanded, it shows a hierarchy of sub-groups, leading down to the set of messages that enforce a particular rule. Hence, this is where the rule mapping can be viewed.

In this view, the 'Active' and 'Total' column counts are slightly different. **Each level of the Rule Group tab sums all its child levels, but only counts *unique* diagnostic messages; this means that if a single message violates, for example, rules 2-4 and 2-2, the summation level above those two rules will count the offending message only *once*.** The rule group tab is intended to show users which diagnostics violate which rules, but ultimately counts unique diagnostics only; the number of rules violated may be greater than the number of diagnostics if a diagnostic violates more than one rule.

Violation counts are the numbers of times a rule has been violated and diagnostic counts are the number of instances of a message. They are not the same metric.

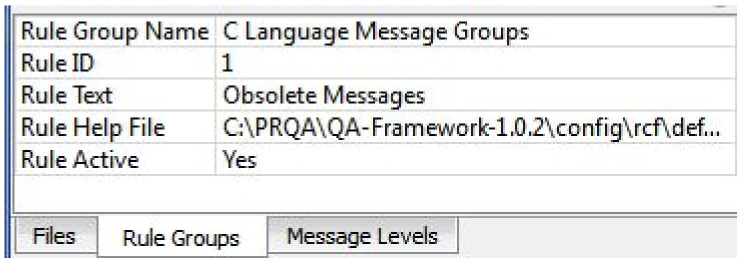
Right-clicking on a message or rule gives the user the option to select a single message or all messages for a given rule group or warning level. This will then be applied to the diagnostics shown in the Analysis Results/Diagnostics window. By this means, a user could choose, for example, to show all instances of a single message for the entire project, or all messages from a single rule group for a given subset of files.

The Summary table in the Files panel shows the following summary diagnostics for a single selected rule:

And for a single selected message:

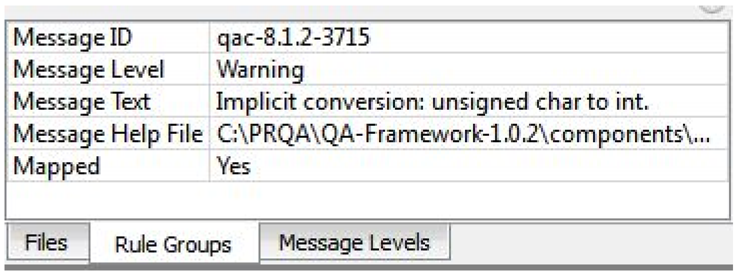
3.1.3 Viewing Analysis Results

The Analysis Results\Diagnostics window is populated when files(s) are selected in the Files panel. It shows all the diagnostic messages, and sub-msgs, found in each file. The data may be sorted by clicking any column header.



Rule Group Name	C Language Message Groups
Rule ID	1
Rule Text	Obsolete Messages
Rule Help File	C:\PRQA\QA-Framework-1.0.2\config\rcf\def...
Rule Active	Yes

Figure 3.2: Rule Groups View



Message ID	qac-8.1.2-3715
Message Level	Warning
Message Text	Implicit conversion: unsigned char to int.
Message Help File	C:\PRQA\QA-Framework-1.0.2\components\...
Mapped	Yes

Figure 3.3: Message Levels View

Each message is categorized with an icon indicating an:

- Error message (red)
- Warning message (yellow)
- User message (green)
- Information message (blue)

The presentation of the diagnostics can be controlled by using the control panel in the top left corner.

Each diagnostic is described by the following information:

- Category
- Message ID (define by component-version-ID)
- The Rule(s) that it enforces - if more than one, a context menu is shown
- The Message Text
- The filename
- The line number on which the error occurs
- The column number
- The message group to which it belongs (i.e. the Rule Configuration)

When the Message ID is selected, a QA·Framework Message help window is opened.

When the Rule is selected, a Rule help window is opened.

When the Message text is selected, the offending source line is highlighted in the central Editor window.

3.2 Menu-based Operations

Note: Not all menu items apply to all programming languages.

3.2.1 The Project Menu

- New Project
  This opens the 'New Project' window, where the user browses to the directory of the project source code and selects initial configuration files. See section Create a PRQA Framework Project for details.

  The QA·Framework software stores information about a project by adding two files (prqaproject.xml and prqaproject.xml.stamp) and a directory (prqa) to the project directory. These should not be altered by the user.

  The user is warned if the directory already contains a prqaproject.xml file.

- Open Project
  Enables the user to browse to an existing project directory. An error is shown if the directory does not contain a prqaproject.xml file.

- Close Project
  The currently active project is closed and the screen is cleared.

- Synchronize
  Opens this Synchronize Analysis with Build Process dialog.

  For C and C++, the PRQA Framework software is intended to infer the source code contents of a code project by observing a build process. The user specifies a script to be run, and should read the list of constraints to ensure that the script adheres to them. Pressing "Synchronize" will cause the PRQA Framework software to execute the script and monitor the subsequent build process, identifying the source code files of interest.

  Note that because the PRQA Framework software informs itself by watching the build process, if the build process does not build something (for example, because the code project has been recently built and nothing needs to be recompiled), the PRQA Framework software will gather no information. Accordingly, it is recommended to clean the build first, or to have the scripted build processes conduct such a cleaning operation.

Upon successful completion of this 'synchronization' process, the 'Files' panel is populated with the source and header files found, along with any definitions needed.

- Open Project Properties
  This opens the Project Properties window, where the following configuration functions may be performed:
  - Set the Project and Source Root Directories
  - Set Baseline Diagnostic Suppression
  - Add file extension mappings for the programming language
  - Perform Analysis Configuration
  - Perform Rule Configuration
  - Select Compiler Compatibility templates
  - Perform Synchronization configuration
  - Perform Version Control configuration
- CMA Project Editor
  This opens the CMA Project Editor window where configuration of CMA may be performed.
- Upload Results to a Structure101 project
  The level for upload may be specified as:
  - Upload project
  - Upload selected files
  - Upload files to be specified
- Exit Application

3.2.2 The Admin Menu

- License Server
  Specify the License Server host address and port. If the license server is not available, or if a particular component is not available in the license file, the appropriate component icon appears 'grayed' out on the Welcome Page.
- Languages
  Set the preferred language of this installation of PRQA Framework.

- Set Logging Level
  Controls whether log files are written, and how much detail is written to them. The recommended logging level is 'error'. Log files are written to the <local> \PRQA\PRQA-Framework-xxx\app\logs directory, where <local> is your local application data directory, and xxx is the PRQA Framework version number.

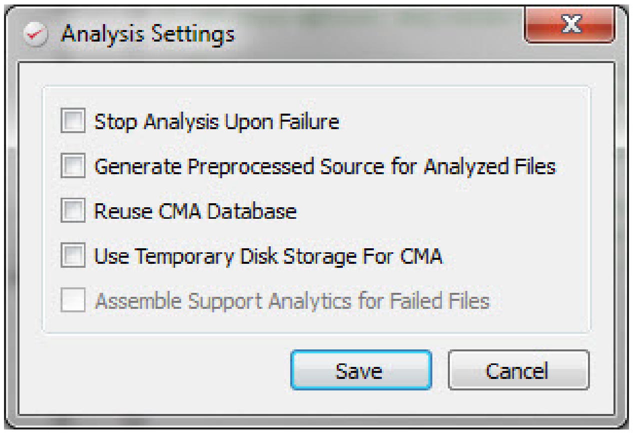
3.2.3 The Analyze Menu

For C and C++, the source files can be analyzed individually, or in a group. There are two kinds of analysis available; "file based" and "CMA". "File based" analysis operates at the translation unit level and checks for issues such as correct language use, dataflow and layout. CMA analysis operates across translation unit boundaries and checks for issues like duplicate definitions, incompatible declarations and unused variables. CMA Analysis is described in Cross-Module Analysis.

For other languages, analysis operates on all the files in the project.

- File-Based Analysis
  - File-based Analysis of project
  - File-based Analysis of selected files
  - File-based Analysis of files to be specified
- Run Current Project CMA Analysis
  Run CMA analysis on the currently-active project.
- MTR Analysis
  Run multi-threading analysis on the currently-active project
- Run CMA Analysis
  CMA Analysis is performed on CMA projects, which are created using the CMA Project Editor, found under the Project menu.
- Clean Analysis Results
  - Clean project
  - Clean selected files
  - Clean files to be specified
- Raw Source Analysis
  This feature is primarily for analyzing pre-processed (.i) files:
  - Analyze Raw Source File - opens a browser to select a file
  - Raw Source Analysis of file - to analyze a Raw Source file

- Analysis Settings
  This feature is intended to provide analysis information to aid debugging.



Stop Analysis Upon Failure [ ]
Generate Preprocessed Source for Analyzed Files [ ]
Reuse CMA Database [ ]
Use Temporary Disk Storage For CMA [ ]
Assemble Support Analytics for Failed Files [ ]

Figure 3.4: Analysis Settings View

  – 'Generate Preprocessed Source for Analyzed Files' - produces '.i' files for each file
  – 'Assemble Support Analytics for Failed Files' - produces a '.zip' file containing the .met, .i and .via files for the file. Note: this is only produced for files that have a parse failure.
  – Reuse CMA Database is described in section CMA Options.
  – Use Temporary Disk Storage for CMA is described in section CMA Options.

3.2.4 The Report Menu

- Generate report for project
- Generate report for selected files
- Generate report for files to be specified
  The selection of available reports is:
  – Suppression Report
  – Code Review Report
  – Metric Data Report
  – Rule Compliance Report

Please see the Reports section for details of the report content.

3.2.5 The QA·Verify Menu

This allows interaction with QA·Verify. QA·Verify is a centralised application that manipulates, summarises and gives wide visibility to the results generated through PRQA Framework.

- QAV Server Connection
- Upload Project Definition

This allows projects to be shared, and also enables the downloading of suppressions, and snapshots to be used as baselines. Uploading a new definition creates a project on QA·Verify, which together with the PRQA Framework project forms a Unified project. The definitions for existing Unified projects can also be updated.

- Upload Results

  The level for upload may be specified as:

  - Upload project
  - Upload selected files
  - Upload files to be specified

- Download Project Definition

  This will create a new PRQA Framework project as part of a Unified project. It can also be used to update an existing Unified project.

- Download Baseline

  Downloads a snapshot and the corresponding source code, which together are used to generate baseline diagnostic suppressions.

- Download Suppressions

  Downloads suppressions that have been defined within QA·Verify. These suppressions are applied against diagnostics in the local source code.

3.2.6 The View Menu

The user has the following options to customize the layout of the PRQA Framework GUI:

- Analysis Results/Diagnostics

- Message Levels
- Rule Groups
- Files
- Recent Projects
- Toolbar
- Reset Layout
- Close All tabs
- Welcome Page

3.2.7 The Help Menu

The Help menu provides access to on-line documentation:

- PRQA Framework Help
- Rules (the default configuration)
- Individual Component Manuals (dependent on installation)
- The Quick Start guide for QAGUI
- About
  Version: x.y.z.nnnn . e.g. 1.0.2.1754

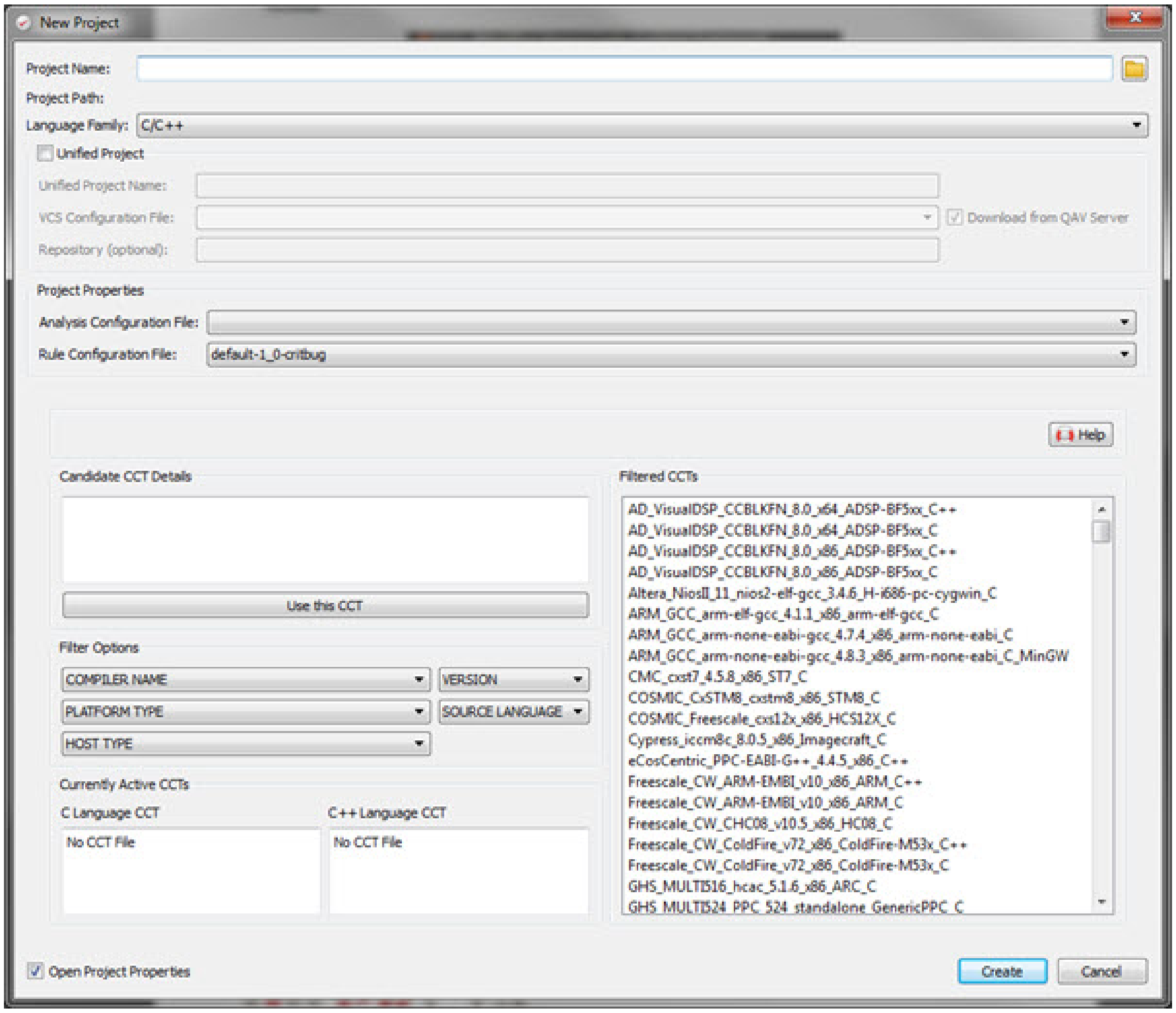
4 Projects

4.1 New Projects

A new PRQA Framework project can be created through the "New Project" button in the toolbar:

Alternatively, by selecting the Project Menu option or by typing its associated keyboard shortcut:

The user will be presented with the New Project dialog, which gathers information from the user necessary to create a new, empty PRQA Framework project:



Project Name:
Project Path:
Language Family: C/C++
☐ Unified Project
Unified Project Name:
VCS Configuration File:
☑ Download from QAV Server
Repository (optional):

Project Properties
Analysis Configuration File:
Rule Configuration File: default-1_0-critbug

Candidate CCT Details

Filter Options
COMPILER NAME
VERSION
PLATFORM TYPE
SOURCE LANGUAGE
HOST TYPE

Currently Active CCTs

C Language CCT	C++ Language CCT
No CCT File	No CCT File

Filtered CCTs

```
AD_VisualDSP_CCBLKFN_8.0_x64_ADSP-BF5xx_C++
AD_VisualDSP_CCBLKFN_8.0_x64_ADSP-BF5xx_C
AD_VisualDSP_CCBLKFN_8.0_x86_ADSP-BF5xx_C++
AD_VisualDSP_CCBLKFN_8.0_x86_ADSP-BF5xx_C
Altera_NiosII_11_nios2-elf-gcc_3.4.6_H-i686-pc-cygwin_C
ARM_GCC_arm-elf-gcc_4.1.1_x86_arm-elf-gcc_C
ARM_GCC_arm-none-eabi-gcc_4.7.4_x86_arm-none-eabi_C
ARM_GCC_arm-none-eabi-gcc_4.8.3_x86_arm-none-eabi_C_MinGW
CMC_cxst7_4.5.8_x86_ST7_C
COSMIC_CxSTM8_cxstm8_x86_STM8_C
COSMIC_Freescale_cxs12x_x86_HCS12X_C
Cypress_iccm8c_8.0.5_x86_Imagecraft_C
eCosCentric_PPC-EABI-G++_4.4.5_x86_C++
Freescale_CW_ARM-EMBI_v10_x86_ARM_C++
Freescale_CW_ARM-EMBI_v10_x86_ARM_C
Freescale_CW_CHC08_v10.5_x86_HC08_C
Freescale_CW_ColdFire_v72_x86_ColdFire-M53x_C++
Freescale_CW_ColdFire_v72_x86_ColdFire-M53x_C
GHS_MULTI516_hcac_5.1.6_x86_ARC_C
GHS_MULTI524_PPC_524_standalone_GenericPPC_C
```

☑ Open Project Properties

Figure 4.1: New Project Dialog

The following sections describe each User Interface panel and the information that must be provided.

4.1.1 Project Name

A PRQA Framework project name is the name of the directory the PRQA Framework project file is placed in.

For example, if a PRQA Framework project were created in the directory:

```
C:\Users\CodeProjects\Project_Alpha\
```

the PRQA Framework project would be named "Project_Alpha". This means that this element requires the user to nominate the directory in which to place the PRQA Framework project; this can be done easily by using the directory browser, accessible through the button at the right of this element.

Once a directory is nominated, an informational message will be displayed beneath this element; for example, indicating that the nominated directory does not exist and will be created, or that the directory name entered contains an unacceptable character or exceeds a size limit.

4.1.2 Project Path

This is a non-interactive element that shows the user the directory path of the nominated directory.

4.1.3 Language Family

The user must select a programming language family. The options available will be dependent on the components available to the PRQA Framework; for example, if components are installed for both C/C++ and C# , the user will be presented with these options.

4.1.4 Unified Projects

Project definitions can be centralized, by storing them in QA·Verify. These projects are called Unified projects. More detail on this subject can be found in Working with QA·Verify.

Although it is not the recommended approach, projects can be specified as Unified projects directly from the New Project dialog. See Creating from 'New Project' if you wish to do this, otherwise leave the Unified Project checkbox unticked.

4.1.5 Project Properties - Analysis Configuration File

The user can select the Analysis Configuration File (ACF) to be associated with the project. There is a default ACF available; other ACF files to address specific requirements can be written by Programming Research Ltd, or can be created by the user.

4.1.6 Project Properties - Rule Configuration File

The user can select the Rule Configuration File (RCF) to be associated with the project. There are default RCFs available, and if the user has compliance modules installed, additional RCF files will be available. Custom RCF files to address specific requirements can be written by Programming Research Ltd, or can be created by the user.

4.1.7 Compiler Compatibility Templates

The user must select at least one Compiler Compatibility Template (CCT) to be associated with the project. The interface presents all available CCT files, filtered according to filter selections the user can make.

Upon identifying at least one suitable CCT file, the user selects it with the “Use This CCT” button. At least one CCT file must be selected during this process.

Note: The user can change the CCT files selected after a project has been created through the "Project Properties" dialog.

4.1.8 Open Project Properties (checkbox)

If this is checked when the project "Create" button is pressed, then after creating the new project, the system will launch the "Open Project Properties" dialog. From this dialog the user can make more choices about the configuration of their PRQA Framework project.

4.2 C and C++ Compile Dependencies

Source files, more often than not, contain include statements. Included files may themselves include other files. To analyze a particular file, PRQA Framework needs to know where to locate the files that have been specified by all of these include statements, otherwise the analyzers will fail to resolve symbols that are defined in those files. The locations to search for include files are known in PRQA Framework as include paths.

Files are often compiled with definitions that are specified on the command line that compiles the file. For example, the following compile command specifies that the preprocessor is to substitute all uses of the 'MACRO' token with the literal '1' in the source file to be compiled, as well as in all files that have been included in its include dependency chain.

```
gcc -o file.o -I/include/directory/path  -DMACRO=1 file.c
```

The files that you wish to analyze, and the include paths and definitions used to compile those files are normally contained within your build automation or IDE project files. It is necessary to extract this information to populate your project.

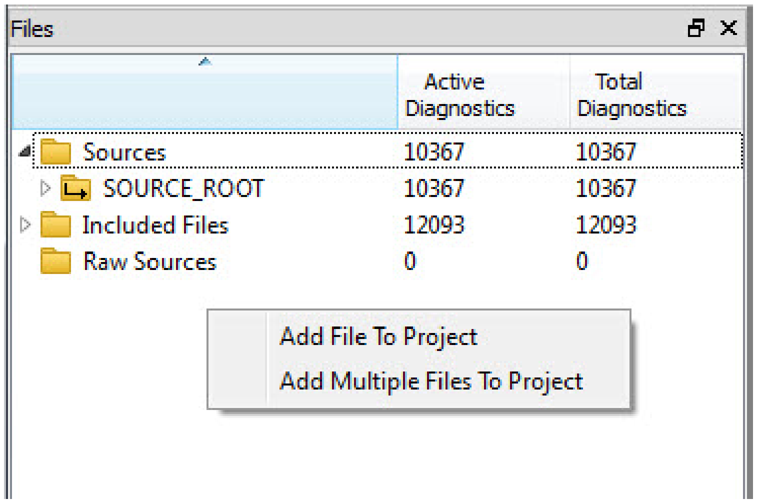
4.3 Populating Your Project

When first created, PRQA Framework projects only consist of a basic project directory structure and project configuration files. See subsection Create a PRQA Framework Project for details on how to create a project. The next step is to populate the project with the source files that you wish to analyze and the compile dependency information that is needed to analyze those files.

Populating an empty PRQA Framework project can be done in two ways through the QA·GUI. Adding files can be done for any language; observing a build is for C and C++ only.

4.3.1 Adding Files

Right-clicking in the "Files" panel (typically at left) will present the option to add a single file, or to add files recursively from a directory with a file filter.



	Active Diagnostics	Total Diagnostics
Sources	10367	10367
SOURCE_ROOT	10367	10367
Included Files	12093	12093
Raw Sources	0	0

Figure 4.2: Adding Files to a PRQA Framework Project

Files added must have their file name extensions recognised by the PRQA Framework project; file extensions can be set in the Project Properties "General" tab. After adding files, the user must set any project wide include paths or definitions in the Project Properties "Sync Settings" tab. Alternatively, they can be set on a file basis using the collapsible panel in the lower section of the "Files" panel.

4.3.2 Observing a Build

The second method, for C and C++ only, is to have the QA·GUI observe a build of the code project; the user triggers a build, and the QA·GUI monitors the process and identifies files being compiled, paths being used, and various other information. Note that triggering this cleans the PRQA Framework project of all existing files first, and only files that are used in the build will be added. If the code project's build is up-to-date and no compilation takes place, then the PRQA Framework project will be empty. For this reason, a user should *clean* their code project first.

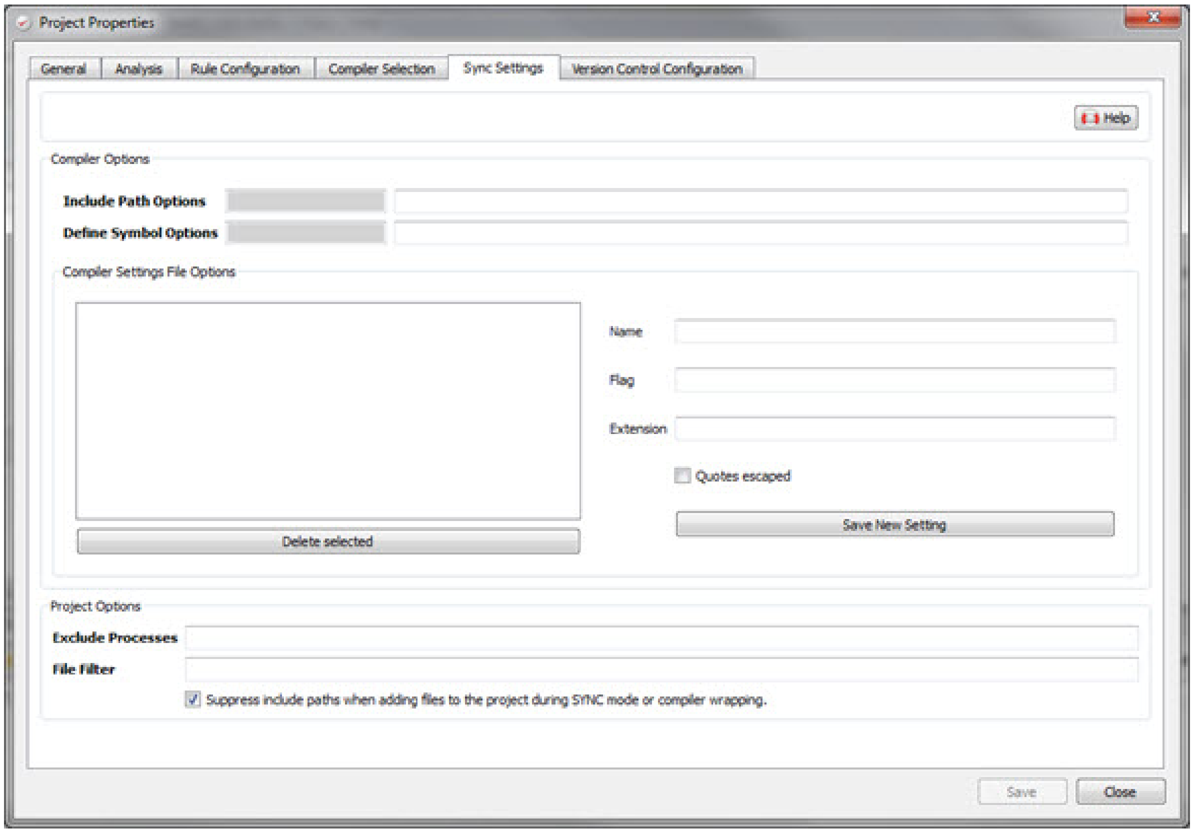


Figure 4.3: Preparing to Populate a PRQA Framework Project by Observing a Code Build

Various setting in preparation for this are set in the Project Properties "Sync Settings" tab. Many of these will have been set already when the user selected a CCT.

- Include Path Options: these should be the switches the compiler recognises as indicating an *include path*.

- Define Options: these should be the switches the compiler recognises as *defining* a value.
- Compiler Settings File Options: this allow the user, for compilers which store options in a file, to identify that file[9]. The `Quotes Escaped` option helps PRQA Framework decide how to parse quotes[10].
- Exclude processes: the user can nominate process names that should be ignored. For example, if the user's build script ran *rm filename.cpp* to delete a file, the user may wish to place *rm* here.
- File Filter: the user can nominate files that should not be added to the PRQA Framework project.

These settings will persist as part of the project. Once satisfied, the user may trigger the operation from the "Project -> Synchronize" menu option.

- Enter Build Command: A command-line build command must be entered in the *Enter Build Command* line. The complete path should be given; for convenience, a file browser button is at right. Note that this could be simply an IDE to run; that IDE would then be observed for code building activity.
- Optional Working Directory: An optional directory from which to run the build command.

Upon pressing *Synchronize* the user will be presented with information regarding the identification and addition of files to the PRQA Framework project.

4.4 Project Portability & Sharing

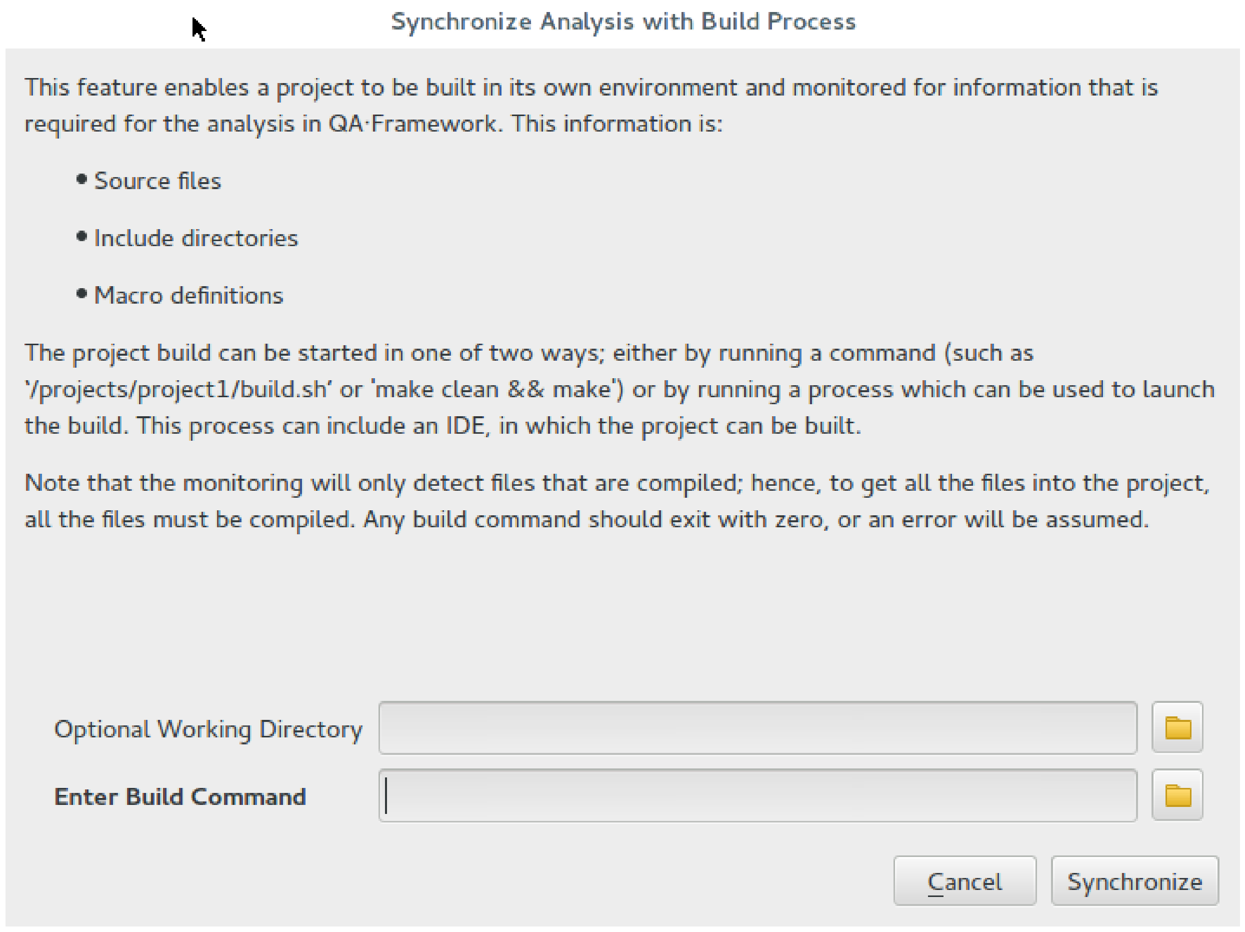
PRQA Framework provides project portability in two ways:

4.4.1 Generic Project Portability

In the case of generic project portability - without relative source files (a headless project), it is just the configuration which is being distributed. Create a project and edit any or all the settings using the project properties, then the project can be copied anywhere. The Sync Analysis will then have to occur on any machine the project is copied to. This case allows for easy configuration portability that can be used for various projects or passed

[9]For example, Microsoft's Visual Studio compiler stores defines, includes and other compiler options in `file.rsp`

[10]If it is ticked any unescaped quotes will be used to encase spaced options e.g `/DMAX(A,B)="A < B"` Will create a macro which checks if A is less than B. If it is unticked all quotes are taken as literal e.g. `/DMAX(A,B)="A < B"` will create a macro which makes a string `"A < B"`


Synchronize Analysis with Build Process

This feature enables a project to be built in its own environment and monitored for information that is required for the analysis in QA·Framework. This information is:

- Source files
- Include directories
- Macro definitions

The project build can be started in one of two ways; either by running a command (such as '/projects/project1/build.sh' or 'make clean && make') or by running a process which can be used to launch the build. This process can include an IDE, in which the project can be built.

Note that the monitoring will only detect files that are compiled; hence, to get all the files into the project, all the files must be compiled. Any build command should exit with zero, or an error will be assumed.

Figure 4.4: Preparing to Trigger a Code Build to Observe

between team members.

4.4.2 Project-Specific Portability

For a project-specific portability, the type of portability is for redistributing a complete project, including information extracted from build environments, such as location of files, include paths and CLI macro definitions. As a result the setting for the 'source code project root' needs to be set to a location that suits the particular organization's build dependency environment. See section Set the Root Directory of Your Sources for information on how to do this.

A PRQA Framework project consists of the following as a minimum:

- A directory that contains the prqaproject.xml file - the directory serves as the project identifier, among others.

- The prqaproject.xml file itself.
- If project-specific portability - the files (source files) must be in the same relative location (relative to the source code project roots) on the new machine.
- The PRQA directory containing the following directories:
  - Output - may be empty
  - Reports - may be empty
  - Config - must contain the following:
    - Exactly one acf.
    - Exactly one rcf.
    - One or more CCTs

Everything else is non-portable and is not needed to re-analyze the project on a different machine - therefore, does not need to be checked-in.

The CIP file is a non-portable part of the configuration and is completely dependent on where someone has installed their compiler and is automatically generated on each machine for the same project. Everything needed to analyze on a different machine is contained within the project root directory. This is the parent directory of the prqaproject.xml file.

Most Version Control Systems have an 'ignore' mechanism that is used to ignore certain files and file types, this same mechanism should be used to ignore all non-portable files and directories contained in the project directory structure.

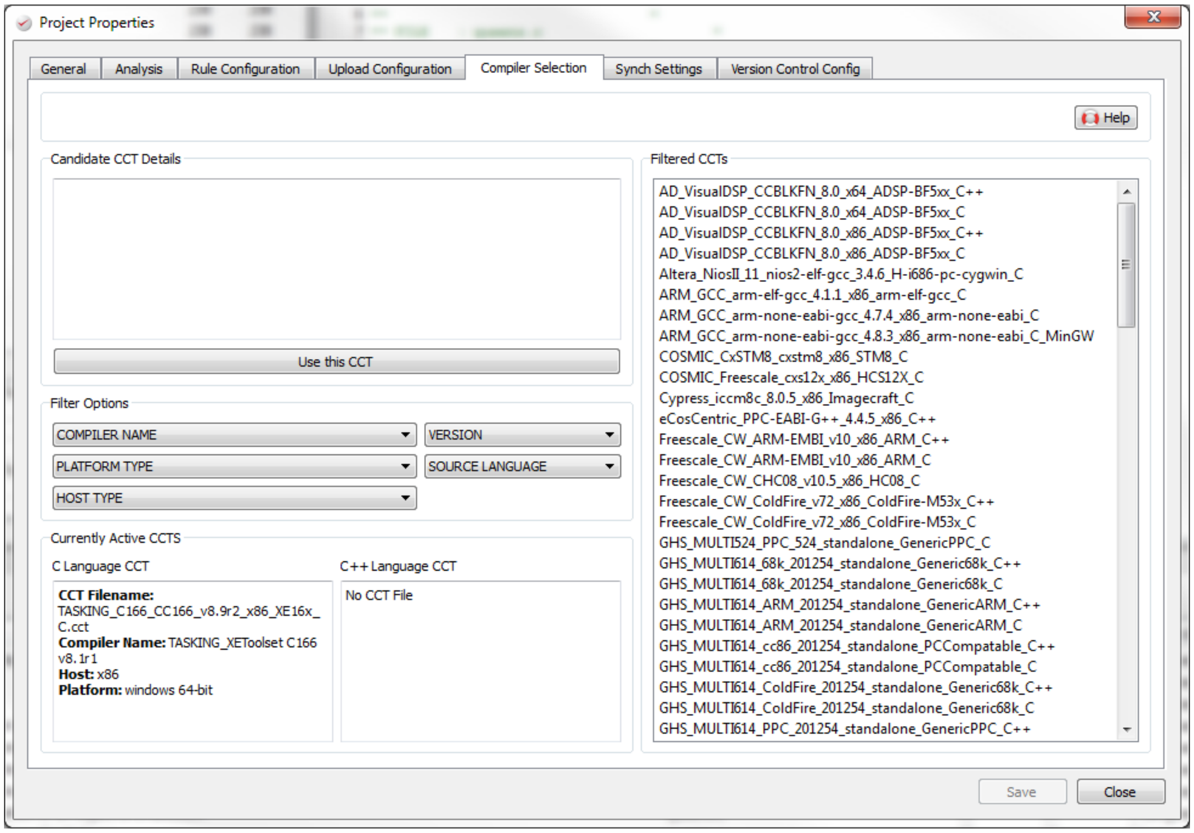
4.5 Compiler Selection

The compiler selection allows for searching and selection of CCTs which match the user's compiler.

For C/C++, active CCTs for both C and C++ are shown at the bottom, with a list of available CCTs shown to the right.

4.5.1 Selecting a CCT

To choose a new CCT the user selects one from the list of filtered CCTs for some languages. The details will then be displayed in the Candidate CCT Details panel. If the CCT matches the system requirements then select it using the 'Use this CCT' button.



Currently Active CCTS

C Language CCT

CCT Filename: TASKING_C166_CC166_v8.9r2_x86_XE16x_C.cct
Compiler Name: TASKING_XEToolset C166 v8.1r1
Host: x86
Platform: windows 64-bit

C++ Language CCT

No CCT File

Filtered CCTs

```
AD_VisualDSP_CCBLKFN_8.0_x64_ADSP-BF5xx_C++
AD_VisualDSP_CCBLKFN_8.0_x64_ADSP-BF5xx_C
AD_VisualDSP_CCBLKFN_8.0_x86_ADSP-BF5xx_C++
AD_VisualDSP_CCBLKFN_8.0_x86_ADSP-BF5xx_C
Altera_NiosII_11_nios2-elf-gcc_3.4.6_H-i686-pc-cygwin_C
ARM_GCC_arm-elf-gcc_4.1.1_x86_arm-elf-gcc_C
ARM_GCC_arm-none-eabi-gcc_4.7.4_x86_arm-none-eabi_C
ARM_GCC_arm-none-eabi-gcc_4.8.3_x86_arm-none-eabi_C_MinGW
COSMIC_CxSTM8_cxstm8_x86_STM8_C
COSMIC_Freescale_cxs12x_x86_HCS12X_C
Cypress_iccm8c_8.0.5_x86_Imagecraft_C
eCosCentric_PPC-EABI-G++_4.4.5_x86_C++
Freescale_CW_ARM-EMBI_v10_x86_ARM_C++
Freescale_CW_ARM-EMBI_v10_x86_ARM_C
Freescale_CW_CHC08_v10.5_x86_HC08_C
Freescale_CW_ColdFire_v72_x86_ColdFire-M53x_C++
Freescale_CW_ColdFire_v72_x86_ColdFire-M53x_C
GHS_MULTI524_PPC_524_standalone_GenericPPC_C
GHS_MULTI614_68k_201254_standalone_Generic68k_C++
GHS_MULTI614_68k_201254_standalone_Generic68k_C
GHS_MULTI614_ARM_201254_standalone_GenericARM_C++
GHS_MULTI614_ARM_201254_standalone_GenericARM_C
GHS_MULTI614_cc86_201254_standalone_PCCompatable_C++
GHS_MULTI614_cc86_201254_standalone_PCCompatable_C
GHS_MULTI614_ColdFire_201254_standalone_Generic68k_C++
GHS_MULTI614_ColdFire_201254_standalone_Generic68k_C
GHS_MULTI614_PPC_201254_standalone_GenericPPC_C++
```

Figure 4.5: Compiler Selection

The language will be noted automatically and the correct active CCT will be updated.

4.5.2 Filtering

There are many available CCTs for some languages and to narrow the search for the one required for the user's system there are a number of filter options. These options are:

- Compiler Name
- Compiler Version
- Platform type
- Source Language
- Host Type

For C/C++, if the Compiler Name and Compiler Version are known then by selecting the matching options this will reduce the number of available CCTs to a C and C++ version.

Platform and Host Type will enable the user to find the compilers which might be compatible with PRQA Framework on the user's system.

Finally, if development is to be completed in only one language then by selecting that language all non-relevant CCTs will be removed.

4.6 Baseline Diagnostics Suppression

Baseline diagnostics suppression is used to suppress old diagnostic warnings when the decision has been made to begin a new cycle of development.

The baseline consists of the set of diagnostics present when the baseline was generated. In any subsequent analysis, if a diagnostic can be identified as being the same as a diagnostic within the baseline, it is suppressed. Sophisticated diff-style techniques are used to match diagnostics, even if some of the surrounding code has changed.

Baseline suppressions are applied at view time. They do not affect the generation of diagnostics. This means you can see exactly what has been suppressed by selecting the 'Show baseline suppressed diagnostics' on the results panel.

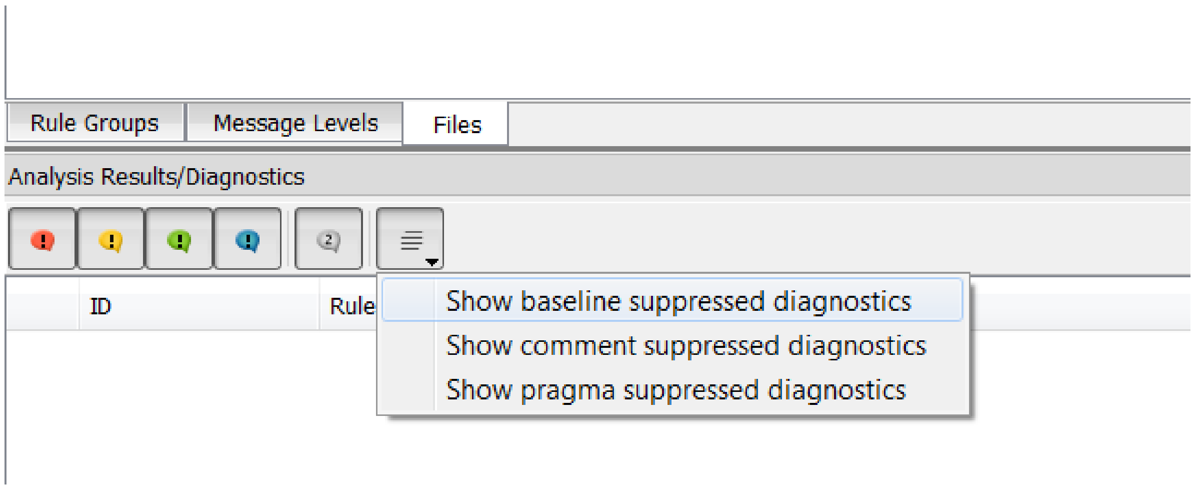

Rule Groups
Message Levels
Files
Analysis Results/Diagnostics
ID
Rule
Show baseline suppressed diagnostics
Show comment suppressed diagnostics
Show pragma suppressed diagnostics

Figure 4.6: Analysis Results/Diagnostics Panel

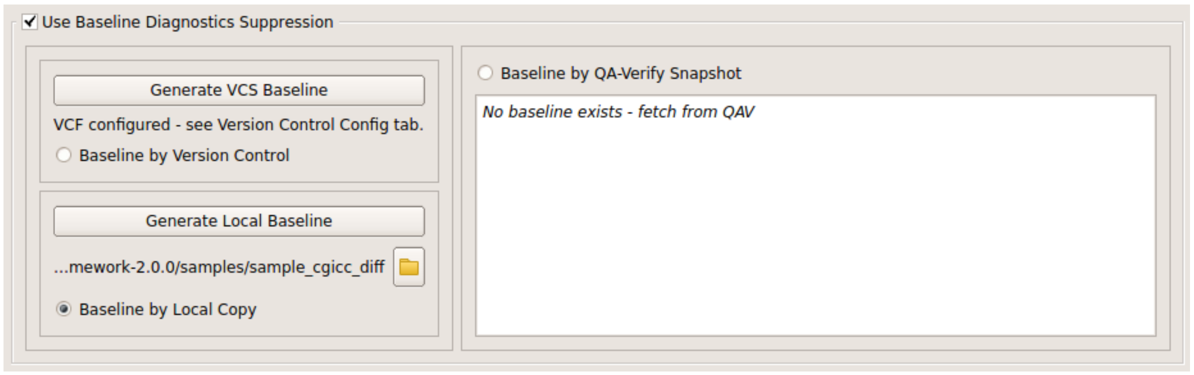
Baselining is particularly useful when dealing with legacy code, where code changes are often seen as undesirable. Apply a baseline to the legacy code. Once the baseline is in place, diagnostics resulting from code changes will be displayed, but diagnostics from the original code will not.

4.6.1 Generating a Baseline

There are three types of Baseline. In order of preference, these are:

- Baseline from a QA·Verify snapshot
- Baseline by Version Control
- Baseline by Local Copy

The options for Baseline Diagnostics Suppression are be found by selecting 'Project/Open Project Properties' from the menu and selecting the General panel, see figure General Panel, Baseline Diagnostics Suppression.



☑ Use Baseline Diagnostics Suppression
Generate VCS Baseline
VCF configured - see Version Control Config tab.
○ Baseline by Version Control
Generate Local Baseline
...mework-2.0.0/samples/sample_cgicc_diff
● Baseline by Local Copy
○ Baseline by QA-Verify Snapshot
No baseline exists - fetch from QAV

Figure 4.7: General Panel, Baseline Diagnostics Suppression

The panel is initially disabled; click the 'Use Baseline Diagnostics Suppression' check-box to enable it.

4.6.2 Baseline by QA·Verify snapshot

This option is available to projects that have been associated with a project on QA·Verify. These are termed Unified projects. At least one snapshot must have been uploaded to QA·Verify from this project.

Generating the baseline involves selecting a suitable snapshot, and downloading the diagnostics from the snapshot. This process is controlled through the QA·Verify : Download Baseline menu option.

A full description of baselining from QA·Verify is given in Downloading a Baseline.

4.6.3 Baseline by Local Copy

This allows the user to indicate that all current diagnostics (excluding CMA diagnostics) are to be suppressed. If there are no current diagnostics (for example, because the project has not yet been analyzed, or has been recently cleaned) no diagnostics will be suppressed.

Ensure that the checkbox “Use Baseline Diagnostics Suppression” is checked (without which this section of the panel remains disabled), and then click ’Generate Local Baseline’; this process will create the suppressions for the project and will automatically update the location of the suppressions information. This location will be visible below the “Generate Local Baseline” button and should require no user assistance. Finally, select the “Baseline by Local Copy” radio button.

Selecting the “Baseline by Local Copy” radio button *without* having generated a baseline will be disallowed, with an error message advising the user to create the baseline first.

Any changes must be saved. Upon closing the “Project Properties” dialog the project will need to be cleaned and re-analyzed in order to suppress the diagnostics. Suppressed diagnostics can be seen by toggling the “Show baseline suppressed diagnostics” button in the “Analysis Results/Diagnostics” panel in the main window, as shown in figure Analysis Results/Diagnostics Panel.

4.6.4 Baseline by Version Control

Before generating a Baseline by Version Control, the project must have been fully analyzed, and a version control script should have been set up in the "Version Control Configuration" tab.

Select ’Baseline by Version Control’. Press ’Generate Project Baseline’ and save the new options. The old warnings will be suppressed.

5 Analysis Configuration

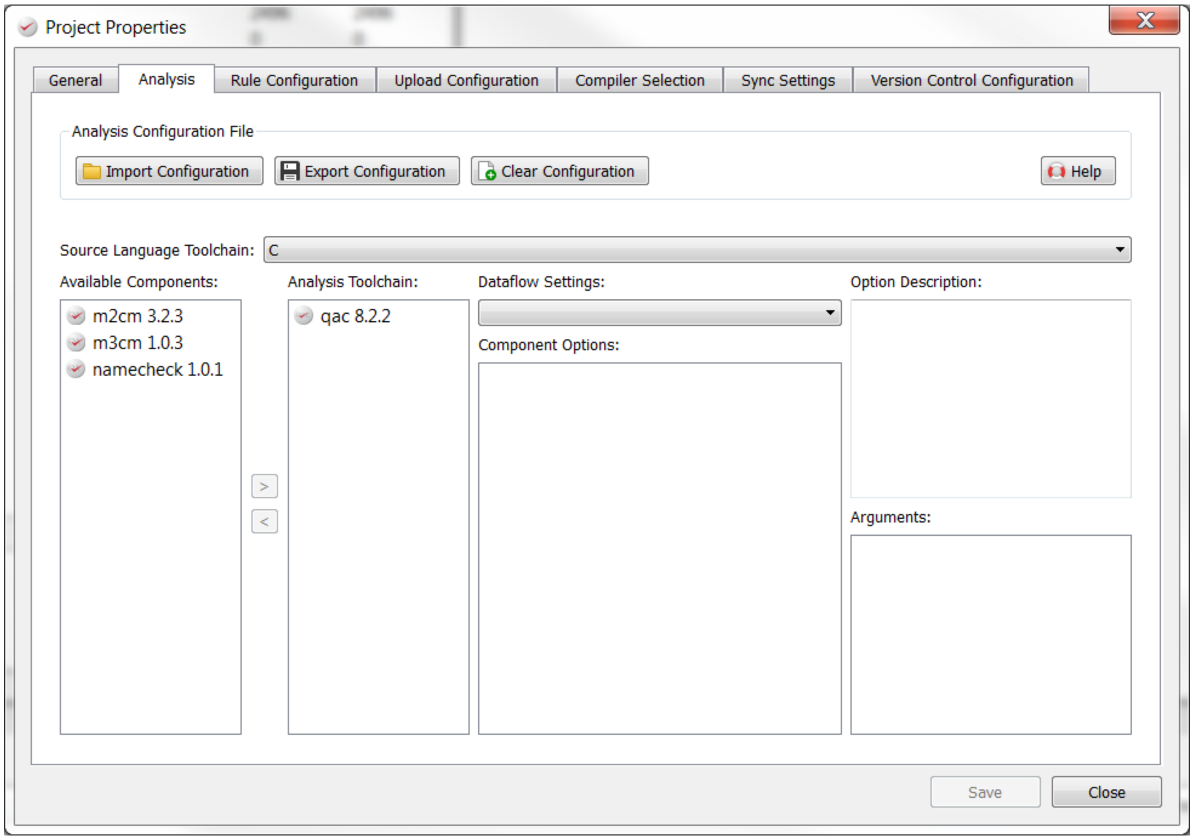
5.1 Introduction

The Analysis panel allows definition of the main parameters of the Analysis toolchain. Each setting established at this level can be exported to a dedicated file with the extension .acf (Analysis Configuration File).

A description of the options available for each individual component can be found in the manual for that component.

5.2 Analysis Configuration

The Analysis Configuration panel is shown below:



Source Language Toolchain: C

Available Components: m2cm 3.2.3, m3cm 1.0.3, namecheck 1.0.1

Analysis Toolchain: qac 8.2.2

Figure 5.1: Analysis Configuration

5.2.1 Importing and Exporting Configuration Settings

It is possible to import a new configuration set or export your current configuration via the Analysis Configuration File controls.

A new Rule Configuration file may be chosen by selecting the Import button.

Any changes made to the configuration can be saved by pressing the Save button. If you wish to share your configuration you can use the Export button. This will allow you to save the configuration; this configuration can then be imported into other projects.

5.2.2 Language Toolchain

For mixed-language projects, there is both a C and a C++ toolchain. The C toolchain will be run against C code, and the C++ toolchain against C++ code. Use the dropdown menu "Source Language Toolchain" to select a toolchain.

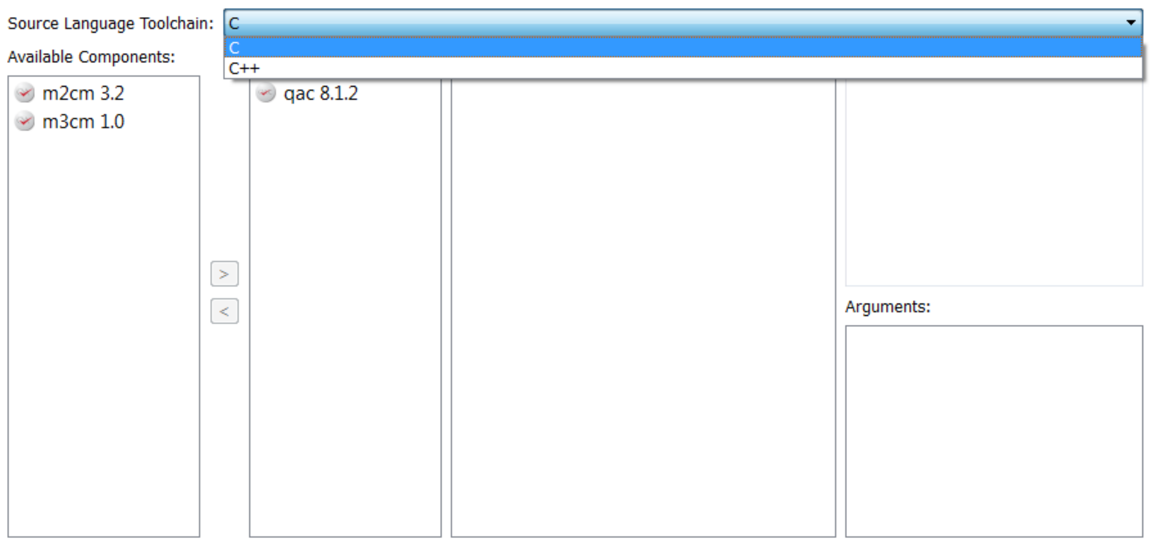


Figure 5.2: Source Language Toolchain

5.2.3 Selection of the Components

Once the language has been selected, it's possible to define which components will be part of the toolchain for that language.

By default the primary analyzers such as C and C++ parsers, QA·C and QA·C++, are already available into the analysis toolchain. Secondary analysis components are available in the left column "Available components". If a secondary component needs to be part of the current analysis, it can be added to it by using the ">" button; it can conversely be removed by using the "<" symbol.

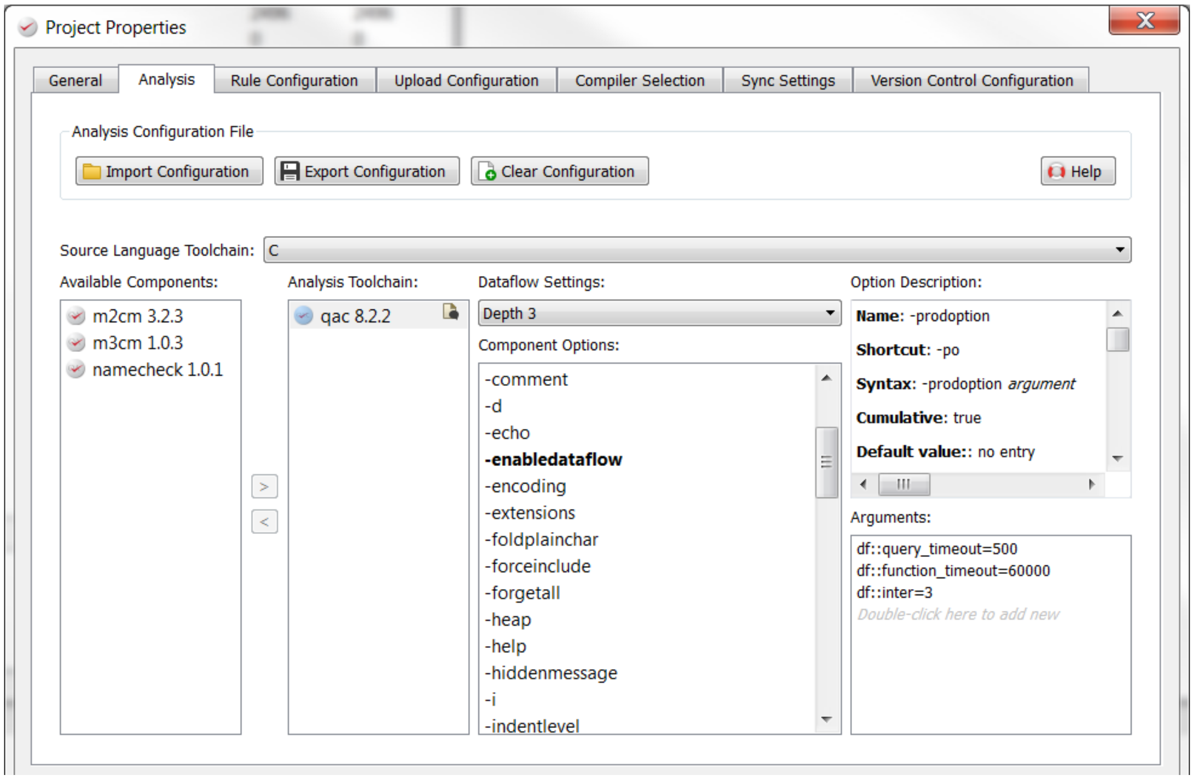
5.2.4 Defining the Component Options

Clicking on a component in the "Analysis Toolchain" list will dynamically populate the "Component Options" panel with the options available for that component. Clicking on one of them, will be possible to read a description of the option which reports:

- The **name** of the option
- A **shortcut** for the option
- The **syntax** for that option
- Its **default value**
- An extensive **description**

To add arguments to an option, just double click into "Double-click here to add new" and write the value to be added. If the option requires a file or directory, a dialog will open up to allow you to navigate to that file or directory.

Note: If the argument contains spaces, and it is not a path, then it should be entirely enclosed between quotes:



Source Language Toolchain: C

Dataflow Settings: Depth 3

Option Description:

Name: -prodoption

Shortcut: -po

Syntax: -prodoption *argument*

Cumulative: true

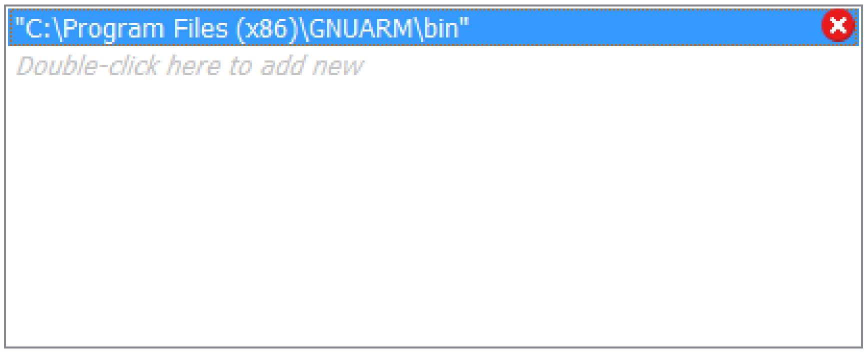
Default value:: no entry

Arguments:

```
df::query_timeout=500
df::function_timeout=60000
df::inter=3
```

Figure 5.3: New Analysis Toolchain Arguments

To remove an argument from the list of current arguments just click on the remove icon:


"C:\Program Files (x86)\GNUARM\bin"

Double-click here to add new

Figure 5.4: Remove Arguments from the List

5.3 User Messages

The user can create new messages to be added to the current group of messages for the analysis component.

To do that, right click on icon on the right of the component:

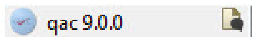


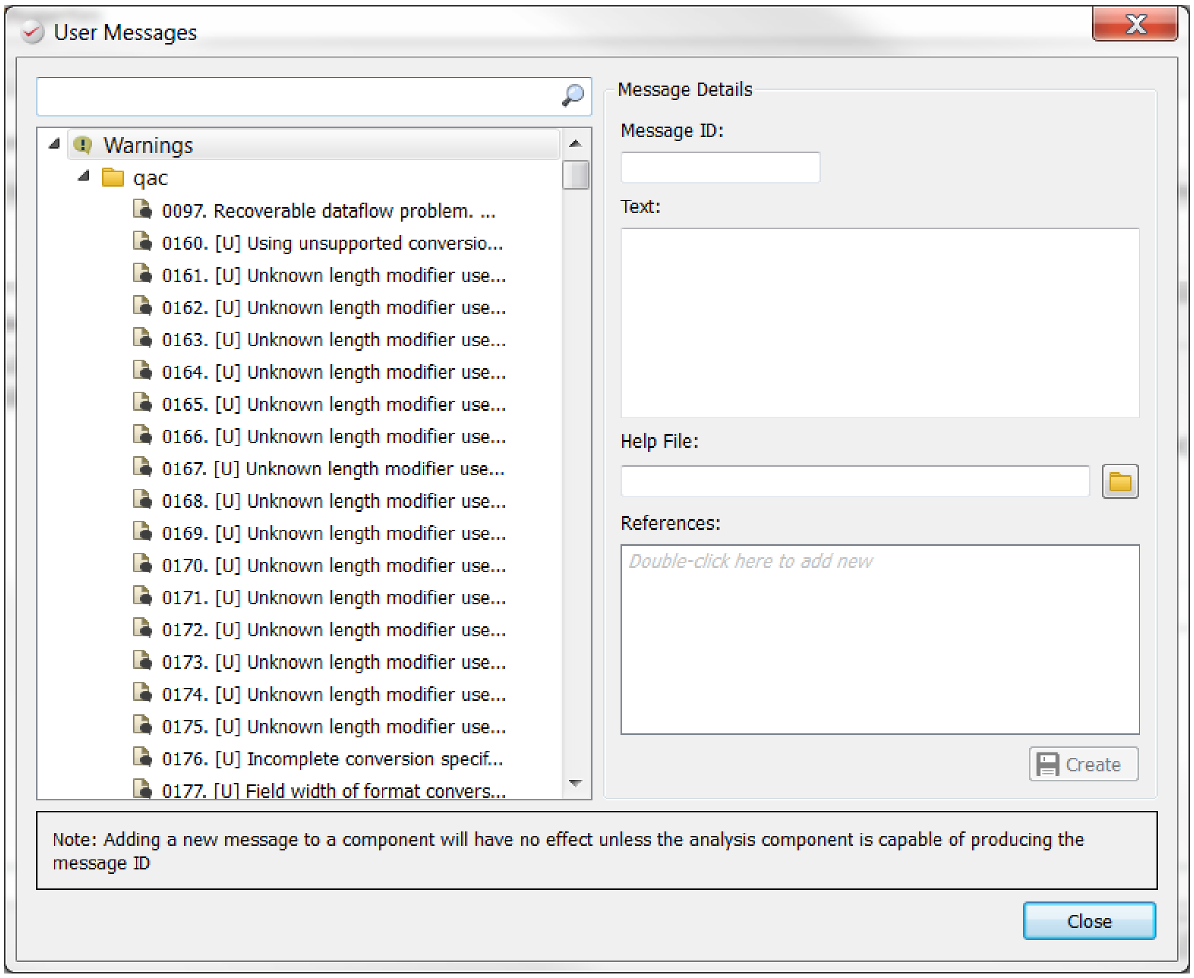
Figure 5.5: New Message Icon

and a "User Messages" panel will be displayed:

To create a new message it will be necessary to insert the message ID [11], the text displayed by the message, a help file in HTML format to be linked to and, optionally, some references to be added to the context of the message.

Once created, the user message, will be added to the specific "User Specified" section of the messages list. To remove the message just right click on it and select "Remove".

[11]Note that the analysis component has to be able to produce the selected message ID, otherwise adding the message itself to the component won't have any effect. User message numbers should be from 6000 to 9999.



0097. Recoverable dataflow problem. ...
0160. [U] Using unsupported conversio...
0161. [U] Unknown length modifier use...
0162. [U] Unknown length modifier use...
0163. [U] Unknown length modifier use...
0164. [U] Unknown length modifier use...
0165. [U] Unknown length modifier use...
0166. [U] Unknown length modifier use...
0167. [U] Unknown length modifier use...
0168. [U] Unknown length modifier use...
0169. [U] Unknown length modifier use...
0170. [U] Unknown length modifier use...
0171. [U] Unknown length modifier use...
0172. [U] Unknown length modifier use...
0173. [U] Unknown length modifier use...
0174. [U] Unknown length modifier use...
0175. [U] Unknown length modifier use...
0176. [U] Incomplete conversion specif...
0177. [U] Field width of format convers...

Message Details

Message ID:

Text:

Help File:

References:

Double-click here to add new

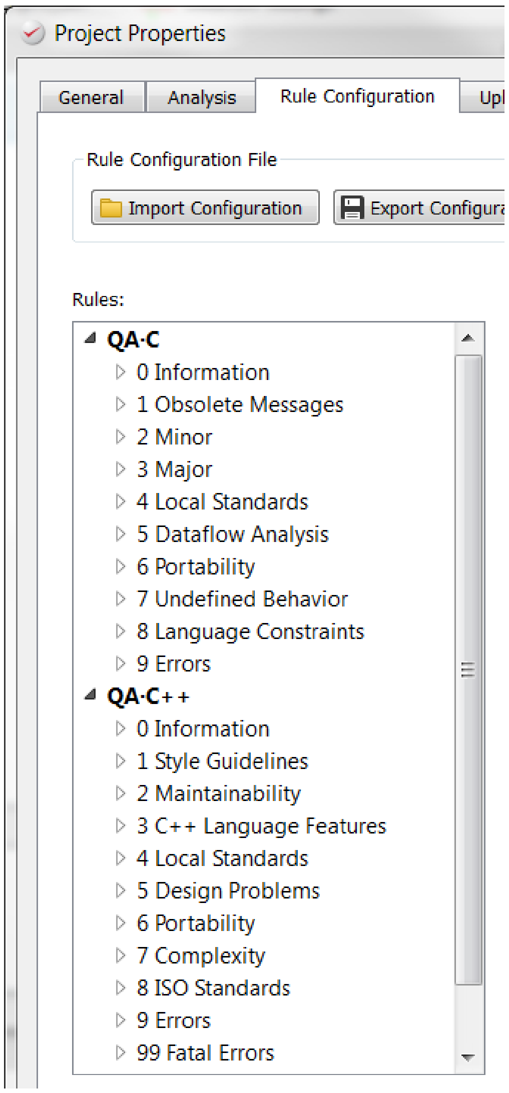
Note: Adding a new message to a component will have no effect unless the analysis component is capable of producing the message ID

Figure 5.6: User Messages

6 Rule Configuration

PRQA Framework uses a Rule Configuration file to define the mapping of warning messages detected by analysis to Rules. This panel presents the rules in a tree-like structure that is organized into levels and sub-groups. When the lowest sub-group/rule is selected the actual enforcement messages are shown.

The Rule Configuration panel is accessed via Projects->Open Project Properties-> Rule Configuration tab. The window opens with the current Rule Configuration file for the project:



- QA·C
  - 0 Information
  - 1 Obsolete Messages
  - 2 Minor
  - 3 Major
  - 4 Local Standards
  - 5 Dataflow Analysis
  - 6 Portability
  - 7 Undefined Behavior
  - 8 Language Constraints
  - 9 Errors
- QA·C++
  - 0 Information
  - 1 Style Guidelines
  - 2 Maintainability
  - 3 C++ Language Features
  - 4 Local Standards
  - 5 Design Problems
  - 6 Portability
  - 7 Complexity
  - 8 ISO Standards
  - 9 Errors
  - 99 Fatal Errors

Figure 6.1: Top-Level Default Rule Configuration Groups

A new Rule Configuration file may be chosen by selecting Open Configuration button.

The file must have the extension .rcf. When the file is opened, the mapping is shown in the tree structure described below.

6.1 The Default and Critical Rule Configuration File

Some Rule Configuration files are delivered with the product.

The Default file defines mappings for all the message in each language. This can be used to see what messages are available. The Critical file contains a reduced set of messages for QA·C and QA·C++. The reduced set is aimed at the more important issues. Users working with legacy C or C++ code are recommended to start off with the Critical file.

Several sections are shown in these files because there is a section in the configuration file for each language.

Other Rule Configuration files, for example: m3cm, may only show one message group because the Compliance Module only applies to one target language.

6.2 Rule Configuration Operations

In the Rules panel, a number of operations are available from the context menu that appears when a level is selected.

Note: These operations have a direct affect on the project specific configuration file.

6.2.1 Edit

Every level in the Rules tree has an associated ID, text, rule help file and category associated with it. These entities may be modified within the Rule Editor:

Rules may be de-activated within this panel and any of the associated values may be modified by the user: ID, text, the link to the rule help file, and categories.

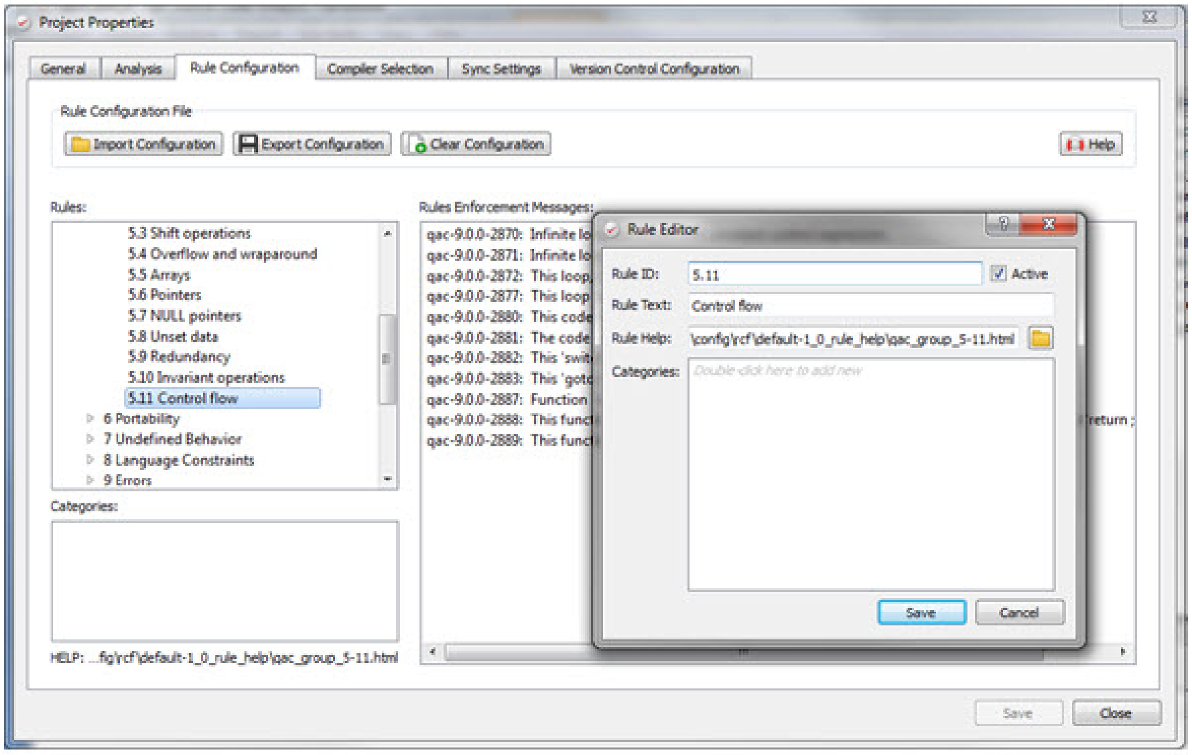
Changes that are made here are reflected in the Rule Groups panel after analysis.

Note: Rule IDs may be duplicated.

6.2.2 Remove

This operation removes a level, and all its sub-levels, from the Rule Configuration file.

Note: This should be used with caution as the blocks of enforcement message are removed too. Only the local copy of the Rule Configuration in the project configuration



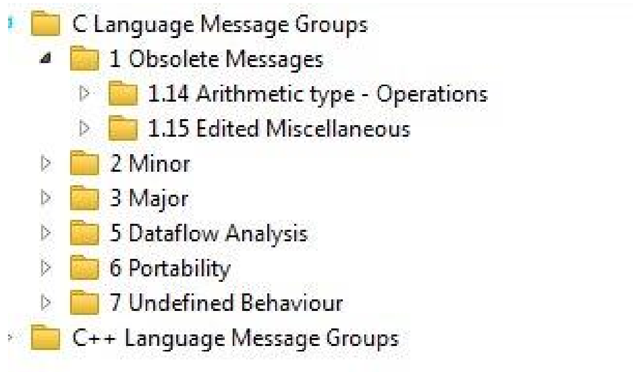
5.3 Shift operations
5.4 Overflow and wraparound
5.5 Arrays
5.6 Pointers
5.7 NULL pointers
5.8 Unset data
5.9 Redundancy
5.10 Invariant operations
5.11 Control flow
6 Portability
7 Undefined Behavior
8 Language Constraints
9 Errors

HELP: ...fig\rcf\default-1_0_rule_help\qac_group_5-11.html

```
qac-9.0.0-2870: Infinite lo
qac-9.0.0-2871: Infinite lo
qac-9.0.0-2872: This loop
qac-9.0.0-2877: This loop
qac-9.0.0-2880: This code
qac-9.0.0-2881: The code
qac-9.0.0-2882: This 'swit
qac-9.0.0-2883: This 'goto
qac-9.0.0-2887: Function
qac-9.0.0-2888: This func
qac-9.0.0-2889: This func
```

Rule ID: 5.11 ☑ Active
Rule Text: Control flow
Rule Help: \config\rcf\default-1_0_rule_help\qac_group_5-11.html
Categories: Double-click here to add new

Figure 6.2: The Rule Editor



C Language Message Groups
- 1 Obsolete Messages
  - 1.14 Arithmetic type - Operations
  - 1.15 Edited Miscellaneous
- 2 Minor
- 3 Major
- 5 Dataflow Analysis
- 6 Portability
- 7 Undefined Behaviour

C++ Language Message Groups

Figure 6.3: Rule Groups Display

directory is affected, and the original can be restored by reloading it using Open Configuration.

6.2.3 Disable/Enable

Disable the rule so that the Analysis results, that is to say the warning messages are no longer mapped to it. Initially, the Rule Groups panel shows a disabled rule as follows:

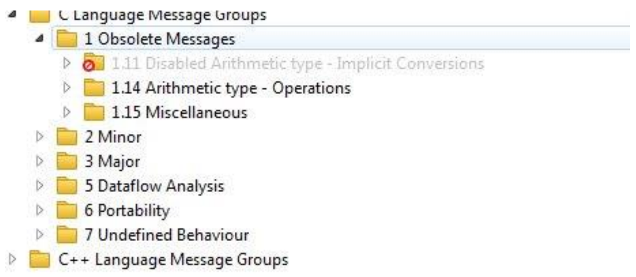

C Language Message Groups
1 Obsolete Messages
1.11 Disabled Arithmetic type - Implicit Conversions
1.14 Arithmetic type - Operations
1.15 Miscellaneous
2 Minor
3 Major
5 Dataflow Analysis
6 Portability
7 Undefined Behaviour
C++ Language Message Groups

Figure 6.4: Disabled Rule

When the project is re-analyzed, warning messages are no longer mapped to the rule, and it is not shown in the tree. The message counts are reduced accordingly. When a rule is disabled, the context menu presents an Enable option.

6.2.4 New Rule

A new Rule may be defined at any level in the tree. This is a really useful feature for defining customized rule groups in a hierarchy, for example:

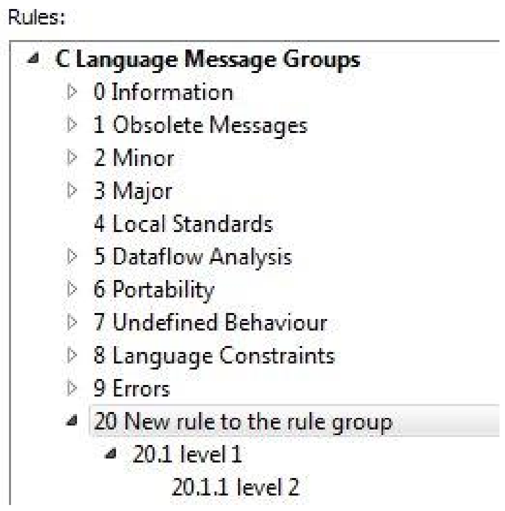
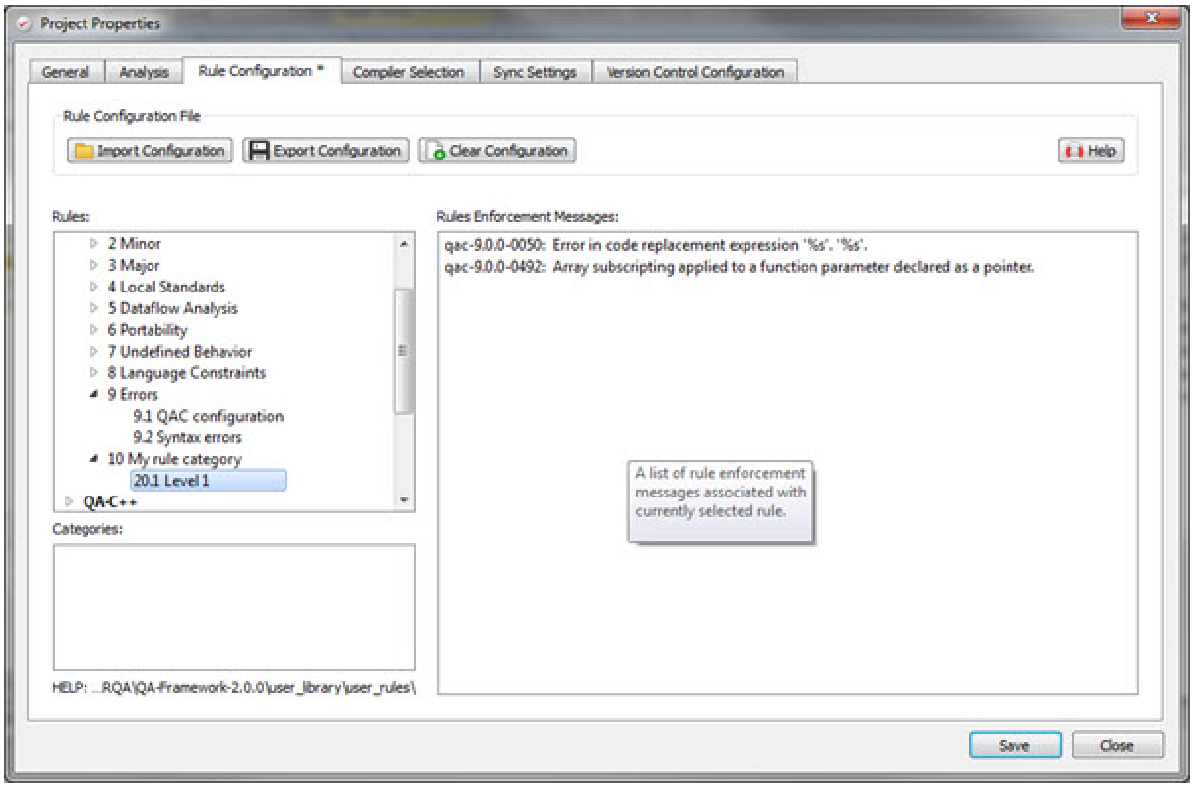


Figure 6.5: New Rules Hierarchy

The Rule Editor panel described previously is presented so that the user can define the new Rule ID, required text, link to the Rule Help file and category.

6.2.5 Messages

The rule levels in the hierarchy may contain either sub-groups or actual enforcement messages, or both. For example, although the new rule 20 contains sub-levels, we can define an enforcement message as well, for example: qac-8.1.2-1831.



qac-9.0.0-0050: Error in code replacement expression '%s'. '%s'.
qac-9.0.0-0492: Array subscripting applied to a function parameter declared as a pointer.

HELP: ...RQA\QA-Framework-2.0.0\user_library\user_rules\

Figure 6.6: Message Added to Rule

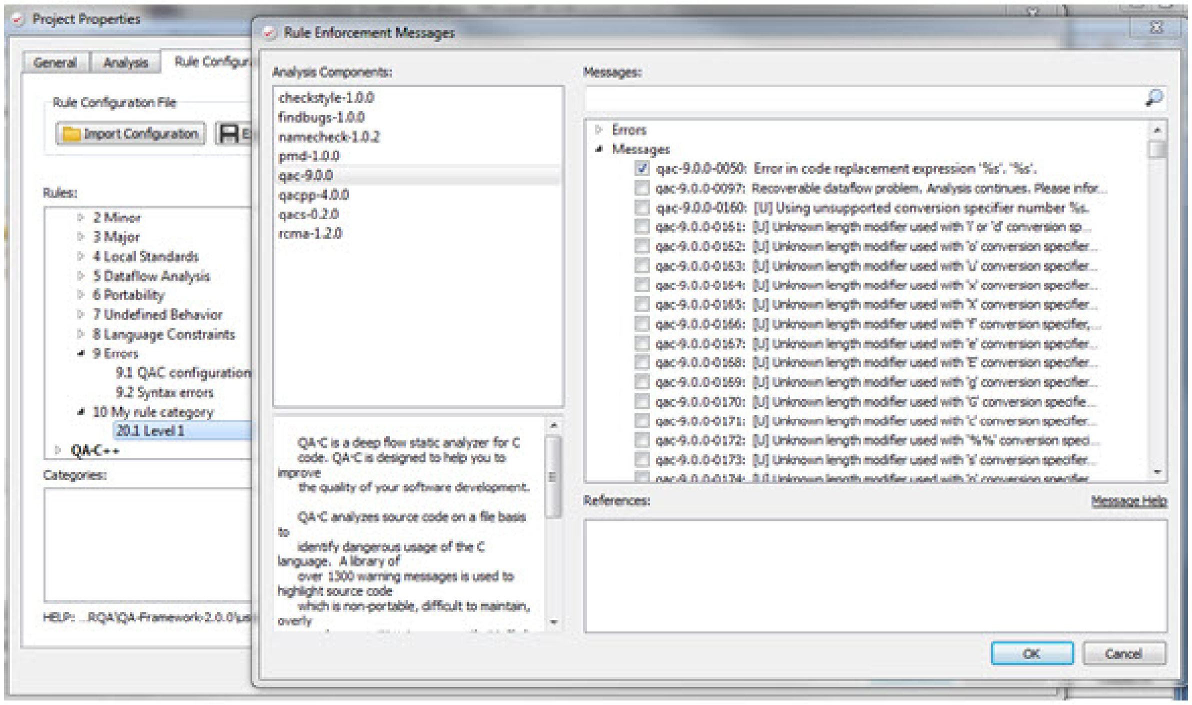
To associate any message to any level, the Rule Enforcement Messages panel is presented:

Within the Rule Enforcement Message panel, all the available Analysis Components have their own set of produced messages for errors, warnings and user messages. User messages are defined via the Analysis Configuration tab, for a specific Analysis Component.

When a message group, Errors/User or Warnings, is opened by selecting the marker, the available message set is presented and the user selects those required.

In the example above, two enforcement messages were allocated for enforcement of the new rule 20.1.1. Equally, the selection of enforcement messages for any existing rule may be changed by selecting/de-selecting entries in the panel below:

Removing Rules from a Rule Configuration file is achieved by the Remove menu option in the Rules panel.



QA·C is a deep flow static analyzer for C code. QA·C is designed to help you to improve the quality of your software development.

QA·C analyzes source code on a file basis to identify dangerous usage of the C language. A library of over 1300 warning messages is used to highlight source code which is non-portable, difficult to maintain, overly

```
[✓] qac-9.0.0-0050: Error in code replacement expression '%s'. '%s'.
[ ] qac-9.0.0-0097: Recoverable dataflow problem. Analysis continues. Please infor...
[ ] qac-9.0.0-0160: [U] Using unsupported conversion specifier number %s.
[ ] qac-9.0.0-0161: [U] Unknown length modifier used with 'i' or 'd' conversion sp...
[ ] qac-9.0.0-0162: [U] Unknown length modifier used with 'o' conversion specifier...
[ ] qac-9.0.0-0163: [U] Unknown length modifier used with 'u' conversion specifier...
[ ] qac-9.0.0-0164: [U] Unknown length modifier used with 'x' conversion specifier...
[ ] qac-9.0.0-0165: [U] Unknown length modifier used with 'X' conversion specifier...
[ ] qac-9.0.0-0166: [U] Unknown length modifier used with 'f' conversion specifier,...
[ ] qac-9.0.0-0167: [U] Unknown length modifier used with 'e' conversion specifier...
[ ] qac-9.0.0-0168: [U] Unknown length modifier used with 'E' conversion specifier...
[ ] qac-9.0.0-0169: [U] Unknown length modifier used with 'g' conversion specifier...
[ ] qac-9.0.0-0170: [U] Unknown length modifier used with 'G' conversion specifie...
[ ] qac-9.0.0-0171: [U] Unknown length modifier used with 'c' conversion specifier...
[ ] qac-9.0.0-0172: [U] Unknown length modifier used with '%%' conversion speci...
[ ] qac-9.0.0-0173: [U] Unknown length modifier used with 's' conversion specifier...
[ ] qac-9.0.0-0174: [U] Unknown length modifier used with 'n' conversion specifier
```

Figure 6.7: New Rule Enforcement Messages

Removing a message from a rule is achieved by right-click the message in the Rules Enforcement Messages panel, and selecting the Remove option here. This has the same effect as de-selecting the message through the Rules Enforcement Messages panel, where it may be selected again.

Note: It is possible to Remove All Messages from a rule.

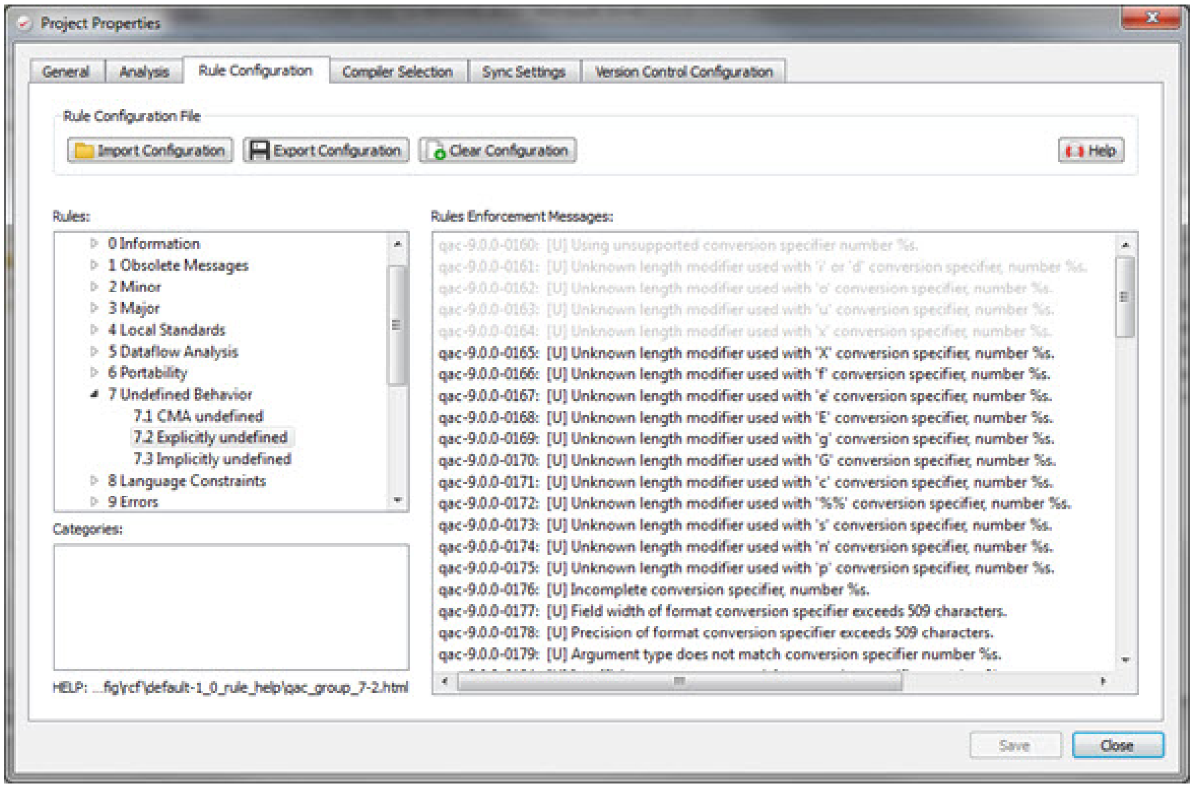
Disabling a message from a rule is achieved by right-clicking the message in the Rules Enforcement Messages panel, and selecting the Disable option here. It can be re-enabled by right clicking the message and selecting the Enable option.

6.3 New Rule Configuration

It is possible to define a completely new Rule Configuration hierarchy, by selecting New Configuration. The Rule Group is started in the Rules panel. When the rule group marker is clicked, a top level group is started. When its marker is clicked, a sub-group level is started.

The rule group levels can be deepened by using New Rule to define the lower levels and then finally applying enforcement messages, via the Messages menu option.

The resultant hierarchy is saved in the local project configuration area as template.rcf,



```
qac-9.0.0-0160: [U] Using unsupported conversion specifier number %s.
qac-9.0.0-0161: [U] Unknown length modifier used with 'i' or 'd' conversion specifier, number %s.
qac-9.0.0-0162: [U] Unknown length modifier used with 'o' conversion specifier, number %s.
qac-9.0.0-0163: [U] Unknown length modifier used with 'u' conversion specifier, number %s.
qac-9.0.0-0164: [U] Unknown length modifier used with 'x' conversion specifier, number %s.
qac-9.0.0-0165: [U] Unknown length modifier used with 'X' conversion specifier, number %s.
qac-9.0.0-0166: [U] Unknown length modifier used with 'f' conversion specifier, number %s.
qac-9.0.0-0167: [U] Unknown length modifier used with 'e' conversion specifier, number %s.
qac-9.0.0-0168: [U] Unknown length modifier used with 'E' conversion specifier, number %s.
qac-9.0.0-0169: [U] Unknown length modifier used with 'g' conversion specifier, number %s.
qac-9.0.0-0170: [U] Unknown length modifier used with 'G' conversion specifier, number %s.
qac-9.0.0-0171: [U] Unknown length modifier used with 'c' conversion specifier, number %s.
qac-9.0.0-0172: [U] Unknown length modifier used with '%%' conversion specifier, number %s.
qac-9.0.0-0173: [U] Unknown length modifier used with 's' conversion specifier, number %s.
qac-9.0.0-0174: [U] Unknown length modifier used with 'n' conversion specifier, number %s.
qac-9.0.0-0175: [U] Unknown length modifier used with 'p' conversion specifier, number %s.
qac-9.0.0-0176: [U] Incomplete conversion specifier, number %s.
qac-9.0.0-0177: [U] Field width of format conversion specifier exceeds 509 characters.
qac-9.0.0-0178: [U] Precision of format conversion specifier exceeds 509 characters.
qac-9.0.0-0179: [U] Argument type does not match conversion specifier number %s.
```

HELP: ...fig\rcf\default-1_0_rule_help\qac_group_7-2.html

Figure 6.8: Existing Rule Enforcement Messages

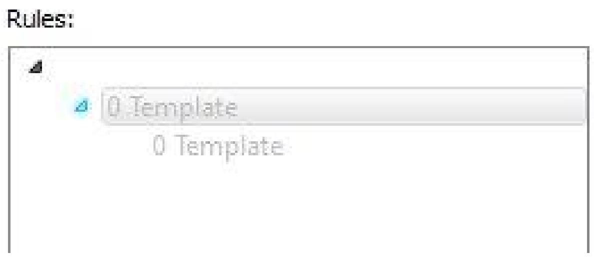


Figure 6.9: New Rule Configuration Structure

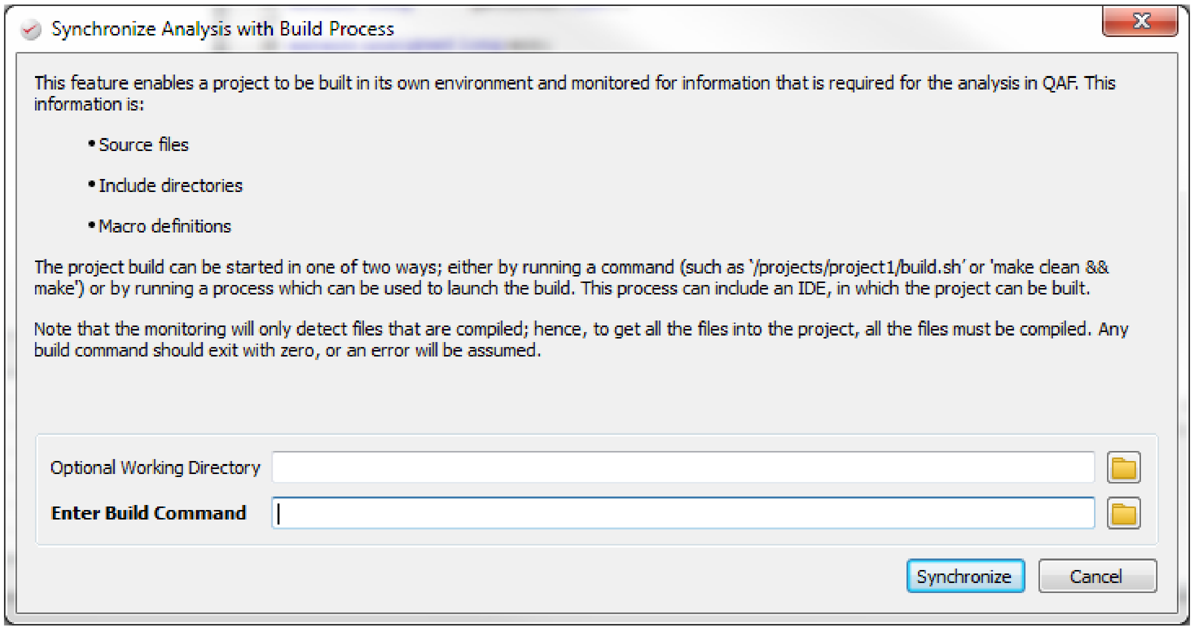
unless Save Configuration As is used.

7 Project Synchronization

PRQA Framework synchronization is used to get source files, include paths and definitions from an existing C/C++ project. It works with the majority of Integrated Development Environments; build systems, such as Make, scons and CMake; and compilers.

7.1 Synchronizing

To perform synchronization select the Project→Synchronize from the main menu. The Synchronize Analysis with Build Process panel will be displayed.



Synchronize Analysis with Build Process

This feature enables a project to be built in its own environment and monitored for information that is required for the analysis in QAF. This information is:

- Source files
- Include directories
- Macro definitions

The project build can be started in one of two ways; either by running a command (such as '/projects/project1/build.sh' or 'make clean && make') or by running a process which can be used to launch the build. This process can include an IDE, in which the project can be built.

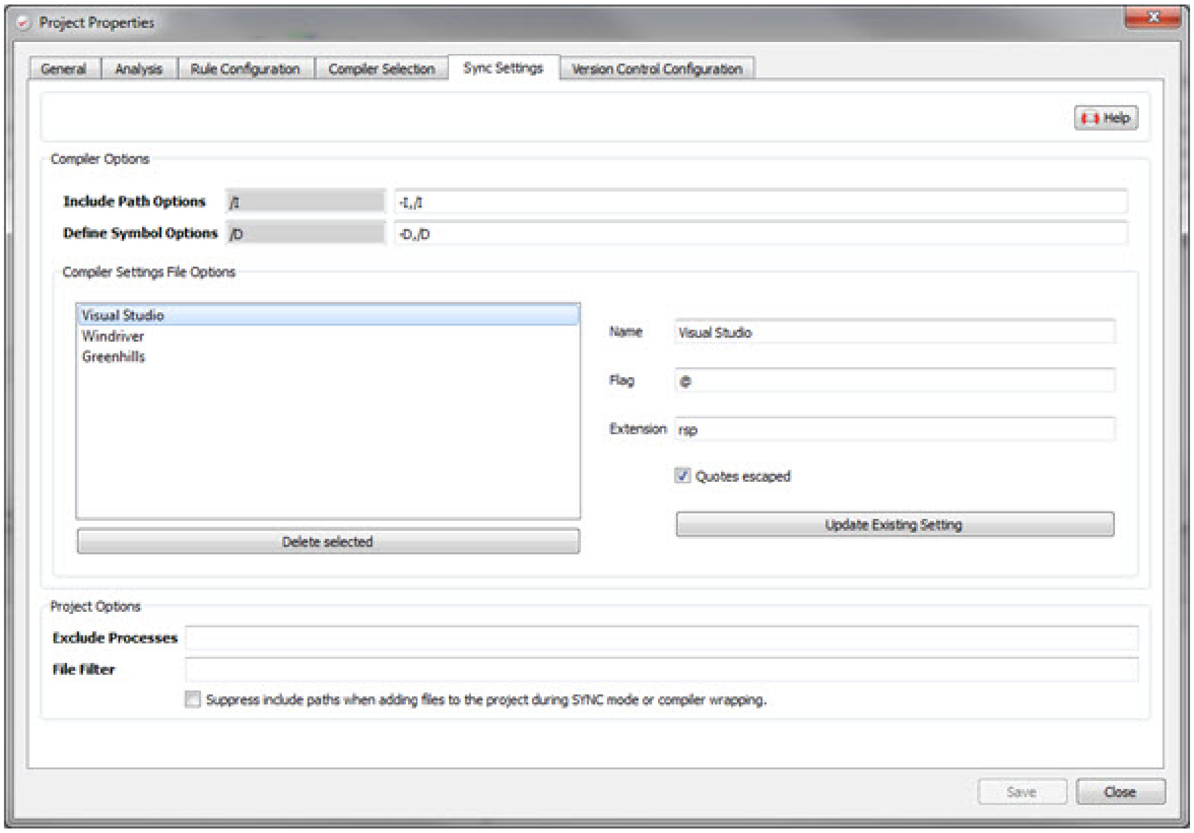
Note that the monitoring will only detect files that are compiled; hence, to get all the files into the project, all the files must be compiled. Any build command should exit with zero, or an error will be assumed.

Figure 7.1: Synchronize Analysis with Build Process

From this panel the browse button can open an Integrated Development Environment, a project or a script to run. If a working directory is selected then commands can be entered as if on the command line, for example 'Make' or some other build/compile command. Note that a full build must be performed when doing a synchronization. Any existing files in the QAF project will be removed if they are not in the current build.

7.2 Synchronization Options

In the majority of cases synchronisation works with the default options provided, however, if problems are encountered then they can be solved by changing the existing defaults.



Compiler Options

Include Path Options /I -I,/I

Define Symbol Options /D -D,/D

Compiler Settings File Options

Visual Studio
Windriver
Greenhills

Name: Visual Studio

Flag: @

Extension: rsp

[✓] Quotes escaped

Project Options

Exclude Processes

File Filter

[ ] Suppress include paths when adding files to the project during SYNC mode or compiler wrapping.

Figure 7.2: Synchronization Settings

The figure displays the default synchronization settings that all new projects use. Settings in the disabled (read-only) fields will have been read from the CCT file. If the CCT has settings file options, this set will be displayed along with the ones shown. These synchronization settings can be changed if the project synchronization is not successful.

7.2.1 Include Path Options

The 'Include Path Options' contain the flags used by the compiler to specify an include path. The defaults of -I for Linux and /I for Windows should be suitable for most settings. Some compilers may use other flags or allow additional flags and these can be added in the Include Path Options if required. For example GCC also allows -iquote to be used.

7.2.2 Define Symbol Options

'Define Symbol Options' contains the flags used by the compiler to specify and define a macro on the command line. The defaults of -D for Linux and /D for Windows should be suitable for most requirements. If your compiler uses a different flag it may be added to the comma separated list or used to replace the defaults.

7.2.3 Compiler Settings File Options

Compilers may store the command line options in a file. The 'Compiler Settings File Options' contains the flag used to specify this, the file extension used and the file format. If a file extension flag is not used then this can be left blank.

If the 'Quotes escaped' checkbox is ticked the synchronisation will expect literal quotes to be escaped. If it is not ticked then all quotes will be treated as literal and passed to the analyzers as written in the settings file.

A set of options can be named; selecting the name in the list box will show the options. If the name already exists then clicking 'Update Existing setting' will overwrite the old settings. If a new name is entered this can be saved as a new entry.

To delete an entry select the name in the list box and click 'Delete selected'.

7.2.4 Exclude Processes

Synchronization may detect false positives for compiler processes. This can cause source files to be incorrectly added to the project, or for the Defines and Include paths to be incorrectly formatted. To stop this from happening the "Exclude Processes" option can be used. Any process which mentions a source file may cause a false positive. For example, `rm`, `ln` or `/bin/sh`. Compile commands are often launched with `/bin/sh` but the command line may not be fully resolved at this point which can lead to incorrect information being found.

After a synchronisation all compiler processes used to create the project will be displayed. Any processes which do not match your compiler should be added to the "Exclude Processes" list and Synchronization run again.

7.2.5 File Filter

Using the file filters it is possible to filter out individual source files and folders. String matching is used so 'project/subproject' will filter the folder '/home/project/subproject', as well as 'project/subproject1.cpp'.

7.2.6 Suppress Included Paths

This is the same option as seen on the General panel. For a description of this option see Analysis Configuration.

8 Version Control Configuration

PRQA Framework operates on the code that is currently being worked on locally by the developer. In most cases, this will have originally come from a Version Control System, but it is the local version of the code that the analysis results are generated for. Analysis only needs to know the location of the local code, it does not need to know where it came from. For most people working on the desktop, there is no need to tell PRQA Framework about the Version Control System.

By contrast, QA·Verify does normally work directly against a Version Control System. QA·Verify deals not just with the current code, but it holds a series of snapshots from the lifetime of the project. Extracting historical code from a Version Control System is the most efficient way of handling this. It also ensures its reporting is accurate and traceable. Results uploaded to QA·Verify normally come from a server-side automated process that extracts code from the Version Control System before generating the results. There is then a clear correspondence between results and versioned code.

A QA·Verify Unified project (see Working with QA·Verify) will have a configuration file attached to it that gives details on the Version Control System. This is known as a VCF file. There is no need to create this file locally, this is done centrally within QA·Verify, and the necessary file is just pulled down from the server as part of the project definition.

The only circumstances where a VCF file needs to be created or adjusted locally are:

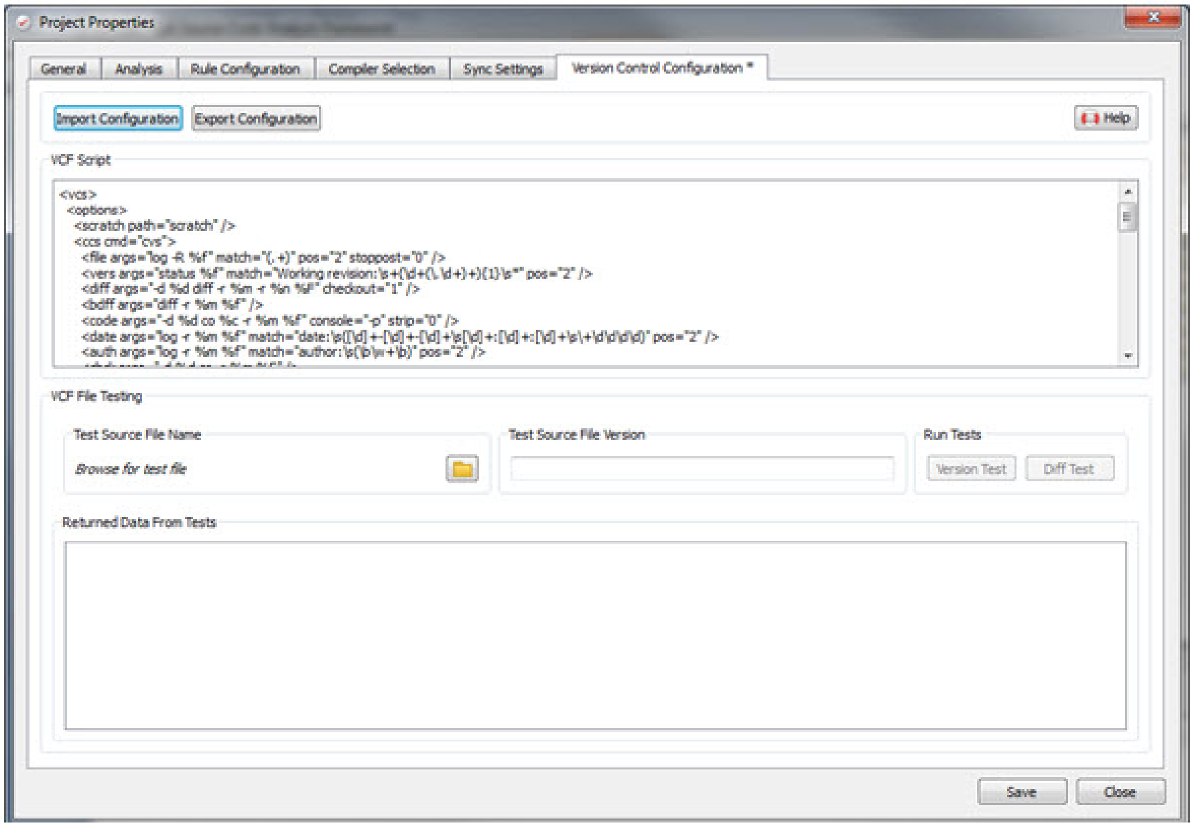
1. If the Baseline by Version Control feature is being used. Baselining needs a reference copy of the code, so that it can compare the current code with the code that existed when the baseline was taken.
2. The current project is not a Unified project, but results generated against checked-in code need to be uploaded to QA·Verify
3. For some Version Control Systems, a VCF file pulled down from QA·Verify may need adjustment for local usage.

To provide the necessary information, select 'Open Project Properties' and then select the 'Version Control Configuration' panel.

A number of scripts exists for popular version control systems. They can be imported using the 'Import Configuration' button. When imported, the script is displayed in the 'VCF Script' box. The script can be edited if necessary to work with the user's Version Control system.

The user should test that the script works (see VCF File Testing below).

Once it has been confirmed that the script works correctly, be sure to save the project properties. The script will then be used by PRQA Framework.



```
<vcs>
 <options>
  <scratch path="scratch" />
  <ccs cmd="cvs">
   <file args="log -R %f" match="(.+)" pos="2" stoppost="0" />
   <vers args="status %f" match="Working revision:\s+(\d+(\.\d+)+){1}\s*" pos="2" />
   <diff args="-d %d diff -r %m -r %n %f" checkout="1" />
   <bdiff args="diff -r %m %f" />
   <code args="-d %d co %c -r %m %f" console="-p" strip="0" />
   <date args="log -r %m %f" match="date:\s([\d]+-[\d]+-[\d]+\s[\d]+:[\d]+:[\d]+\s\+\d\d\d\d)" pos="2" />
   <auth args="log -r %m %f" match="author:\s(\b\w+\b)" pos="2" />
```

Figure 8.1: Version Control Configuration Panel

8.1 VCF File Testing

To check that the VCF file is configured correctly it can be tested using the options provided.

First ensure that PRQA Framework can interface to the project's Version Control System. Select a source file using the Browse button and select the 'Version Test' button. Data from running the test will be displayed at the bottom of the Version Control Config panel. Errors will be shown if there are problems.

To ensure PRQA Framework can retrieve a file enter a version number into the 'Test Source File Version' field. This should be a valid variant of the source file selected and then press the 'Diff Test' button. The changes in the file will be shown in the display box if the retrieve was successful.

8.2 Using PRQA Framework Without a Version Control System

If a Version Control System is not used but it is required to upload to QA·Verify then the ’single-comb’ VCF script can be imported. This will include file versions and authors information in the snapshot. If file versions and authors information is not required then the ’single-sop’ VCF script can be imported.

9 Working with QA·Verify

QA·Verify is a centralised application that manipulates, summarises and gives wide visibility to the results generated through PRQA Framework.

For PRQA Framework users, the key interactions with QA·Verify are:

- QA·Verify can provide centralised project definitions.
- Analysis results can be uploaded to QA·Verify. A single set of uploaded results becomes a snapshot in QA·Verify.
- A snapshot from QA·Verify can be used as a baseline.
- Suppressions defined in QA·Verify can be downloaded and applied to local results.

9.1 Connecting to QA·Verify

Data is uploaded to or downloaded from QA·Verify through a QA·Verify connection. To acquire a connection, you need to log on. Each user should have a unique logon ID for QA·Verify, both for licensing reasons and because QA·Verify has a permissions scheme that allows administrators to control the entitlements given to individual users. If you do not have a logon ID, ask the QA·Verify administrator to create an account for you. To log on to QA·Verify, choose the menu item:

QA·Verify menu : Connect

This brings up a connection dialog, where you should enter the address of the QA·Verify server, and your credentials. The QA·Verify server address is the same one you would use to access QA·Verify through a browser. Your QA·Verify administrator will have the details.

The default port number is 8080, but this can be different on different sites if this clashes with another application, or violates a policy.

You can ask for the details to be remembered, to simplify your next logon. For security reasons, the password will not be stored. The dialog will in any case store the history of previous connection details, which simplifies logon if you access different QA·Verify servers for different projects. Connection details are stored separately for each user on the machine. Other users will not be able to pick up your logon credentials.

Once the logon details have been entered, click Connect. For QA·GUI, the connection status will be shown bottom right of the main screen. If the connection attempt fails, two error codes are shown. The second one is a standard http error code. Clicking on this connection status area brings up the connection dialog.

Some QA·Verify menu items will be disabled until a successful connection is made.

If you connect from PRQA Framework to QA·Verify from a second machine, the connection from the first machine will be broken. You can however have simultaneous connections from PRQA Framework and from a browser (which is the standard way of accessing QA·Verify).

9.2 Centralising Project Definitions

Most code projects are worked on by more than one person. The PRQA Framework project associated with that code project would be almost the same for all developers. For reasons of efficiency and consistency, it is recommended for one person to set up the PRQA Framework project, and then distribute it to anyone else who needs it. The work done by the first person to ensure that all the correct options have been chosen need not be repeated by the others.

Project definitions are just files, and can be shared through a source control system, or simply by copying. However, using QA·Verify to share these definitions has additional advantages. It becomes what is known as a Unified project. On the QA·Verify server, a Unified project has both the normal QA·Verify project definition and a stored PRQA Framework project definition. The PRQA Framework part of the Unified project can be downloaded onto a local machine. A permanent linkage is maintained between the local project and the project on the QA·Verify server. The linkage helps with maintaining the project definition, but also with uploading results, and downloading baselines and suppressions.

9.2.1 Creating a Unified Project

This process starts with a single user creating a normal PRQA Framework project. That user should add all the relevant files to the project, ensure the options are set correctly for that particular set of code, and that the results are as expected. The user needs to be aware that the project definition will be shared, so the project roots should be set up with the full set of users in mind.

A particularly important 'user' is the automated analysis which takes place on a central server, as described in Desktop and Server-side Working. Whereas desktop users can make minor adjustments to a project definition once they have downloaded it, this is much harder to do in a fully-scripted automated solution. If there is an automated system, and that system will regularly download a project definition, then the definition should be set up with that system in mind. An example might be a Dataflow setting, which is often set at a deeper level for the automated system than for the desktop.

It is recommended that before uploading, all rules should be set as enabled - see Rule

Configuration Operations. The disabling of individual rules within a ruleset is a feature of a local project definition, rather than a centralized definition. Any disable flag will be ignored when the project is uploaded to QA·Verify. If a rule should not be applied, then delete it before the upload starts. Users downloading the definition onto their local machine can disable rules if they wish, but all downloaded rules will initially be enabled.

Once the user is satisfied with the project definition, it can be uploaded to QA·Verify:

QA·Verify menu : Upload Project Definition : Create

If the command is disabled, connect to QA·Verify first.

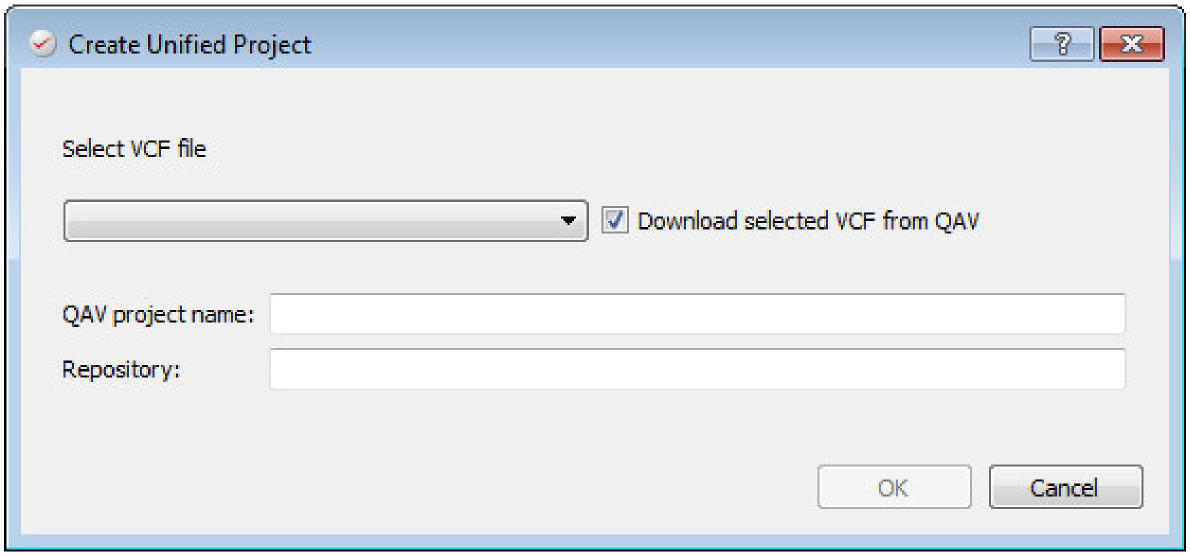


Figure 9.1: QA·Verify Upload Configuration

In the dialog that appears:

1. Select a version control file (VCF) from the list. This file holds all the details used by QA·Verify to connect to the version control system to access the source files. The VCF file selected needs to match the version control system used for this code project.
   - Normally, you would leave the 'Download selected VCF from QAV' check-box selected. This makes the VCF file available to the local system for operations such as baselining.
2. Provide a name for the Unified project. Most often, this would be the same name as used for the code project, but it could be any name that is meaningful to the users of the project.
3. Some version control systems need additional project-specific information to access

the source code. Set this into the Repository field. Refer to the QA·Verify documentation for more detail.

The act of uploading the project definition will turn that project into a Unified project:

1. A project will be created within QA·Verify.
2. The PRQA project definition will be stored centrally in QA·Verify so that other users can download it.
3. The local project will change its name to that of the Unified project.

9.2.2 Setting Project Options in QA·Verify

The upload will create a QA·Verify project, using the default options for project creation. In most cases, the project definition will need adjustment, to set permissions, and to add definitions to make use of the QA·Verify flexibility in areas such as reporting and code review workflows.

9.2.3 Downloading a Unified Project

Other users can download the Unified project, and start using it to analyse code.

Before downloading, ensure that the code project is in place and has all the necessary code files. To download, close your current project in PRQA Framework if you have it open, and navigate to:

QA·Verify menu : Download Project Definition : Create

If the command is disabled, connect to QA·Verify first.

In the dialog, just select the project you want to download, and click OK.

Once the project is downloaded, it may be necessary to make some adjustments to suit the local environment. For example, if the source files are in a slightly different location from other people working on the project, adjust the project roots as described in Analyzing Your Project. If additional files have been added to the code project since the Unified project definition was uploaded, the project should be updated as described in Populating Your Project.

In most cases, once the project definition has been downloaded, it should be ready for immediate use.

9.2.4 Working on a Unified Project

Once downloaded, a Unified project is just like any other PRQA project on the local machine. The only difference is that it happens to have permanent linkage to the associated project on QA·Verify. The linkage is not active, in the sense that there is no automatic updating through that linkage. The user is at all times in control.

As a result, the user is able to make changes to the project configuration privately.

Just like any other PRQA project, a user is able to adjust the project settings for a Unified project. For example, dataflow could be turned on or off, or rules could be enabled or disabled to concentrate attention on one particular area during an initial development phase.

9.2.5 Updating a Unified Project

Projects change over time, and it may well be that the project definition held centrally in QA·Verify becomes out of date. The definition can be updated through the command:

QA·Verify menu : Upload Project Definition : Update

Just as with project creation, the project should be put into a state suitable for sharing before upload.

Once updated, other users can download the project definition through:

QA·Verify menu : Download Project Definition : Update

This will overwrite their existing definition, so as with the initial download, they may need to make some adjustments to suit the local environment.

9.2.6 Maintaining a Ruleset

Once the project definition has been uploaded to QA·Verify and made into a Unified project, the ruleset belonging to that project needs to be maintained centrally, through a QA·Verify tool. The tool to be used is ConfigGUI, and can be found in the following location:

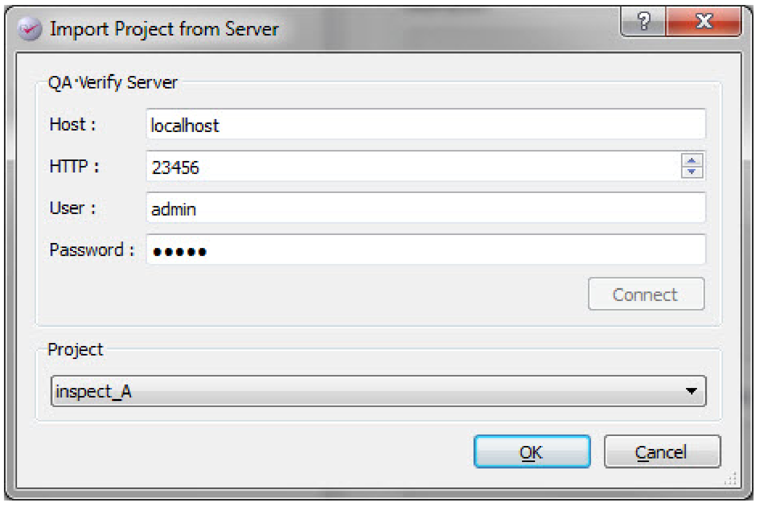
```
<install>\PRQA-Framework-xxx\components\qaverify-yyy\configgui.exe
```

where <install> is the installation directory for PRQA Framework, and xxx and yyy are the version numbers for PRQA Framework and QA·Verify respectively.

Run ConfigGUI, and connect to the QA·Verify server through:

File menu : Load Configuration : Import from Server.

This brings up the dialog:


Import Project from Server

QA·Verify Server

Host : localhost

HTTP : 23456

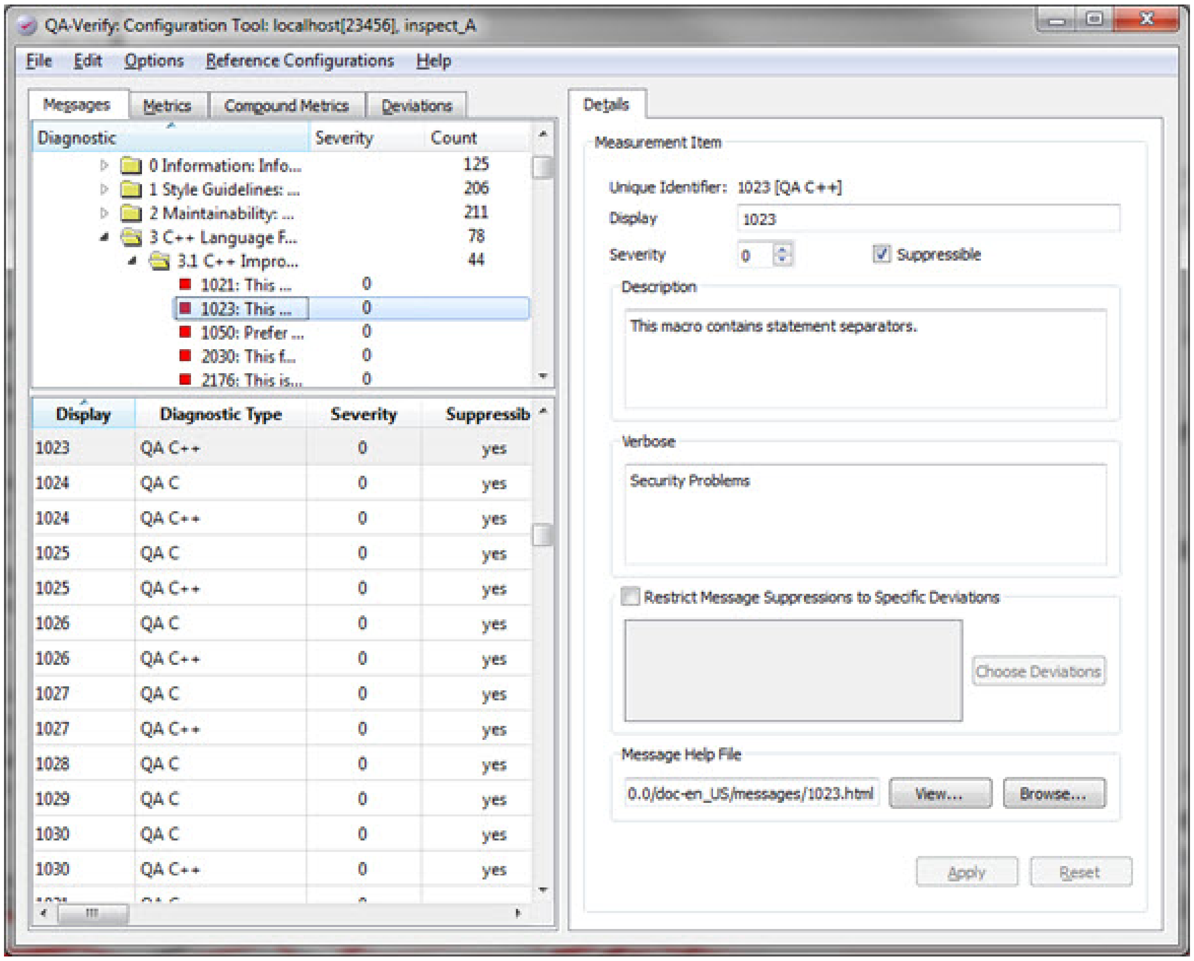
User : admin

Password : •••••

Figure 9.2: Import Project From Server

Enter the connection details and click **Connect**. The project drop-down list will then be populated with a list of projects. Select the one you want to update, and click OK.

ConfigGUI will import the ruleset from QA·Verify, and show them as follows:



QA-Verify: Configuration Tool: localhost[23456], inspect_A

File Edit Options Reference Configurations Help
Messages Metrics Compound Metrics Deviations
Details

Diagnostic	Severity	Count
0 Information: Info...		125
1 Style Guidelines: ...		206
2 Maintainability: ...		211
3 C++ Language F...		78
3.1 C++ Impro...		44
🟥 1021: This ...	0	
🟥 1023: This ...	0	
🟥 1050: Prefer ...	0	
🟥 2030: This f...	0	
🟥 2176: This is...	0	

Display	Diagnostic Type	Severity	Suppressib
1023	QA C++	0	yes
1024	QA C	0	yes
1024	QA C++	0	yes
1025	QA C	0	yes
1025	QA C++	0	yes
1026	QA C	0	yes
1026	QA C++	0	yes
1027	QA C	0	yes
1027	QA C++	0	yes
1028	QA C	0	yes
1029	QA C	0	yes
1030	QA C	0	yes
1030	QA C++	0	yes

Measurement Item

Unique Identifier: 1023 [QA C++]

Display: 1023

Severity: 0 ☑ Suppressible

Description: This macro contains statement separators.

Verbose: Security Problems

☐ Restrict Message Suppressions to Specific Deviations

Message Help File: 0.0/doc-en_US/messages/1023.html

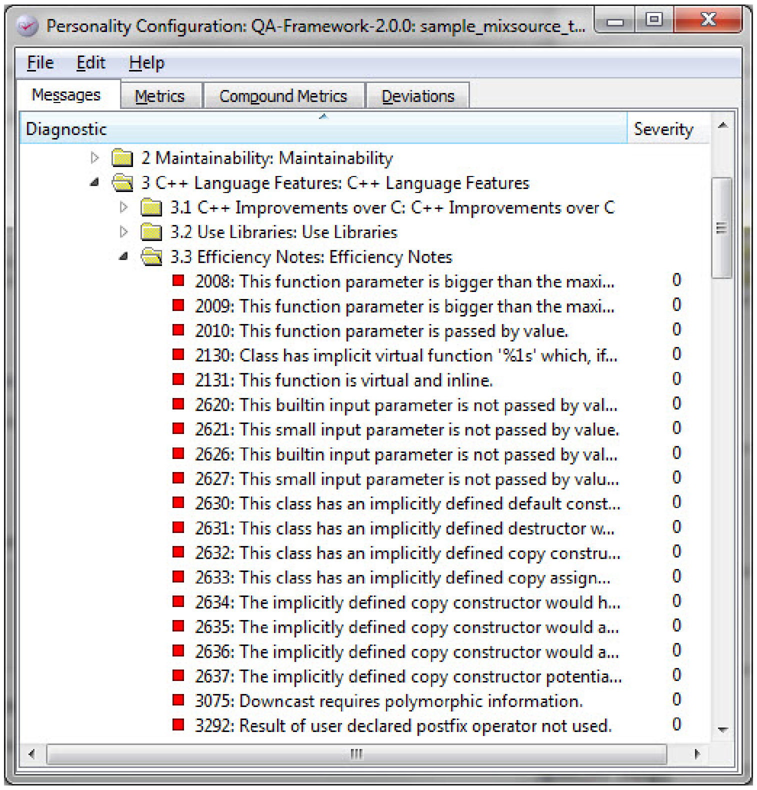
Figure 9.3: QA·Verify Configuration Tool

To remove a message from a ruleset, just select it in the top-level navigation pane, and bring up the context menu (normally through a right-click). The Delete option removes the message.

To add a message from the ruleset, if you already know the message number, the context menu has an Add diagnostic option which allows you to bring up a list of the possible messages, from which you can choose one. If you are not sure of the message number, or just want to see what other messages are available, it is best to work with a Reference Configuration. For example, to see what messages were delivered with the product, you can view the messages in a sample project (this is particularly useful after a version upgrade):

ReferenceConfigurations menu: PRQA Framework Project

This brings up a dialog from which you can browse the projects on your system. Select the root directory for the project, click OK, and the Personality Configuration dialog is shown:


Personality Configuration: QA-Framework-2.0.0: sample_mixsource_t...

File Edit Help

Messages | Metrics | Compound Metrics | Deviations

Diagnostic	Severity
2 Maintainability: Maintainability	
3 C++ Language Features: C++ Language Features	
3.1 C++ Improvements over C: C++ Improvements over C	
3.2 Use Libraries: Use Libraries	
3.3 Efficiency Notes: Efficiency Notes	
2008: This function parameter is bigger than the maxi...	0
2009: This function parameter is bigger than the maxi...	0
2010: This function parameter is passed by value.	0
2130: Class has implicit virtual function '%1s' which, if...	0
2131: This function is virtual and inline.	0
2620: This builtin input parameter is not passed by val...	0
2621: This small input parameter is not passed by value.	0
2626: This builtin input parameter is not passed by val...	0
2627: This small input parameter is not passed by valu...	0
2630: This class has an implicitly defined default const...	0
2631: This class has an implicitly defined destructor w...	0
2632: This class has an implicitly defined copy constru...	0
2633: This class has an implicitly defined copy assign...	0
2634: The implicitly defined copy constructor would h...	0
2635: The implicitly defined copy constructor would a...	0
2636: The implicitly defined copy constructor would a...	0
2637: The implicitly defined copy constructor potentia...	0
3075: Downcast requires polymorphic information.	0
3292: Result of user declared postfix operator not used.	0

Figure 9.4: Personality Configuration

This shows the categorised list of messages. Navigate to a message that you want to add to the current ruleset. Drag-and-drop (or copy-paste) a message from the Personality Configuration dialog into a rule displayed in ConfigGUI.

When all the changes have been made, you should upload the new configuration to the server using:

File menu : Upload to Server...

This brings up a dialog, where you can connect to the server and specify the project name to upload to - normally the same project that you downloaded from.

Once the server has been updated, any users of the project can download the new definition as described in Updating a Unified Project.

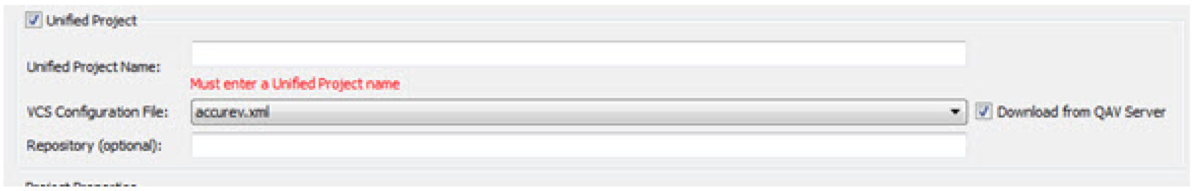
9.2.7 Creating from 'New Project'

The recommended approach for creating a new Unified project is to first create a local project, and then convert it to a Unified project, as described in Creating a Unified Project. This allows the user to test out the project definition locally, before uploading the definition to the QA·Verify server.

It is also possible to create a project as a Unified project immediately, from the New Project dialog. If this is done, the project definition is uploaded as soon as the dialog closes - and before any source code files have been added to the project. Users downloading this project definition will need to do their own project population as described in Extraction of Configuration Data. Any changes made to your local project after the New Project dialog closes (including in the Project Properties dialog that opens directly after the New Properties dialog, if you have ticked the 'Open Project Properties' check-box) will not by default be uploaded to QA·Verify. You need to upload these changes through:

QA·Verify menu : Upload Project Definition : Update

To create the project as a Unified project from within the New Project dialog, tick the Unified Project checkbox.


☑ Unified Project

Unified Project Name:

Must enter a Unified Project name

VCS Configuration File: accurev.xml ☑ Download from QAV Server

Repository (optional):

Figure 9.5: Unified Project

Additional fields then become accessible. These are as described in Creating a Unified Project.

9.3 Uploading Results to QA·Verify

Analysis results can be distributed more widely by uploading them to QA·Verify. A single set of uploaded results becomes a **snapshot** in QA·Verify, which represents the project status at a point in time. If results are uploaded regularly, then QA·Verify holds a series of snapshots, one for each upload. Together they form a record of progress over a period of time.

The value of these snapshots is greatly enhanced if the results come from known versions of the source code, with a consistent set of options used to produce those results. In practice, this means that the results should be uploaded from the automated server-side analysis run as part of a build-test-distribute workflow. This will make use of the QACLI upload functions described in Upload.

Uploading results from a GUI is most commonly used as part of a code review process. This situation is described in the Code Reviews and QA·Verify Solo Projects section.

9.3.1 Viewing the Snapshot

Once the snapshot has been uploaded, it is available for viewing within QA·Verify. The snapshot represents the status at a point in time. Each snapshot uploaded to QA·Verify is preserved, and so together they form a record of progress over a period of time. Trend reports can highlight the changes.

QA·Verify also adds value onto the current snapshot, such as metric calculations. It can show additional information on functions, such as a Function Structure diagram, and calls to and from the function. Annotations can be positioned against source code, and used as part of a code review process.

QA·Verify functionality is accessed through a browser. Refer to the QA·Verify documentation for the detail on the functionality available in QA·Verify.

9.3.2 Code Reviews and QA·Verify Solo Projects

QA·Verify Solo projects are projects that do not retain history. There is only ever one snapshot in a Solo project. The next upload will overwrite the data from the previous upload. Solo projects need to be created from the QA·Verify browser interface. Refer to the QA·Verify documentation on how to do this.

Solo projects are ideal for code reviews of code that has not yet been checked in. Uploads to Solo projects are normally done from a GUI. Snapshots uploaded from automated server-side analysis are always post-check-in, and while code reviews are often performed against a checked-in code for audit reasons, sometimes there is a need to review before the code is checked in.

Any number of files can be uploaded to a Solo project. There is no need to upload the whole code project. If the review is just to be focused on 3 or 4 files, just upload those files.

To upload, navigate to:

QA·Verify menu : Upload Results

There are three sub-menu options available to specify what to upload: the whole project; the files you currently have selected in the File Navigation pane; or you have the option to specify the files in a selection dialog.

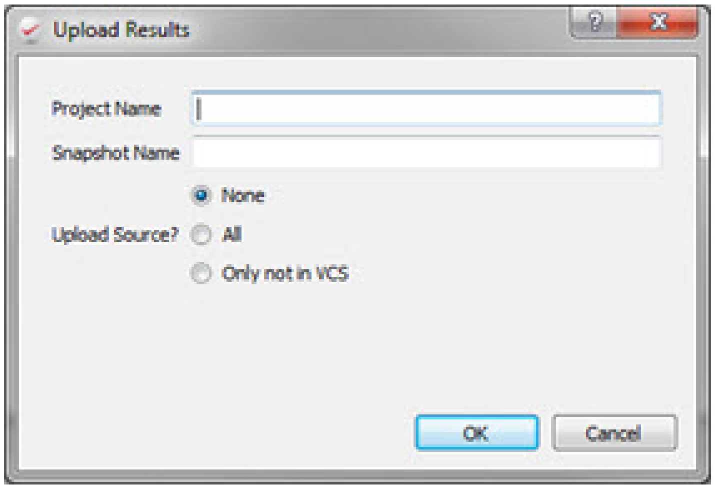


Figure 9.6: Upload Results

Within the Upload dialog, enter the name of the Solo project, and choose a snapshot name that is meaningful for this upload. Any previous snapshot will be removed from QA·Verify as part of the upload process. Care should be taken not to remove a snapshot that is still needed - create a new Solo project if necessary.

If the results you want to upload are for code not yet checked in to the version control system, select either All or 'Only not in VCS' for the 'Upload Source?' radio button.

The analysis results will be uploaded, along with the source that the results were produced against. Reviewers of the code can view source and diagnostics from a browser, from anywhere in the world. They can position comments against the code in the form of annotations, and assign actions to other users. Refer to the QA·Verify documentation on annotations, and how to conduct the code review.

9.3.3 Uploading to Standard QA·Verify Projects

While uploading from the server-side analysis workflow is the recommended approach for non-Solo projects, it is also possible to upload results from the GUI. Results can be uploaded whether or not your project is a Unified project. If it is not a Unified project, you are recommended to first create the destination QA·Verify project through the QA·Verify user interface before trying to upload.

For non-Solo projects, care should be taken to upload the full project. If only part of the project is uploaded, this affects the inheritance aspect of the snapshot sequence representing the project history. Trend reports will be affected by the gap in data, but more importantly, QA·Verify suppressions can also be affected. Suppressions against diagnostics in a file are by default inherited by the next snapshot. If that file is not uploaded as part of the snapshot, suppressions are not inherited by this snapshot nor by any subsequent snapshots.

To upload to a non-Solo project, choose the option:

QA·Verify menu : Upload Results : Upload Project

The upload dialog will then appear:

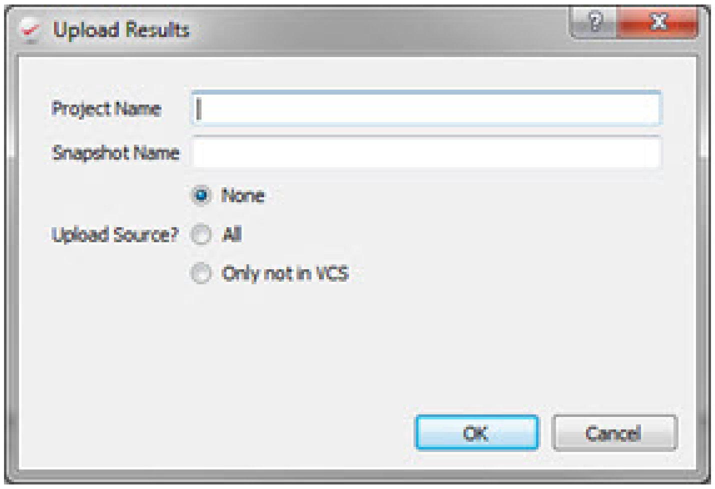


Figure 9.7: Upload Results

For a Unified project, the Project Name will default to the corresponding project in QA·Verify. You can change the project name if you wish. Normally, all you need to set is the name of the snapshot. If the source code has been checked in to the version control system, leave the 'Upload Source?' radio button value as None.

9.4 Downloading a Baseline

Baselines were introduced in the section Baseline Diagnostics Suppression. Downloading a baseline from a QA·Verify snapshot is the recommended way of baselining because each snapshot comes from a known set of source code processed with known analysis options. Baselines can only be downloaded for Unified projects.

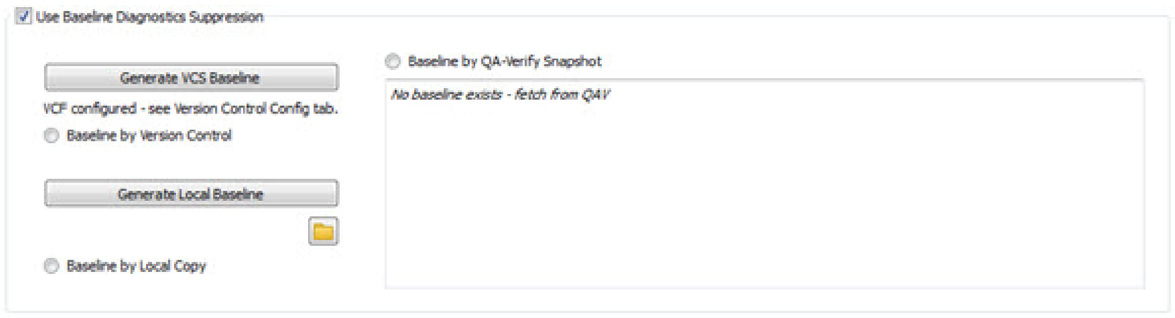
QA·Verify will hold many snapshots from throughout the project's history. Any of these can be used as the basis of the baseline. One approach is to choose a snapshot representing a release, on the grounds that the diagnostics still present at the point of release were regarded as not significant. Another approach is to use a recent snapshot, so that the only diagnostics seen on the desktop are newly-created issues in freshly- written code.

To download a baseline, navigate to:

QA·Verify menu : Download Baseline

In the dialog, choose the snapshot and click OK to start the download.

Once downloaded, you can at any time make the baseline active, from the Project Properties General tab:


☑ Use Baseline Diagnostics Suppression

VCF configured - see Version Control Config tab.

◯ Baseline by Version Control

◯ Baseline by Local Copy

◯ Baseline by QA-Verify Snapshot

No baseline exists - fetch from QAV

Figure 9.8: Using Baseline Diagnostic Suppression

Click the 'Use Baseline Diagnostics Suppression' checkbox, and the 'Baseline by QA·Verify Snapshot' radio button. Once the baseline is set, those diagnostics that appear both in the baseline snapshot and in your own analysis results will be suppressed. Section Baseline Diagnostics Suppression describes how these suppressions are displayed.

Baselines can be downloaded as often as needed. Downloading a new baseline will overwrite the old.

9.5 Downloading Suppressions

Individual diagnostics can be hidden from view through Suppressions. There are two main mechanisms available:

1. Put instructions into comments in the source code
2. Define suppressions against the results held in QA·Verify

How to do this is described in the documentation for the individual analysis tools, and the QA·Verify documentation, respectively. Source code suppressions are easy for developers to apply and use, as they are defined directly in the development environment. QA·Verify suppressions are less immediate, but have the advantage that they are more formalised, and in particular they leave an audit trail.

Suppressions defined within the source code are applied automatically. Suppressions defined within QA·Verify need to be downloaded from QA·Verify before they can be applied by PRQA Framework.

To download suppressions, navigate to:

QA·Verify menu : Download Suppressions

In the dialog, choose the snapshot and click OK to start the download. The suppressions downloaded are those that cause a diagnostic to be suppressed within the chosen snapshot. Because of the carry-forward functionality of QA·Verify suppressions, the actual definition of the suppression may come from an earlier snapshot. See the QA·Verify documentation for detail on defining suppressions in QA·Verify.

Once the suppressions have been downloaded, they will automatically be applied when you view analysis results. Even if you have edited your local source code since the snapshot was uploaded, PRQA Framework can identify where the same diagnostic has been generated in a slightly different location. The suppression in the snapshot will still be applied within your own code.

New suppressions will be created in QA·Verify regularly. To obtain the latest suppressions, just download a newer snapshot. However, if much of the code you have locally is older than the latest check-in - which can happen if you need stability in the code you are not actually working on - you should choose a snapshot whose source code is most similar to your own. This is because QA·Verify only carries forward suppressions while the diagnostic is still present. A suppression needed for your older code may no longer be present in the latest 'fixed' code.

10 Cross-Module Analysis

Cross-Module Analysis (CMA) applies only to C/C++ projects.

With C and C++, source files can be compiled independently of each other. A source file will include header files to ensure that all the necessary information for compilation is available, without needing to look at other source files. A source file together with its included header files is called a translation unit. Just like the compiler, QA·C and QA·C++ analyse one translation unit at a time. This approach is efficient because the developer can ask for an analysis of just the single file they are working on.

While single-file analysis can provide most of the checks made, some additional checks require that the files are considered together. These checks include finding duplicate or conflicting declarations; finding unused code; reporting on potential name confusion; and detection of recursion. This type of analysis is called Cross-Module Analysis. A component called RCMA can be used to carry out these checks. Just like a linker within a build, RCMA will operate on all the files within a PRQA project.

Multi-threading is another category of checks that generally needs to consider all the files in the project. A specialized module MTR is available to perform these checks. This is described in MTR Analysis.

10.1 Single-Project and Multi-Project CMA

C and C++ allow executables to be built either in a single step, or in multiple steps e.g. by adding libraries into the definition of the executable. The libraries and the executable are all separate code projects, with the build scripts set up to build them sequentially, rather than as a single unit. Often a developer will only work on a library, or only work on the executable.

PRQA projects normally correspond to code projects. If the executable is built from several libraries, there would often be a PRQA Framework project for each library, as well as a PRQA Framework project for the executable itself.

In the context of CMA, it is important to work with what will become a buildable entity, e.g. a final executable will all libraries linked. RCMA needs to see all the code that will be made into the executable, including code within the libraries that get built into the executable. This means that RCMA needs to work with several PRQA Framework projects. To allow for this, CMA projects can be created that contain many individual PRQA Framework projects. Section CMA Project Editor describes how to set up a CMA project.

Not all C and C++ executables are built on several steps. Often, there is just a single code project that contains all the code that goes into the executable. There would then just be a single PRQA Framework project corresponding to this code project, and which contains all the code files. It is possible to create a CMA project that just contains this one PRQA Framework project. However, to keep things simple, PRQA Framework provides the ability to run the CMA checks against a single PRQA Framework project, rather than a CMA project. Running RCMA against a PRQA Framework project is called single-project CMA; running RCMA against a CMA project is called multi-project CMA.

10.2 Single-Project CMA Analysis

Single-project CMA analysis is performed against a PRQA Framework project. To run it, choose:

Analyze menu: Current Project CMA Analysis

This will run RCMA against the currently-active PRQA Framework project.

Note that RCMA is not included in the PRQA Framework toolchain for the project (see Language Toolchain). That is because the toolchain operates on one translation unit at a time, whereas RCMA operates on the project as a whole.

RCMA makes use of information calculated by QA·C and QA·C++. All files must have been analysed by one of these before RCMA can do its own checks. Calling RCMA will cause QA·C or QA·C++ to be automatically run on those source files with out-of-date or non-existent results.

10.3 Multi-Project CMA Analysis

Multi-project CMA analysis is performed against a CMA Framework project. See CMA Project Editor for details of how to create a CMA project.

To run CMA analysis, choose:

Analyze menu: Run CMA Analysis...

You need to have an active PRQA Framework project to be able to use the command. It brings up a dialog, listing the available CMA projects. Choose one and click OK to start the analysis.

The results of the CMA analysis will be stored against the current project, and can be viewed alongside the normal analysis results for that project. If you open a different project, you will not be able to view the CMA results just calculated. Instead, perform a CMA analysis against the new project to see CMA results.

If for example a CMA project X contains PRQA Framework projects A and B, then:

a) Open project A, and run CMA analysis against CMA project X. The results will be visible from project A.
b) Open project B, and no CMA results will be visible. Analyse CMA project X from B, and the results will then become visible from B.
c) If the results are then cleaned from project B, this will have no effect on the results still stored against project A.

If a single PRQA Framework project belongs to several CMA projects, you will need to analyse each of those CMA projects to get the full set of results for the PRQA Framework project.

10.4 CMA Project Editor

CMA analysis operates across translation unit boundaries and checks for issues like duplicate definitions, incompatible declarations and unused variables. CMA only applies to C and C++ projects.

The CMA Project Editor is used to manage the CMA projects. From this panel the following can be processed:

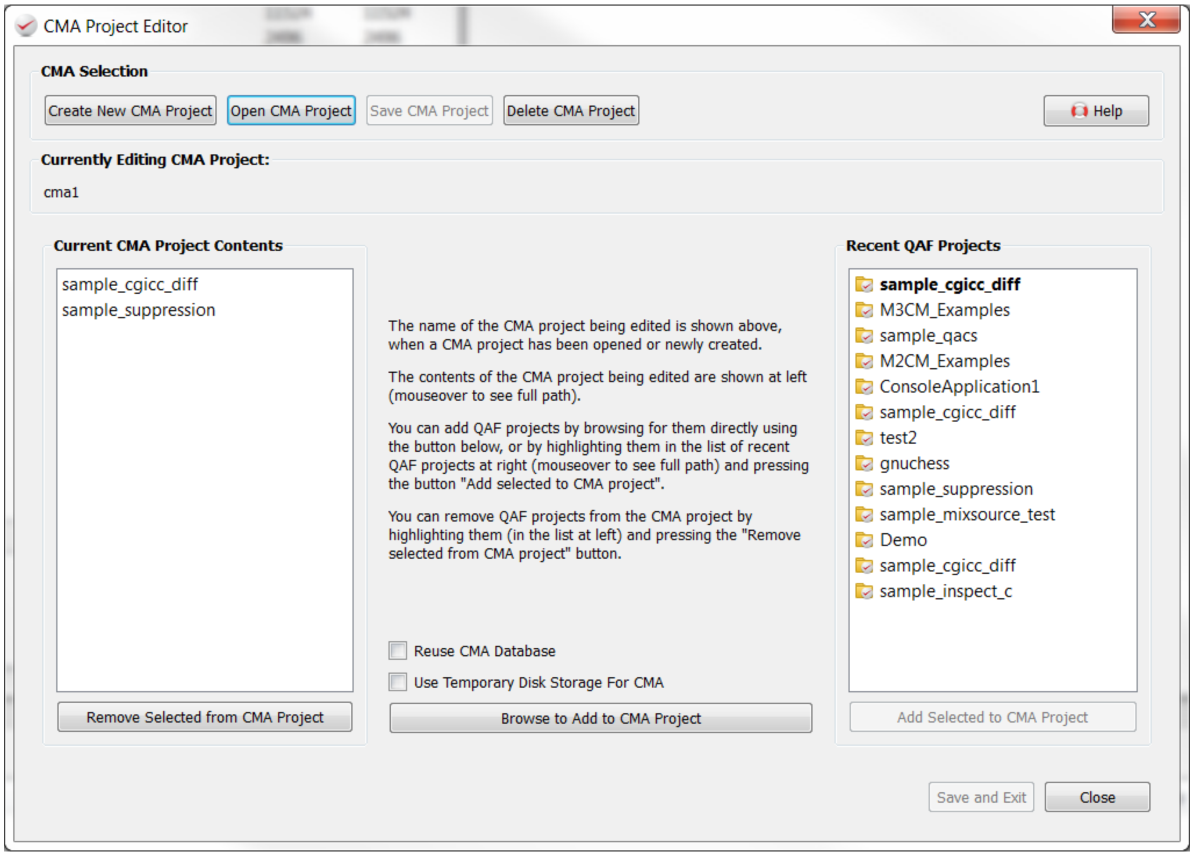
- Create New CMA Project
- Open CMA Project
- Edit the CMA Project
- Save CMA Project
- Delete CMA Project

To add or remove QA·Framework projects to a CMA project an existing CMA project must be opened or a new CMA project must be created. On clicking the 'Open CMA Project' button a panel entitled Open CMA Project will be displayed that allows the user to select existing projects, the desired CMA project can then be selected and it will be opened.

QA·Framework Projects can be added to a CMA Project by selecting from an entry in the list of Recent QAF Projects and clicking on the Add Selected to CMA Project, or by using the 'Browse to Add to CMA Project' button to search for a QA·Framework project folder.

Removing a QA·Framework project can be achieved by selecting the desired project in 'Current CMA Project Contents' list box and pressing the 'Remove Selected from CMA Project' button.

The CMA project can be saved at any point during the change process. However, it should be noted that it is necessary to save the CMA project when all changes have been completed.



The name of the CMA project being edited is shown above, when a CMA project has been opened or newly created.

The contents of the CMA project being edited are shown at left (mouseover to see full path).

You can add QAF projects by browsing for them directly using the button below, or by highlighting them in the list of recent QAF projects at right (mouseover to see full path) and pressing the button "Add selected to CMA project".

You can remove QAF projects from the CMA project by highlighting them (in the list at left) and pressing the "Remove selected from CMA project" button.

Figure 10.1: CMA Project Editor Panel

If a CMA project is not required it can be deleted using the 'Delete CMA Project' button. This will bring up a panel entitled 'Delete CMA Project' from which the CMA project that is to be deleted can be selected. Once confirmed the project details will be deleted.

10.4.1 CMA Options

There are two options which allow control of how CMA is run.

- Reuse CMA Database
- Use Temporary disk storage

Reusing the CMA database will allow for faster re-analysis. By default CMA will use a DB in memory; if you have a large project or limited memory the temporary disk storage option can be used.

These options are set through the Analysis Settings dialog, found under the Analyze menu.

10.5 MTR Analysis

MTR provides multi-threading checks for a C/C++ project. Just like RCMA, it runs against a whole project, rather than individual files. To run it, choose:

Analyze menu: MTR Analysis

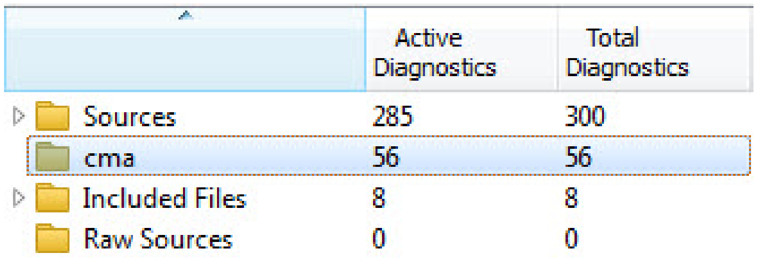
This will run MTR against the currently-active PRQA Framework project.

MTR has similarities with dataflow, and makes use of information calculated by QA·C and QA·C++ for dataflow. All files must have been analysed by QA·C or QA·C++ before MTR can do its own checks. Further, the QA·C and QA·C++ analysis must have been run with the Dataflow option turned on. Only level 1 dataflow is needed. PRQA Framework will return an error if any files have not been analysed with Dataflow set.

10.6 Viewing of CMA Data

Some diagnostics calculated by RCMA or MTR relate to multiple locations. For example, if RCMA detects a duplicate declaration, it will generate a top-level diagnostic highlighting the problem, plus sub-diagnostics that identify the locations of each duplicate. The top-level diagnostic does not have a location of its own.

Diagnostics that do not have a single location are shown under a 'cma' branch of the Files view in the Files panel:



	Active Diagnostics	Total Diagnostics
Sources	285	300
cma	56	56
Included Files	8	8
Raw Sources	0	0

Figure 10.2: Diagnostics with no single location

Clicking on the cma branch will cause the messages to be displayed in the Analysis Results/Diagnostics panel. For any given message, click on the sub-messages in the panel to see the locations in the source code.

10.7 Cleaning CMA Data

To clean the diagnostics generated from RCMA and MTR, use:

Analyze: Clean Analysis Results: Clean project

Cleaning will only affect the current PRQA Framework project.

Note that cleaning any source file will also clean RCMA and MTR results for the whole project. This is to avoid having old RCMA / MTR results left behind when the file is re-analysed.

11 Reports

11.1 Introduction

The Reports functionality inside PRQA Framework provides four PRQA Standard reports for QA·C and QA·C++ and allow users to write their own plug-in for customised reports.

11.2 Standard Report Types

RCR A Rule Compliance Report contains data on violations of rules that are specified in a PRQA Framework project's Rule Configuration File.

MDR A Metrics Data Report generates an XML file that a user can use as a source of metrics data for their own further examination.

SUR A Suppression Report provides information on messages diagnostics which have been suppressed during analysis.

CRR The Code Review Report summarizes metrics and messages from files, functions and classes. It can also display some code visualizations: includes, calls, relations and function structure. It gives a broad overview of the code.

A Rule Compliance Report is generated for all files in a project against the project's Rule Configuration File and html file have dependency to jquery library located under report directory. If a user wish to share Rule Compliance Report then he/she shall bundle jquery folder also along with html report.

Note: If jquery folder is missing from the report folder then pie charts in rule compliance report will not be shown to user.

Note that the analysis results for a PRQA Framework project will include any CMA results relating to the specified project.

Reports, when generated, can be found in the 'prqa/reports' directory of a PRQA Framework project.

11.2.1 Components of Function Structure

Function structure diagrams show the control flow structures within source code. Decisions (if, switch, and the condition part of a loop) are displayed as forks in the structure. The corresponding join, further to the right, indicates where the paths rejoin. Backward arcs, caused by loops, are shown as dotted lines.

The function structure diagrams show the following code structures:

- straight code
- if
- if else
- switch
- while loop
- for loop
- nested structures
- break in a loop
- return in a loop
- continue in a loop
- unreachable code

Each component progresses from the left to the right as the control flow would progress through the source code.

11.2.1.1 Straight Code

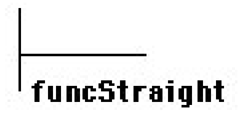


Figure 11.1: Straight Code

```
static int code = 0;
void funcStraight (void)
{
    code = 1;
}
```

There is only one path through this particular function, which is represented as lying along the x-axis.

11.2.1.2 If

```
static int code = 0;
void funcIf (void)
{
```

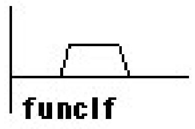


Figure 11.2: If

```
        if (code > 0)
        {
            code = 1;
        }
    }
```

This function has two paths. The first path is through the if statement and is shown by the raised line. The second path is where the if condition is false and is represented by the x-axis.

11.2.1.3 If-Else

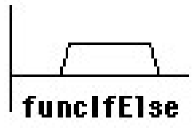


Figure 11.3: If-Else

```
    static int code = 0;
    void funcIfElse (void)
    {
        if (code > 0)
        {
            code = 3;
        }
        else
        {
            code = 4;
        }
    }
```

This function has two execution paths. The first path is the *if* sub-statement represented by the raised line. The second path is the else sub-statement represented by the x-axis.

Note that the body of this structure is longer than the body of the *if* statement. If the two arms of the if-else were not straight line code, the body of the *if* branch would appear at the left hand end of the raised line and the body of the else branch would appear to the right of the lower line.

11.2.1.4 Switch

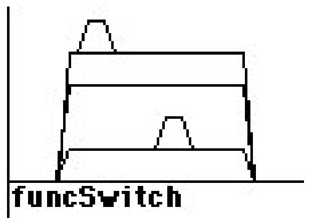


Figure 11.4: Switch

```
static int code = 0;
void funcSwitch (void)
{
    switch (code)
    {
        case 1:
            if (code == 1)
            {
                /* block of code  */
            }
            break;
        case 2:
            break;
        case 3:
            if (code == 3)
            {
                /* block of code  */
            }
            break;
        default:
            break;
    }
}
```

In the switch statement, the x-axis represents the default action, and each case statement is shown by a raised line. The two briefly raised lines represent the *if* statements within case 1 and case 3.

The diagram shows how the *if* statements are staggered. The *if* of case 1 is shown on the left and the *if* of case 3 is shown on the right.

11.2.1.5 While Loop

```
static int code = 0;
void funcWhile (void)
{
    while (code > 0)
    {
        --code;
    }
}
```

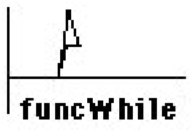


Figure 11.5: While Loop

In the *while* loop, the x-axis represents the path straight through the function as though the while loop had not been executed. The solid raised line shows the path of the while body. The dotted line shows the loop to the beginning of the while statement.

11.2.1.6 For Loop

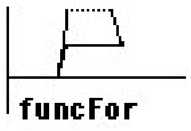


Figure 11.6: For Loop

```
static int code = 0;
void doSomethingWith(int);
void funcFor (void)
{
    for (int i = 0; i >  code; ++i)
    {
        doSomethingWith(i);
    }
}
```

The *for* loop is similar to the *while* loop as for loops can be rewritten as *while* loops. For example, funcFor in the above example could be written as:

```
void funcFor (void)
```

```
    {
        int i=0;
        while (i > code)
        {
            /* body */
            ++i;
        }
    }
```

11.2.1.7 Nested Structures

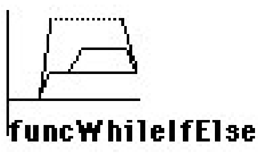


Figure 11.7: Nested Structures

```
static int code = 0;
    void funcWhileIfElse (void)
    {
        while (code > 0)
        {
            if (code == 1)
            {
                code = 0;
            }
            else
            {
                --code;
            }
        }
    }
```

This is an *if-else* contained within a *while* loop. The first solid raised line represents the *while* loop while the inner raised solid line represents the *if-else* loop. Other function structure components can be similarly nested.

11.2.1.8 Break in a Loop

```
static int code = 0;
```

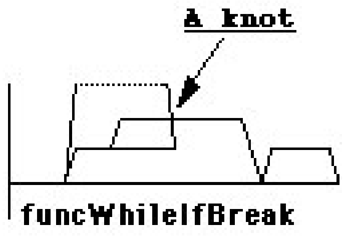


Figure 11.8: Break in a Loop

```
void funcWhileIfBreak (void)
{
    while (code > 0)
    {
        if (code == 3)
        {
            break;
        }
        --code;
    }
    if (code == 0)
    {
        code++;
    }
}
```

The break jumps to the end of the *while* statement and causes a knot. A knot is where the control flow crosses the boundary of another statement block and indicates unstructured code. In this case, break jumps to the end of the *while* statement and the next part of the program, the *if* statement, is executed.

11.2.1.9 Return in a Loop

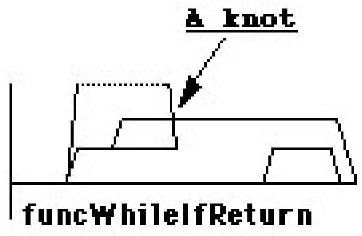


Figure 11.9: Return in a Loop

```
static int code = 0;
void funcWhileIfReturn (void)
```

```
    {
        while (code > 0)
        {
            if (code == 3)
            {
                return;
            }
            --code;
        }
        if (code == 0)
        {
            code++;
        }
    }
```

The *return* statement causes the program to jump to the end of the function. This jump breaks the control flow and causes a knot in the code.

11.2.1.10 Continue in a Loop

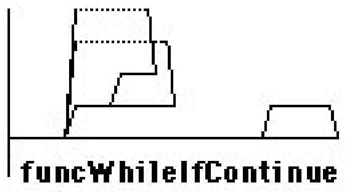


Figure 11.10: Continue in a Loop

```
static int code = 0;
    void funcWhileIfContinue (void)
    {
        while (code > 0)
        {
            if (code == 3)
            {
                continue;
            }
            --code;
        }
        if (code == 0)
        {
            code++;
        }
}
```

The *continue* statement causes the program to jump back to the beginning of the *while* loop.

11.2.1.11 Unreachable Code

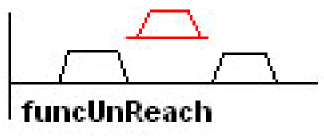


Figure 11.11: Unreachable Code

```
static void funcUnReach (int i)
{
    if (i)
    {
        i = 1;
    }
    goto Bad;

    if (i)
    {
        i = 2;
    }

    Bad:
    if (i)
    {
        i = 3;
    }
    return;
}
```

The raised red section represents code that is unreachable.

The structure and approximate position of the unreachable code is shown. It occurs after the first *if* condition and before the last *if* condition. Its position above the main structure shows that the control flow misses the middle *if* condition.

11.3 Customized Report Plug-in

The report plug-in feature allows users to write plug-ins in python scripting language and generate customized reports by parsing results_data.xml file located under the report directory of PRQA Project. Report plug-in scripts shall follow the pre-defined conversions and script file(s) shall be stored in the "report_plugins" folder located under QA·Framework installation folder. The report names are derived from the plug-in script file name, so ensure that that script files are following proper naming convention and acronyms for reports are not overlapping with other report types.

Report plug-in script shall always take result-data, output-file and rcf-file as input option and write data into the specified output file.

Example: Customized Report plug-in script.

```
def write_report_file(options):
    f = open(options.output_file, 'w')
    message = """<html>
    <head>Customized_Report_Sample</head>
    <body><p>Customized Report Plug-in Example!</p></body>
    </html>"""

    try:
        f.write(message)
    finally:
        f.close()


if __name__ == "__main__":
    usage ="""
    Given a results data xml file, this script generates a Sample Report.
    Type Customized_Report_Sample.py --help for more information.
    """

    # set up proper command line parsing
    parser = OptionParser(usage=usage, version="%prog 0.1")
    parser.add_option("-d", "--results-data", dest="results_data",
                      help="An xml file containing results data for a PRQA Framework
                            project.", metavar="FILE")
    parser.add_option("-o", "--output-file", dest="output_file",
                      help="Specify the output file path for the generated
                            document.", metavar="PATH")
    parser.add_option("-r", "--rcf-file", dest="rcf_file",
                        help="Specify the Rule Configuration File for the
                              project.", metavar="FILE")

    # parse the command line
    (options, args) = parser.parse_args()
    enc = locale.getpreferredencoding()
    #---------------------------------------------------------------------------
    #Add conditional check to make sure that all options are passed if you want to
    #run the script on command line and not as plug-in refer other report plug-ins
    #---------------------------------------------------------------------------

    options.results_data = options.results_data.decode(enc)

    options.output_file = options.output_file.decode(enc)
    options.rcf_file = options.rcf_file.decode(enc)

    # touch the output file to make sure that the user has write permission.
    try:
        f = open(options.output_file, 'a')
        f.close()
        os.utime(options.output_file, None)
    except:
        print "could not open output file for writing. exiting"
        sys.exit(1)

    write_report_file(options)
```

Note: The above script will write "Customized Report Plug-in Example!" in to the generated html report.

Result data for analyzed file(s) is/are stored in results-data.xml, you can write python functions to parse results-data.xml file along with .json file and create your own HTML report.

11.4 Report Name

The reports are Python scripts located in

`\report_plugins`

Any file in here will be used as a report, so they can be written in any language - though the existing ones are python. Underscores in the file name are replaced with spaces in the GUI and the ’first’ letter of each word in the file name can be used as the abbreviation on the command line. Hence

```
Jasons_Report.py
```

will be presented as

```
Jasons Report.py
```

in the GUI and can be generated on the command line with

```
qacli report -P . -t JR
```

Note that if there are duplicated names, on the command line it will use only one of them. In the GUI the extensions are stripped, so

```
Jasons_Report.py
```

and

```
Jasons_Report.py~
```

are both shown exactly the same.

12 QA·CLI

12.1 Introduction

The QA·CLI tool is the command line interface to PRQA Framework.

The tool is designed to be used for integrating PRQA Framework with builds on build servers but can also be used for desktop use.

It features a command line interface of the form:

```
qacli <subcommand> [options...]
```

where related features are grouped under a subcommand, not unlike most version control system interfaces. Each subcommand and its options conform to the GNU/posix command line interface standard, chosen because of of its widespread adoption and maturity for such use cases.

Features are grouped under six subcommands which can be viewed by issuing the '*qacli -h*' command and are as follows:

admin	Manage PRQA Framework and CMA projects and the PRQA Framework installation.
analyze	Run analysis on existing projects.
help	Provide help and information.
report	Generate analysis reports.
upload	Send results to external systems.
view	View analysis results.

Table 12.1: Available Subcommands

Verbose usage information on the options for each subcommand can be accessed by issuing the following command:

```
qacli <subcommand> --help
```

There is one option that appears across all subcommands, except 'help'. This option is '- -qaf-project' and its shortcut is '-P'. Its argument is the path to the directory of the PRQA Framework project you wish to work with. For all subcommands except 'admin', this option defaults to the current working directory.

12.2 Admin

The QA·CLI *admin* subcommand provides features for creating and managing projects and for managing the PRQA Framework installation as a whole.

Verbose help on the command can be accessed as follows:

```
qacli admin -h
```

12.2.1 PRQA Framework Installation Management

All of the following settings are per-user settings (i.e. they will not apply to a different user on the same machine).

To set the address of a PRQA license server from where the PRQA Framework system should look to retrieve a license:

```
qacli admin --set-license-server <port@hostname|port@ip>
```

where the argument refers to an application layer address, for example, *5055@192.168.1.10* or *5055@myhost*.

PRQA Framework is packaged with sample projects and sample configuration as well as compatibility configuration for various compilers. Upon starting PRQA Framework for the first time, a copy of this data is made in what's called the user data area. The location of this area is dependent on your desktop environment:

- Windows: %LOCALAPPDATA%\PRQA\prqa-framework-<version>\
- FreeDesktop.org: $XDG_CONFIG_HOME/PRQA/prqa-framework-<version>/
- Other: $HOME/.config/PRQA/prqa-framework-<version>

Due to the nature of this data, and the fact that it can be modified and extended by the user, the data is not deleted during uninstallation. If you wish to remove it, or you wish to have PRQA Framework re-create it the next time it runs, issue the following command:

```
qacli admin --remove-user-data
```

This will delete all contents on the user data area, so be careful that you do not inadvertently delete something that you wish to retain.

The PRQA Framework system has the ability to write log files during processing. The size of the log files, and the information that they contain, are dependent on the debug level that is set for the current user. Logs are generated in a logs directory within the PRQA Framework user data area. By default, the debug level is set to ERROR. The following command can be used to change this:

```
qacli admin --debug-level <NONE|ERROR|INFO|DEBUG|TRACE>
```

The arguments are displayed in the order of the amount of data that is written to log files, with NONE specifying not to write logs and TRACE specifying to write everything that is possible to log. Large logs that are written frequently can take up a lot of disk space very quickly. Hence, only use logging if necessary or if requested by a PRQA support engineer.

Some PRQA Framework user interfaces (i.e. QA·GUI, QA·Eclipse, QA·Visual Studio) can be presented in different languages. The following setting tells PRQA Framework to present user interfaces in a specified language:

```
qacli admin --set-language <JP|EN>
```

If a particular user interface does not have a translation for the specified language, it will be presented in English.

PRQA Framework projects maintain configuration data that is grouped into various configuration files. This is so that the same configuration file can be used when creating many projects, which may require similar configuration for certain features. Each configuration file type has a directory in the user data area where they are maintained, and this is where the PRQA Framework graphical user interfaces look for files to present as choices when creating/modifying a PRQA Framework project. The following command is used to create a new configuration file of a specific type from a template:

```
qacli admin --create-config-file <acf|rcf|vcf>
```

The configuration file types that can be created correspond to:

acf Analysis Configuration File

rcf Rule Configuration File

vcf Version Control Compatibility File

12.2.2 PRQA Framework Project Management

A PRQA Framework project is identified by the directory in which it resides and does not have a specific name that identifies it, other than that directory path. A PRQA Framework project contains configuration files which contain related settings. Creating a new PRQA Framework project is a case of specifying the directory where the project should be created and specifying the configuration files that it should contain. A PRQA Framework project consists of:

- at least one Compiler Compatibility Template (CCT)
- only one Rule Configuration File (RCF)
- only one Analysis Configuration File (ACF)
- zero or one Version Control Compatibility File (VCF)

The following command creates a new A PRQA Framework project:

```
qacli admin --qaf-project-config --qaf-project <directory> --cct <path>
--rcf <path> --acf <path>
```

For the creation of a new project the following constraints apply:

- a PRQA Framework project must not exist at the specified directory location.
- the specified directory need not exist but its parent directory must exist.
- the '- -rcf' option must be specified and its argument must be a path to a valid Rule Configuration File.
- the '- -acf' option must be specified and its argument must be a path to a valid Analysis Configuration File.
- the '- -cct' option must be specified at least once in the command and its argument must be a path to a valid Compiler Compatibility Template.

The '- -cct' argument may be specified more than once in order to add support to a project for more than one compiler (e.g. a C compiler and a C++ compiler). Other optional configuration files may be specified also:

- the '- -vcf' option can be used to specify a path to a valid Version Control Compatibility File (for retrieving version control information that is used for baselining and performing uploads to a QA·Verify server).

The same command can be used on an existing PRQA Framework project, but does not have a constraint on the configuration files that must be specified. This has the effect of:

- if a RCF is specified, replacing the RCF in the project with the one specified
- if an ACF is specified, replacing the ACF in the project with the one specified
- if a VCF is specified, replacing the VCF in the project, if one exists, else adding the specified VCF
- if a CCT is specified, adding the specified CCT to the project. It will not remove any CCTs currently in the project. A project may have multiple CCTs, but only one may be *active* for each source language at any one time. Currently, changing the *active* status of CCTs must be done through the QA·Project Properties interface (command: qapp <project-path>)

Internally, PRQA Framework maintains paths as relative to source code roots, as described in Set the Root Directory of Your Sources. The default source code root is SOURCE_ROOT. This location can be set by the following command:

```
qacli admin -P <project-path> --set-source-code-root <directory>
```

To list the configuration files that are used within a project, issue the following command:

```
qacli admin -P <project-path> --list-config-files
```

Sometimes it is necessary to quickly edit a configuration file within a PRQA Framework project. To do so, the project and configuration file type must be specified. Issue the following command:

```
qacli admin -P <project-path> --edit-config-file <acf|rcf|cct-c|cct-cpp>
```

where the following is a description of the various possible arguments:

acf the project's Analysis Configuration File

rcf the project's Rule Configuration File

cct-c the project's 'active' Compiler Compatibility Template for the C language

cct-cpp the project's 'active' Compiler Compatibility Template for the C++ language

If you wish to delete a PRQA Framework (i.e. delete only PRQA Framework related data from a directory), issue the following command:

```
qacli admin -P <project-path> --remove-project-files
```

Normally when a PRQA Framework interface is operating with a specific project, it has a lock on the project, preventing other PRQA Framework processes from acting on it simultaneously. If the same process ends abruptly, it may leave the lock in place, prevent any PRQA Framework process from working with it. If you find that you are locked out of a project, but are certain that there are no other PRQA Framework process running, the following command will remove the locks that are on the project.

```
qacli admin -P <project-path> --remove-locks
```

12.2.3 CMA Project Management

A CMA project is identified by a name. When a CMA project is first created, it is empty, with no PRQA Framework projects (modules) associated with it. The following command creates a new CMA project, which can be subsequently used to associate modules with it and conduct analyses:

```
qacli admin --create-cma-project <NAME>
```

This will create a new empty CMA project that can be referred to with the name specified. The following command deletes an existing CMA project:

```
qacli admin --delete-cma-project <NAME>
```

Note that no change to any modules associated with the specified CMA project name will occur as a result of this call. It simply removes a name that is used to refer to a set of modules for the purpose of CMA.

The following command can be used to associate a module (PRQA Framework project) with a CMA project:

```
qacli admin -P <project-path> -M <cma-project-name> -i
```

In order to dissociate a module from a CMA project, the following command can be used:

```
qacli admin -P <project-path> -M <cma-project-name> -x
```

12.2.4 Extraction of C/C++ Analysis Data

After creating a project, the next thing to be done is to populate it with the source files and analysis data (i.e. the include paths and macro definitions used to compile each source file) from your build. This data needs to be extracted in some way from your build system. There are a number of ways to achieve this without having to resort to manual atomic datum addition.

The easiest, and recommended, way is to use build process monitoring. PRQA Framework will monitor your build process, extract the relevant data and add it to a specified project. The following command can be used to accomplish this:

```
qacli admin -P <project-path> --build-command <command>
```

where *project-path* is a path to a valid PRQA Framework project and *build-command* is a command that will set your build in motion and compile your files. If your build command contains more that one word, wrap it in quotes. If your build command is complex (e.g. has quotes within it), place the command in a script and make it executable and pass the path of this script as the argument. The above command can be reused on any existing project and will replace the old data with the data gathered from the build. This is useful for keeping PRQA Framework projects synchronized with you build projects.

An alternative way to extract data from your build and add it to a PRQA Framework project is to use the compiler-wrapping mode. This works only with build systems that can alias the compilation command. An example is a Make-based system, where the CC or CXX variables can be set. An example of a command for a Make build is as follows:

```
make CC=’’qacli admin -P <project-path> -compile-command ’gcc -c’
--add-file --ignore_rest’’
```

In this situation, instead of your normal compiler receiving the arguments in the build, qacli will receive them directly, where they will be placed after the ’- -ignore_rest’ switch. qacli will parse the arguments, search for a source file path, include paths and macro definitions and, if a source file is found, it will be added to the project along with the other data. As soon as this is done, qacli will use the argument to the ’- -compile-command’ option and issue it to the system, compiling your files as normal. If the ’- -compile-command’ option is omitted, qacli will not issue any compilation command after extracting data from arguments.

Another alternative is to add data from your build with a build log. If your build system is capable of producing a build log that abides by the following constraints:

- the build log is newline-separated for each source file that was compiled
- lines that contain a path to a source file also contain the include paths and macro definitions needed to compile that file.

then the resultant log file can be used to add the data to a PRQA Framework project using the following command:

```
qacli admin -P <project-path> --add-files <build-log-path>
```

If your build system does not support the creation of a build log in this format, for most IDEs and build systems, a log can be created using the following mechanism.

Create a script that wraps your compiler, for example (Bash variant):

```
#!/bin/bash
echo ''$@'' >> build.log
exec /usr/bin/gcc ''$@''
```

In this example, the script must be named 'gcc', made executable and placed on the PATH before *gcc*. During a build process, the build system will call the wrapper script instead of *gcc* and a build log will be dropped in each directory that the wrapper is executed from. These build logs will be suitable for using with the above qacli command. Just ensure to use an absolute path for the *project-path* argument and issue the command from the same directory as the log file. The same method can be used for the Windows platform, using a Windows version of the above script example.

Finally, if all else fails, files can be added explicitly.

12.2.5 Adding Files Explicitly

Files can be added to a project explicitly using the following command:

```
qacli admin -P <project-path> --add-file --ignore_rest <arguments>
```

where arguments represent the data to be added. For example, for a C project:

```
qacli admin -P <project-path> --add-file -- -Isrc/ -Ithird/party/lib/include -DMACRO=1 src/
```

For C#:

```
qacli admin -P <project-path> --add-file -- src/file.cs
```

Note that '-P' is a shortcut for '- -qaf-project' and '- -' is a shortcut for '- -ignore_rest'.

12.2.6 Specifying QA·Verify Credentials

PRQA Framework is able to upload data to QA·Verify and download data from QA·Verify using qacli commands, as described below. All of the QA·Verify commands involve authenticating to QA·Verify. There are two ways to do this: add the credentials into each command; or use a token. A command such as listing the Unified projects has the following two forms:

```
qacli admin --list-unify-project --url <url> --username <uname> --password <pwd>
```

or

```
qacli admin --list-unify-project --url <url> --token <tkn>
```

For an automated set of scripts, the advantage of the token is that it avoids having a password listed in each command, possibly scattered across many scripts. A token is created in a single place, using the following command, where the url parameter is the address of the QA·Verify server:

```
qacli admin --get-token --expire-token <days> --url <url>
--username <uname> --password <pwd>
```

This returns a string, which can be used in subsequent commands, as shown in the list-unified-project example just above. Tokens are time-limited, with a default duration of 1 day. Use the –expire-token parameter to specify the number of days before expiry.

For brevity, all of the examples in the following sections will use the token mechanism, but in each case it is also possible to use the full set of url/username/password options.

12.2.7 Working with Unified Projects

The section Working with QA·Verify describes how project definitions can be shared amongst other users by storing them on QA·Verify as Unified projects. This can all be done through command line options.

To convert your local project into a Unified project, use the convert-to-unify command. This will upload the project definition to QA·Verify, and is equivalent to using the dialog described in the section 'Creating a Unified Project':

```
qacli admin -P  <project_path> --convert-to-unify
--project-name <unifyname>   --server-vcf <file>
--use-local-vcs --repository-path <path > --url
<url> --token <tkn>
```

where:

P(or - -qaf-project) gives the location of the PRQA Framework project.

project-name will be the name given to the project on QA·Verify. Once the project has been uploaded, the local PRQA Framework project will be adjusted to use this name.

server-vcf gives the name of the version control system file to be used. The list of files can be found by running the command:

```
qacli admin --list-server-vcf --url <url> --token <tkn>
```

use-local-vcs specifies that the VCS file should not be downloaded onto the local machine. If not set, the file specified for the server-vcf option will be downloaded, and used as the local VCS file.

repository-path is any additional information needed by the version control system holding the source code. This is an optional parameter.

url is the address of the QA·Verify server.

token is the string value returned from the get-token function. Alternatively, –username and –password parameters can be used.

Once the project has been uploaded to QA·Verify, it is available for other users to download. Users can see what Unified projects are present and downloadable by the following command:

```
qacli admin --list-unify-project --url <url> --token <tkn>
```

The actual download is performed by the following command:

```
qacli admin -P  <project_path>  --pull-unify-project --project-name
<unifyname>  --url <url> --token <tkn>
```

If the definition of the Unified project needs to be updated, the new definition can be uploaded to QA·Verify through the command:

```
qacli admin -P  <project_path>  --push-unify-project --url <url> --token <tkn>
```

12.2.8 Upgrade

Each version of PRQA Framework that is released is, more often than not, released with new and/or updated analysis components. Projects that have been created by older versions of PRQA Framework will have a Rule Configuration Files and Analysis Configuration Files that refer to the versions of analysis components that are not shipped with the new PRQA Framework version. If you wish to convert your existing projects to refer to the new versions of components instead of the old versions, the following command can be issued:

```
qacli admin --qaf-project <project-path> --upgrade
```

This will only work if PRQA Framework possesses an upgrade map for the specified project version. This command will not add any new features or components (e.g. new messages or analysis components) to your configuration files. It will replace corresponding features from the older versions with the newer versions of those features. It will also remove any features or components that no longer exist in the newer versions.

If instead you wish to continue using older versions of components, do not use the -upgrade command. You will need to tell PRQA Framework where to find the old components through the following command:

```
qacli admin --add-component-search-path <path>
```

This will cause the older component to appear in the toolchain editing screen (the Analysis tab of the Project Properties dialog). Use the arrow button to add the older component into the active toolchain.

You should not mix old and new versions of components. If you have used the –add-component-search-path option, set the older component as active for all the projects run on that machine.

12.3 Analyze

The QA·CLI 'analyze' subcommand allows the user to perform analysis on existing PRQA Framework and CMA projects.

Verbose help on the command can be accessed as follows:

```
qacli analyze -h
```

12.3.1 Analysis of PRQA Framework Projects

The following command can be issued to perform analysis on a PRQA Framework project:

```
qacli analyze -P <project-path> --file-based-analysis
```

The '-P' option can be omitted if your current working directory is the same directory as the project. This applies to all subcommands except the 'admin' subcommand. The shortcut for the '- -file-based-analysis' switch is '-f'.

The following command can be issued to clean a PRQA Framework project:

```
qacli analyze -P <project-path> --clean
```

The shortcut for this switch is '-c'.

The following switch can be used to stop analysis upon the first analysis failure:

```
qacli analyze -P <project-path> -f --stop-on-fail
```

The shortcut for this switch is '-s'.

For C and C++, pre-processed source is useful for debugging configurations and code in general. PRQA Framework will generate pre-processed source files in the ’prqa/output’ directory structure of a project if it is specified. The following switch can be used to generate a pre-processed source file for each file that is analyzed:

```
qacli analyze -P <project-path> -f --generate-preprocessed-source
```

The shortcut for this switch is ’-g’.

When analysis configuration problems result in source files failing to analyze successfully, it is useful to gather information that can be relayed to a Programming Research support engineer to expedite a response. The following switch specifies that PRQA Framework is to create an archive for each file that fails analysis, containing the pre-processed file, the met file and the settings (.via) file:

```
qacli analyze -P <project-path> -f --assemble-support-analytics
```

The shortcut for this switch is ’-a’. The archive will be created in the ’prqa/output’ directory structure of the PRQA Framework project.

Each CMA project maintains a database containing analysis data of all the PRQA Framework projects (modules) that are associated with it. The population of this data can occur a various stages. However, it is sometimes beneficial to conduct population during analysis, thereby cutting down the time needed to perform a CMA. The following switch can be used to populate the databases of all of the CMA projects that the specified PRQA Framework project is associated with:

```
qacli analyze -P <project-path> -f --populate
```

The shortcut for this switch is ’-p’.

All previous switches can be chained:

```
qacli analyze -P <project-path> -cfsgap
```

will clean the project, conduct file-based analysis until the first failure is encountered, generate pre-processed source for each file that is analyzed, create an archive with support data for any file that fails to analyze successfully and populate the databases for any CMA projects that the specified PRQA Framework project is associated with.

When running whole program dataflow analysis (refer to the QA·C or QA·C++ manual), it can be beneficial to run analysis more than once. Each subsequent analysis run will result in more information being stored for project functions. This should result in a reduction of the number of Possible issues and a potential increase in the number of Definite, Apparent and Suspicious issues found.

To run more than once, use the repeat option:

```
qacli analyze -P <project-path> -f --repeat <n>
```

where n is the number of times analysis should be run.

To specify a specific file to be analyzed, the path to the source file can be provided:

```
qacli analyze -P <project-path> -f <source-file-path>
```

This cannot be used with CMA.

To specify a set of files to be analyzed, the path to each source file must be placed in a text file, one path per line - a filelist. To analyze those files:

```
qacli analyze -P <project-path> -f --files <path-to-filelist>
```

The shortcut for this option is '-F'. This option cannot be used with CMA.

12.3.2 Single-Project CMA Analysis

Cross-module analysis can be done on a single PRQA Framework project, without the need for a CMA project to be set up. See the section on Cross-Module Analysis for more detail.

Single-project CMA analysis can be invoked by:

```
qacli analyze -P <project-path> --project-based-analysis
```

If the individual source files have out-of-date or non-existent results, these will be analysed before the CMA part of the analysis is done.

To ensure that all files are individually analysed prior to the CMA analysis, use the clean option:

```
qacli analyze -P <project-path> --project-based-analysis --clean
```

12.3.3 Analysis of CMA Projects

As outlined in sub-section Admin, PRQA Framework projects can be associated with CMA projects as modules. Performing analysis on a CMA project implies conducting file-based analysis on each of the modules with the '- -populate' switch turned on and then conducting the CMA on the resulting database. As a useful side-effect, this feature can be used to conduct all analyses on a number of PRQA Framework projects. To conduct CMA, specify the name of the CMA project and the name of the PRQA Framework project against which the results are to be stored:

```
qacli analyze -P <project-path> -C <cma-project-name>
```

If no PRQA Framework project is specified, the default is to use the first in the list of PRQA Framework projects in the CMA project.

If file-based analysis of the associated modules has previously been conducted with the '- -populate' switch turned on, the performance of CMA is dramatically increased. All of the normal switches apply for CMA (except '- -file-based-analysis' and '- -populate', which are implicit):

```
qacli analyze -P <project-path> -C <cma-project-name> -csga
```

will clean the associated modules first before performing file-based analysis on each, halting if an analysis failure is encountered, generating pre-processed source for each source file that is analyzed and creating an archive containing support data for any file that fails analysis.

12.3.4 MTR Analysis

MTR analysis on a project can be invoked by:

```
qacli analyze -P <project-path> --mtr-analysis
```

You should ensure that all the source files in the project have previously been analysed with the dataflow option set. The dataflow options are held in the ACF file, and are normally set through QAGUI.

12.3.5 Raw Source Analysis

Sometimes it is beneficial to analyze a file that is not part of a PRQA Framework project, for example, to analyze a pre-processed file to debug configuration or to analyze a source file directly, without any configuration (i.e. include paths and macro definitions). The following command can be used to do so:

```
qacli analyze --raw-source <file-path>
```

Also, it is sometimes beneficial to supply only compiler-specific configuration for raw source analysis. The following command can used to accomplish this:

```
qacli analyze --raw-source <file-path> -language-cct <cct-path>
```

12.4 Baseline

The general functionality of baselines is described in the section Baseline Diagnostics Suppression.

12.4.1 Setting a QA·Verify Baseline

This involves downloading code and diagnostics from a QA·Verify snapshot to form the actual baseline. Once this has been done, you need to tell PRQA Framework to apply the baseline. These baselines are only available for Unified projects, which have linkage to a project within QA·Verify.

The commands that get data from QA·Verify need to connect to QA·Verify using the URL for the QA·Verify server, as well as logon credentials. As described in subsubsec:specify_qav_credentials, the logon credentials can either involve specifying username and password, or the use of a token. The example calls listed here use the token method.

The available snapshots for the Unified project can be listed through the command:

```
qacli baseline -P <project_path> --list-snapshots --url <url> --token<tkn>
```

To download the baseline, use:

```
qacli baseline -P <project_path> --pull-baseline <snapshot> --url <url> --token<tkn>
```

Once the baseline has been downloaded, it can be applied through the command:

```
qacli baseline -P <project_path> --set-baseline --baseline-type UNIFIED
```

To turn the baseline off, use:

```
qacli baseline -P <project_path> --set-baseline --baseline-type OFF
```

12.4.2 Setting a Version Control Baseline

This involves generating the baseline from the current analysis results, and then applying the baseline. To generate:

```
    qacli baseline -P <project_path> --generate-baseline --baseline-type VCS
```

To apply the baseline, use:

```
    qacli baseline -P <project_path> --set-baseline --baseline-type VCS
```

To turn the baseline off, use:

```
    qacli baseline -P <project_path> --set-baseline --baseline-type OFF
```

12.4.3 Setting Local Baseline

This involves generating the baseline from the current analysis results, and storing the source code from which the baseline diagnostics were generated. Once this has been done, you need to tell PRQA Framework to apply the baseline.

To generate:

```
    qacli baseline -P <project_path> --generate-baseline
```

The source code associated with the baseline must be stored, so that when the baseline is actually applied diff-style techniques can be used to match baselined diagnostics with the diagnostics in the latest code. To store:

```
qacli baseline -P <project_path> --local-source <path> --baseline-type LOCAL
```

The path must already exist.

To apply the baseline, use:

```
qacli baseline -P <project_path> --set-baseline --baseline-type LOCAL
```

To turn the baseline off, use:

```
qacli baseline -P <project_path> --set-baseline --baseline-type OFF
```

12.4.4 Downloading QA·Verify Suppressions

The general functionality of QA·Verify suppressions is described in the section Downloading Suppressions. QA·Verify suppressions are only available for Unified projects, which have linkage to a project within QA·Verify.

The suppressions are downloaded from a QA·Verify snapshot. The available snapshots for the Unified project can be listed through the command:

```
qacli baseline -P <project_path> --list-snapshots --url <url> --token<tkn>
```

To download the suppressions, use:

```
qacli baseline -P <project_path> --pull-suppressions --url <url> --token<tkn>
```

Once the suppressions have been downloaded, they will automatically be applied when the analysis results are viewed.

12.5 Help

The QA·CLI 'help' subcommand provides information on qacli return codes and PRQA Framework compiler support.

To access a listing of supported compilers:

```
qacli help --compatibility
```

The shortcut for this switch is '-c'.

To access descriptions of qacli return codes:

```
qacli help --return-codes
```

The shortcut for this switch is '-r'

12.6 Report

The QA·CLI 'report' subcommand allows the user to generate analysis reports based on the analysis results of a PRQA Framework project. Note that the analysis results for a PRQA Framework project will include any CMA results relating to the specified project.

Currently, there are four report types that can be generated:

RCR A Rule Compliance Report contains data on violations of rules that are specified in a PRQA Framework project's Rule Configuration File.

MDR A Metrics Data Report generates an XML file that a user can use as a source of metrics data for their own further examination.

SUR A Suppression Report provides information on messages diagnostics which have been suppressed during analysis.

CRR The Code Review Report summarizes metrics and messages from files, functions and classes. It can also display some code visualizations: includes, calls, relations and function structure. It gives a broad overview of the code.

Reports, when generated, can be found in the 'prqa/reports' directory of a PRQA Framework project.

To generate a report, issue the following command, where `acronym` is one of the report types (i.e. RCR, MDR, SUR or CRR):

```
qacli report -P <project-path> --type <acronym>
```

The shortcut for the '- -type' option is '-t'.

To specify a report to be generated for a specific set of source files within a PRQA Framework project, the path to each source file must be placed in a text file, one path per line - a filelist. To generate the report:

```
qacli report -P <project-path> -t <acronym>  --files <path-to-filelist>
```

The shortcut for this option is '-F'. This option will be ignored if '- -type' is set to 'RCR'.

12.7 Upload

The QA·CLI 'upload' subcommand allows the user to upload analysis results from a PRQA Framework project to an external application. Note that the analysis results for a PRQA Framework project will include any CMA results relating to the specified project.

At present, supported external applications are Structure101 and QA·Verify.

12.7.1 Upload to QA·Verify

The following command can be used to upload the analysis results of a PRQA Framework project to a QA·Verify project.

```
qacli upload -P <project-path> --qav-upload --upload-project
<qav_name> --snapshot-name < snapshot> --upload-source <flag>
--url <url> --username <uname> --password <pwd>
```

where

qav-upload flags that this upload is going to QA·Verify. The shortcut for this option is '-q'.

upload-project is the name of the QA·Verify project that should receive the results. The QA·Verify project may or may not be a Unified project. If the project does not exist, it will be created.

snapshot-name is the name that will be given to the QA·Verify snapshot created to hold the uploaded results.

upload-source can be set to one of: NONE, ALL, NOT_IN_VCS. This determines which code files get uploaded to QA·Verify.

url is the address of the QA·Verify server.

username is the QA·Verify identifier for the uploading user. It is not possible to use a token with the upload command, username and password must be used instead.

password is the user's QA·Verify password.

Note that PRQA Framework no longer uses a QCF file to specify connection and QA·Verify details.

The general functionality of the upload command is described in the section Uploading Results to QA·Verify.

12.7.2 Upload to Structure101

The following command can be used to upload the analysis results of a PRQA Framework project to a Structure101 project.

```
qacli upload -P <project-path> --s101-upload --upload-location <directory>
```

The shortcut for the '- -s101-upload' option is '-s' and the shortcut for the '- -upload-location' option is '-u'. Also note that there are constraints on this feature:

- A Structure101 license must be available from the license server.
- The project must be a C or C++ project.

- The project must have been previously analyzed and analysis results must be available.
- The argument to the '- -upload-location' option must specify a directory path that exists.

12.7.3 Upload of Selected Files

To specify upload for a specific source file within a PRQA Framework project, the path to the source file can be provided:

```
qacli upload -P <project-path> -q <path-to-file>
```

To specify upload for a specific set of source files within a PRQA Framework project, the path to each source file must be placed in a text file, one path per line - a filelist. To upload:

```
qacli upload -P <project-path> -q --files <path-to-filelist>
```

The shortcut for this option is '-F'.

The file options are available with both the –qav-upload and –s101-upload switches.

12.8 View

The QA·CLI 'view' subcommand allows the user to view the diagnostics from the analysis results of a PRQA Framework project. The 'view' subcommand applies only to C/C++ projects. Note that the analysis results for a PRQA Framework project will not include any CMA results relating to the specified project.

Note that currently all features in this subcommand are, by default, not available to most users. PRQA Framework provides three graphical user interfaces for the consumption of analysis results - QA·GUI, QA·Eclipse and QA·Visual Studio. Results can also be consumed from a QA Verify Server, if they have been uploaded to it. There are four modes of output:

STDOUT output results to standard output (restricted)

STDERR output results to standard error (restricted)

HTML for each source file, output results to a HTML file (restricted)

For those that possess a completely unrestricted license, it is possible to view results on the command line or output them to a HTML file, but note that there are technical limitations to what is presented in this way. First, CMA results are not available from the command line. Second, diagnostics that are suppressed by a baseline will be presented as if they were not suppressed. These caveats are due to the technical considerations of a non-reentrant command line tool, memory and the performance of a tool primarily designed for build server usage.

Issue the following command to output the results for a PRQA Framework project to standard output:

```
qacli view -P <project-path> --medium STDOUT
```

The shortcut for this option is ’-m’. To output to standard error, replace STDOUT with STDERR in the above command.

Data sets can be quite large and difficult to navigate on the command line, hence there are a series of switches that hide or show various subsets of data:

To show rule information with each primary diagnostic (not sub-diagnostics):

```
qacli view -P <project-path> -m STDOUT --rules
```

The shortcut for this switch is ’-r’.

To present diagnostics that are suppressed, along with all other diagnostics (i.e all diagnostics)

```
qacli view -P <project-path> -m STDOUT --suppressed-messages
```

The shortcut for this switch is ’-s’. Note that diagnostics that are suppressed by baselining will always be presented, regardless of this switch, and CMA diagnostics will never be presented.

To prevent diagnostics from header files being presented:

```
qacli view -P <project-path> -m STDOUT --no-header-messages
```

The shortcut for this switch is ’-n’.

To present diagnostics as annotations in the original source code (the original source is only read and will never be modified or have its modification time altered under any circumstances):

```
qacli view -P <project-path> -m STDOUT --annotated-source
```

The shortcut for this switch is ’-a’. This switch implies ’- -no-header-messages’.

Switches may be chained:

```
qacli view -P <project-path> -m STDOUT -rsa
```

To specify results for a specific source file within a PRQA Framework project, the path to the source file can be provided:

```
qacli view -P <project-path> -m STDOUT <path-to-file>
```

To specify results for a specific set of source files within a PRQA Framework project, the path to each source file must be placed in a text file, one path per line - a filelist. To view:

```
qacli view -P <project-path> -m STDOUT --files <path-to-filelist>
```

The shortcut for this option is '-F'.

Finally, when the argument to the '- -medium' option is set to 'HTML', a directory can be specified. HTML files will be output to this directory:

```
qacli view -P <project-path> -m HTML -output-path <directory>
```

The directory must exist and the shortcut for this option is '-o'.

12.8.1 Formatting the Diagnostic String

To alter the format of the diagnostic string, the following option can be used:

```
qacli view -P <project-path> -m STDOUT --format <format-string>
```

where *format-string* is a quoted string in which the following specifiers are recognised:

```
%f - file name
%F - file name (absolute, including path)
%B - base file name
%l - line number
%c - column number
%C - column number less 1 (left column is 0)
%g - message level
%n - message number (raw integer format)
%N - message number (zero padded to four digits)
%t - message text
%H - message help file path
%v - verbose text
%p - producer component (e.g. qacpp-3.1)
%r - rule number
%G - rule group number
%^ - if the column is not zero and the message is not a submessage
for different line or filename, update last location, show  the
caret and insert a newline. It is equivalent to the following
sequence:
%?c>0%(?u==0%(%q%R(%C, )^\n%:%?L%(%:%q%R(%C, )^\n%)%)%)
```

12.8.2 Conditional Formatting

Message formatting can also be configured so as to apply only in certain conditions. This allows you to specialize message display, for example according to the message level. You can enter a conditional statement in the format:

```
%?<condition> %(<true condition> [%:<false condition>] %)
```

The false branch is optional and if it is not supplied, nothing is displayed. The condition field consists of a letter, and optionally a conditional operator and a value.

The following conditional variables can hold any legitimate value, and must be tested using a relational operator:

l	Line number
c	Column number
g	Message level
n	Message number
u	Context message depth

The possible values for Message level (%g) are:

0	Error
1	Warning
2	Information
3	User Message

For example, to display Err for level 0 messages and Msg for all others, the conditional statement would look like:

```
%?g==0%(Err%:Msg%)
```

All warnings in the message level 0 will be generated with Err in the message prefix. All other levels will be generated with Msg.

Primary messages have a context message depth (u) of zero, and context messages have level 1 or greater. Thus, typical conditional processing of context messages is:

```
%?u==0%( primary message %: context message %)
```

The following conditional variables can only be truth-tested for a change in their value:

C	File name, line and column number
L	File name and line number
F	File name

For example, a simple check for a file name is:

```
%?F%(\n%f%)
```

In this example, if the file name has changed (F), the new file name will be displayed on a new line (\n%f). The conditional variable C can be used to determine the positioning of the warning location caret (^). In the default message format, the caret is omitted for messages with the same location as the previous message. To replicate this, use the following conditional statement:

```
%?C%(%^%)
```

This statement will compare the file name, line and column number (C) of the current message with the previous message. If the location is different, the statement is considered true and the caret is displayed, followed by a new line, which is added implicitly.

Finally, properties of a message can be queried with these Boolean conditional variables:

h	Is message a hard error (level 9)
v	Is verbose text present?

For example, the following statement:

```
%t%?v%(\n%v)
```

will print the message text (%t) followed by the verbose text on a new line (\n%v), if it exists (%?v).

12.8.3 Message Text Control

There is another set of format specifiers that control how text is handled within warning messages. These should appear at the beginning of the format string.

%s(n)	limit message size: text is truncated to n characters.
%w(n)	character wrap: message text wraps at column n.
%W(n)	word wrap: message text wraps after the last word before column n.
%i(str)	indent wrapped text: when text is wrapped, it is indented with <str>.
%q	save the location for the next message (used for %F, %L, %C tests).
%Q	reset the location for the next message (%F, %L, %C will yield true when formatting the next message). Note: %Q takes precedence over %q.
%R(num,char)	print num copies of char.

To limit the size of message output to 80 characters per line without word breaks, include the specifier %W(80) in the format string.

The %q and %Q entries can be used to override the default behavior of %^, or to implement similar functionality from scratch. As an example of %R(num,char) usage:

```
%?c>0%(%?C%(%R(%C,.)^\n%)%)%)%
```

will cause the following display of caret lines:

```
.....................^
```

12.8.4 An Example Message Format

The following is a typical message format that displays the most relevant information about a warning message:

```
%W(80)%i( )%?C%(%^%)%f(%l) ++ %?h%(ERROR%:WARNING%) ++: <=%g=(%n) %t
```

The components of this string are:

%W(80)	Message text wraps after the last word before column 80.
%i( )	Each hanging indent is indented by three columns.
%?C%(%^%)	A caret is displayed if the location of the error is different from that of the previous error.
%f(%l) ++	The file name and line number (in brackets) is displayed, followed by '++'.
%?h%(ERROR%: WARNING%) ++:	If the message is a hard error, ERROR is displayed, otherwise WARNING is displayed, followed by '++:'.
<=%g=(%n)	The message level is displayed (preceded by '<=') and followed by the message number (preceded by '=').
%t	The message text is displayed.

A warning message with the above format would look like:

```
 int i;
^
test.c (1) ++ WARNING ++: <=2=(3408) 'i' is externally
visible. Functions and variables with external
linkage should be declared in a header file.
```

12.8.5 Default Format Strings

The default format string for standard output and error, which is compatible with Eclipse and GNU Emacs C++ mode (i.e. will link to the correct file, line and column), is:

```
%F:%l:%c: %?u==0%(%?h%(Err%:Msg%)%:-->%)(%G:%N) %R(%u, )%t%?v%(\n%v%)
```

The default format string for annotated source is:

```
%?C%(%^%)%?u==0%(%?h%(Err%:Msg%)%:-->%F:%l:%c %)(%g:%N) %R(%u, )%t%?v%(\n%v%)
```

The default format string for HTML output is:

```
%?C%(%^%)%?u==0%(%?h%(Err%:Msg%)%:-->%F:%l:%c %)(<a href=\"%H\">%g:%p-%N</a>)
%R(%u, )%t%?v%(<br/>%v%)
```