Software Process for Rapid Development of HPC Software Using CMake

Bill Hoffman and David Cole
Kitware, Inc., Clifton Park, NY
{bill.hoffman, david.cole}@kitware.com

John Vines
US Army Research Laboratory (ARL), Aberdeen Proving Ground, MD
jvines@arl.army.mil

Abstract

We are developing and extending the CMake family of software development tools (www.cmake.org) for use in the Mobile Network Institute and the Multi-Scale Reactive Modeling Institute. These tools are used to build, test, and package C/C++ and FORTRAN software in a cross platform manner. By using CMake, a software project can be built just as easily on a Windows PC as on a Cray XT5 super computer. In addition, CMake's CTest facility can test and then populate the testing dashboard CDash (www.cdash.org), which is a web-based tool used to monitor and display the "health" of a software system. In combination with CTest, CDash provides a continuous integration testing system. Finally, CPack can be used to package and deploy software across multiple computing platforms.

This paper will describe these tools, how they are used in the software process, and provide specific application of their usage in support of Multi-Scale Reactive Modeling (MSRM) and the Mobile Network Modeling (MNM) High Performance Computing Software Applications Institute. The tools described in this paper are open source and available to any high performance computing (HPC) project. The paper will provide a high level overview of the CMake tools with enough specifics to enable any HPC development effort to begin working with them. We will also describe how these tools and the associated software process provide the computational infrastructure required to rapidly develop next generation HPC software.

1. Objective/Tools

This section describes the objectives of the CMake (cmake.org) project and the tools used to reach those objectives. The project aims to provide a framework for software development that provides a consistent approach to the building, testing, and packaging of C, C++, and FORTRAN high performance computing (HPC) software. Modern software requires a build system to drive the compiler and other tools that are used to create the final end user application. Once the software can be built reliably, it is essential that the software is tested on a regular basis. Finally, once the software can be built and tested, it must be packaged for use by the end users.

1.1. The CMake Build Tool

Given the heterogeneous nature of software development across Linux, Windows, Mac, and HPC platforms, the build system should not be a proprietary solution for one platform. Maintaining a separate build system for each of the supported platforms, although common in many projects, is brittle and does not encourage collaboration and code reuse. CMake is an open source build tool that has been under development since 2000, initially conceived as part of the Insight Segmentation and Registration Toolkit (itk.org) funded by the National Library of Medicine. Over the past nine years, CMake has grown in popularity and is used by well-known projects, such as KDE, ParaView, Visualization Toolkit (VTK), ITK, SecondLife, Open Scene Graph, Boost, and many others. CMake development also continues at a rapid pace due to funding from sponsors, such as the NA-MIC NIH National Center for Biomedical Computing (na-mic.org), Sandia and the Army Research Laboratory (ARL).

1.2. CMake Features

CMake aims to simplify the cross platform building of software. Instead of having a separate build file for each targeted platform as is common place in many projects, CMake needs only a single set of input files written in the CMake language to control the build for all platforms. This reduces duplication, and helps avoid the problem of software not building correctly on all of its target platforms. For example, if you look at a typical project, it might contain a set of files like this: Makefile.in, Makfile.am, Makefile.bc32, Makefile.m32, configure.in, configure, Makefile.netware, Makefile.vc8,

978-0-7695-3946-1 2010
DOI 10.1109/HPCMP-UGC.2009.62

project.dsp, project.vcproj, and possibly even more files. It is a significant burden on the project's software developers to maintain N distinct build files. At any given point in time many of those build files might be broken. In comparison, that same project could have one CMakeLists.txt file to drive the build for all if not more of the platforms. In addition, it now becomes the responsibility of the CMake community, experts in the build process, to maintain support for the platform build specifics.

CMake is different than many other build tools in that it is a "Meta" build tool. It does not actually build the software, but rather generates makefiles or project files targeted to the native build environment on each platform. This enables a project to best use the most valuable resource of any project, the human developers. Each developer typically deploys set of tools with which they are most efficient, and CMake enables a project to support a heterogeneous set of development tools. In this way, each developer can use the tools that work best for them. CMake has support for most common Integrated Development Environments (IDEs), and makefiles are supported on all platforms. The generated projects and makefiles are very efficient. CMake can generate Makefiles, nmake files, Visual Studio 6, 7, 8, and 9 projects, Xcode, Eclipse, KDevelop, CodeBlocks, msys, and cygwin makefiles.

The Makefiles that are generated by CMake include dependency analysis of the C, C++, and FORTRAN codes being built. This ensures that the software is built in the correct order. This is very important for incremental software development. FORTRAN 90 dependencies automatically make sure that files are built in the correct order so that module files are produced by the compiler before they are used.

CMake also supports the notion of "out of source" builds. This works by creating a parallel build tree to the source tree containing all of the derivative build files (e.g., object code, libraries and executables). This means that a single source tree can be built using many different compilers and compiler options in multiple, parallel build directories.

The input to CMake is a simple scripting language contained in one or more CMakeLists.txt file. The language has built in commands for common rules like adding a library or an executable, which look like this:

```
add_library(MyLib MyLib.cxx)
add_executable(MyExe MyMain.cxx)
```

CMake also allows for the easy use of large software packages from smaller ones. This following code fragment example shows a very small C++ program that uses the Boost C++ thread and signals library. This example will work on any supported platform for CMake and Boost. This feature is very important for allowing developer to quickly try and reuse existing software.

```
cmake_minimum_required(VERSION 2.6)
project(MyProject)
find package(Boost REQUIRED thread signals)
include directories(${Boost INCLUDE DIRS})
add_executable(MyExe MyProjectMain.cxx)
target_link_libraries(MyExe ${Boost_LIBRARIES})
```

CMake also supports arbitrary custom commands that can be executed at build time. This allows for the generation of code during the build process. Code generation is common in tools like Qt which has pre-processors that add code to user developed C++ code as the code is processed.

CMake can also perform system introspection and configure files with the results of the inspection of the system. This allows developers to write code to a canonical system, and avoid system specific #ifdef code. In other words, the code can be tuned to features of the current system as part of the configuration process. This allows for targets and libraries to be included based on the hardware and software inventory of the current build machine. In addition, this gives a project the ability to support R&D efforts involving new and novel computational technology including multi-core CPUs, GPUs, and Cell processors, and FPGAs.

1.3. Installing CMake

Since CMake must be installed on all machines that build the software, the CMake developers provide pre-built binaries for all major platforms at cmake.org. In addition, most Linux distributions include CMake binaries as an optional binary install. CMake can be installed and run on all supported systems without root or administrative permissions making it easy to use for any project.

1.4. Running CMake

Once initiated, CMake processes a software source tree and creates a tree of build files for the target build system. CMake can be run from either the command line, or from an interactive GUI. CMake ships with two such GUIs—one is based on Qt (Figure 1), and the other is a Curses interface (Figure 2). The interfaces enable users to select options for building the software. These options are stored in a cache file at the top of the binary tree. Once a project has been configured by CMake, the developer then uses native tools to actually build the project, and if any of the input files to CMake change, CMake will automatically regenerate the build files.

1.5. CTest/CDash Testing Tools

Although many projects include tests, those tests are many times only run by the developer as the code is

constructed. CMake ships with a tool called CTest which is used to run tests, and send test run information to the web application CDash. CDash (www.cdash.org) is a web application that can display and analyze testing data. CDash supports the several different types of builds; they can be Nightly, Experimental or Continuous.

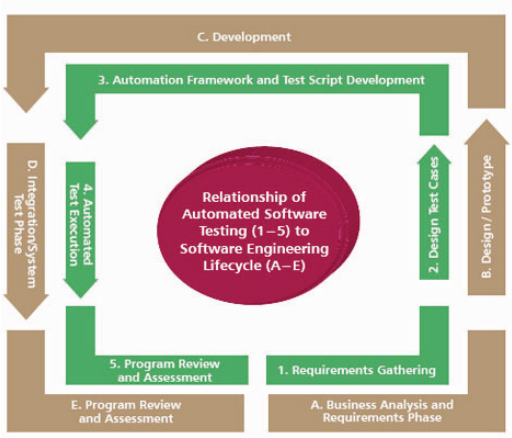

Figure 1. Automated Testing Lifecycle parallels the System Engineering Lifecycle

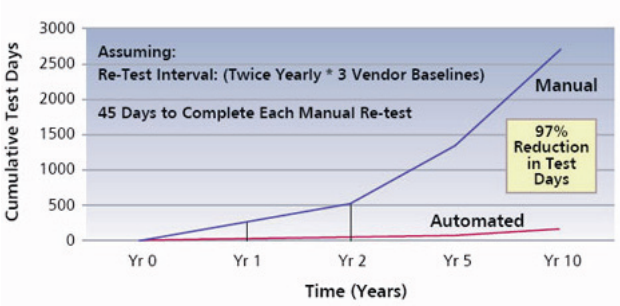

Figure 2. Example Automated Software Testing Savings over time

Adding a test to a CMake based project is as simple as calling the add_test command.

```
add_test ( testname exename arg1 arg2 arg3 …)
```

The add_test command creates a test with the given name, and runs the given executable with the specified arguments. By default, a test passes if the program returns 0, and fails for any non-zero return. Test properties can be used to make the test pass or fail based on string matching in the output of the program as well.

When a project is configured in CDash, a Nightly start time is specified. CTest then coordinates with the version control system for the project to checkout snap shots of the software at the given time each day. For example, if the project nightly start time was 9:00 pm EST, any CTest client doing a Nightly build any time in the 24 hour period after 9:00pm EST, CTest would check out and test a copy of the software from the point closest to 9:00pm EST. Developers and users can use this to gauge the health of the software. An experimental build can be any copy of the software including, local modifications by a developer. This is useful for sharing results or issues with other developers on the team. CTest can also be configured to perform Continuous tests which are run any time new code is checked into the version control repository. CDash can automatically send emails when tests, builds errors, or warnings occur alerting the team as problems occur.

1.6. The CPack Packaging Tool

In addition to building the software, the ability to package and deploy the software in a consistent manner is important to the software being used to its fullest potential. CPack takes advantage of the existing install rules in a project to create professional platform specific installers for a project. CPack can generate installer packages on Windows, Mac OSX, and Linux. To use CPack in a project, you need to set a few CPack specific variables and then include CPack as follows:

```
set (CPACK_RESOURCE_FILE_LICENSE
"${CMAKE_CURRENT_SOURCE_DIR}/License.txt")
set (CPACK PACKAGE VERSION MAJOR
"${Tutorial VERSION MAJOR}")
set (CPACK PACKAGE VERSION MINOR
"${Tutorial VERSION MINOR}")
set (CPACK_PACKAGE_CONTACT       "foo@bar.org")
set (CPACK_PACKAGE_EXECUTABLES "Tutorial"
"Tutorial" )
include (CPack)
```

This will use the projects install commands and work with CPack to create a Windows installer program, Mac OSX installer, Debian package, RPM package, a self extracting shell script, a compressed tar or zip file.

2. Methodology

The tools described in the previous section, can be easily deployed in what we have called the Kitware Quality Software Process. This process is illustrated in Figure 3. The process implements a continuous integration testing framework. It uses both CMake and CTest to automatically test software as it is checked into a version control system.

The process provides what is known as a continuous integration testing system, which makes it much easier to track down and correct software bugs as they are uncovered by the testing process. Often times in cross platform applications what works on one system breaks another system. With this process, developers can get quick feedback on what is not working on other platforms, and frequently can diagnose issues without ever having direct access to those machines. In addition it is very easy to localize which check in caused a bug. CDash enables developers to see what changes were made to the repository just prior to a test or build failure.

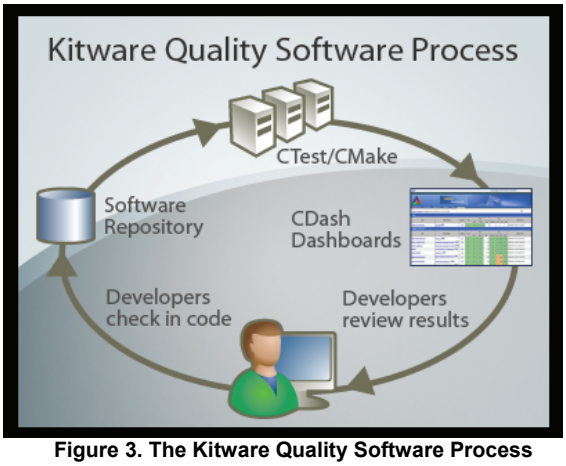

Figure 3. The Kitware Quality Software Process

3. Results

Kitware has encouraged use of this quality software process for years on the following open source projects: CMake itself, ITK (itk.org), VTK (vtk.org) and ParaView (paraview.org). The open source Trilinos project at Sandia National Labs recently migrated to a CMake based build system and is actively using CTest and CDash for the project (http://trilinos-dev.sandia.gov/cdash).

Kitware and ARL are creating a CMake build system for the Computational Science Environment (CSE). CSE is a large conglomeration of interdependent projects. They must be built in a certain order and they must be tested to validate that they work. Prior to converting to the CMake build system, no CSE developer could answer the question, "How do you know its working like it's supposed to be working?" The previous CSE build system could test each individual package after it was built, but was lacking feedback to developers committing code to the repository. Now, after setting up a CDash dashboard for CSE and adding some simple "smoke tests" in the CMake build system, a developer committing code can see the results of these tests on a continuous basis via the CSE dashboard web page. Furthermore, if a developer commits results in a build error or test failure, he will receive an email notification that there was a problem possibly caused by his commit. Now when you ask a CSE developer "How do you know its working?" he can point to the dashboard, and legitimately claim that the software is working as intended.

The CoreXMD project also has a CDash dashboard at https://arlpartners2.arl.army.mil/hsai/CDash. Clients that submit the dashboards via the "https" protocol are also located inside the secure network. These clients must be granted access to send the dashboard submissions using the "https" protocol. To enable secure transmission of dashboard results from a CTest client to a CDash server, support for the "https" protocol was added to CTest. To enable "https" submission, set CMAKE_USE_OPENSSL to ON when configuring CMake. Submission types using "scp" and "cp" are also available.

Kitware also runs a continuous dashboard for the Conceptual Model Builder (CMB) project for ERDC, the US Army Corps of Engineers Engineer Research and Development Center. The dashboard results are publicly available at https://www.kitware.com/CDash/index.php?project=CMB. CMB depends on ParaView for its GUI/visualization framework. During the development of this project, there have been several instances where a change committed to the ParaView repository introduced a problem in the CMB build or failing CMB regression tests. Because the project runs a continuous integration testing dashboard, problems caused by ParaView changes are detected and dealt with in a timely manner. When a problem (build error or failed test) occurs, the CDash system notifies the developers who committed the code. Presumably, the new code is the most likely cause of the problem. Without such a system in place, problems may not be discovered until possibly much later, perhaps interfering with a scheduled release, causing delays, wasted time, and lost opportunity.

The CMB project also runs a nightly coverage dashboard. The data reported on CMB's CDash dashboard indicates what source code files need further testing. The CMB developers are able to see what code is not being called during their test runs and add code to the test suite appropriately. Using this system night after night, one CMB developer was able to add tests to the test suite until the coverage level was increased to the point where most code in the system is being exercised every night.

ERDC has also switched to using CMake to build its Adaptive Hydrology/Hydraulics system. ADH is a modular, parallel, adaptive finite-element model for one-, two-, and three-dimensional flow and transport. The first goal of switching to a CMake environment was to ease the complexity of compiling for the Windows platform. Other benefits became immediately apparent, as well. CMake provided a simple way to delineate the source code among the different models, which include groundwater, heat transfer, two-dimensional (2D) heat transfer, Navier-Stokes, shallow water 2D, and shallow water three-dimensional (3D) portions. By separating the code, developers can work on their portions of the code and are less likely to have merging conflicts with other developers.

The abilities to avoid problems altogether, or detect problems quickly, and to measure code coverage are key attributes of the Kitware Quality Software Process. When a project combines those with the capability of communicating information effectively to the developers, via email and a web-based dashboard interface, it

becomes clear: using CDash yields better, higher quality software.

4. Significance to the DoD

The High Performance Computing Modernization Program (HPCMP) was created to modernize the Department of Defense (DoD) laboratories' HPC capabilities. As HPC capabilities evolve, the complexity of the software applications, and corresponding development process, also increases. A significant amount of time can be spent of thoroughly testing software. As stated by Hailpern and Santhanam: "… debugging, testing, and verification activities can easily range from 50 to 75 percent of the total development cost.[1]"

One recent testing improvement initiative is the establishment of a task force to improve development test and evaluation. A "Memorandum for Chairman, Defense Science Board" with the subject "Terms of Reference – Defense Science Board (DSB) Task Force on Development Test and Evaluation (DT&E)", states that "approximately 50% of programs entering Initial Operational Test and Evaluation (IOT&E) in recent years have not been evaluated as Operationally Effective or Operationally Suitable." Because of this memorandum, dated 2007, it was requested that "DSB establish a task force to examine T&E roles and responsibilities, policy and practices, and recommend changes that may contribute to improved success in IOT&E along with quicker delivery of improved capability and sustainability to Warfighters."[2]

A software testing survey (2) conducted by IDT, LLC (3), emphasized the relationship between long software test timelines and high testing time percentages relative to the rest of the software engineering process. The survey was sent to tens of thousands of engineers worldwide. Responses were compiled from around the world, 74% from the US and the other 26% from countries including India, Pakistan, Canada, South Africa, China, and Europe. Over half of the responses were from engineers working for companies with 1,000 or more employees. Nearly 50% of the surveyed engineers report that nearly 50% of time is spent of software testing in relation to overall software development, and 25% state they spent more than 50% of their time is spent on testing.

An automated software testing return on investment was undertaken for a Navy application. The application is used for communications onboard ships and to other DoD areas. Software components were developed by various vendors and delivered to the Navy labs for testing. Currently testing the millions of lines of code takes several months.

Figure 1 provides an overview of how an automated software testing lifecycle parallels the software engineering lifecycle.[3]

Figure 2 shows the initial findings: Based on the initial component testing actual results, one can project that a 97% reduction in test days would occur over the course of ten years. Implementing Automated Testing to conduct testing in new and innovative ways, while shortening the testing and certification timeline, while maintaining or improving product quality, can accomplish a significant reduction in overall software development costs.[4]

References

1. Software debugging, testing and verification by Hailpern and Santhanam, see http://www.research.ibm.com/ journal/sj/411/ hailpern.pdf, 2002.

2. http://www.surveymonkey.com/s.asp?u=267783652245.

3. http://www.idtus.com.

4. For a detailed explanation of how Automated Testing parallels the Engineering Lifecycle, see the book, *Automated Software Testing*, Dustin, et al., Addison Wesley, 1999.

5. Hendrick, D., IDC, July 2006

6. *Lessons in Test Automation*, Dustin–see http://www.stickyminds.com/s.asp?F=S5010_ MAGAZINE_62, 2001.

7. *Automated Software Testing*, Dustin, et al, Addison Wesley, 1999.

8. See http://www.opensourcetesting.com for information on various open-source testing tools.

9. Martin, Ken and Bill Hoffman, "An Open Source Approach to Developing Software in a Small Organization." *IEEE Software*, Vol. 24, Number 1. IEEE, January 2007.

10. Martin, K. and B. Hoffman, "Mastering CMake: A Cross-Platform Build System." *Kitware Inc.*, 2003

11. http://www.martinfowler.com/articles/continuous Integration .html.

12. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber =4668112.

13. For a detailed explanation please see "Mastering CMake" available at www.cmake.org.

[1] DoD Software Tech News, "The Case for Automated Software Testing"

[2] DoD Software Tech News, "The Case for Automated Software Testing, Too Much Time Spent on Testing"

[3] DoD Software Tech News, "The Case for Automated Software Testing, Return on Investment"

[4] DoD Software Tech News, "The Case for Automated Software Testing: Automated Software Testing Return on Investment"