

Technical Note

From: Roscoe A. Bartlett

Number:

Date: June 11, 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 designed to handle large software development projects involving multiple independent development teams and multiple source repositories which is built on top of the open-source CMake set of tools. TriBITS also defines a complete software development, testing, and deployment system supporting processes consistent with modern agile software development best practices. This paper provides an overview of TriBITS that describes its roots, the types of problems and projects it is designed to address, and then describes the major architectural 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 its advantages and disadvantages are, and what a typical TriBITS project looks like.

Contents

1	Introduction	2
1.1	Why CMake?	3
1.2	Why TriBITS?	4
1.3	What about CMake ExternalProject?	7
2	Overview of the TriBITS system	7
2.1	TriBITS Requirements, Architecture and Design Principles	7
2.2	Partitioning of TriBITS into Core and Extended Functionality	8
3	TriBITS Core	9
3.1	TriBITS Package Architecture and Dependency Handling	10
3.2	Example TriBITS Project	10
3.3	TriBITS Support for Multiple Source Repositories and Distributed Development Groups	10
4	Extended TriBITS	10
4.1	Agile Software Development Process Support	10
4.2	TriBITS Testing and Integration	10
4.3	TriBITS Project Development Workflow	10

1 Introduction

Developing, testing, and deploying complex computational software involving multiple compiled languages created by multiple distributed 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 practices to manage such projects (e.g. agile methods [10, 9, 7, 6, 5]), 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 challenging. 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 top of (and implemented using) the Kitware CMake tools [??] to address many of these challenges across a potentially large number of semi-independent development efforts while still allowing for seamless integration and deployment for large stacks of related software. At 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-push and post-push continuous integration [6, 7]. At 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 [4, 3].

The scope of TriBITS, and the material in this paper, is focused on the development and deployment of software written using primarily the compiled languages C, C++ (93, 03, 11), and Fortran (77, 90, 95, 2003, 2008) (and mixed language programs involving these) 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 very difficult.

TriBITS was initially developed as a package-based architecture build and test system for the Trilinos project (see Section 1.1). This system was later factored out of Trilinos as the reusable TriBITS system and adopted as the build architecture for the Consortium for the Advanced Simulation of Light Water Reactors (CASL) VERA code [??], a larger multi-institution, multi-repository development project. Since the initial extraction of TriBITS from Trilinos, the TriBITS system was further extended and refined, driven by CASL VERA development and expansion. After the initial extraction of TriBITS, it was quickly adopted by the ORNL CASL-related projects Denovo [??] and SCALE [??] as their native development build systems. TriBITS was then adopted as the native build system for the new CASL-related University of Michigan code MPACT [1]. Because these different repositories used the same TriBITS build system, it proved relatively easy to keep these various codes integrated together in the CASL VERA code meta-build. In addition to being used in CASL, all of these codes also have a significant life outside of CASL. TriBITS additionally well served these independent development teams and non-CASL projects apart from CASL. Unrelated to CASL, an early version of TriBITS was adopted by the LiveV project¹.

The remainder of this section is broken down as follows. More background is given in Section 1.1 on why CMake was chosen as the foundation for TriBITS which is followed by a discussion of what TriBITS provides in addition to raw CMake in Section 1.2. The introduction is concluded by a short mention and

¹<https://github.com/lifev/cmake>

comparison with TriBITS with the standard CMake ExternalProject module in Section 1.3.

1.1 Why CMake?

Many different build and test tools have been created and are available in the open-source community. Many computational science 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 object files, libraries and executables when C and C++ header-files change and they will not build Fortran files in the correct order given Fortran module dependencies. 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. [8]). 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 in computational software projects.

Then there was CMake. What is CMake and what does it offer for these types of complex mixed-language 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 (source and/or binary) packaging tool, with installer support for all systems currently supported by CMake.

In addition, **CDash** is a compatible free open-source build and test reporting web-based dashboard tool 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 [2] 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 or key capabilities of CMake and CTest over autotools and the old Trilinos perl-based test harness are/were:

- Simplified build system and easier build and test maintenance.
- Improved mechanism for extending capabilities using simple, fast CMake scripting language (as compared to M4 in autotools).
- Better built-in support for all major C, C++, and Fortran compilers.
- Automatic 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).
- Built-in automatic dependency tracking for Fortran 90+ module files and object files to allow parallel Fortran compilation and linking.
- Built-in support for shared libraries on a variety of platforms and compilers.
- Built-in support for MS Windows (i.e. generates Visual Studio projects, NMake files, Windows installers, etc.).
- Support for cross-compiling (i.e. build on compile nodes, run on compute nodes).
- Built-in support for portable determination of C/C++/Fortran mixed-language bindings.
- Faster configure times (e.g. from minutes with autotools to seconds with CMake).

- Parallel running and scheduling of tests and test time-outs (i.e. run 1000 test executables 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 with optional submitting to CDash.
- 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 considerations, etc.).
- Leverages open-source tools maintained and used by a large community and supported by a professional software development company (Kitware).

While there are other build and test systems that have been developed with significant capabilities and advantages over raw makefiles and autotools, the CMake set of tools is more widely used and its usage is growing in the CSE community. While CMake is not perfect, 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 on many fronts.

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 compatible 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 beyond what any basic C++ software package would have. However, building CMake from source is usually not necessary as Kitware provides free public 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 of 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 project that uses CMake.

1.2 Why TriBITS?

So if CMake has so many advantages and features over autotools and may other common alternatives, then why not just use raw CMake for all CSE projects? While the built-in features that one gets with just the straightforward raw usage of CMake (as mentioned in Section 1.1) are significant, 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 its 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 additional features (in relative order of significance):

- TriBITS provides a set of wrapper CMake functions and macros to reduce boiler-plate CMake code and enforces consistency 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 some basic additional functionality missing in raw CMake.
- TriBITS changes default CMake behavior when necessary and beneficial for a given project (or set of related projects) in a consistent way.

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 executable and a unit test executable 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 familiar 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 approaches this from a basic software coding perspective (see [10]), 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 misspell 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 variables 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 to 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 executables 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 modified to put in if statements around the header and library `INSTALL()` commands based on a CMake variable 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 policies 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 `CMakeLists.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. At the same time, these macros cover 99% of the use cases for defining libraries, executables, 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 inside of `TRIBITS_ADD_EXECUTABLE()`). In addition, by default, the macros `TRIBITS_ADD_EXECUTABLE()` and `TRIBITS_ADD_TEST()` namespace the executable targets and the test names by the package name, so the names passed into these functions 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 libraries 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 inside 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/functions around raw CMake commands, it is impossible to consistently add these types of features or enforce policies across a large (distributed) CMake project.

1.3 What about CMake ExternalProject?

TriBITS is not the only set of macros/functions that has 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 ExternalProject module that provides a set of macros and functions that use CMake's custom command and custom target features to streamline the acquisition (download, SVN checkout, etc.), configure, build, test, and install of bits of external software (which will be referred to here as third-party libraries (TPLs)). This is used to create a "Super Build" of a set of possibly 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 to get the various external 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 downstream TPLs that uses it can begin. All of the steps of the acquisition, configuration, 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 viewed more as meta-build and deployment system for pre-developed software and is not as attractive to use to develop on an aggregate set of related software projects that use CMake as their build/test system.

What differentiates TriBITS from CMake ExternalProject is that TriBITS provides an architecture for the development of the different CMake projects themselves that scales to large CMake projects. TriBITS provides a development environment with all the bells and whistles of a single CMake project including complete dependency tracking from between any library, executable, object file, etc. TriBITS and CMake ExternalProject are mostly complementary frameworks that have a little (but not a lot) of 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 laid out, and what various features are offered by TriBITS.

2 Overview of the TriBITS system

First and foremost, TriBITS defines an architecture and framework for single (large, partitioned) 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. **ToDo:** Finish summary of subsections.

2.1 TriBITS Requirements, Architecture and Design Principles

The most important requirement for TriBITS when it was first designed for 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 for the new CMake-based build system for 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 (see Section 1.1), 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 individual `CMakeLists.txt` files.
- While aggregating as much common functionality as possible into 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 `CMakeLists.txt` files have a lot of duplication (as demonstrated in Listing 1).
- 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 trade-off 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 as possible. This includes proper and complete pre-push 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 precursor for TriBITS was first designed. TriBITS supports many more features than these and some of these features are given later in this paper but are given in more detail in ???.

ToDo: Edited up to here on 5/20/2014!

2.2 Partitioning of TriBITS into Core and Extended Functionality

TriBITS is divided into a few different bits of functionality that can be adopted incrementally (or not at all). With each part, additional functionality is provided but at the cost of additional external dependencies and less flexibility. TriBITS is divided into the follow major pieces, listed in order of increasing functionality and dependency:

- *TriBITS Core*: Basic CMake package-based architecture, configure, build, test, install, distribution support, etc. This part only depends on raw CMake with no other dependencies (i.e. no Python or git dependence).
- *TriBITS Basic Continuous Integration Support*: Consists of the `checkin-test.py` script which requires Python 2.4+ and git 1.7.0.4+.
- *TriBITS Automated Package-based CTest/CDash Testing*: Additional CMake/CTest and python code to support the creation of `ctest -S` scripts that do package-by-package configure/build/test, submissions, and display on CDash.

ToDo: Show figure of the different parts of TriBITS and how they depend on each other and what they depend on externally.

These different pieces of TriBITS are shown in Figure ??? and are described in a little more detail below and some parts are described in even more detail in following sections. One can depend-on/extract/snapshot only the parts of TriBITS that are desired for a given TriBITS project or repository. The extended TriBITS functionality is only applicable to a full TriBITS project, not to individual TriBITS Repositories or Packages. The partitions of TriBITS are described in more detail below.

TriBITS Core is set of CMake modules which is used to construct CMake projects and provides the basic package-based architecture. This is used to configure, build, test, install, and deploy software that

uses TriBITS/CMake. This part of TriBITS only depends on raw CMake/CTest and is the only part that is used by customers of the software in installation and deployment. This is the first part of TriBITS that every project has to adopt in order to use any part of the TriBITS system. This is the part of TriBITS that allows for the seamless configuration, build, and testing of different integrated (but independent) software development efforts. However, this basic framework does not assume any software development process at all or make any assumptions about the version control tools or processes used. There should only be small learning curve to apply this framework for a smaller, simpler CMake project compared to just using raw CMake/CTest. This “core” part of TriBITS is described in more detail in Section 3.

In addition to the core package-based TriBITS framework (i.e. “TriBITS Core”) described in Section 3, TriBITS also provides additional extended functionality that is needed to flesh out a fully functional Agile software development process or to support sophisticated multi-group, multi-repository integration processes. These different pieces can be adopted in phases or not at all. The usage of each additional piece requires extra external dependencies (e.g. Python 2.4+, git 1.7.0.4+, etc.), imposes more assumptions on the software development process or tools (e.g. assumes all VC repos use git), and involves an additional learning curve to use correctly. Each of these additional pieces (or feature sets) of TriBITS are described briefly below and some in some more detail in later sections.

TriBITS Basic Continuous Integration Support is essentially provided in a single python script called `checkin-test.py` (and a set of python modules that it uses). The `checkin-test.py` script implements robust pre-push continuous integration and can be used to implement light-weight post-push continuous integration and nightly testing processes that rely on email notifications. Almost every software development project that uses TriBITS should consider using the `checkin-test.py` tool to implement pre-push CI testing for the project. At the high end, the `checkin-test.py` tool has all the functionality needed to implement sophisticated multi-repository integration processes that take only minutes-to-hours to set up and maintain (verses weeks-to-months of developer time to set up and support integration processes for arbitrary software projects). The `checkin-test.py` tool requires that Python 2.4+ and git 1.7.0.4+ be installed on the system and it requires all of the VC repos to use git (or be snapshotted into git repos).

TriBITS Automated Package-based CTest/CDash Testing support exploits the package-based architecture of a TriBITS CMake project to provide incremental testing for a large project with targeted regression email addresses and CDash display on a package-by-package basis. This extra functionality requires Python 2.4+ and is implemented in the CMake/CTest function `TRIBITS_CTEST_DRIVER()`. It also requires, of course, the setup of a CDash server (where CDash has a number of external dependencies such as MySQL, PHP, etc.). However, a project that uses TriBITS can just use raw CTest and CDash to test the project as whole that but does not take advantage of the package-based architecture. ToDo: Put link to later section on the CTest/CDash server.

3 TriBITS Core

The “Core” of TriBITS is set of CMake modules which provides a basic package-based architecture for a CMake project. This is used to configure, build, test, install, and deploy software that uses TriBITS/CMake. The basic architectural components for a TriBITS project are:

- ***TriBITS Package***: A collection of related software that typically includes one or more source files built into one or more libraries and/or executables and has associated tests to help define and protect the functionality provided.
- ***TriBITS TPL***: The specification for a particular external dependency that is required or can be used in one or more *TriBITS Packages*. A TPL (Third Party Library) typically provides a list of libraries or a list include directories for header files but can also be manifested in other ways as well.
- ***TriBITS Repository***: A collection of one or more *TriBITS Packages* and zero or more TriBITS TPL specifications referenced in those packages.

- **TriBITS Project:** A collection of one or more *TriBITS Repositories* and *TriBITS Packages* which constitutes a complete CMake “PROJECT“ defining software which can be directly configured with `cmake` and then be built, tested, installed, etc.

Many TriBITS CMake projects like Trilinos only involve a single TriBITS Repository. More complex multi-repository TriBITS projects like CASL VERA are described in Section ??? Every TriBITS project must define at least one package and most TriBITS projects have at least one (perhaps optional) external dependency such as a library like BLAS or LAPACK and therefore will specify at least one TriBITS TPL.

3.1 TriBITS Package Architecture and Dependency Handling

- * Packages, subpackages, and TPLs
- * Use ReducedMockTrilinos as the example for dependency handling.
- * Enabling upstream and downstream packages and tests

3.2 Example TriBITS Project

ToDo: * Use the TribitsExampleProject to demonstrate the structure.

3.3 TriBITS Support for Multiple Source Repositories and Distributed Development Groups

4 Extended TriBITS

4.1 Agile Software Development Process Support

In addition to the “core” of TriBITS which provides the basic CMake project package-based architecture, TriBITS also includes a number of tools to support Agile software development.

4.2 TriBITS Testing and Integration

4.3 TriBITS Project Development Workflow

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

5 Summary

ToDo: Fill in!

There are, however, a few disadvantages to using TriBITS over raw CMake that must be considered. First, using the encapsulating TriBITS functions, one can almost forget that one is using CMake and can even avoid leaning CMake basics. This becomes a problem when one needs to do something slightly outside of the normal 99% use case covered by the basic brain-dead usage of TriBITS. Second, TriBITS, as a framework, provides a lot of functionality at low cost for the supported use cases but inherently limits flexibility. This restricted flexibility is an advantage when it one tries to integrate large collections of independently developed software. However, if a project is not going to be integrated with other TriBITS-based software, some of the lack of flexibility can be a hindrance. Also, if a project is going to stay small and the needs of the project are well served by raw CMake/CTest/CPack, then the extra dependency

on TriBITS may not be worth the added functionality. TriBITS is really designed to construct large CMake projects and manage the integration between multiple independent by related CMake projects.

References

- [1] *Overview of development and design of MPACT: Michigan parallel characteristics transport code*, vol. 1 of *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering*, Elsevier, 2013.
- [2] 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.
- [3] 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.
- [4] 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.
- [5] K. BECK, *Test Driven Development*. Addison Wesley, 2003.
- [6] K. BECK, *Extreme Programming (Second Edition)*. Addison Wesley, 2005.
- [7] P. DUVAL and ET. AL., *Continuous Integration*. Addison Wesley, 2007.
- [8] M. HEROUX and ET. AL., “An overview of the Trilinos project,” *ACM TOMS*, 2005.
- [9] R. MARTIN, *Agile Software Development (Principles, Patterns, and Practices)*. Prentice Hall, 2003.
- [10] S. MCCONNELL, *Code Complete: Second Edition*. Microsoft Press, 2004.

Distribution

CAUTION

This document has not been given final patent clearance and is for internal use only. If this document is to be given public release, it must be cleared through the site Technical Information Office, which will see that the proper patent and technical information reviews are completed in accordance with the policies of Oak Ridge National Laboratory and UT-Battelle, LLC.