OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE DEPARTMENT OF ENERGY

Technical Note

From: Roscoe A. Bartlett
Number:
Date: February 28, 2014

Subject:

TriBITS Overview

The Tribal Build, Integrate, and Test System (Rev. 1)

Executive Summary

The Tribal Build, Integrate, and Test System (TriBITS) is a framework built on top of the open-source CMake set of tools designed to handle large software development projects involving multiple independent development teams and multiple source repositories. TriBITS also defines a complete software development, testing, and deployment system supporting processes consistent with modern agile software development best practicies. This paper provides an overview of TriBITS that describes its roots, what problems and types of projects it is designed to address, and then describes the major archetectural design and features of the system. This presentation should provide the reader with a fairly good idea what TriBITS is, that it is not, what the advantages and disadvantages are, and what projects that use TriBITS look like.

Contents

1 Introduction

Developing, testing, and deploying complex computational software involving multiple compiled languages developed by multiple distrubuted teams of developers that must be deployed on many different platforms on an aggressive schedule is a daunting endeavor. While the modern software engineering (SE) community has made great strides in developing principles, processes and best practicies to manage such projects (e.g. agile methods [9, 8, 6, 5, 4]), it takes non-trivial tools to effectively implement such processes. In addition, just the mechanics of configuring, building, and installing complex compiled multi-language software on a variety of platforms is very challanging. Today's computational software must be able to be run on platforms ranging from basic Linux workstations and Microsoft Windows machines to the largest bleeding-edge massively parallel super computers.

TriBITS is an attempt to create an organized framework built on the Kitware CMake tools to address many of these challanges across a potentially large number of semi-independent development efforts while still allowing for seamless integration and deployment for large stacks of related software. On the low end, TriBITS can be used to quickly develop a small independent software product with all the bells and whistles of agile software development including pre- and post-test continuous integration [5, 6]. On the high end, TriBITS can be used as a meta-build system to integrate several smaller semi-independent TriBITS-enabled software projects. In addition, TriBITS directly supports the customized modern Lean/Agile research-to-production TriBITS lifecycle processes defined in [3, 2].

The scope of TriBITS and the material in this paper is the development and deployment of software written in multi-language compiled lanuages, primarily C, C++ (93, 03, 11), and Fortran (77, 90, 95, 2003, 2008) which use MPI and various local threading approaches to achieve parallel computation. Creating an effective development and deployment system for these types of software projects and languages is difficult.

TriBITS was initially developed as a package-archetecture based build system in CMake part of the Trilinos project (see Section 1.1). It was then refactored out as the new TriBITS system and adopted as the build architecture for CASL VERA [???], a larger multi-institution multi-repository code coupling project. After the initial extration of this system as TriBITS, it was quickly adopted by the ORNL CASL-related projects Denovo [???] and SCALE [???] as their native development build systems. Then TirBITS was adopted the native build system for the the new CASL-related Univeristy of Michigan code MPACT [???]. In addition to being used in CASL, all of these codes also had a significant life outside of CASL. Because they used the same TriBITS build system, it proved relatively easy to keep these various codes integrated together in the CASL VERA code meta-build. At the same time, TriBITS well served the independent development teams and non-CASL projects independent from CASL. Since the initial extraction of TriBITS from Trilinos, the TriBITS system was further extended and refined, driven by CASL VERA development and expansion. Independently, an early version of TriBITS was adopted by the LiveV project[1].

In the following subsections, more background is given in Section 1.1 on why CMake was chosen as the foundation for TriBITS which followed by a discussion in Section 1.2 of what TriBITS provides in addition to raw CMake. The introduction is concluded by a short mention and comparison with the official CMake ExteranlProject system in Section 1.3.

[1] https://github.com/lifev/cmake

1.1 Why CMake?

Many different build and test tools have been created and are avaiable in the open-source community. Many computational projects just use raw Make or GNU Make and devise their own add-on scripts to drive configuration, building, and testing. For simple projects that don't need to be very portable and only need to run on Linux, writing raw makefiles is attractive. However, for example, raw makefiles will not automatically rebuild objects files, libraries and executables when C and C++ header-files change and they will not build Fortran files in the correct order given module dependenices. Another popular set of tools used in the computational community are the so-called GNU autotools which are comprised of autoconf, automake, and related programs. Using autotools over raw (GNU) Make offers several advantages but these tools were never designed to manage the development and deployment of large complex software projects. We have worked with projects that have extensive experience using raw makefiles, autotools, and other home-grown tools based on these (e.g. [7]). There are other build and test systems as well but we will not mention those here (this paper is not meant to be an exhaustive literature review of build and test tools). We only mention raw makefiles and GNU autotools because in our observation, these are the most popular approaches currently in use on computational software projects.

Then there was CMake. What is CMake and what does it offer for these types of complex mixed-lanauge software projects? When CMake is installed (which just requires a basic C++ compiler and little else to get the key functionality), it actually provides the following tools:

- ***CMake***: Portable configuration and build manager that includes a complete scripting language for configuring and building software libraries and executables that leverages native build systems and IDEs (e.g. cmake can generate project files for Make, Ninja, MS Visual Studio, Eclipse, and XCode).
- ***CTest***: Executable to handle running tests in the shell and reporting results to CDash.
- ***CPack***: Cross-platform software packaging tool, with installer support for all systems currently supported by CMake.

In addition, ***CDash*** is a complementaory open-source build and test reporting dashboard provided by Kitware built on PHP, CSS, XSL, MySQL, and Apache HTTPD.

When the Trilinos project was first considering adopting CMake, a detailed evaluation was performed [1] comparing its usage to the existing customized Trilinos autotools system which included a home-grown perl-based test harness and dashboard. Through that evaluation and the through years of usage by the Trilinos and related projects, we have found that the major (but by no means all of the) advantages/capabilities of CMake and CTest over autotools and the existing Trilinos perl-based test harness were:

- Simplified build system and easier build and test maintenance.
- Improved mechanism for extending capabilities using simple, fast CMake scripting lanaguage (as compared to M4 in autotools).
- Built-in support for all major C, C++, and Fortran compilers.
- Automatactic built-in full dependency tracking of every kind possible on all platforms (i.e. header to object, object to library, library to executable, and any build system changes).
- Faster configure times (e.g. from minutes with autotools to seconds with CMake).
- Built-in support for shared libraries on a variety of platforms and complers.
- Built-in support for MS Windows (i.e. generates Visual Studio projects, Windows installers, etc.).
- Support for cross-compiling (i.e. build on compile nodes but run on a compute nodes).
- Built-in automatic depencency tracking for Fortran 90+ module files and object files to allow parallel Fortran compilation and linking.
- Built-in support for portable determination of C/C++/Fortran mixed-langauge bindings.
- Parallel running and scheduling of tests and test time-outs (i.e. run 1000 test executabes each using different numbers of cores effectively on 16 processes keeping all 16 processes busy).
- Built-in support for memory testing and reporting with Valgrind.

- Built-in support for line coverage testing and reporting using gcov and bulls-eye.
- Better integration between the test system and the build system (e.g. natural and flexible test specification based on configuration settings, platform consideraitons, etc.).
- Leverages open-source tools maintained and used by a large community and supported by a profession software development company (Kitware).

While there are other build and test systems with significant capabilities and advantages over raw makefiles and autotools that have been developed, the CMake set of tools is more widely used and its usage is growing in the CSE community. There is safety in numbers and the decision to adopt CMake for Trilinos has generally been considered to be a very positive development in the project and has allowed for the scalable growth of Trilinos in every way.

One drawback of CMake over autotools is that every client that needs to build and install software on a given system needs to have a minimum compable version of CMake installed first. (Raw makefiles, autotools, and CMake all require a recent version of Make installed on a Unix/Linux system.) CMake is easy to configure and install from source with just a C++ compiler and therefore CMake has no extra dependencies that any basic C++ program would have. However, building CMake from source is usually not necssary as Kitware provides public free downloads of binary installers for CMake for a variety of platforms. In general, we have found CMake to be much easier to configure and install on any given platform than the subsequent configure, build, and install any project that uses CMake. Therefore, we have found the additional overhead of having to install CMake is in the noise compared to installing the target software.

1.2 Why TriBITS?

So if CMake has so many advantages and featuers over autotools and may other alternatives, then why not just use raw CMake? While the built-in features that one gets with just the strightforward raw usage of CMake (as mentioned in Section 1.1) are signficiant, there are several problems and shortcomings with directly using only raw CMake commands in a large project. TriBITS is designed to address these problems and shortcomings and has been demonstrated to do so successfully (for at least the projects where TriBITS has been used).

At is most basic, TriBITS provides a framework for CMake-based projects that leverages all the advantages/features of raw CMake, CTest, CPack, CDash, but in addition provides the following (in relative order of significance):

- TriBITS provides a set of wrapper CMake functions and macros to reduce boiler-plate CMake code and enforces consistancy across large distributed projects.
- TriBITS provides a subproject dependency and namespacing architecture (i.e. packages with required and optional dependencies and namespaced names for tests and other targets).
- TriBITS provides additional tools to enable better and more efficient agile software development and deployment processes.
- TriBITS adds basic additional functionality missing in raw CMake.
- TriBITS changes default CMake behavior when neccesary and beneficial for a given project.

To see the potential problems and shortcoming of using raw CMake, consider the simple CMakeLists.txt file shown in Listing 1 that builds and installs a library from a set of C++ sources and then creates a user exectuable and a set of unit test exectuables and tests.

Listing 1 *: Simple example hello world raw CMakeLists.txt file*

```
# Build and install library
SET(HEADERS hello_world_lib.hpp)
```

```
SET(SOURCES hello_world_lib.cpp)
ADD_LIBRARY(hello_world_lib ${SOURCES})
INSTALL(TARGETS hello_world_lib DESTINATION lib)
INSTALL(FILES ${HEADERS} DESTINATION include)

# Build and install user executable
ADD_EXECUTABLE(hello_world hello_world_main.cpp)
TARGET_LINK_LIBRARIES(hello_world hello_world_lib)
INSTALL(TARGETS hello_world DESTINATION bin)

# Test the executable
ADD_TEST(test ${CMAKE_CURRENT_BINARY_DIR}/hello_world)
SET_TESTS_PROPERTIES(test PROPERTIES PASS_REGULAR_EXPRESSION "Hello World")

# Build and run some unit tests
ADD_EXECUTABLE(unit_tests hello_world_unit_tests.cpp)
TARGET_LINK_LIBRARIES(unit_tests hello_world_lib)
ADD_TEST(unit_test ${CMAKE_CURRENT_BINARY_DIR}/unit_tests)
SET_TESTS_PROPERTIES(unit_test
  PROPERTIES PASS_REGULAR_EXPRESSION "All unit tests passed")
```

Looking at the raw CMake code in Listing 1, it should be pretty clear what it is doing, even if one is not familar with CMake. A large project will have hundreds if not thousands of CMakeLists.txt files like this that define various libraries, executables, and tests (hopefully lots of tests if it is good software). So what is wrong with this raw CMake code? If one appraoches this from a basic software coding perspective (see [9]), there are several problems with these raw CMakeLists.txt files. First, there is a lot of duplication of the same information. For example, it duplicates the library name `hello_world_lib` four times just in this very short file. With duplication comes the opportunity to mispell names, forgetting to update the name when it changes under maintenance, etc. Also, when you define a test and set a regex to check the output, you have to list the test name at least twice; again, more duplication. Another problem is that while the library name `hello_world_lib` and the executable name `hello_world` are likely pretty well namespaced, the target names for the tests `test` and `unit_tests` are not. If another directory uses the same test names, their CMake target names will clash which is an error in CMake. (While CMake varibles are scoped based on `ADD_SUBDIRECTORY()` and `FUNCTION` commands, the targets and tests that are created are not; they must be globally unique!) The problem is that CMake provides no namespacing facility for these types of entities. It is up the the developers of a large project to come up with their own namespacing scheme for global targets, libraries, and executables.

Another set of problems with raw CMake code like in Listing 1 relates to needing to set consistent policies and change global behaviors across all CMake code inside of large projects. For example, what if an executable-only installation mode needs to be added to the project where only the created execuables and supporting scripts are installed and not the header files and libraries? How would that be accomplished with raw CMakeLists.txt files like this? In order to support that, the raw CMakeLists.txt files would need to be modifed to put in if statements around the header and library `INSTALL()` commands based on a CMake varible set by the user. However, if shared libraries are used, one must install the shared libraries as well but not so when static libraries are used. Adding this feature to a project using raw CMake commands would mean updating hundreds if not thousands of CMakeLists.txt files in a large project. Similarly, there are several other project-wide polcies and features that one would like the ability to add without having to manually modify hundreds if not thousands of raw CMakeLists.txt files.

The above example raw CMakeList.txt file in Listing 1 and the above discussion tries to demonstrate that for many large projects, using raw CMake commands has a number of disadvantages/shortcomings. That is where TriBITS comes in. To see how TriBITS helps, the equivalent TriBITS CMakeLists.txt file for this example hello-world project is given in Listing 2.

Listing 2 *: Simple example hello world TriBITS CMakeLists.txt file*

```
TRIBITS_PACKAGE(HelloWorld)
TRIBITS_ADD_LIBRARY(hello_world_lib
  HEADERS hello_world_lib.hpp SOURCES hello_world_lib.cpp)
TRIBITS_ADD_EXECUTABLE(hello_world NOEXEPREFIX SOURCES hello_world_main.cpp
  INSTALLABLE)
TRIBITS_ADD_TEST(hello_world NOEXEPREFIX PASS_REGULAR_EXPRESSION "Hello World")
TRIBITS_ADD_EXECUTABLE_AND_TEST(unit_tests SOURCES hello_world_unit_tests.cpp
  PASS_REGULAR_EXPRESSION "All unit tests passed")
TRIBITS_PACKAGE_POSTPROCESS()
```

Comparing Listing 2 and Listing 1, one can immediately see the elimination of a lot of boiler-plate CMake code and a reduction in duplication resulting in far fewer CMake commands and lines of CMake code. The macros `TRIBITS_ADD_LIBRARY()`, `TRIBITS_ADD_EXECUTABLE()`, `TRIBITS_ADD_TEST()`, `TRIBITS_ADD_EXECUTABLE_AND_TEST()` (and others that are defined by TriBITS not shown in this example) eliminate a lot of boiler-plate CMake code and reduce duplication while covering 99% of the use cases for defining libraries, executbles, and tests for the large projects where TriBITS has been applied. To briefly describe how this works, TriBITS defines packages as a unit of encapsulation and namespacing (see Section ??? for details). In the simple example in Listing 2, an entire TriBITS package is defined in a single CMakeLists.txt file. Within a TriBITS package, all executables are assumed to require the capabilities of the libraries in the given package (in this case, the HelloWorld TriBITS package). Therefore, there is no need to continually manually call `TARGET_LINK_LIBRARY()` for every executable in a package (it is called automatically). In addition, by default, the macros `TRIBITS_ADD_EXECUTABLE()`, `TRIBITS_ADD_TEST()` namespace the executable targets and the test names by the package name, so names passed into these macros only need to be unique inside of the TriBITS package, not across other TriBITS packages. (As shown, the names of executables can have the package name prefix removed by passing in `NOEXEPREFIX`.) By default, headers and libraries are automatically installed under `<install>/include/` and `<install>/lib/` which is by far the most common use case (at least for the projects where TriBITS has been applied). More details about these macros and what they do are given later in Section ???.

At the same time, these TriBITS macros provide a hook for adding additional features in a consistent way across large projects (and every project, repository, and package that uses TriBITS; more on that in Section ???). For example, adding a new policy to skip installing header files and libaries when only executables should be installed is implemented within these macros and is then applied to all packages automatically without having to change a single line of existing CMakeLists.txt files that use TriBITS. Other features that have been implemented in TriBITS inside of these basic macros include:

- Performing backward compatibility testing (see Section ???).
- Automatically calling MPI-enabled tests using the correct MPI run command and syntax.
- ???

Without having a set of basic wrapper macros around raw CMake commands, it is impossible to consistently add these types of features or enforce policies across a large CMake project.

1.3 What about CMake ExternalProject?

TriBITS is not the only set of macros that have been built on top of raw CMake to enable the building and installing of large pieces of developed software. In particular, there is the standard CMake ExteranlProject module that provides a set of macros and functions that use CMake's custom command and custom targert features to streamline the aquisition (download, SVN checkout, etc.), configure, build, and install of bits of external software (which will be refered to as third-party libraries (TPLs) here). This is used to create a "Super Build" of a set of possibily related TPL software (i.e. one TPL may depend on one or more of the

other TPLs). Each TPL can use any build system and the primary development task in setting up a super-build is get the various exteranl TPL build systems to use a consistent set of compilers, compiler options, and common TPL dependencies. In this type of system, there is little advantage to having each TPL use CMake as their native build system as all of this info has to be passed to the inner builds just as one has to for an autotools TPL build, for example. With CMake ExternalProject, the entire configure and build of one TPL must be complete before the configure and build of a downstream TPL that uses it begins. All of the steps of the acquisition, confguration, building, and installing take place at build time in the outer CMake super-build.

Because of the nature of CMake ExternalProject and the way it is typically used, it can be through more of a meta-build and deployment system for pre-developed software and is of little help in setting up and developing on CMake projects.

What differentates TriBITS is that it provides an architecture for the developed CMake projects themselves that scales to large CMake projects. TriBITS provides a development environment with all the bells and whistles of a CMake project including complete dependency tracking from between any library, executable, object file, etc. Suffice to say that TriBITS and CMake ExternalProject are mostly complementary frameworks that have a little (but not a lot) overlap. More could be said to compare and contrast TriBITS and CMake ExternalProject but that is beyond the scope of this paper.

Now that basic background on CMake and TriBITS has been presented and a basic value statement for TriBITS has been stated, the remainder of this paper seeks to provide more details about how TriBITS projects are layed out, and what various features are offered by TriBITS.

2 Overview of the TriBITS system

First and foremost, TriBITS defines an archetecture for single (large) CMake projects. In such a project, there is a single 'cmake' configure step, followed by a single build step (e.g. 'make') followed by a single test suite invocation ('ctest'). In the following subsections, the requirements for TriBITS are described

2.1 TriBITS Requirements, Architecture and Design Principles

The most important requirement for TriBITS when it was first designed to support Trilinos was to support the Trilinos concept of a **package** and support required and optional dependencies between Trilinos packages. Prior to CMake and TriBITS, it was difficult to maintain the dependency structure for packages in Trilinos. The primary goal was to provide explicit support for defining and managing Trilinos packages and to make a "package" a first-class citizen in the build, test, and deployment system.

Beyond requirements that CMake automatically satisfies, the primary extra requirements for TriBITS are:

- Make it exceedingly easy to write CMakeLists.txt files for new packages and to define libraries, tests, and examples in those packages.
- Automatically provide uniformity of how things are done and allow changes to logic and functionality that apply to all packages without having to touch each individual package CMakeLists.txt files.
- While aggregating as much common functionality as possible to the the TriBITS system and top-level project files, allow individual packages (and users) to refine the logic globally and on a package-by-package (or finer-grained) basis if needed.
- Provide 100% automatic intra-package dependencies handling. This helps to avoid mistakes, avoid duplication, and robustifies a number of important features.
- Avoid duplication of all kinds as much as possible. This is just a fundamental software maintenance issue. Raw CMakeList.txt files have a lot of duplication.

- The build system should be able to reproduce 100% update-to-date output by simply rebuilding a given target (i.e. typing 'make'). Endeavor to provide 100% correct dependency management in all situations (e.g. coping test input files to binary directory so tests can run).
- Where there is a tradeoff between extra complexity at the global framework level verses at the package level, always prefer greater complexity at the framework level where solid software engineering design principles can be applied to manage the complexity and spare package developers.
- Provide built-in automated support for as many beneficial software engineering practices a possible. This includes proper and complete pre-checkin testing when synchronous continuous integration is being performed.

These requirements were explicitly listed at the very beginning of the design of TriBITS way back in 2008 before the the precursor for TriBITS was first desgined. TriBITS supports many more features than these

2.2 TriBITS Package Dependency Handling

* Packages, subpackages, and TPLs * Enabling upstream and downstream packages and tests

2.3 Example TriBITS Project

ToDo: * Use the TribitsExampleProject to demonstrate the structure.

2.4 TriBITS Support for Multiple Source Respositories and Distributed Development Groups

3 Agile Software Development Process Support

3.1 TriBITS Project Development Workflow

3.2 TriBITS Testing and Integration

* Show diagram of differnet test categories * Describe checkin-test.py

4 Summary

ToDo: Fill in!

There are, however, a few disadvantages to using TriBITS over raw CMake that must be calked. First, using these encapsulating scripts, one can almost forget they are using CMake and can even avoid leaning CMake basics. This becomes a problem one needs to do something slightly outside of the normal 99% use case coverged by basic TriBITS usage.

References

[1] R. A. Bartlett, D. Dunlavy, and T. Shead, "Trilinos CMake evaluation," Technical report SAND2008-7593, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2008.

[2] R. A. Bartlett, M. A. Heroux, and J. M. Willenbring, "Overview of the tribits lifecycle model: A lean/agile software lifecycle model for research-based computational science and engineering software," *e-science*, vol. 2012 IEEE 8th International Conference on E-Science, pp. 1–8, 2012.

[3] R. A. Bartlett, M. A. Heroux, and J. M. Willenbring, "Tribits lifecycle model: A lean/agile software lifecycle model for research-based computational science and engineering software," Technical report SAND2012-0561, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2012.

[4] K. Beck, *Test Driven Development.* Addison Wesley, 2003.

[5] K. Beck, *Extreme Programming (Second Edition).* Addison Wesley, 2005.

[6] P. Duvall and et. al., *Continuous Integration.* Addison Wesley, 2007.

[7] M. Heroux and et. al., "An overview of the Trilinos project," *ACM TOMS*, 2005.

[8] R. Martin, *Agile Software Development (Principles, Patterns, and Practices).* Prentice Hall, 2003.

[9] S. McConnell, *Code Complete: Second Edition.* Microsoft Press, 2004.

Distribution