Automation & Testing with CMake

10.1 Testing with CMake, CTest, and CDash

Testing is a key tool for producing and maintaining robust, valid software. This chapter will examine the tools that are part of CMake to support software testing. We will begin with a brief discussion of testing approaches and then discuss how to add tests to your software project using CMake. Finally we will look at additional tools that support creating centralized software status dashboards.

The tests for a software package may take a number of forms. At the most basic level there are smoke tests, such as one that simply verifies that the software compiles. While this may seem like a simple test, with the wide variety of platforms and configurations available, smoke tests catch more problems than any other type of test. Another form of smoke test is to verify that a test runs without crashing. This can be handy for situations where the developer does not want to spend the time creating more complex tests, but is willing to run some simple tests. Most of the time these simple tests can be small example programs. Running them verifies not only that the build was successful, but that any required shared libraries can be loaded (for projects that use them) and that at least some of the code can be executed without crashing.

Moving beyond basic smoke tests leads to more specific tests such as regression, black, and white box testing. Each of these has its strengths. Regression testing verifies that the results of a test do not change over time, or platform. This is very useful when performed frequently, as it provides a quick check that the behavior and results of the software have not changed. When a regression test fails a quick look at recent code changes can usually identify the culprit. Unfortunately, regression tests typically require more effort to create than other tests.

White and black box testing refer to tests written to exercise units of code (at various levels of integration) with and without knowledge of how those units are implemented respectively. White box testing is designed to stress potential failure points in the code knowing how that code was written, and hence its weaknesses. As with regression testing this can take a substantial amount of effort to create good tests. Black box testing typically knows little or nothing about the implementation of the software other than its public API. Black box testing can provide a lot of code coverage without too much effort in developing the tests. This is especially true for libraries of object oriented software where the APIs are well defined. A black box test can be written to go through and invoke a number of typical methods on all the classes in the software.

The final type of testing we will discuss is software standard compliance testing. While the other test types we have discussed are focused on determining if the code works properly, compliance testing tries to determine if the code adheres to the coding standards of the software project. This could be a check to verify that all classes have implemented some key method, or that all functions have a common prefix. The options for this type of test are limitless and there are a number of ways to perform such testing. There are software analysis tools that can be used, or specialized test programs (maybe python scripts etc) could be written. The key point to realize is that the tests do not necessarily have to involve running some part of the software. The tests might run some other tool on the source code itself.

There are a number of reasons why it helps to have testing support integrated into the build process. First, complex software projects may have a number of configuration or platform-dependent options. The build system knows what options can be enabled and can then enable the appropriate tests for those options. For example, the Visualization Toolkit (VTK) includes support for a parallel processing library called MPI. If VTK is built with MPI support then additional tests are enabled that make use of MPI and verify that the MPI-specific code in VTK works as expected. Secondly, the build system knows where the executables will be placed and it has tools for finding other required executables (such as perl, python etc). The third reason is that with UNIX Makefiles it is common to have a test target in the Makefile so that developers can type make test and have the test(s) run. In order for this to work, the build system must have some knowledge of the testing process.

10.2 How Does CMake Facilitate Testing?

CMake facilitates testing your software through special testing commands and the CTest executable. First we will discuss the key testing commands in CMake. To add testing to a CMake-based project is fairly simple using the `add_test` command. The `add_test` command has a simple syntax as follows:

```
add_test (TestName ExecutableToRun arg1 arg2 arg3 ...)
```

The first argument is simply a string name for the test. This is the name that will be displayed by testing programs. The second argument is the executable to run. The executable can be built as part of the project or it can be a standalone executable such as python, perl, etc. The remaining arguments will be passed to the running executable. A typical example of testing using the `add_test` command would look like this:

```
add_executable (TestInstantiator TestInstantiator.cxx)
target_link_libraries (TestInstantiator vtkCommon)
add_test (TestInstantiator
          ${EXECUTABLE_OUTPUT_PATH}/TestInstantiator)
```

The `add_test` command is typically placed in the CMakeLists file for the directory that has the test in it. For large projects there may be multiple CMakeLists files with `add_test` commands in them. Once the `add_test` commands are present in the project, the user can run the tests by invoking the "test" target of Makefile, or the `RUN_TESTS` target of Visual Studio or Xcode. An example of running tests on the CMake tests using the Makefile generator on Linux would be:

```
$ make test
Running tests...
Test project
     Start 2: kwsys.testEncode
 1/20 Test  #2: kwsys.testEncode ..........   Passed    0.02 sec
     Start 3: kwsys.testTerminal
 2/20 Test  #3: kwsys.testTerminal ........   Passed    0.02 sec
     Start 4: kwsys.testAutoPtr
 3/20 Test  #4: kwsys.testAutoPtr .........   Passed    0.02 sec
```

10.3 Additional Test Properties

By default a test passes if all of the following conditions are true:

- The test executable was found
- The test ran without exception
- The test exited with return code 0

That said, these behaviors can be modified using the `set_property` command:

```
set_property (TEST test_name
              PROPERTY prop1 value1 value2 ...)
```

This command will set additional properties for the specified tests. Example properties are:

ENVIRONMENT

Specify environment variables that should be defined for running a test. If set to a list of environment variables and values of the form `MYVAR=value` those environment variables will be defined while the test is running. The environment is restored to its previous state after the test is done.

LABELS

Specify a list of text labels associated with a test. These labels can be used to group tests together based on what they test. For example you could add a label of MPI to all tests that exercise MPI code.

WILL_FAIL

If this option is set to true, then the test will pass if the return code is not 0 and fail if it is. This reverses the third condition of the pass requirements.

PASS_REGULAR_EXPRESSION

If this option is specified, then the output of the test is checked against the regular expression provided (a list of regular expressions may be passed in as well). If none of the regular expressions match, then the test will fail. If at least one of them matches, then the test will pass.

FAIL_REGULAR_EXPRESSION

If this option is specified, then the output of the test is checked against the regular expression provided (a list of regular expressions may be passed in as well). If none of the regular expressions match, then the test will pass. If at least one of them matches, then the test will fail.

If both `PASS_REGULAR_EXPRESSION` and `FAIL_REGULAR_EXPRESSION` are specified, then the `FAIL_REGULAR_EXPRESSION` takes precedence. The following example illustrates using the `PASS_REGULAR_EXPRESSION` and `FAIL_REGULAR_EXPRESSION`:

```
add_test (outputTest ${EXECUTABLE_OUTPUT_PATH}/outputTest)

set (passRegex "^Test passed" "^All ok")
set (failRegex "Error" "Fail")

set_property (TEST outputTest
              PROPERTY PASS_REGULAR_EXPRESSION "${passRegex}")
set_property (TEST outputTest
              PROPERTY FAIL_REGULAR_EXPRESSION "${failRegex}")
```

10.4 Testing Using CTest

When you run the tests from your build environment what really happens is that the build environment runs CTest. CTest is an executable that comes with CMake, it handles running the tests for the project. While CTest works well with CMake, you do not have to use CMake in order to use CTest. The main input file for CTest is called `CTestTestfile.cmake`. This file will be created in each directory that was processed by CMake (typically every directory with a CMakeLists file). The syntax of `CTestTestfile.cmake` is like the regular CMake syntax, with a subset of the commands available. If CMake is used to generate testing files, they will list any subdirectories that need to be processed as well as any `add_test` calls. The subdirectories are those that were added by `subdirs` or `add_subdirectory` commands. CTest can then parse these files to determine what tests to run. An example of such a file is shown below:

```
# CMake generated Testfile for
#  Source directory: C:/CMake
#  Build directory: C:/CMakeBin
#
# This file includes the relevent testing commands required
# for testing this directory and lists subdirectories to
# be tested as well.

ADD_TEST (SystemInformationNew …)

SUBDIRS (Source/kwsys)
SUBDIRS (Utilities/cmzlib)
…
```

When CTest parses the `CTestTestfile.cmake` files it will extract the list of tests from them. These tests will be run, and for each test CTest will display the name of the test and its status. Consider the following sample output:

```
$ ctest
Test project C:/CMake-build26
        Start 1: SystemInformationNew
 1/21 Test  #1: SystemInformationNew ......   Passed    5.78 sec
        Start 2: kwsys.testEncode
 2/21 Test  #2: kwsys.testEncode ..........   Passed    0.02 sec
        Start 3: kwsys.testTerminal
 3/21 Test  #3: kwsys.testTerminal ........   Passed    0.00 sec
        Start 4: kwsys.testAutoPtr
 4/21 Test  #4: kwsys.testAutoPtr .........   Passed    0.02 sec
        Start 5: kwsys.testHashSTL
```

```
 5/21 Test  #5: kwsys.testHashSTL .........   Passed    0.02 sec
...
100% tests passed, 0 tests failed out of 21
Total Test time (real) =  59.22 sec
```

CTest is run from within your build tree. It will run all the tests found in the current directory as well as any subdirectories listed in the `CTestTestfile.cmake`. For each test that is run CTest will report if the test passed and how long it took to run the test.

The CTest executable includes some handy command line options to make testing a little easier. We will start by looking at the options you would typically use from the command line.

```
-R <regex>              Run tests matching regular expression
-E <regex>              Exclude tests matching regular expression
-L <regex>              Run tests with labels matching the regex
-LE <regex>             Run tests with labels not matching regexp
-C <config>             Choose the configuration to test
-V,--verbose            Enable verbose output from tests.
-N,--show-only          Disable actual execution of tests.
-I [Start,End,Stride,test#,test#|Test file]
                        Run specific tests by range and number.
-H                      Display a help message
```

The `-R` option is probably the most commonly used. It allows you to specify a regular expression, only the tests with names matching the regular expression will be run. Using the `-R` option with the name (or part of the name) of a test is a quick way to run a single test. The `-E` option is similar except that it excludes all tests matching the regular expression. The `-L` and `-LE` options are similar to `-R` and `-E` except that they apply to test labels that were set using the `set_property` command as described in section 10.3. The `-C` option is mainly for IDE builds where you might have multiple configurations, such as Release and Debug in the same tree. The argument following the `-C` determines which configuration will be tested. The –V argument is useful when you are trying to determine why a test is failing, with –V CTest will print out the command line used to run the test as well as any output from the test itself. The –V option can be used with any invocation of CTest to provide more verbose output. The –N option is useful if you want to see what tests CTest would run without actually running them.

Running the tests and making sure they all pass before committing any changes to the software is a sure fire way to improve your software quality and development process. Unfortunately, for large projects the number of tests and the time required to run them may be prohibitive. In these situations the –I option of CTest can be used. The –I option allows you to

flexibly specify a subset of the tests to run. For example, the following invocation of CTest will run every seventh test.

```
ctest -I ,,7
```

While this is not as good as running every test, it is better than not running any and it may be a more practical solution for many developers. Note that if the start and end arguments are not specified, as in this example, then they will default to the first and last tests. In another example, assume that you always want to run a few tests plus a subset of the others. In this case you can explicitly add those tests to the end of the arguments for –I. For example:

```
ctest -I ,,5,1,2,3,10
```

will run tests 1, 2, 3, and 10, plus every fifth test. You can pass as many test numbers as you want after the stride argument.

10.5 Using CTest to Drive Complex Tests

Sometimes to properly test a project you need to actually compile code during the testing phase. There are several reasons for this. First, if test programs are compiled as part of the main project, they can end up taking up a significant amount of the build time. Also, if a test fails to build, the main build should not fail as well. Finally, IDE projects can quickly become too large to load and work with. The CTest command supports a group of command line options that allow it to be used as the test executable to run. When used as the test executable, CTest can run CMake, run the compile step, and finally run a compiled test. We will now look at the command line options to CTest that support building and running tests.

```
--build-and-test  src_directory build_directory
Run cmake on the given source directory using the specified
build directory.
--test-command          Name of the program to run.
--build-target          Specify a specific target to build.
--build-nocmake         Run the build without running cmake first.
--build-run-dir         Specify directory to run programs from.
--build-two-config      Run CMake twice before the build.
--build-exe-dir         Specify the directory for the executable.
--build-generator       Specify the generator to use.
--build-project         Specify the name of the project to build.
--build-makeprogram     Specify the make program to use.
--build-noclean         Skip the make clean step.
--build-options         Add extra options to the build step.
```

For an example, consider the following add_test command taken from the CMakeLists.txt file of CMake itself. It shows how CTest can be used both to compile and run a test.

```
add_test (simple ${CMAKE_CTEST_COMMAND}
   --build-and-test "${CMake_SOURCE_DIR}/Tests/Simple"
                    "${CMake_BINARY_DIR}/Tests/Simple"
   --build-generator ${CMAKE_GENERATOR}
   --build-makeprogram ${CMAKE_MAKE_PROGRAM}
   --build-project Simple
   --test-command simple)
```

In this example, the add_test command is first passed the name of the test, "simple". After the name of the test, the command to be run is specified. In this case, the test command to be run is CTest. The CTest command is referenced via the CMAKE_CTEST_COMMAND variable. This variable is always set by CMake to the CTest command that came from the CMake installation used to build the project. Next, the source and binary directories are specified. The next options to CTest are the --build-generator, and --build-makeprogram options. These are specified using the cmake variables CMAKE_MAKE_PROGRAM and CMAKE_GENERATOR. Both CMAKE_MAKE_PROGRAM and CMAKE_GENERATOR are defined by CMake. This is an important step as it makes sure that the same generator is used for building the test as was used for building the project itself. The --build-project option is passed Simple which corresponds to the project command used in the Simple test. The final argument is the --test-command which tells CTest the command to run once it gets a successful build. That should be the name of the executable that will be compiled by the test.

10.6 Handling a Large Number of Tests

When a large number of tests exist in a single project, it is cumbersome to have individual executables available for each test. That said, the developer of the project should not be required to create tests with complex argument parsing. This is why CMake provides a convenience command for creating a test driver program. This command is called create_test_sourcelist. A test driver is a program that links together many small tests into a single executable. This is useful when building static executables with large libraries to shrink the total required size. The signature for create_test_sourcelist is as follows:

```
create_test_sourcelist (SourceListName
                        DriverName
                        test1 test2 test3
                        EXTRA_INCLUDE include.h
                        FUNCTION function
                        )
```

The first argument is the variable which will contain the list of source files that must be compiled to make the test executable. The DriverName is the name of the test driver program (e.g. the name of the resulting executable). The rest of the arguments consist of a list of test source files. Each test source file should have a function in it that has the same name as the file with no extension (`foo.cxx` should have `int foo(argc, argv);`) The resulting executable will be able to invoke each of the tests by name on the command line. The `EXTRA_INCLUDE` and `FUNCTION` arguments support additional customization of the test driver program. Consider the following CMakeLists file fragment to see how this command can be used:

```
# create the testing file and list of tests
create_test_sourcelist(Tests
  CommonCxxTests.cxx
  ObjectFactory.cxx
  otherArrays.cxx
  otherEmptyCell.cxx
  TestSmartPointer.cxx
  SystemInformation.cxx
  ...
  )

# add the executable
add_executable(CommonCxxTests ${Tests})

# remove the test driver source file
set (TestsToRun ${Tests})
remove (TestsToRun CommonCxxTests.cxx)

# Add all the ADD_TEST for each test
foreach (test ${TestsToRun})
  get_filename_component(TName ${test} NAME_WE)
  add_test (${TName} CommonCxxTests ${TName})
endforeach ()
```

The `create_test_sourcelist` command is invoked to create a test driver. In this case it creates and writes `CommonCXXTests.cxx` into the binary tree of the project using the rest of the arguments to determine its contents. Next the `add_executable` command is used to add that executable to the build. Then a new variable called `TestsToRun` is created with an initial value of the sources required for the test driver. The `remove` command is used to remove the driver program itself from the list. Then a `foreach` command is used to loop over the remaining sources. For each source its name without a file extension is extracted and put in the variable `TName`, then a new test is added for `TName`. The end result is that for each source file in the `create_test_sourcelist` an `add_test` command is called with the name of

the test. As more tests are added to the `create_test_sourcelist` command, the `foreach` loop will automatically call `add_test` for each one.

10.7 Producing Test Dashboards

As your project's testing needs grow, keeping track of the test results can become overwhelming. This is especially true for projects that are tested nightly on a number of different platforms. In these cases we recommend using a test dashboard to summarize the test results. (see Figure 30)

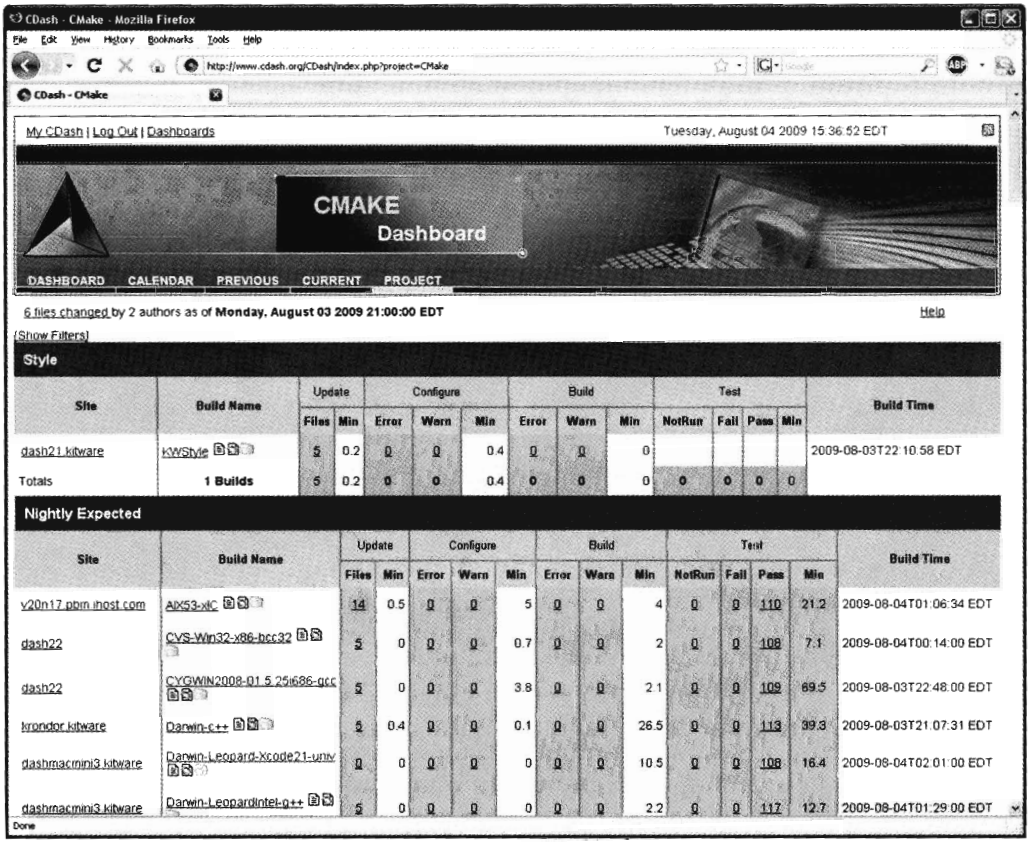

Figure 30 - Sample Testing Dashboard

A test dashboard summarizes the results for many tests on many platforms, and its hyperlinks allow people to drill down into additional levels of detail quickly. The CTest executable includes support for producing test dashboards. When run with the correct options, CTest will

produce XML-based output recording the build and test results and post them to a dashboard server. The dashboard server runs an open source software package called CDash. CDash collects the XML results and produces HTML web pages from them.

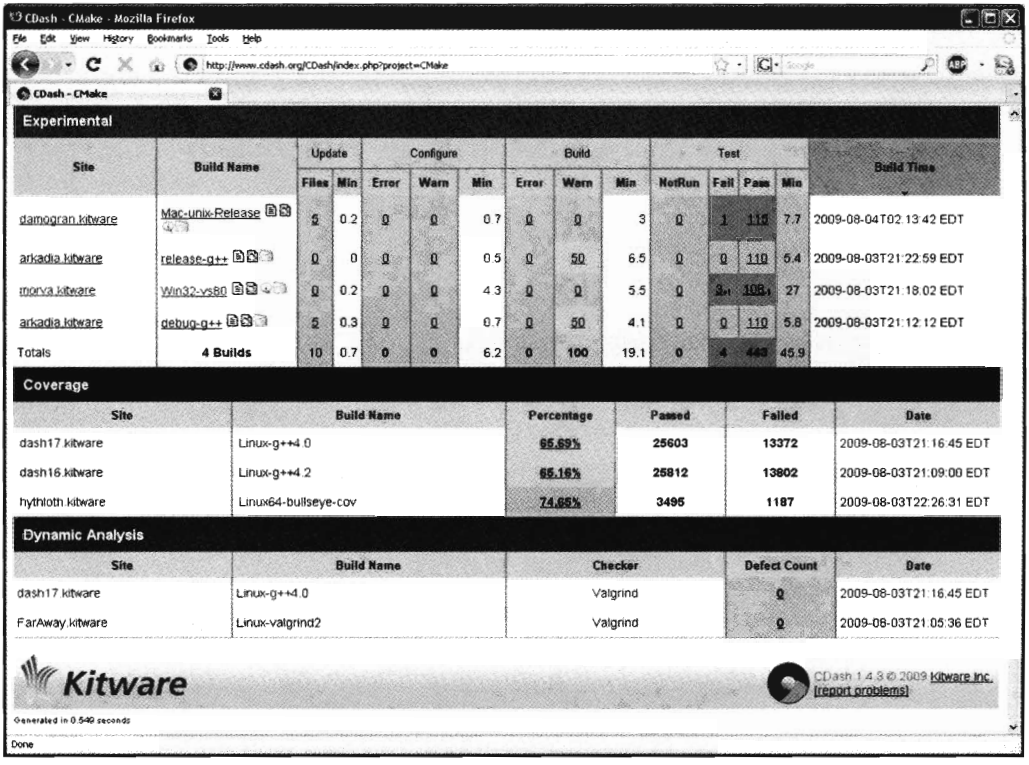

Figure 31 - Experimental, Coverage, and Dynamic Analysis Results

Before discussing how to use CTest to produce a dashboard let us consider the main parts of a testing dashboard. Each night at a specified time the dashboard server will open up a new dashboard, so each day there is a new web page showing the results of tests for that twenty-four hour period. There are links on the main page that allow you to quickly navigate through different days. Looking at the main page for a project (such as CMake's dashboard off of www.cmake.org) you will see that it is divided into a few main components. Near the top you will find a set of links that allow you to step to previous dashboards as well as links to project pages such as the bug tracker, documentation etc.

Below that you will find groups of results. Typically groups that you will find include Nightly, Experimental, Continuous, Coverage, and Dynamic Analysis (see Figure 31). The category into which a dashboard entry will be placed depends on how it was generated. The simplest are Experimental entries which represent dashboard results for someone's current

copy of the project's source code. With an experimental dashboard the source code is not guaranteed to be up to date. In contrast a Nightly dashboard entry is one where CTest tries to update the source code to a specific date and time. The expected result is that all nightly dashboard entries for a given day should be based on the same source code.

A continuous dashboard entry is one that is designed to run every time new files are checked in. Depending on how frequently new files are checked in a single day's dashboard could have many continuous entries. Continuous dashboards are particularly helpful for cross platform projects where a problem may only show up on some platforms. In those cases a developer can commit a change that works for them on their platform and then another platform running a continuous build could catch the error, allowing the developer to correct the problem promptly.

Dynamic Analysis and Coverage dashboards are designed to test the memory safety and code coverage of a project. A Dynamic Analysis dashboard entry is one where all the tests are run with a memory access/leak checking program enabled. Any resulting errors or warnings are parsed, summarized and displayed. This is important to verify that your software is not leaking memory, reading from un-initialized memory, etc. Coverage dashboard entries are similar in that all the tests are run, but as the tests are run the lines of code executed are tracked. When all the tests have been run, a listing of how many times each line of code was executed is produced and displayed on the dashboard.

Adding CDash Dashboard Support to a Project

In this section we show how to submit results to the CDash dashboard. You can either use the Kitware CDash servers at my.cdash.org or you can setup your own CDash server as described in section 10.11. If you are using my.cdash.org you can click on the "Start My Project" button which will ask you to create an account (or login if you already have one), and then bring you to a page to start creating your project. If you have installed your own CDash server then you should login to your CDash server as administrator and select "Create New Project" from the administration panel. Regardless of which approach you use the next few steps will be to fill in information about your project as shown in Figure 32. Many of the items below are optional, so do not be concerned if you do not have a value for them.

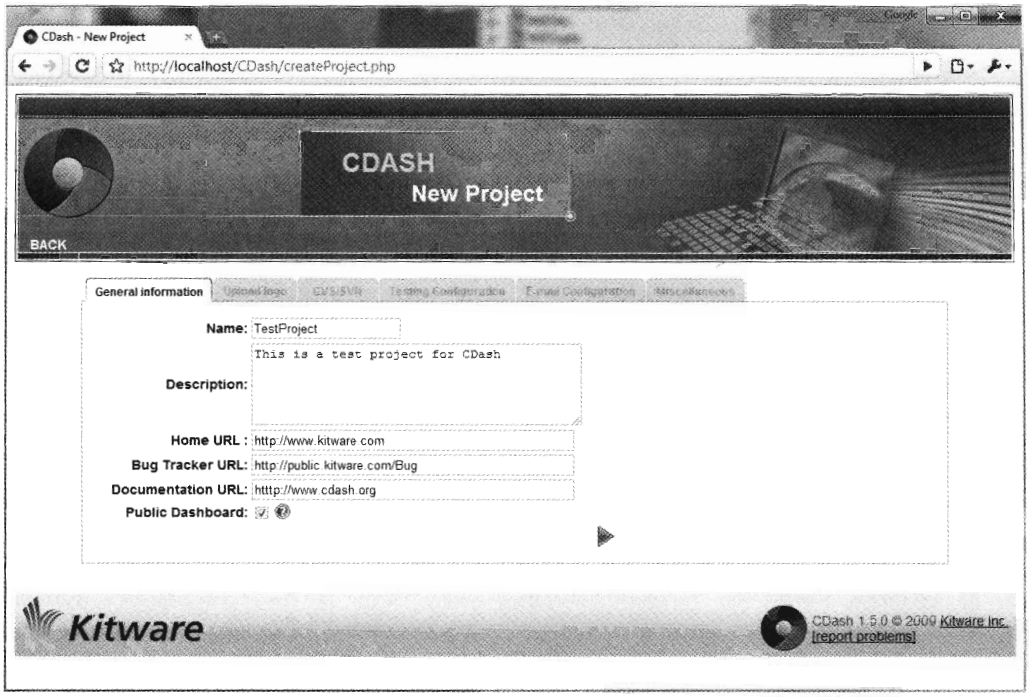

Figure 32 – Creating a new project in CDash

Name: what you want to call the project.

Description: description of the project to be shown on the first page.

Home URL: home URL of the project to appear in the main menu of the dashboard.

Bug Tracker URL: URL to the bug tracker. Currently CDash supports Mantis (http://www.mantisbt.org/), and if a bug is entered in the repository with the message "BUG: 132456", CDash will automatically link to the appropriate bug.

Documentation URL: URL to where the project's documentation is kept. This will appear in the main menu of the dashboard.

Public Dashboard: if checked, the dashboard is public and anybody can see the results of the dashboard. If unchecked, only users assigned to this project can access the dashboard.

Logo: logo of the project to be displayed on the main dashboard. Optimal size for a logo is 100x100 pixels. Transparent GIFs work best as they can blend in with the CDash background.

Repository Viewer URL: URL of the web repository browser. CDash currently supports: ViewCVS, Trac, Fisheye, ViewVC, WebSVN and Loggerhead. For instance a URL for ViewVC is http://public.kitware.com/cgi-bin/viewcvs.cgi/?cvsroot=CMake, or for WebSVN: https://www.kitware.com/websvn/listing.php?repname=MyRepository

Repositories: in order to display the daily updates CDash gets a diff version of the modified files. Current CDash supports only anonymous repository access. A typical URL is `:pserver:anoncvs@myproject.org:/cvsroot/MyProject`.

Nightly Start Time: CDash displays the current dashboard using a 24 hour window. The nightly start time defines the beginning of this window. Note that the start time is expressed in the form HH:MM:SS TZ, e.g. 21:00:00 EDT.

Coverage Threshold: CDash marks that coverage has passed (green) if the global coverage for a build or specific files is above this threshold. It is recommended to set the coverage threshold to a high value and decrease it as you focus on improving your coverage.

Enable Test Timing: enable/Disable test timing for this project See "Test timing" in the next section for more information.

Test Time Standard Deviation: set a multiplier for the standard deviation of a test time. If the time for a test is higher than the mean + multiplier*standard deviation, the test time status is marked as failed. The default value is 4 if not specified. Note that changing this value does not affect previous builds, only builds submitted after the modification.

Test Time Standard Deviation Threshold: set a minimum standard deviation for a test time. If the current standard deviation for a test is lower than this threshold then the threshold is used instead. This is particularly important for tests that have a very low standard deviation but still some variability. The default threshold is set to 2 if not specified. Note that changing this value does not affect previous builds, only builds submitted after the modification.

Test Time # Max Failures Before Flag: some tests might take longer from one day to another depending on the client machine load. This variable defines the number of times a test should fail because of timing issues before being flagged.

Email Submission Failures: enable/disable sending email when a build fails (configure, error, warnings, update, test failings) for this project. This is a general feature.

Email Redundant Failure: by default CDash does not send email for the same failures. For instance if a build continues to fail over time, only one email would be sent. If the email redundant failures is checked then CDash will send an email every time a build has a failure. (not all versions of CDash have this option).

Email Administrator: enable/disable sending email when submission to CDash is invalid. This can help tracking down misconfigured clients. This is particularly useful when CTest is not used to submit to CDash..

Email Build Missing: enable/disable sending email when a build has not submitted.

Email Low Coverage: enable/disable sending email when the coverage for files is lower than the default threshold value specified above.

Email Test Timing Changed: enable/disable sending email when a test's timing has changed.

Maximum Number of Items in Email: how many failures should be sent in an email.

Maximum Number of Characters in Email: how many characters from the log should be sent in the email.

Google Analytics Tracker: CDash supports visitor tracking through Google analytics. See "Adding Google Analytics" for more information.

Show Site IP Addresses: Enable/disable the display of IP addresses of the sites submitting to this project.

Display Labels: as of CDash 1.4, and recent versions of CTest, labels can be attached to build files, these can displayed and retrieved on CDash.

AutoRemove Timeframe: set the number of days to keep for this project. If the timeframe is less than 2 days, CDash will not remove any builds (not all versions of CDash have this option).

AutoRemove Max Builds: set the maximum number of builds to remove when performing the auto removal of builds (not all versions of CDash have this option).

After providing this information you can click on "Create Project" to create the project in CDash. At this point the server is ready to accept dashboard submissions. The next step is to provide the dashboard server information to your software project. This information is kept in a file named `CTestConfig.cmake` at the top level of your source tree. You can download this file by clicking on the "Edit Project" button for your dashboard (it looks like a pie chart with a wrench underneath it), then click on the miscellaneous tab and select "Download CTestConfig" and save the `CTestConfig.cmake` in your source tree. In the next section we review this file in more detail.

Client Setup

To support dashboards in your project you need to include the CTest module as follows.

```
# Include CDash dashboard testing module
include (CTest)
```

The CTest module will then read settings from the `CTestConfig.cmake` file you downloaded from CDash. If you have added `add_test` command calls to your project creating a dashboard entry is as simple as running:

```
ctest -D Experimental
```

The `-D` option tells CTest to create a dashboard entry. The next argument indicates what type of dashboard entry to create. Creating a dashboard entry involves quite a few steps that can be run independently or as one command. In this example the Experimental argument will cause CTest to perform a number of different steps as one command. The different steps of creating a dashboard entry are summarized below.

Start

Prepare a new dashboard entry. This creates a Testing subdirectory in the build directory. The testing subdirectory will contain a subdirectory for the dashboard results with a name that corresponds to the dashboard time. The testing subdirectory will also contain a subdirectory for the temporary testing results called Temporary.

Update

Perform a source control update of the source code (typically used for nightly or continuous runs). Currently CTest supports Concurrent Versions System (CVS), Subversion, Git, Mercurial and Bazaar.

Configure

Run CMake on the project to make sure the Makefiles or project files are up to date.

Build

Build the software using the specified generator.

Test

Run all the tests and record results.

MemoryCheck

Perform memory checks using Purify or valgrind.

Coverage

Collect source code coverage information.

Submit

Submit the testing results as a dashboard entry to the server.

Each of these steps can be run independently for a Nightly or Experimental entry using the following syntax:

```
ctest -D NightlyStart
ctest -D NightlyBuild
ctest -D NightlyCoverage -D NightlySubmit
```

or

```
ctest -D ExperimentalStart
ctest -D ExperimentalConfigure
ctest -D ExperimentalCoverage -D ExperimentalSubmit
```

etc. Alternatively you can use shortcuts that perform the most common combinations all at once. The shortcuts that CTest has defined include:

ctest -D Experimental

performs a start, configure, build, test, coverage, submit

ctest -D Nightly

performs a start, update, configure, build, test, coverage, submit

ctest -D Continuous

performs a start, update, configure, build, test, coverage, submit

ctest -D MemoryCheck

performs a start, configure, build, memory check, coverage, submit

When first setting up a dashboard it is often useful to combine the –D option with the –V option. This will allow you to see the output of all the different stages of the dashboard process. Likewise, CTest maintains log files in the `Testing/Temporary` directory it creates in your binary tree. There you will find log files for the most recent build, update, test, etc. The actual dashboard results are stored in the Testing directory too.

10.8 Customizing Dashboards for a Project

CTest has a few options that can be used to control how it processes a project. If, when CTest runs a dashboard, it finds `CTestCustom.ctest` files in the binary tree, it will load these files and use the settings from them to control its behavior. The syntax of a CTestCustom file is the same as regular CMake syntax. That said, only set commands are normally used in this file. These commands specify properties that CTest will consider when performing the testing.

Dashboard Submissions Settings

A number of the basic dashboard settings are provided in the file that you download from CDash. You can edit these initial values and provide additional values if you wish. The first value that is set is the nightly start time. This is the time that dashboards all around the world will use for checking out their copy of the nightly source code. This time also controls how dashboard submissions will be grouped together. All submissions from the nightly start time until the next nightly start time will be included on the same "day".

```
# Dashboard is opened for submissions for a 24 hour period
# starting at the specified NIGHLY_START_TIME. Time is
# specified in 24 hour format.
set (CTEST_NIGHTLY_START_TIME "21:00:00 EDT")
```

The next group of settings control where to submit the testing results. This is the location of the CDash server.

```
# CDash server to submit results (used by client)
set (CTEST_DROP_METHOD http)
set (CTEST_DROP_SITE "my.cdash.org")
set (CTEST_DROP_LOCATION "/submit.php?project=KensTest")
set (CTEST_DROP_SITE_CDASH TRUE)
```

The `CTEST_DROP_SITE` specifies the location of the CDash server. Build and test results generated by CDash clients are sent to this location. The `CTEST_DROP_LOCATION` is the directory or the HTTP URL on the server where CDash clients leave their build and test reports. The `CTEST_DROP_SITE_CDASH` specifies that the current server is CDash which prevents CTest from trying to trigger the submission (backwards compatibility with Dart and Dart 2).

Currently CDash supports only the HTTP drop submission method; however CTest supports other submission types. The `CTEST_DROP_METHOD` specifies the method used to submit testing results. The most common setting for this will be HTTP which uses the Hyper Text Transfer Protocol (HTTP) to transfer the test data to the server. Other drop methods are supported for special cases such as FTP and SCP. In the example below, clients that are

submitting their results using the HTTP protocol use a web address as their drop site. If the submission is via FTP, this location is relative to where the CTEST_DROP_SITE_USER will log in by default. The CTEST_DROP_SITE_USER specifies the FTP username the client will use on the server. For FTP submissions this user will typically be "anonymous". However, any username that can communicate with the server can be used. For FTP servers that require a password this can be stored in the CTEST_DROP_SITE_PASSWORD variable. The CTEST_DROP_SITE_MODE (not used in this example) is an optional variable that you can use to specify the FTP mode. Most FTP servers will handle the default passive mode but you can set the mode explicitly to active if your server does not.

CTest can also be run from behind a firewall. If the firewall allows FTP or HTTP traffic, then no additional settings are required. If the firewall requires an FTP/HTTP proxy or uses a SOCKS4 or SOCKS5 type proxy, some environment variables need to be set. HTTP_PROXY and FTP_PROXY specify the servers that service HTTP and FTP proxy requests. HTTP_PROXY_PORT and FTP_PROXY_PORT specify the port on which the HTTP and FTP proxies reside. HTTP_PROXY_TYPE specifies the type of the HTTP proxy used. The three different types of proxies supported are the default, which is a generic HTTP/FTP proxy, "SOCKS4", and "SOCKS5", which specify SOCKS4 and SOCKS5 compatible proxies.

Filtering Errors and Warnings

By default CTest has a list of regular expressions that it matches for finding the errors and warnings from the output of the build process. You can override these settings in your CTestCustom.ctest files using several variables as shown below.

```
set (CTEST_CUSTOM_WARNING_MATCH
  ${CTEST_CUSTOM_WARNING_MATCH}
  "{standard input}:[0-9][0-9]*: Warning: "
  )

set (CTEST_CUSTOM_WARNING_EXCEPTION
  ${CTEST_CUSTOM_WARNING_EXCEPTION}
  "tk8.4.5/[^/]+/[^/]+.c[:\"]"
  "xtree.[0-9]+. : warning C4702: unreachable code"
  "warning LNK4221"
  "variable .var_args[2]*. is used before its value is set"
  "jobserver unavailable"
  )
```

Another useful feature of the CTestCustom files is that you can use it to limit the tests that are run for memory checking dashboards. Memory checking using purify or valgrind is a CPU intensive process that can take twenty hours for a dashboard that normally takes one hour. To

help alleviate this problem CTest allows you to exclude some of the tests from the memory checking process as follows:

```
set (CTEST_CUSTOM_MEMCHECK_IGNORE
     ${CTEST_CUSTOM_MEMCHECK_IGNORE}
  TestSetGet
  otherPrint-ParaView
  Example-vtkLocal
  Example-vtkMy
  )
```

The format for excluding tests is simply a list of test names as specified when the tests were added in your CMakeLists file with `add_test`.

In addition to the demonstrated settings, such as `CTEST_CUSTOM_WARNING_MATCH`, `CTEST_CUSTOM_WARNING_EXCEPTION`, and `CTEST_CUSTOM_MEMCHECK_IGNORE`, CTest also checks several other variables.

CTEST_CUSTOM_ERROR_MATCH

Additional regular expressions to consider a build line as an error line

CTEST_CUSTOM_ERROR_EXCEPTION

Additional regular expressions to consider a build line not as an error line

CTEST_CUSTOM_WARNING_MATCH

Additional regular expressions to consider a build line as a warning line

CTEST_CUSTOM_WARNING_EXCEPTION

Additional regular expressions to consider a build line not as a warning line

CTEST_CUSTOM_MAXIMUM_NUMBER_OF_ERRORS

Maximum number of errors before CTest stops reporting errors (default 50)

CTEST_CUSTOM_MAXIMUM_NUMBER_OF_WARNINGS

Maximum number of warnings before CTest stops reporting warnings (default 50)

CTEST_CUSTOM_COVERAGE_EXCLUDE

Regular expressions for files to be excluded from the coverage analysis

CTEST_CUSTOM_PRE_MEMCHECK
CTEST_CUSTOM_POST_MEMCHECK
List of commands to execute before/after performing memory checking

CTEST_CUSTOM_MEMCHECK_IGNORE
List of tests to exclude from the memory checking step

CTEST_CUSTOM_PRE_TEST
CTEST_CUSTOM_POST_TEST
List of commands to execute before/after performing testing

CTEST_CUSTOM_TESTS_IGNORE
List of tests to exclude from the testing step

CTEST_CUSTOM_MAXIMUM_PASSED_TEST_OUTPUT_SIZE
Maximum size of test output for the passed test (default 1k)

CTEST_CUSTOM_MAXIMUM_FAILED_TEST_OUTPUT_SIZE
Maximum size of test output for the failed test (default 300k)

Commands specified in `CTEST_CUSTOM_PRE_TEST` and `CTEST_CUSTOM_POST_TEST`, as well as the equivalent memory checking ones, are executed once per CTest run. These commands can be used for example if all tests require some initial setup and cleanup to be performed.

Adding Notes to a Dashboard

CTest and CDash support adding notes to a dashboard submission. These notes will appear on the dashboard as a graphical icon that when clicked expands to the text of the notes. Using CTest you can use the –A option followed by a semicolon separated list of filenames. The contents of these files will be submitted as notes for the dashboard. For example:

```
ctest -D Continuous -A C:/MyNotes.txt;C:/OtherNotes.txt
```

Another way to submit notes with a dashboard is to copy or write the notes as files into a Notes directory under the Testing directory of your binary tree. Any files found there when CTest submits a dashboard will also be uploaded as notes.

10.9 Setting up Automated Dashboard Clients

CTest has a built-in scripting mode to help make the process of setting up dashboard clients even easier. CTest scripts will handle most of the common tasks and options that CTest –D Nightly does not. The dashboard script is written using CMake syntax and mainly involves setting up different variables or options, or creating an elaborate procedure, depending on the complexity of testing. Once you have written the script you can run the nightly dashboard on Windows, Mac OS X or UNIX systems as follows:

```
ctest -S myScript.cmake
```

First we will consider the most basic script you can use, and then we will cover the different options you can make use of. There are four variables that you must always set in your scripts. The first two variables are the names of the source and binary directories on disk, `CTEST_SOURCE_DIRECTORY` and `CTEST_BINARY_DIRECTORY`. These should be fully specified paths. The next variable, `CTEST_COMMAND`, specifies which CTest command to use for running the dashboard. This may seem a bit confusing at first. The –S option of CTest is provided to do all the setup and customization for a dashboard, but the actual running of the dashboard is done with another invocation of CTest –D. Basically once the CTest script has done what it needs to do to setup the dashboard, it invokes CTest –D to actually generate the results. You can adjust the value of `CTEST_COMMAND` to control what type of dashboard to generate (Nightly, Experimental, Continuous) as well as to pass other options to the internal CTest process such as -I ,,7 to run every 7th test. To refer to the CTest that is running the script, use the variable: `CTEST_EXECUTABLE_NAME`. The last required variable is `CTEST_CMAKE_COMMAND` which specifies the full path to the cmake executable that will be used to configure the dashboard. To refer to the CMake command that corresponds to the CTest command running the script, use the variable: `CMAKE_EXECUTABLE_NAME`. The CTest script does an initial configuration with CMake in order to generate the `CTestConfig.cmake` file that CTest will use for the dashboard. The following example demonstrates the use of these four variables and is an example of the simplest script you can have.

```
# these are the source and binary directories on disk
set (CTEST_SOURCE_DIRECTORY C:/martink/test/CMake)
set (CTEST_BINARY_DIRECTORY C:/martink/test/CMakeBin)

# which CTest command to use for running the dashboard
set (CTEST_COMMAND
  "\"${CTEST_EXECUTABLE_NAME}\" -D Nightly"
  )

# what cmake command to use for configuring this dashboard
set (CTEST_CMAKE_COMMAND
```

```
  "\"${CMAKE_EXECUTABLE_NAME}\""
  )
```

The script above is not that different to running CTest –D from the command line yourself. All it adds is that it verifies that the binary directory exists and creates it if it does not. Where CTest scripting really shines is in the optional features it supports. We will consider these options one by one, starting with one of the most commonly used options `CTEST_START_WITH_EMPTY_BINARY_DIRECTORY`. When this variable is set to true it will delete the binary directory and then recreate it as an empty directory prior to running the dashboard. This guarantees that you are testing a clean build every time the dashboard is run. To use this option you simply set it in your script. In the example above we would simply add the following lines:

```
# should CTest wipe the binary tree before running
set (CTEST_START_WITH_EMPTY_BINARY_DIRECTORY TRUE)
```

Another commonly used option is the `CTEST_INITIAL_CACHE` variable. Whatever values you set this to will be written into the CMakeCache file prior to running the dashboard. This is an effective and simple way to initialize a cache with some preset values. The syntax is the same as what is in the cache with the exception that you must escape any quotes. Consider the following example:

```
# this is the initial cache to use for the binary tree, be
# careful to escape any quotes inside of this string
set (CTEST_INITIAL_CACHE "

//Command used to build entire project from the command line.
MAKECOMMAND:STRING=\"C:/PROGRA~1/MICROS~1.NET/Common7/IDE/devenv
.com\" CMake.sln /build Debug /project ALL_BUILD

//make program
CMAKE_MAKE_PROGRAM:FILEPATH=C:/PROGRA~1/MICROS~1.NET/Common7/IDE
/devenv.com

//Name of generator.
CMAKE_GENERATOR:INTERNAL=Visual Studio 7 .NET 2003

//Path to a program.
CVSCOMMAND:FILEPATH=C:/cygwin/bin/cvs.exe

//Name of the build
BUILDNAME:STRING=Win32-vs71
```

```
//Name of the computer/site where compile is being run
SITE:STRING=DASH1.kitware

")
```

Note that the above code is basically just one `set` command setting the value of `CTEST_INITIAL_CACHE` to a multiline string value. For Windows builds these are the most common cache entries that need to be set prior to running the dashboard. The first three values control what compiler will be used to build this dashboard (Visual Studio 7.1 in this example). `CVSCOMMAND` might be found automatically, but if not it can be set here. The last two cache entries are the names that will be used to identify this dashboard submission on the dashboard.

The next two variables work together to support additional directories and projects. For example, imagine that you had a separate data directory that you needed to keep up to date with your source directory. Setting the variables `CTEST_CVS_COMMAND` and `CTEST_EXTRA_UPDATES_1` tells CTest to perform a cvs update on the specified directory, with the specified arguments prior to running the dashboard. For example:

```
# what cvs command to use for configuring this dashboard
set (CTEST_CVS_COMMAND "C:/cygwin/bin/cvs.exe")

# set any extra directories to do an update on
set (CTEST_EXTRA_UPDATES_1
     "C:/Dashboards/My Tests/VTKData" "-dAP")
```

If you have more than one directory that needs to be updated you can use `CTEST_EXTRA_UPDATES_2` through `CTEST_EXTRA_UPDATES_9` in the same manner. The next variable you can set is called `CTEST_ENVIRONMENT`. This variable consolidates several `set` commands into a single command. Setting this variable allows you to set environment variables that will be used by the process running the dashboards. You can set as many environment variables as you want using the syntax shown below.

```
# set any extra environment variables here
set (CTEST_ENVIRONMENT
  "DISPLAY=:0"
  "USE_GCC_MALLOC=1"
)
# is the same as
set (ENV{DISPLAY} ":0")
set (ENV{USE_GCC_MALLOC} "1")
```

The final general purpose option we will discuss is CTest's support for restoring a bad dashboard. In some cases you might want to make sure that you always have a working build of the software. Or you might use the resulting executables or libraries from one dashboard in the build process of another dashboard. In these cases if the first dashboard fails it is best to drop back to the last previously working dashboard. You can do this in CTest by setting `CTEST_BACKUP_AND_RESTORE` to true. When this is set to true CTest will first backup the source and binary directories, it will then check out a new source directory and create a new binary directory. After that it will run a full dashboard. If the dashboard is successful the backup directories are removed, if for some reason the new dashboard fails the new directories will be removed and the old directories restored. To make this work you must also set the `CTEST_CVS_CHECKOUT` variable. This should be set to the command required to check out your source tree. This doesn't actually have to be cvs but it must result in a source tree in the correct location. Consider the following example:

```
# do a backup and should the build fail restore,
# if this is true you must set the CTEST_CVS_CHECKOUT
# variable below.
set (CTEST_BACKUP_AND_RESTORE TRUE)

# this is the full cvs command to checkout the source dir
# this will be run from the directory above the source dir
set (CTEST_CVS_CHECKOUT
  "/usr/bin/cvs -d /cvsroot/FOO co -d FOO FOO"
)
```

Note that whatever checkout command you specify will be run from the directory above the source directory. A typical nightly dashboard client script will look like this:

```
set (CTEST_SOURCE_NAME CMake)
set (CTEST_BINARY_NAME CMake-gcc)
set (CTEST_DASHBOARD_ROOT "$ENV{HOME}/Dashboards/My Tests")

set (CTEST_SOURCE_DIRECTORY
     "${CTEST_DASHBOARD_ROOT}/${CTEST_SOURCE_NAME}")
set (CTEST_BINARY_DIRECTORY
     "${CTEST_DASHBOARD_ROOT}/${CTEST_BINARY_NAME}")

# which ctest command to use for running the dashboard
set (CTEST_COMMAND
   "\"${CTEST_EXECUTABLE_NAME}\"
   -D Nightly
   -A \"${CTEST_SCRIPT_DIRECTORY}/${CTEST_SCRIPT_NAME}\"")
```

```
# what cmake command to use for configuring this dashboard
set (CTEST_CMAKE_COMMAND "\"${CMAKE_EXECUTABLE_NAME}\"")

# should ctest wipe the binary tree before running
set (CTEST_START_WITH_EMPTY_BINARY_DIRECTORY TRUE)
# this is the initial cache to use for the binary tree
set (CTEST_INITIAL_CACHE "
SITE:STRING=midworld.kitware
BUILDNAME:STRING=DarwinG5-g++
MAKECOMMAND:STRING=make -i -j2
")

# set any extra environment variables here
set (CTEST_ENVIRONMENT
   "CC=gcc"
   "CXX=g++"
  )
```

Settings for Continuous Dashboards

The next three variables are used for setting up continuous dashboards. As mentioned earlier a continuous dashboard is designed to run continuously throughout the day providing quick feedback on the state of the software. If you are doing a continuous dashboard you can use `CTEST_CONTINUOUS_DURATION` and `CTEST_CONTINUOUS_MINIMUM_INTERVAL` to run the continuous repeatedly. The duration controls how long the script should run continuous dashboards, and the minimum interval specifies the shortest allowed time between continuous dashboards. For example, say that you want to run a continuous dashboard from 9AM until 7PM and that you want no more than one dashboard every twenty minutes. To do this you would set the duration to 600 minutes (ten hours) and the minimum interval to 20 minutes. If you run the test script at 9AM it will start a continuous dashboard. When that dashboard finishes it will check to see how much time has elapsed. If less than 20 minutes has elapsed CTest will sleep until the 20 minutes are up. If 20 or more minutes have elapsed then it will immediately start another continuous dashboard. Do not be concerned that you will end up with 300 dashboards a day (10 hours * three times an hour). If there have been no changes to the source code, CTest will not build and submit a dashboard. It will instead start waiting until the next interval is up and then check again. Using this feature just involves setting the following variables to the values you desire.

```
set (CTEST_CONTINUOUS_DURATION 600)
set (CTEST_CONTINUOUS_MINIMUM_INTERVAL 20)
```

Earlier we introduced the `CTEST_START_WITH_EMPTY_BINARY_DIRECTORY` variable that can be set to start the dashboards with an empty binary directory. If this is set to true for a

continuous dashboard then every continuous where there has been a change in the source code will result in a complete build from scratch. For larger projects this can significantly limit the number of continuous dashboards that can be generated in a day, while not using it can result in build errors or omissions because it is not a clean build. Fortunately there is a compromise, if you set `CTEST_START_WITH_EMPTY_BINARY_DIRECTORY_ONCE` to true CTest will start with a clean binary directory for the first continuous build but not subsequent ones. Based on your settings for the duration this is an easy way to start with a clean build every morning, but use existing builds for the rest of the day.

Another useful feature to use with a continuous dashboard is the –I option. A large project may have so many tests that running all the tests limits how frequently a continuous dashboard can be generated. By adding –I „7 (or –I „5 etc) to the `CTEST_COMMAND` value the continuous dashboard will only run every seventh test significantly reducing the time required between continuous dashboards. For example:

```
# which ctest command to use for running the dashboard
set (CTEST_COMMAND
  "\"${CTEST_EXECUTABLE_NAME}\" -D Continuous -I ,,7"
  )
```

As you can imagine there is a compromise to be made between the coverage of the continuous and the frequency of its updates. Depending on the size of your project and the compute resources at your disposal these variables can be used to fine tune a continuous dashboard to meet your needs. An example of a CTest script for a continuous dashboard looks like this:

```
# these are the names of the source and binary directories
set (CTEST_SOURCE_NAME CMake-cont)
set (CTEST_BINARY_NAME CMakeBCC-cont)
set (CTEST_DASHBOARD_ROOT "c:/Dashboards/My Tests")
set (CTEST_SOURCE_DIRECTORY
     "${CTEST_DASHBOARD_ROOT}/${CTEST_SOURCE_NAME}")
set (CTEST_BINARY_DIRECTORY
     "${CTEST_DASHBOARD_ROOT}/${CTEST_BINARY_NAME}")

# which ctest command to use for running the dashboard
set (CTEST_COMMAND
    "\"${CTEST_EXECUTABLE_NAME}\"
    -D Continuous
    -A \"${CTEST_SCRIPT_DIRECTORY}/${CTEST_SCRIPT_NAME}\"")

# what cmake command to use for configuring this dashboard
set (CTEST_CMAKE_COMMAND "\"${CMAKE_EXECUTABLE_NAME}\"")
```

```
# this is the initial cache to use for the binary tree
set (CTEST_INITIAL_CACHE "
SITE:STRING=dash14.kitware
BUILDNAME:STRING=Win32-bcc5.6
CMAKE_GENERATOR:INTERNAL=Borland Makefiles
CVSCOMMAND:FILEPATH=C:/Program Files/TortoiseCVS/cvs.exe
CMAKE_CXX_FLAGS:STRING=-w- -whid -waus -wpar -tWM
CMAKE_C_FLAGS:STRING=-w- -whid -waus -tWM
")

# set any extra environment variables here
set (ENV{PATH} "C:/Program
Files/Borland/CBuilder6/Bin\;C:/Program
Files/Borland/CBuilder6/Projects/Bpl"
  )
```

Variables Available in CTest Scripts

There are a few variables that will be set before your script executes. The first two variables are the directory the script is in, `CTEST_SCRIPT_DIRECTORY`, and name of the script itself `CTEST_SCRIPT_NAME`. These two variables can be used to make your scripts more portable. For example if you wanted to include the script itself as a note for the dashboard you could do the following:

```
set (CTEST_COMMAND
  "\"${CTEST_EXECUTABLE_NAME}\" -D Continuous
    -A \"${CTEST_SCRIPT_DIRECTORY}/${CTEST_SCRIPT_NAME}\""
  )
```

Another variable you can use is `CTEST_SCRIPT_ARG`. This variable can be set by providing a comma separated argument after the script name when invoking `CTest -S`. For example `CTest -S foo.cmake,21` would result in `CTEST_SCRIPT_ARG` being set to 21.

10.10 Advanced CTest Scripting

CTest scripting described in section Setting up Automated Dashboard Clients10.9 should provide support for most dashboards. That said it has some limitations that advanced users may want to circumvent. This section describes how to write CTest scripts that allow the dashboard maintainer to have much more control.

Limitations of Traditional CTest Scripting

Let us start with the limitations of traditional CTest scripting. The first limitation is that the dashboard will always fail if the Configure step fails. The reason for that is that the input files for CTest are actually generated by the Configure step. To make things worse, the update step will not happen and the dashboard will be stuck. To prevent this, an additional update step is necessary. This can be achieved by adding `CTEST_EXTRA_UPDATES_1` variable with "-D yesterday" or similar flag. This will update the repository prior to doing a dashboard. Since it will update to yesterday's time stamp, the actual update step of CTest will find the files that were modified since the previous day.

The second limitation of traditional CTest scripting is that it is not actually scripting. We only have control over what happens before the actual CTest run, but not what happens during or after. For example, if we want to run the testing and then move the binaries somewhere, or if we want to build the project, do some extra tasks and then run tests or something similar, we need to perform several complicated tasks, such as run CMake with –P option as a part of `CTEST_COMMAND`.

Extended CTest Scripting

To overcome the limitations of traditional CTest scripting, CTest provides an extended scripting mode. In this mode, the dashboard maintainer has access to individual CTest handlers. By running these handlers individually, she can develop relatively elaborate testing schemes. An example of an extended CTest script would be something like this:

```
cmake_minimum_required (VERSION 2.2)

set (CTEST_SITE          "andoria.kitware")
set (CTEST_BUILD_NAME    "Linux-g++")
set (CTEST_NOTES_FILES
     "${CTEST_SCRIPT_DIRECTORY}/${CTEST_SCRIPT_NAME}")

set (CTEST_DASHBOARD_ROOT   "$ENV{HOME}/Dashboards/My Tests")
set (CTEST_SOURCE_DIRECTORY "${CTEST_DASHBOARD_ROOT}/CMake")
set (CTEST_BINARY_DIRECTORY "${CTEST_DASHBOARD_ROOT}/CMake-gcc
")

set (CTEST_UPDATE_COMMAND    "/usr/bin/cvs")
set (CTEST_CONFIGURE_COMMAND
     "\"${CTEST_SOURCE_DIRECTORY}/bootstrap\"")
set (CTEST_BUILD_COMMAND     "/usr/bin/make -j 2")


ctest_empty_binary_directory (${CTEST_BINARY_DIRECTORY})
```

```
ctest_start (Nightly)
ctest_update (SOURCE "${CTEST_SOURCE_DIRECTORY}")
ctest_configure (BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_build (BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_test  (BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_submit ()
```

The first line is there to make sure an appropriate version of CTest is used, the advanced scripting was introduced in CTest 2.2. The CMake parser is used, and so all scriptable commands from CMake are available. This includes the `cmake_minimum_required` command:

```
cmake_minimum_required (VERSION 2.2)
```

Overall the layout of the rest of this script is similar to the traditional one. There are several settings that CTest will use to perform its tasks. Then, unlike with traditional CTest, there are the actual tasks that CTest will perform. Instead of providing information in the project's CMake cache, in this scripting mode, all the information is provided to CTest. For compatibility reasons we may choose to write the information to the cache, but that is up to the dashboard maintainer. The first block contains the variables about the submission.

```
set (CTEST_SITE              "andoria.kitware")
set (CTEST_BUILD_NAME        "Linux-g++")
set (CTEST_NOTES_FILES
     "${CTEST_SCRIPT_DIRECTORY}/${CTEST_SCRIPT_NAME}")
```

These variables serve the same role as the `SITE` and `BUILD_NAME` cache variables. They are used to identify the system once it submits the results to the dashboard. `CTEST_NOTES_FILES` is a list of files that should be submitted as the notes of the dashboard submission. This variable corresponds to the –A flag of CTest.

The second block describes the information that CTest handlers will use to perform the tasks:

```
set (CTEST_DASHBOARD_ROOT   "$ENV{HOME}/Dashboards/My Tests")
set (CTEST_SOURCE_DIRECTORY "${CTEST_DASHBOARD_ROOT}/CMake")
set (CTEST_BINARY_DIRECTORY "${CTEST_DASHBOARD_ROOT}/CMake-gcc
")
set (CTEST_UPDATE_COMMAND    "/usr/bin/cvs")
set (CTEST_CONFIGURE_COMMAND
    "\"${CTEST_SOURCE_DIRECTORY}/bootstrap\"")
set (CTEST_BUILD_COMMAND     "/usr/bin/make -j 2")
```

The CTEST_SOURCE_DIRECTORY and CTEST_BINARY_DIRECTORY serve the same purpose as in the traditional CTest script. The only difference is that we will be able to overwrite these variables later on when calling the CTest handlers. The CTEST_UPDATE_COMMAND is the path to the command used to update the source directory from the repository. Currently CTest supports Concurrent Versions System (CVS), Subversion, Git, Mercurial and Bazaar.

Both the configure and build handlers support two modes. One mode is to provide the full command that will be invoked during that stage, this is designed to support projects that do not use CMake as their configuration or build tool. In this case you specify the full command lines to configure and build your project by setting the CTEST_CONFIGURE_COMMAND and CTEST_BUILD_COMMAND variables respectively. This is similar to specifying CTEST_CMAKE_COMMAND in the traditional CTest scripting.

For projects that use CMake for their configuration and build steps you do not need to specify the command lines for configuring and building your project. Instead you will specify the CMake generator to use by setting the CTEST_CMAKE_GENERATOR variable. This way CMake will be run with the appropriate generator. One example of this is:

```
set (CTEST_CMAKE_GENERATOR "Visual Studio 8 2005")
```

For the build step you should also set the variables CTEST_PROJECT_NAME and CTEST_BUILD_CONFIGURATION to specify how to build the project. In this case CTEST_PROJECT_NAME will match the top level CMakeLists file's PROJECT command. The CTEST_BUILD_CONFIGURATION should be one of Release, Debug, MinSizeRel, or RelWithDebInfo. Additionally CTEST_BUILD_FLAGS can be provided as a hint to the build command. An example of testing for a CMake based project would be:

```
set (CTEST_CMAKE_GENERATOR "Visual Studio 8 2005")
set (CTEST_PROJECT_NAME "Grommit")
set (CTEST_BUILD_CONFIGURATION "Debug")
```

The final block performs the actual testing and submission:

```
ctest_empty_binary_directory (${CTEST_BINARY_DIRECTORY})

ctest_start (Nightly)

ctest_update (SOURCE
              "${CTEST_SOURCE_DIRECTORY}" RETURN_VALUE res)
ctest_configure (BUILD
                 "${CTEST_BINARY_DIRECTORY}" RETURN_VALUE res)
ctest_build (BUILD "${CTEST_BINARY_DIRECTORY}" RETURN_VALUE res)
```

```
ctest_test (BUILD "${CTEST_BINARY_DIRECTORY}" RETURN_VALUE res)
ctest_submit (RETURN_VALUE res)
```

The `ctest_empty_binary_directory` command empties the directory and all subdirectories. Please note that this command has a safety measure built in, which is that it will only remove the directory if there is a CMakeCache.txt file in the top level directory. This is to prevent CMake from mistakenly removing a directory.

The rest of the block contains the calls to the actual CTest handlers. Each of them corresponds to a CTest –D option. For example, instead of:

```
ctest -D ExperimentalBuild
```

The script would contain:

```
ctest_start (Experimental)
ctest_build (BUILD "${CTEST_BINARY_DIRECTORY}" RETURN_VALUE res)
```

Each step can return a return value, which can then be used to determine if the step was successful. For example the return value of the Update stage can be used in a continuous dashboard to determine if the rest of the dashboard should be run.

To demonstrate some advantages of using extended CTest scripting, let us examine a more advanced CTest script. This script drives testing of an application called Slicer. Slicer uses CMake internally, but it drives the build process through a series of Tcl scripts. One of the problems of this approach is that it does not support out-of-source builds. Also, on Windows, certain modules come pre-built, so they have to be copied to the build directory. To test a project like that, we would use a script like this:

```
cmake_minimum_required (VERSION 2.2)

# set the dashboard specific variables -- name and notes
set (CTEST_SITE                "dash11.kitware")
set (CTEST_BUILD_NAME          "Win32-VS71")
set (CTEST_NOTES_FILES
     "${CTEST_SCRIPT_DIRECTORY}/${CTEST_SCRIPT_NAME}")

# do not let the test run for more than 1500 seconds
set (CTEST_TIMEOUT "1500")

# set the source and binary directories
```

```
set (CTEST_SOURCE_DIRECTORY  "C:/Dashboards/MyTests/slicer2")
set (CTEST_BINARY_DIRECTORY  "${CTEST_SOURCE_DIRECTORY}-build)

set (SLICER_SUPPORT
     "//Dash11/Shared/Support/SlicerSupport/Lib")
set (TCLSH   "${SLICER_SUPPORT}/win32/bin/tclsh84.exe")
# set the complete update, configure and build commands
set (CTEST_UPDATE_COMMAND
     "C:/Program Files/TortoiseCVS/cvs.exe")
set (CTEST_CONFIGURE_COMMAND
    "\"${TCLSH}\"
     \"${CTEST_BINARY_DIRECTORY}/Scripts/genlib.tcl\"")
set (CTEST_BUILD_COMMAND
    "\"${TCLSH}\"
     \"${CTEST_BINARY_DIRECTORY}/Scripts/cmaker.tcl\"")

# clear out the binary tree
file (WRITE "${CTEST_BINARY_DIRECTORY}/CMakeCache.txt"
      "// Dummy cache just so that ctest will wipe binary dir")
ctest_empty_binary_directory (${CTEST_BINARY_DIRECTORY})

# special variables for the Slicer build process
set (ENV{MSVC6}              "0")
set (ENV{GENERATOR}          "Visual Studio 7 .NET 2003")
set (ENV{MAKE}               "devenv.exe ")
set (ENV{COMPILER_PATH}
     "C:/Program Files/Microsoft Visual Studio .NET
2003/Common7/Vc7/bin")
set (ENV{CVS}                "${CTEST_UPDATE_COMMAND}")

# start and update the dashboard
ctest_start (Nightly)
ctest_update (SOURCE "${CTEST_SOURCE_DIRECTORY}")

# define a macro to copy a directory
macro (COPY_DIR srcdir destdir)
  exec_program ("${CMAKE_EXECUTABLE_NAME}" ARGS
                "-E copy_directory \"${srcdir}\" \"${destdir}\"")
endmacro ()

# Slicer does not support out of source builds so we
# first copy the source directory to the binary directory
# and then build it
copy_dir ("${CTEST_SOURCE_DIRECTORY}"
          "${CTEST_BINARY_DIRECTORY}")
```

```
# copy support libraries that slicer needs into the binary tree
copy_dir ("${SLICER_SUPPORT}"
          "${CTEST_BINARY_DIRECTORY}/Lib")

# finally do the configure, build, test and submit steps
ctest_configure (BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_build (BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_test (BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_submit ()
```

With extended CTest scripting we have full control over the flow, so we can perform arbitrary commands at any point. For example, after performing an update of the project, the script copies the source tree into the build directory. This allows it to do an "out-of-source" build.

10.11 Setting up a Dashboard Server

For many people making use of Kitware's my.cdash.org dashboard hosting will be sufficient, if that is the case for you then you can skip this section. If you wish to setup your own server, then this section will walk you through the process. There are a few options for what to run on the server to process the dashboard results. The preferred option is to use CDash, a new dashboard server based on PHP, MySQL, CSS, and XSLT. Predecessors to CDash such as DART 1 and DART 2 can also be used, information on them can be found at http://www.itk.org/Dart/HTML/Index.shtml.

CDash Server

CDash is a new dashboard server developed by Kitware that is based on the common LAMP platform. It makes use of PHP, CSS, XSL, MySQL/PostgreSQL and of course your web server, normally Apache. CDash takes the dashboard submissions as XML and stores them in an SQL database (currently MySQL and PostgreSQL are supported). When the web server receives requests for pages the PHP scripts extract the relevant data from the database and produce XML that is sent to XSL templates, that in turn convert it into HTML. CSS is used to provide the overall look and feel for the pages. CDash can handle large projects ,and has been hosting up to 20 projects on a typical server (dual-core PC running Ubuntu), with about 300M records stored in the database.

Server requirements

- MySQL (5.x and higher) or PostgreSQL (8.3 and higher)
- PHP (5.0 recommended)
- XSL module for PHP (apt-get install php5-xsl)

- cURL module for PHP
- GD module for PHP

Gettting CDash

You can get CDash from the www.cdash.org website, or you can get the latest code from SVN using the following command:

```
svn co https://www.kitware.com:8443/svn/CDash/trunk CDash
```

Quick installation

1. Unzip or checkout CDash in your webroot directory on the server. Make sure the web server has read permission to the files
2. Create a `cdash/config.local.php` and add the following lines, adapted for your server configuration

```
// Hostname of the database server
$CDASH_DB_HOST = 'localhost';
// Login for database access
$CDASH_DB_LOGIN = 'root';
// Password for database access
$CDASH_DB_PASS = '';
// Name of the database
$CDASH_DB_NAME = 'cdash';
// Database type
$CDASH_DB_TYPE = 'mysql';
```

3. Point your web browser to the `install.php` script

```
http://mywebsite.com/CDash/install.php
```

4. Follow the installation instructions
5. When the installation is done, add the following line in the `config.local.php` to ensure the installation script is no longer accessible

```
$CDASH_PRODUCTION_MODE = true;
```

Testing the installation

In order to test the installation of the CDash server, you can download a small test project and test the submission to CDash, by following these steps:

1. Download the test project at

```
http://www.cdash.org/download/CDashTest.zip
```

2. Create a CDash project named "test" on your CDash server (see 10.7 Producing Test Dashboards)
3. Download the `CTestConfig.cmake` file from the CDash server
4. Run CMake on CDashTest to configure the project
5. Run

```
make experimental
```

6. Go to the dashboard page for the "test" project, you should see the submission in the experimental section.

Advanced Server Management

Project Roles: CDash supports three role levels for users:

- Normal users are regular users with read and/or write access to the project's code repository.
- Site maintainers are responsible for periodic submissions to CDash.
- Project administrators have reserved privileges to administer the project in CDash.

The first two levels can be defined by the users themselves. Project administrators access must be granted by another administrator of the project, or a CDash administrator.

In order to change the current role for a user:

1. Select [Manage project roles] in the administration section
2. If you have more than one project, select the appropriate project
3. In the "current users" section change the role for a user
4. Click "update" to update the current role
5. In order to completely remove a user from a project, click "remove"

6. If the CVS login is not correct it can be changed from this page. Note that users can also change their CVS login manually from their profile

In order to add a current role for a user:

1. Select [Manage project roles] in the administration section
2. Then, if you have more than one project, select the appropriate project
3. In the "Add new user" section type the first letters of the first name, last name or email address of the user you want to add. Or type '%' in order to show all the users registered in CDash
4. Select the appropriate user's role
5. Optionally enter the user's CVS login
6. Click on "add user"

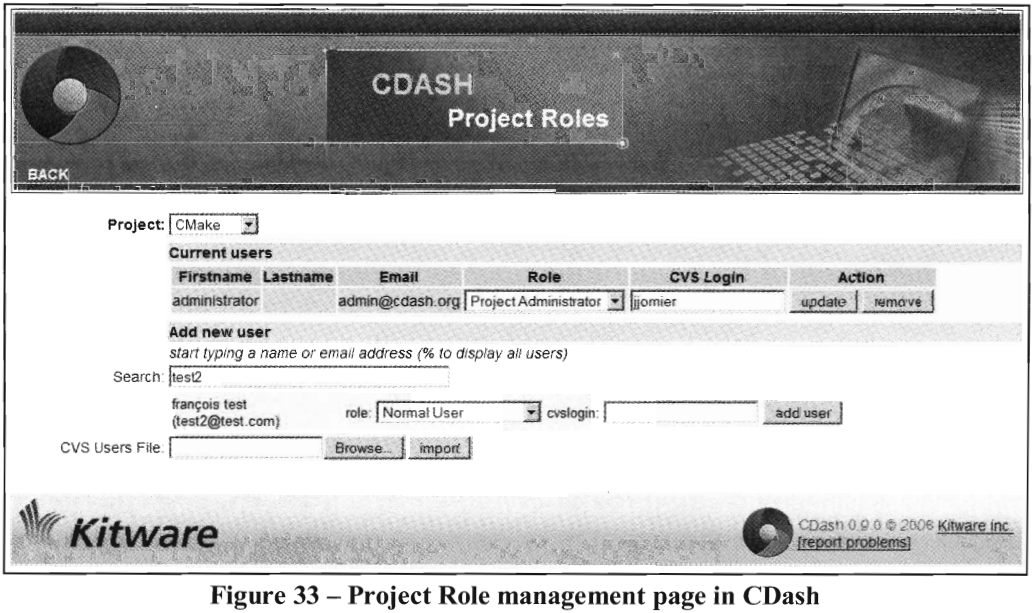

Figure 33 – Project Role management page in CDash

Importing users: to batch import a list of current users for a given project

1. Click on [manage project role] in the administration section
2. Select the appropriate project
3. Click "Browse" to select a CVS users file.
4. The current file should be formatted as follows:

```
cvsuser:email:first_name last_name
```

5. Click "import"
6. Make sure the reported names and email addresses are correct, deselect any that should not be imported
7. Click on "Register and send email". This will automatically register the users, set a random password and send a registration request to the appropriate email addresses

Google Analytics

Usage statistics of the CDash server can be assessed using Google Analytics. In order to setup google analytics:

1. Go to http://www.google.com/analytics/index.html
2. Setup an account if necessary
3. Add a website project
4. Login into CDash as the administrator of a project
5. Click on "Edit Project"
6. Add the code from Google into the Google Analytics Tracker (i.e. UA-43XXXX-X) for your project

Submission backup

CDash backups all the incoming XML submissions and places them in the `backup` directory by default. The default timeframe is 48 hours. The timeframe can be changed in the `config.local.php` as follows:

```
$CDASH_BACKUP_TIMEFRAME=72;
```

If projects are private it is recommended to set the backup directory outside of the apache root directory to make sure that nobody can access the XML files, or to add the following lines to the .htaccess in the backup directory:

```
<Files *>
order allow,deny
deny from all
</files>
```

Note that the backup directory is emptied only when a new submission arrives. If necessary, CDash can also import builds from the backup directory:

1. Log into CDash as administrator
2. Click on [Import from backups] in the administration section

3. Click on "Import backups"

Build Groups

Builds can be organized by groups. In CDash, three groups are defined automatically and cannot be removed: `Nightly`, `Continuous` and `Experimental`. These groups are the same as the ones imposed by CTest. Each group has an associated description that is displayed when clicking on the name of the group on the main dashboard.

To add a new group:

1. Click on [manage project groups] in the administration section
2. Select the appropriate project
3. Under the "create new group" section enter the name of the new group
4. Click on "create group". The newly created group appears at the bottom of the current dashboard

To order groups:

1. Click on [manage project groups] in the administration section
2. Select the appropriate project
3. Under the "Current Groups" section, click on the [up] or [down] links. The order displayed in this page is exactly the same as the order on the dashboard

To update group description:

1. Click on [manage project groups] in the administration section
2. Select the appropriate project
3. Under the "Current Groups" section, update or add a description in the field next to the [up][down] links
4. Click "Update Description" in order to commit your changes

By default a build belongs to the group associated with the build type defined by CTest, i.e. a nightly build will go in the nightly section. CDash matches a build by its name, site, and build type. For instance a nightly build named "Linux-gcc-4.3" from the site "midworld.kitware will be moved to the nightly section unless a rule on "Linux-gcc-4.3"-"midworld.kitware"-"Nightly" is defined. There are two ways to move a build into a given group by defining a rule: Global Move and Single Move.

Global move allows moving builds in batch.

1. Click on [manage project groups] in the administration section .
2. Select the appropriate project (if more than one) .

3. Under "Global Move" you will see a list of the builds submitted in the past 7 days. (without duplicates). Note that expected builds are also shown, even if they have not been submitting in the past 7 days.
4. You can narrow your search by selecting a specific group (default is All).
5. Select the builds to move. Hold "shift" in order to select multiple builds.
6. Select the target group. This is mandatory.
7. Optionally check the "expected" box if you expect the builds to be submitted on a daily basis. For more information on expected builds see the "Expected builds" section below.
8. Click "Selected Builds to Group" to move the groups.

Single move allows modifying only a particular build. From the main dashboard page, if logged in as an administrator of the project, a small folder icon is displayed next to each build. Clicking on the icon shows some options for each build. In particular, project administrators can mark a build as expected, move a build to a specific group, or delete a bogus build.

Expected builds: Project administrators can mark certain builds as expected. That means builds are expected to submit daily. This allows you to quickly check if a build has not been submitting on today's dashboard, or to quickly assess how long the build has been missing by clicking on the info icon on the main dashboard.

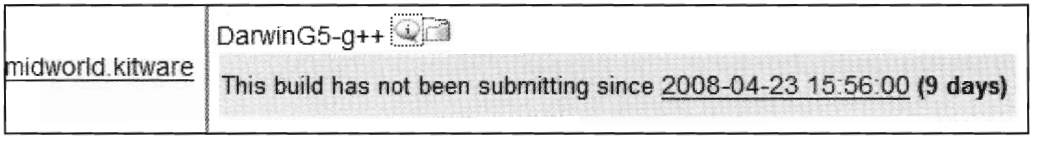

Figure 34 –Information regarding a build from the main dashboard page

If an expected build was not submited the previous day and the option "Email Build Missing" is checked for the project, an email will be sent to the site maintainer and project administrator to alert them (see the Sites section for more information).

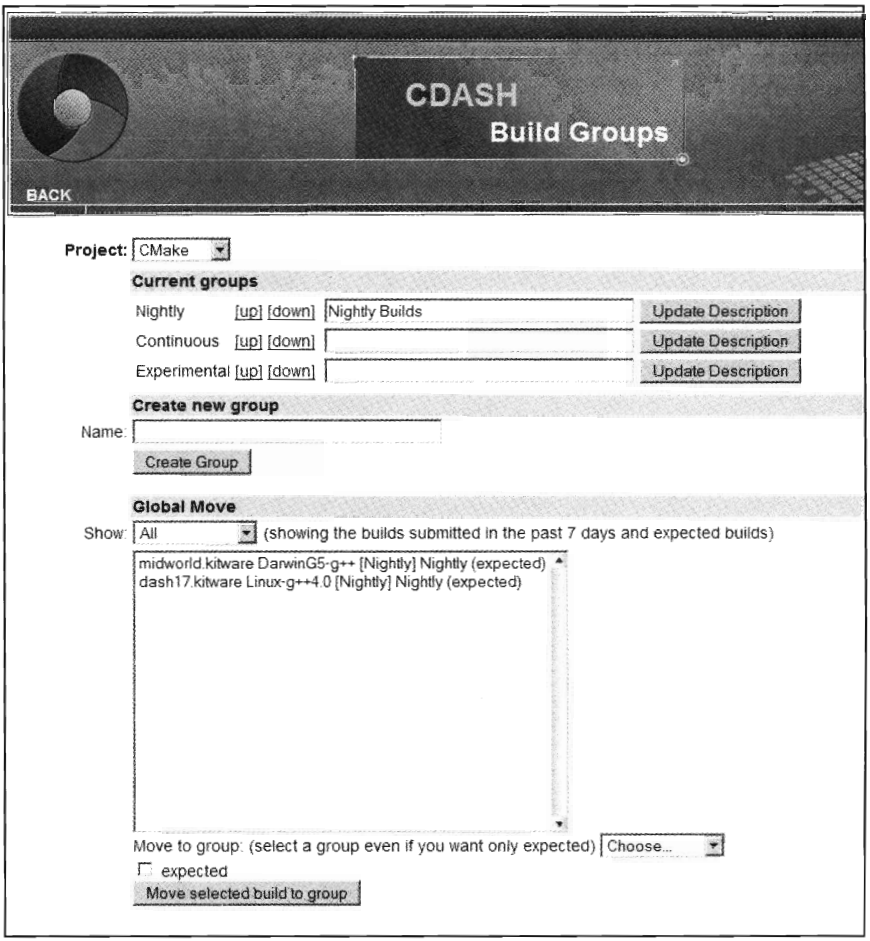

Figure 35 –Build Group Configuration Page

Email

CDash sends email to developers and project administrators when a failure occurs for a given build. The configuration of the email feature is located in three places: the `config.local.php` file, the project administration section and the project's groups section.

In the `config.local.php`, two variables are defined to specify the email address from which email is sent and the reply address. Note that the SMTP server cannot be defined in the current version of CDash, it is assumed that a local email server is running on the machine.

```
$CDASH_EMAIL_FROM = 'admin@mywebsite.com';
$CDASH_EMAIL_REPLY = 'noreply@mywebsite.com';
```

In the email configuration section of the project, several parameters can be tuned to control the email feature. These parameters were described in the previous section, "Adding CDash Support to a Project".

In the "build groups" administration section of a project, an administrator can decide if emails are sent to a specific group, or if only a summary email should be sent. The summary email is sent for a given group when at least one build is failing on the current day.

Sites

CDash refers to a site as an individual machine submitting at least one build to a given project. A site might submit multiple builds (e.g. nightly and continuous) to multiple projects stored in CDash.

In order to see the site description, click on the name of the site from the main dashboard page for a project. The description of a site includes information regarding the processor type and speed as well as the amount of memory available on the given machine. The description of a site is automatically sent by CTest, however in some cases it might be required to manually edit it. Moreover, if the machine is upgraded, i.e. the memory is upgraded; CDash keeps track of the history of the description, allowing users to compare performance before and after the upgrade.

Sites usually belong to one maintainer, responsible for the submissions to CDash. It is important for site maintainers to be warned when a site is not submitting as it could be related to a configuration issue. In order to claim a site, a maintainer should

1. Log into CDash
2. Click on a dashboard containing at least one build for the site
3. Click on the site name to open the description of the site
4. Click on [claim this site]

Once a site is claimed, its maintainer will receive emails if the client machine does not submit for an unknown reason, assuming that the site is expected to submit nightly. Furthermore, the site will appear in the "My Sites" section of the maintainer's profile, facilitating a quick check of the site's status.

Another feature of the site page is the pie chart showing the load of the machine. Assuming that a site submits to multiple projects, it is usually useful to know if the machine has room for other submissions to CDash. The pie chart gives an overview of the machine submission time for each project.

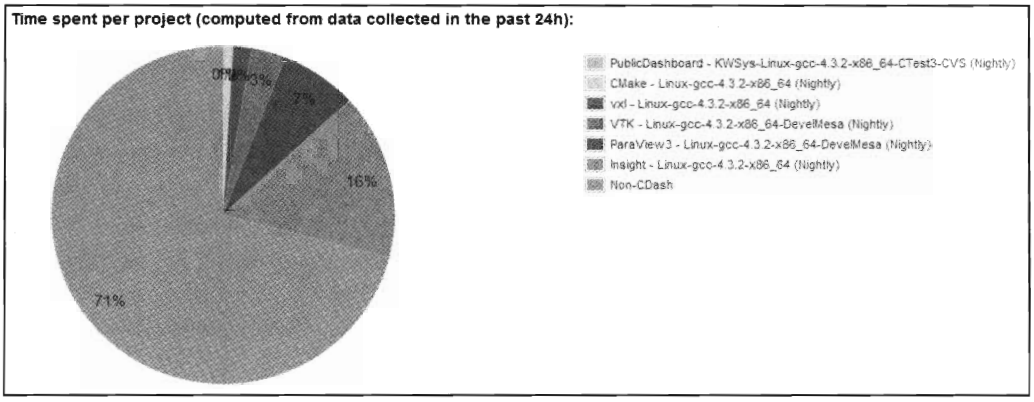

Figure 36 –Pie chart showing how much time is spent by a given site on building CDash projects

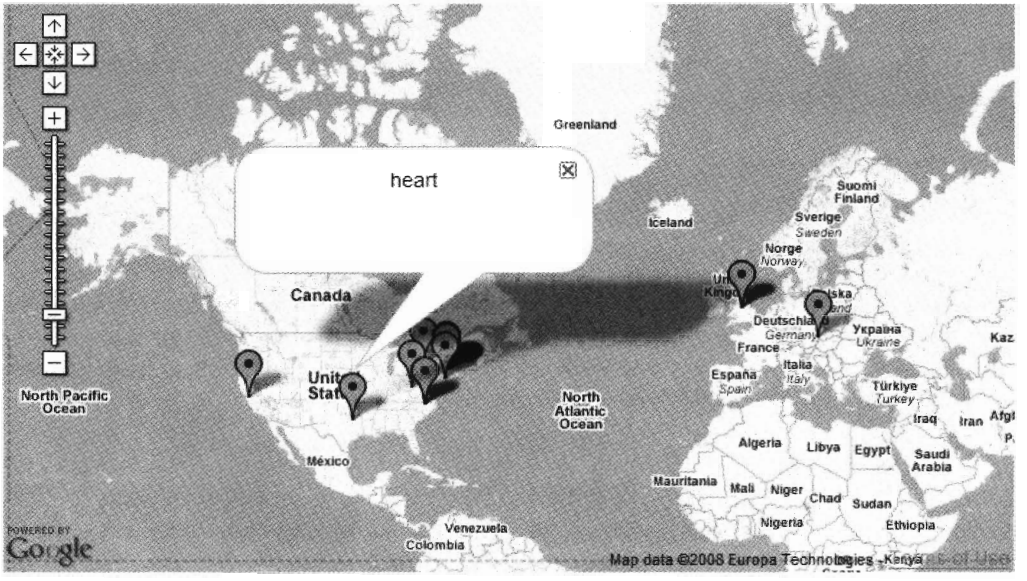

Figure 37 –Map showing the location of the different sites building a project

Graphs

CDash curently plots three types of graph. The graphs are generated dynamically from the database records and are interactive.

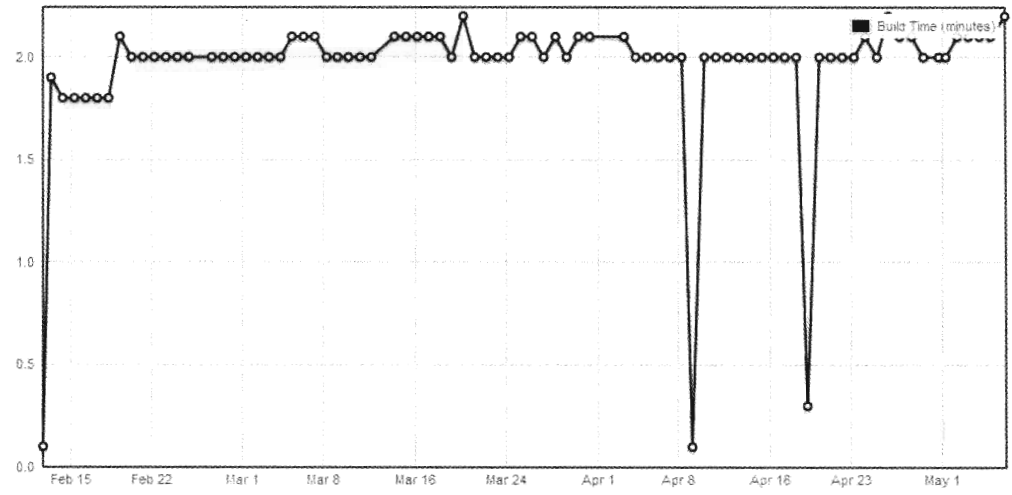

Figure 38 – Example of build time over time

The build time graph displays the time required to build a project over time. In order to display the graph you need to:

1. Go to the main dashboard for the project.
2. Click on the build name you want to track.
3. On the build summary page, click on [Show Build Time Graph].

The test time graphs display the time to run a specific test as well as its status (passed/failed) over time. To display it:

1. Go to the main dashboard for a project.
2. Click on the number of test passed or failed.
3. From the list of tests, click on the status of the test.
4. Click on [Show Test Time Graph] and/or [Show Failing/Passing Graph].

Adding Notes to a Build

In some cases, it is useful to inform other developers that someone is currently looking at the errors for a build. CDash implements a simple note mechanism for that purpose:

1. Login to CDash.
2. On the dashboard project page, click on the build name that you would like to add the note to.
3. Click on the [Add a Note to this Build] link, located next to the current build matrix (see thumbnail).

4. Enter a short message that will be added as a note.
5. Select the status of the note: Simple note, Fix in progress, Fixed.
6. Click on "Add Note".

Logging

CDash supports internal logging mechanism using the error_log() function from PHP. Most of the critical SQL queries are logged. By default the CDash log file is located in the `backup` directory under the name `cdash.log`. The location of the log file can be modified by changing the variable in the `config.local.php` configuration file.

```
$CDASH_BACKUP_DIRECTORY='/var/temp/cdashbackup/log';
```

The log file can be accessed directly from CDash if the log file is in the standard location:

1. Log into CDash as administrator.
2. Click on [CDash logs] in the administration section.
3. Click on cdash.log to see the log file.

Note that CDash does not perform any log rotation.

Test Timing

CDash supports checks on the duration of tests. CDash keeps the current weighted average of the mean and standard deviation for the time each test takes to run in the database. In order to keep the computation as efficient as possible, the following formula is used, which only involves the previous build.

```
newMean = (1-alpha)*oldMean + alpha*currentTime

newSD = sqrt((1-alpha)*SD*SD + alpha*(currentTime-newMean)*
(currentTime-newMean)
```

A test is defined as failing based on the following logic:

```
if previousSD < thresholdSD then previousSD = thresholdSD
if currentTime > previousMean+multiplier*previousSD then fail
```

It should be noted that alpha defines the current "window" for the computation. By default alpha is set to 0.3.

Mobile Support

Since CDash is written using template layers via XSLT, developing new layouts is as simple as adding new rendering templates. As a demonstration, an iPhone web template is provided with the current version of CDash.

```
http://mycdashserver/CDash/iphone
```

The main page shows a list of the public projects hosted on the server. Clicking on the name of a project loads its current dashboard. In the same manner, clicking on a given build displays more detailed information about that build. For the moment, the ability to login and to access private sections of CDash are not supported with this layout.

Figure 39 –Example of dashboard on the iPhone

Backing up CDash

All of the data (except the logs) used by CDash is stored in its database. It is important to backup the database regularly, especially so before performing a CDash upgrade. There are a couple of ways to backup a MySQL database. The easiest is to use the `mysqldump` (http://dev.mysql.com/doc/refman/5.1/en/mysqldump.html) command:

```
mysqldump -r cdashbackup.sql cdash
```

If you are using MyISAM tables exclusively you can copy the CDash directory in your MySQL data directory. Note that you need to shutdown MySQL before doing the copy so that

no file could be changed during the copy. Similarly to MySQL, PostGreSQL has a `pg_dump` utility:

```
pg_dump -U posgreSQL_user cdash > cdashbackup.sql
```

Upgrading CDash

When a new version of CDash is released or if you decide to update from the SVN repository, CDash will warn you on the front page if the current database needs to be upgraded. When upgrading to a new release version the following steps should be taken:

1. Backup your SQL database (see previous section).
2. Backup your `config.local.php` (or `config.php`) configuration files.
3. Replace your current cdash directory with the latest version and copy the `config.local.php` in the cdash directory.
4. Navigate your browser to your CDash page. (e.g.http://localhost/CDash).
5. Note the version number on the main page, it should match the version that you are upgrading to.
6. The following message may appear: "The current database shema doesn't match the version of CDash you are running, upgrade your database structure in the Administration panel of CDash." This is a helpful reminder to perform the following steps.
7. Login to CDash as administrator.
8. In the 'Administration' section, click on '[CDash Maintenance]'.
9. Click on 'Upgrade CDash': this process might take some time depending on the size of your database (do not close your browser).
    - Progress messages may appear while CDash performs the upgrade.
    - If the upgrade process takes too long you can check in the `backup/cdash.log` file to see where the process is taking a long time and/or failing.
    - It has been reported that on some systems the spinning icon never turns into a check mark. Please check the `cdash.log` for the "Upgrade done." string if you feel that the upgrade is taking too long.
    - On a 50GB database the upgrade might take up to 2 hours.
10. Some web browsers might have issues when upgrading (with some javascript variables not being passed correctly), in that case you can perform individual updates. For example, upgrading from CDash 1-2 to 1-4:

```
http://mywebsite.com/CDash/backwardCompatibilityTools.php?upgrad
e-1-4=1
```

CDash Maintenance

Database maintenance: we recommend that you perform database optimization (reindexing, purging, etc.) regularly to maintain a stable database. MySQL has a utility called `mysqlcheck`, and PostgreSQL has several utilities such as `vacuumdb`.

Deleting builds with incorrect dates: some builds might be submitted to CDash with the wrong date, either because the date in the XML file is incorrect or the timezone was not recognized by CDash (mainly by PHP). These builds will not show up in any dashboard because the start time is bogus. In order to remove these builds:

1. Login to CDash as administrator.
2. Click on [CDash maintenance] in the administration section.
3. Click on 'Delete builds with wrong start date'.

Recompute test timing: if you just upgraded CDash you might notice that the current submissions are showing a high number of failing test due to time defects. This is because CDash does not have enough sample points to compute the mean and standard deviation for each test, in particular the standard deviation might be very small (probably zero for the first few samples). You should turn the "enable test timing" off for about a week, or until you get enough build submissions and CDash has calculated an approximate mean and standard deviation for each test time.

The other option is to force CDash to compute the mean and standard deviation for each test for the past few days. Be warned that this process may take a long time, depending on the number of test and projects involved. In order to recompute the test timing:

1. Login to CDash as administrator.
2. Click on [CDash maintenance] in the administration section.
3. Specify the number of days (default is 4) to recompute the test timings for.
4. Click on "Compute test timing". When the process is done the new mean, standard deviation and status should be updated for the tests submitted during this period.

Automatic build removal

In order to keep the database at a reasonable size, CDash can automatically purge old builds. There are currently two ways to setup automatic removal of builds: without a cronjob edit the `config.local.php` and add/edit the following line

```
$CDASH_AUTOREMOVE_BUILDS='1';
```

CDash will automatically remove builds on the first submission of the day. Note that removing builds might add an extra load on the database, or slow down the current

submission process if your database is large and the number of submissions is high. If you can use a cronjob the PHP command line tool can be used to trigger build removals at a convenient time. For example, removing the builds for all the projects at 6am every Sunday:

```
0 6 * * 0 php5 /var/www/CDash/autoRemoveBuilds.php all
```

Note that the 'all' parameter can be changed to a specific project name in order to purge builds from a single project.

CDash XML Schema

The XML parsers in CDash can be easily extended to support new features. The current XML schemas generated by CTest, and their features as described in the book, are located at:

```
http://public.kitware.com/Wiki/CDash:XML
```

10.12 Subprojects

CDash (versions 1.4 and later) supports splitting projects into subprojects. Some of the subprojects may in turn depend on other subprojects. A typical real life project consists of libraries, executables, test suites, documentation, web pages and installers. Organizing your project into well-defined subprojects and presenting the results of nightly builds on a CDash dashboard can help identify where the problems are at different levels of granularity.

A project with subprojects has a different view for its top level CDash page than a project without any. It contains a summary row for the project as a whole, and then one summary row for each subproject.

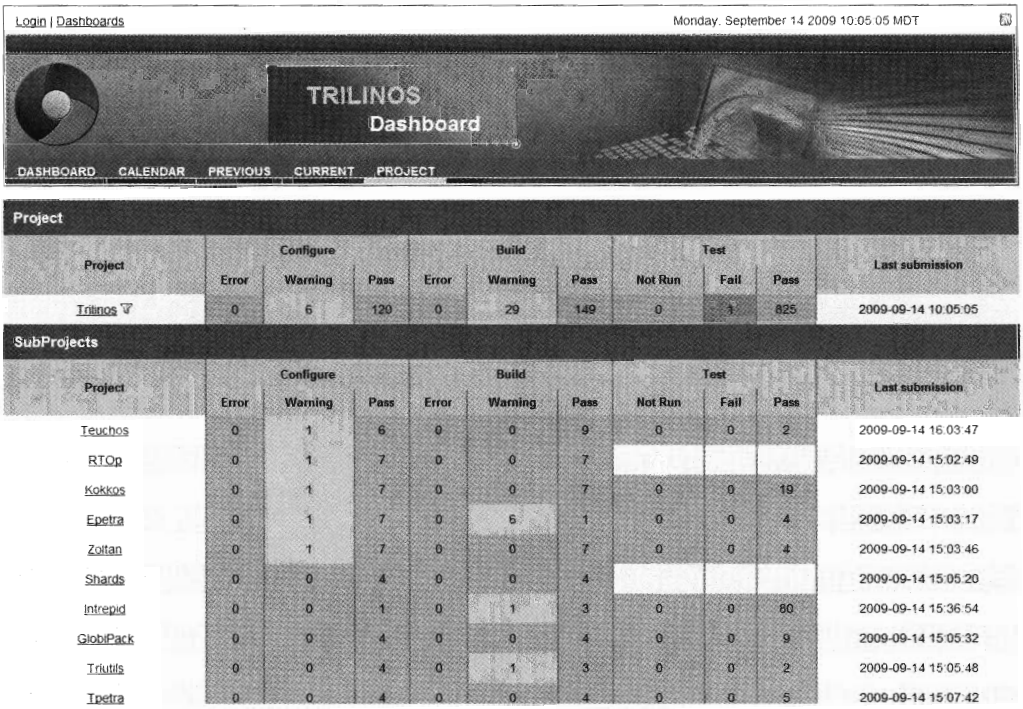

Figure 40 –Main project page with subprojects

Organizing and defining subprojects

To add subproject organization to your project, you must: (1) define the subprojects for CDash, so that it knows how to display them properly and (2) use build scripts with CTest to submit subproject builds of your project. Some (re-)organization of your project's CMakeLists.txt files may also be necessary to allow building of your project by subprojects.

There are two ways to define subprojects and their dependencies: interactively in the CDash GUI when logged in as a project administrator, or by submitting a `Project.xml` file describing the subprojects and dependencies.

Adding Subprojects Interactively

As a project administrator, a "Manage subprojects" button will appear for each of your projects on the My CDash page. Clicking the Manage Subprojects button opens the manage subproject page where you may add new subprojects, or establish dependencies between existing subprojects for any project that you are an administrator of. There are two tabs on this page, one for viewing the current subprojects along with their dependencies, and one for creating new subprojects.

To add subprojects, for instance two subprojects called Exes and Libs, and to make Exes depend on Libs, the following steps are necessary:

- Click the "Add a subproject" tab.
- Type "Exes" in the "Add a subproject" edit field.
- Click the "Add subproject" button.
- Click the "Add a subproject" tab.
- Type "Libs" in the "Add a subproject" edit field.
- Click the "Add Subproject" button.
- In the "Exes" row of the "Current Subprojects" tab, choose "Libs" from the "Add dependency" drop down list and click the "Add dependency" button.

To remove a dependency or a subproject, click on the "X" next to the item you wish to delete.

Adding Subprojects Automatically

Another way to define CDash subprojects and their dependencies is to submit a "Project.xml" file along with the usual submission files that CTest sends when it submits a build to CDash. To define the same two subprojects as in the interactive example above (Exes and Libs) with the same dependency (Exes depend on Libs) the `Project.xml` file would look like the following example:

```
<Project name="Tutorial">
  <SubProject name="Libs"></SubProject>
  <SubProject name="Exes">
    <Dependency name="Libs>
  </SubProject>
</Project>
```

Once the `Project.xml` file is written or generated, it can be submitted to CDash from a ctest -S script using the new `FILES` argument to the `ctest_submit` command, or directly from the ctest command line in a build tree configured for dashboard submission.

From inside a ctest -S script:

```
ctest_submit(FILES "${CTEST_BINARY_DIRECTORY}/Project.xml")
```

From the command line:

```
cd ../Project-build
ctest --extra-submit Project.xml
```

CDash will automatically add subprojects and dependencies according to the `Project.xml` file. CDash will also remove any subprojects or dependencies not defined in the `Project.xml` file. Additionally, if the `Project.xml` is submitted multiple times, the second and subsequent submissions will have no observable effect: the first submission adds/modifies the data, the second and later submissions send the same data, so no changes are necessary. CDash tracks changes to the subproject definitions over time to allow for projects to evolve. If you view dashboards from a past date, CDash will present the project/subproject views according to the subproject definitions in effect on that date.

Using ctest_submit with PARTS and FILES

In ctest version 2.8 and later, the `ctest_submit` command supports new `PARTS` and `FILES` arguments. With `PARTS`, you can send any subset of the xml files with each ctest_submit call. Previously, all parts would be sent with any call to `ctest_submit`. Typically, the script would wait until all dashboard stages were complete and then call `ctest_submit` once to send the results of all stages at the end of the run. Now, a script may call `ctest_submit` with `PARTS` to do partial submissions of subsets of the results. For example, you can submit configure results after `ctest_configure`, build results after `ctest_build` and test results after `ctest_test`. This allows for information to be posted as the builds progress.

With `FILES`, you can send arbitrary XML files to CDash. In addition to the standard build result XML files that CTest sends, CDash also handles the new `Project.xml` file that describes subprojects and dependencies, as described previously. Prior to the addition of the `ctest_submit PARTS` handling, a typical dashboard script would contain a single `ctest_submit()` call on its last line:

```
ctest_start (Experimental)
ctest_update (SOURCE "${CTEST_SOURCE_DIRECTORY}")
ctest_configure (BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_build (BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_test (BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_submit ()
```

Now, submissions can occur incrementally, with each part of the submission sent piecemeal as it becomes available:

```
ctest_start (Experimental)
ctest_update (SOURCE "${CTEST_SOURCE_DIRECTORY}")
ctest_configure (BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_submit (PARTS Update Configure Notes)

ctest_build (BUILD "${CTEST_BINARY_DIRECTORY}" APPEND)
ctest_submit (PARTS Build)
```

```
ctest_test (BUILD "${CTEST_BINARY_DIRECTORY}" APPEND)
ctest_submit (PARTS Test)
```

Submitting incrementally by parts means that you can inspect the results of the configure stage live on the CDash dashboard while the build is still in progress. Likewise, you can inspect the results of the build stage live while the tests are still running.

Splitting Your Project into Multiple Subprojects

One `ctest_build` invocation that builds everything followed by one `ctest_test` invocation that tests everything is sufficient for a project that has no subprojects. But if you want to submit results on a per-subproject basis to CDash, you will have to make some changes to your project and test scripts. For your project you need to identify what targets are part of what sub-projects. If you organize your CMakeLists files such that you have a target to build for each subproject, and you can derive (or look up) the name of that target based on the subproject name, then revising your script to separate it into multiple smaller configure/build/test chunks should be relatively painless. To do this you can modify your CMakeLists files in various ways depending on your needs. The most common changes are listed below.

CMakeLists.txt modifications

- Name targets the same as subprojects, or base target names on subproject names, or provide a look up mechanism to map from subproject name to target name.
- Possibly add custom targets to aggregate existing targets into subprojects, using add_dependencies to say which existing targets the custom target depends on.
- Add the LABELS target property to targets with a value of the subproject name.
- Add the LABELS test property to tests with a value of the subproject name.

Next you need to modify your CTest scripts that run your dashboards. To split your one large monolithic build into smaller subproject builds you can use a foreach loop in your CTest driver script. To help you iterate over your subprojects, CDash provides a variable named `CTEST_PROJECT_SUBPROJECTS` in `CTestConfig.cmake`. Given the above example, CDash produces a variable like this:

```
set (CTEST_PROJECT_SUBPROJECTS Libs Exes)
```

CDash orders the elements in this list such that the independent subprojects (that do not depend on any other subprojects) are first, followed by subprojects that depend only on the independent subprojects. After that subprojects that depend on those. The same logic continues until all subprojects are listed exactly once in this list in an order that makes sense for building them sequentially, one after the other.

To facilitate building just the targets associated with a subproject, use the variable named `CTEST_BUILD_TARGET` to tell `ctest_build` what to build. To facilitate running just the tests associated with a subproject, assign the `LABELS` test property to your tests and use the new `INCLUDE_LABEL` argument to ctest_test.

ctest driver script modifications

- Iterate over the subprojects in dependency order (from independent to most dependent…).
- Set the SubProject and Label global properties – CTest uses these properties to submit the results to the correct subproject on the CDash server.
- Build the target(s) for this subproject: compute the name of the target to build from the subproject name, set CTEST_BUILD_TARGET, call ctest_build.
- Run the tests for this subproject using the INCLUDE or INCLUDE_LABEL arguments to ctest_ctest.
- Use ctest_submit with the PARTS argument to submit partial results as they complete.

To illustrate this, the following example shows the changes required to split a build into smaller pieces. Assume that the subproject name is the same as the target name required to build the subproject's components. For example, here is a snippet from CMakeLists.txt, in the hypothetical Tutorial project. The only additions necessary (since the target names are the same as the subproject names) are the calls to `set_property` for each target and each test.

```
# "Libs" is the library name (therefore a target name) and
# the subproject name
add_library (Libs …)
set_property (TARGET Libs PROPERTY LABELS Libs)
add_test (LibsTest1 …)
add_test (LibsTest2 …)
set_property (TEST LibsTest1 LibsTest2 PROPERTY LABELS Libs)

# "Exes" is the executable name (therefore a target name)
# and the subproject name
add_executable (Exes …)
target_link_libraries (Exes Libs)
set_property (TARGET Exes PROPERTY LABELS Exes)
add_test (ExesTest1 …)
add_test (ExesTest2 …)
set_property (TEST ExesTest1 ExesTest2 PROPERTY LABELS Exes)
```

Here is an example of what the CTest driver script might look like before and after organizing this project into subprojects. Before the changes:

```
ctest_start (Experimental)
ctest_update (SOURCE "${CTEST_SOURCE_DIRECTORY}")
ctest_configure (BUILD "${CTEST_BINARY_DIRECTORY}")
# builds *all* targets: Libs and Exes
ctest_build (BUILD "${CTEST_BINARY_DIRECTORY}")
# runs *all* tests
ctest_test (BUILD "${CTEST_BINARY_DIRECTORY}")
# submits everything all at once at the end
ctest_submit ()
```

After the changes (new lines emphasized):

```
ctest_start (Experimental)
ctest_update (SOURCE "${CTEST_SOURCE_DIRECTORY}")
ctest_submit (PARTS Update Notes)

# to get CTEST_PROJECT_SUBPROJECTS definition:
include ("${CTEST_SOURCE_DIRECTORY}/CTestConfig.cmake")

foreach (subproject ${CTEST_PROJECT_SUBPROJECTS})
  set_property (GLOBAL PROPERTY SubProject ${subproject})
  set_property (GLOBAL PROPERTY Label ${subproject})

  ctest_configure (BUILD "${CTEST_BINARY_DIRECTORY}")
  ctest_submit (PARTS Configure)

  set (CTEST_BUILD_TARGET "${subproject}")
  ctest_build (BUILD "${CTEST_BINARY_DIRECTORY}")
    # builds target ${CTEST_BUILD_TARGET}
  ctest_submit (PARTS Build)

  ctest_test (BUILD "${CTEST_BINARY_DIRECTORY}"
    INCLUDE_LABEL "${subproject}"
  )
# runs only tests that have a LABELS property matching
# "${subproject}"
  ctest_submit (PARTS Test)
endforeach ()
```

In some projects, more than one ctest_build step may be required to build all the pieces of the subproject. For example, in Trilinos, each subproject builds the `${subproject}_libs` target and then builds the all target to build all the configured executables in the test suite. They also

configure dependencies such that only the executables that need to be built for the currently configured packages build when the all target is built.

Normally, if you submit multiple `Build.xml` files to CDash with the same exact build stamp, it will delete the existing entry and add the new entry in its place. In the case where multiple `ctest_build` steps are required, each with their own `ctest_submit(PARTS Build)` call, use the `APPEND` keyword argument in all of the `ctest_build` calls that belong together. The `APPEND` flag tells CDash to accumulate the results from multiple submissions and display the aggregation of all of them in one row on the dashboard. From CDash's perspective, multiple `ctest_build` calls (with the same build stamp and subproject and `APPEND` turned on) result in a single CDash build.

Adopt some of these tips and techniques in your favorite CMake-based project:

- `LABELS` is a new CMake/CTest property that applies to source files, targets and tests. Labels are sent to CDash inside the resulting xml files.
- Use `ctest_submit (PARTS ...)` to do incremental submissions. Results are available for viewing on the dashboards sooner.
- Use `INCLUDE_LABEL` with ctest_test to run only the tests with labels that match the regular expression.
- Use `CTEST_BUILD_TARGET` to build your subprojects one at a time, submitting subproject dashboards along the way.