SAND2020-2718C

Componentized hieratical build and test infrastructure and processes for CASL VERA[1]

Roscoe A. Bartlett,*

Software Engineering and Research, Sandia National Laboratories, P.O. Box 5800, Albuquerque, NM, rabartl@sandia.gov

INTRODUCTION

Developing, testing, and deploying complex computational science and engineering (CSE) software created by distributed teams is a daunting endeavor. This was well demonstrated in the Consortium of the Advanced Simulation of Lightwater reactors (CASL) project. The CASL Virtual Environment for Reactor Applications (VERA) software [1] integrates the development efforts of many different teams, code bases, and institutions into multi-physics executables. The software engineering and process challenges to achieve an integrated product that is continuously upgraded with new contributions are enormous. While the modern software engineering (SE) community has made great strides in developing principles, processes and best practices to manage such projects [2, 3], 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 CSE 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 supercomputers.

In order to address these challenges, a framework called the Tribal Build, Integration, and Test System (TriBITS) was constructed and applied to VERA. TriBITS creates an organized framework built on top of (and implemented using) the Kitware CMake [4] tools to handle 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 [5, 3]. At the high end (such as with VERA), TriBITS can be used as a meta-build, testing and deployment system to integrate several smaller semi-independent TriBITS-enabled software projects. TriBITS is focused on the development and deployment of software written using primarily the compiled languages C, C++, Fortran – and mixed-language programs involving these – which use MPI and various local threading approaches to achieve performant parallel computation.

TriBITS was initially developed as a package-based architecture build and test system for the Trilinos [6] project. This system was later factored out of Trilinos as the reusable TriBITS system and adopted as the build architecture for VERA. Since the initial extraction of TriBITS from Trilinos, the TriBITS system was further extended and refined; driven by VERA development and expansion. After the initial extraction of TriBITS, it was quickly adopted by several CASL-related projects including SCALE [1] at ORNL, MPACT [1] from the University of Michigan, and COBRA-TF [1] from the Pennsylvania State University. Because these different repositories used the same TriBITS build system, it proved relatively easy to keep these various codes integrated together in the VERA TriBITS 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.

Why CMake?

Many different build and test tools have been created and are available in the open-source community. Many CSE 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, 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 CSE 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.

Enter CMake. When CMake is installed (which just requires a basic C++ compiler when building CMake from source to get the key functionality), it provides the executable tools `cmake`, `ctest` and `cpack`. ***CMake*** is a portable configuration and build manager that includes a complete scripting language. CMake configures software for building libraries, executables, and other targets that leverages native build systems and IDEs (e.g. CMake can generate build/project files for Make, Ninja, MS Visual Studio, Eclipse, XCode, and more). ***CTest*** is a tool to handle running tests efficiently in parallel and reporting results locally and (optionally) also to a CDash server. ***CPack*** is a cross-platform source and/or binary packaging tool, with installer support for many different systems. In addition, Kitware also provides ***CDash*** which is a CTest-compatible free open-source web-based dashboard tool provided by Kitware built on PHP, CSS, XSL, MySQL,

[1]This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

and Apache HTTPD. CDash defines various view and query mechanisms that can be used to review build and test data over time. Together, these tools can be used to create a very effective build, test, and deployment process for a complex software project like VERA.

While CMake, CTest, and CPack include numerous features (see [4]), some of the more significant features that make this set of tools attractive for CSE projects include:

- Built-in support for all major C, C++, and Fortran compilers for all major vendors and implementations including GNU, Clang, Intel, PGI, IBM, and Cray.
- Automatic built-in full dependency tracking of every kind possible on all platforms (header file to object file, object file to library, library to executable, and any build system changes to impacted build targets, etc.).
- Built-in automatic dependency tracking for Fortran 90+ module files and object files to allow robust parallel compilation and linking (e.g. using the Ninja build tool).
- Built-in support for shared libraries and library versioning on a variety of platforms and compilers.
- Built-in support for MS Windows (Visual Studio projects, NMake files, Windows installers, etc.) and OSX (bundles, frameworks, code signing, etc.).
- Support for cross-compiling (i.e. build on a compile node and then run on a compute node which has a different CPU architecture).
- Built-in support for portable determination of C/C++/Fortran mixed-language bindings.
- Parallel running and scheduling of tests and robust test time-outs (e.g. schedule 1000s of MPI tests to run on a 64-code node – where each test uses a different number of MPI processes and threads per MPI processes – while keeping all 64 cores busy without overloading the node).
- Built-in support for memory testing (Valgrind, Clang Address Sanitizer, etc.) and line coverage testing (gcov, bulls-eye, etc.) and submitting results to CDash.

In recent years, the adoption of CMake has greatly increased in the broader community, and CMake is now a dominant build configuration system in many communities, including (increasingly) in the CSE community.

Why TriBITS?

While the built-in features that one gets with the straightforward usage of CMake are significant, there are several problems and shortcomings with directly using only raw CMake commands in a large project. Problems include lots of duplicate low-level details, poor management of intra-project optional dependencies, and other issues that will be highlighed below. TriBITS is designed to address these problems and shortcomings and has been demonstrated to do so successfully (at least in the context of CASL VERA).

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 enforce consistency across large distributed CMake projects.
- TriBITS provides a subproject dependency and namespacing architecture (i.e. a package architecture) with required and optional dependencies and namespaced identifiers for tests and other global 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.

OVERVIEW OF THE TRIBITS FRAMEWORK

Primarily, TriBITS defines an architecture and framework for large structured CMake projects. In such a project, there is a single `cmake` configure step, followed by a single `make -j<N>` build step, followed by a single `ctest -j<N>` test suite invocation. A single CMake project greatly simplifies and improves the producitivity of the co-development, testing and deployment of a set of actively developed and integrated software. Such projects are partitioned into smaller pieces to make them easy to work with, even for very large amounts of software.

The most important requirement for TriBITS when it was first designed for Trilinos was to support the Trilinos concept of a **Package** and to support required and optional dependencies between packages. Prior to TriBITS, it was difficult to maintain the dependencies between Trilinos packages. The primary goal for the new CMake-based system was to provide explicit support for defining and managing packages and to make a "package" a first-class citizen in the build, test, and deployment system. (It turned out that this concept of a TriBITS package mapped well with the CMake concept of an external "Package" located with `find_package()`.)

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

- Make it exceedingly easy to write `CMakeLists.txt` files for new packages and to define libraries, executables, 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 inter-package dependency handling. (For example, if package `A` is enabled, recursively enable all of the upstream packages on which it depends. Likewise, if package `B` is disabled, recursively disable all of the downstream packages that depend on it. Such logic applies to required or optional dependencies and to library or test/example dependencies.)
- 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.)

- The build system should be able to reproduce 100% update-to-date output by simply rebuilding a given target (i.e. typing 'make'). (This allows efficient rebuilds of the software for incremental changes over months of development and upgrades, greatly reducing build times.)
- Where there is a trade-off between extra complexity at the global framework level versus 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.
- Provide built-in automated support for as many beneficial software engineering practices as possible. This includes proper and complete pre-push testing when asynchronous continuous integration is being performed.
- As much as possible, make a TriBITS CMake project behave like a raw CMake project (except with different defaults in some cases like build optimized code and shared libraries by default). That is, the user should be able to set any raw CMake variable and it should behave in the same way as with a raw CMake project. (This avoids any extra learning curve for a user just trying to configure and build a CMake project that uses TriBITS as its build system.)

Most of these requirements were explicitly listed at the very beginning of the design of TriBITS way back in 2008 when the precursor for TriBITS was first designed.

The TriBITS package-based architecture is designed according to software engineering (SE) packaging principles. In the book "Agile Software Development" [2], Robert Martin describes six SE principles for packaging software. Of those, the two that most directly apply to the design of TriBITS are the *Common Reuse Principle (CRP)* and the *Acyclic Dependencies Principle (ADP)*. The TriBITS system does not allow cycles in the package dependency graph due to the ADP. Also, the assumption of the CRP is taken advantage of to simplify linking between packages in that if you link against one of the libraries of a package, you link against them all (and therefore most TriBITS SubPackages have just one library).

TriBITS is divided into a few different bits of functionality that can be installed and/or 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 Continuous Integration Support***: Consists primarily of the `checkin-test.py` tool which requires Python 2.6+ and Git 1.7+.
- ***TriBITS CTest Driver Support***: Additional CMake/CTest and Python code to support the creation of `ctest -S` driver scripts that do package-based configure, build, test and submissions to CDash.

TriBITS Core

TriBITS Core is set a of CMake modules which are used to construct CMake projects and provides a basic package-based architecture. This is used to configure, build, test, install, and deploy software that uses TriBITS/CMake. It only depends on raw CMake/CTest/CPack and 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. TriBITS Core allows for the seamless configuration, build, and testing of different integrated (but independent) software development efforts. However, TriBITS Core does not assume any software development process or make any assumptions about the version control (VC) tools or other aspects of the projects that use it.

The basic architectural components of TriBITS Core include *TriBITS Internal Packages*, *TriBITS External Packages (TPLs)*, *TriBITS Repositories* and *TriBITS Projects*.

TriBITS Internal Package: 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: In addition, TriBITS packages can optionally be broken down into ***TriBITS SubPackages*** that allow for finer-grained dependency handling and enable/disable behavior. Each Package and SubPackage contains at least one `CMakeLists.txt` file that specifies the libraries, executables, and tests for the package. Packages and SubPackages list dependencies on other Packages and SubPackages through a `Dependencies.cmake` file. The dependency information in these `Dependencies.cmake` files is used to create a directed acyclic graph (DAG) that is used in dependency analysis.

TriBITS External Package (TPL): Specification for a particular external package/third-party library (TPL) that is required or can be used in one or more downstream *TriBITS Internal Packages*: A TPL typically provides a list of libraries or a list include directories for header files. The specification of how to find an external package/TPL is determined through a provided `FindTPL<tplName>.cmake` file (which can call `find_package(<tplName>)`).

TriBITS Repository: Collection of one or more *TriBITS Internal Packages* and zero or more *TriBITS Internal Package (TPL)* specifications referenced by those packages: A TriBITS Repository contains a file `PackagesList.cmake` that lists the names and source locations of the *TriBITS Internal Packages* in the repository and a file `TPLsList.cmake` that lists the names and locations of `FindTPL<tplName>.cmake` files.

TriBITS Project: Collection of one or more *TriBITS Repositories* and therefore *TriBITS Packages* which constitutes a complete CMake "PROJECT" defining software which can be directly configured with `cmake` and then be built, tested, and installed. A TriBITS project must contain a base-level `CMakeLists.txt` file and a `ProjectSettings.cmake` file but may contain a number of other `*.cmake` files as well to tap into additional TriBITS functionality. For example, a meta-project can be constructed out of multiple TriBITS Repositories by listing them in a `cmake/ExtraRepositoriesList.cmake` file.

Many TriBITS CMake projects like Trilinos only involve a single TriBITS Repository. More complex multi-repository TriBITS projects like VERA contain many TriBITS repositories, each containing TriBITS Packages. Every TriBITS project must define at least one TriBITS package and most TriBITS projects have at least one (perhaps optional) external package/TPL dependency such as a library like BLAS or LAPACK and therefore will specify at least one TriBITS External Package (TPL).

Extended TriBITS

In addition to *TriBITS Core*, TriBITS also contains an extended set of tools to support continuous and nightly testing processes and integration workflows. Using these tools comes with an additional set of software dependencies (just Python 2.6+ and Git 1.7+).

TriBITS Continuous Integration Support contains a small collection of software to support continuous integration which primarily includes the Python `checkin-test.py` tool. The `checkin-test.py` tool implements robust pre-push continuous integration and can be used to implement lightweight post-push continuous integration and nightly testing processes that rely on email notifications. At the low end, 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. The `checkin-test.py` tool requires that Python 2.6+ and Git 1.7+ be installed on the system and it requires all of the VC repositories to use Git (or be snapshotted into Git repositories).

TriBITS CTest Driver Support exploits the package-based architecture of a TriBITS CMake project to provide testing of a large CMake project using CTest and submits result on a package-by-package basis to CDash. In addition, each TriBITS package can have its own targeted regression email list and CDash displays results on a package-by-package basis as well as for the whole project for each build. This extra functionality is primarily implemented in the function `tribits_ctest_driver()` which is called in custom `ctest -S` scripts. It also requires the setup of a CDash server.

VERA USAGE OF TRIBITS

VERA is composed of approximately 25 different Git VC repositories of which 17 are TriBITS Repositories that define more than 500 TriBITS Packages and Sub-Packages (of which the full VERA build enables about 230)[1]. Those TriBITS packages contain more than 3600 `CMakeLists.txt` files and over 2600 `*.cmake` files (not including files in TriBITS itself). There are a total of over 60K C, C++, and Fortran source files and over 12M non-comment lines of code in these repositories. While not all of this code is built, tested and deployed as part of VERA and much of it was not developed under the CASL program, a sizable portion was developed under CASL and other software in these repositories was extended through close collaborations with CASL. The point is that VERA is a large piece of software to build and test. To make changes in the way one builds or tests software of this size, one does not simply go change 3600+ `CMakeLists.txt` files. One needs a systematic approach to deal with build and test issues in a consistent way in a project of this size. Applying changes to build and testing approaches consistently across the entire set of packages is where TriBITS has proven most useful.

[1]These statistics about VERA were taken from a development version of the code base around the beginning of 2019.

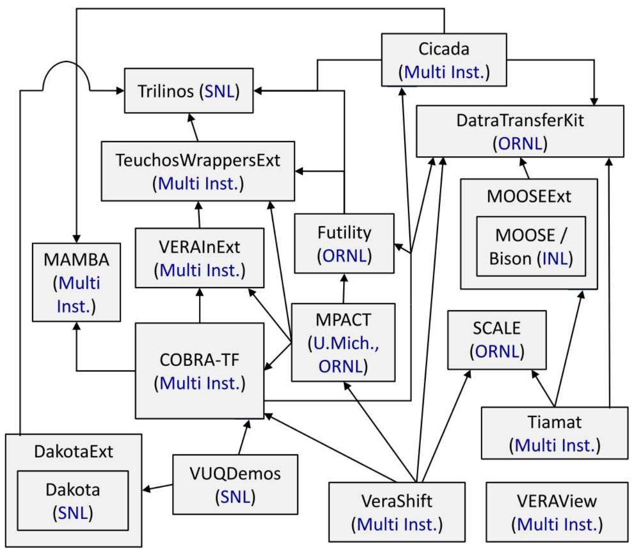


Fig. 1. **Selection of VERA TriBITS and Git Repositories and some dependencies that exist between them**: These repositories are listed in the file `VERA/cmake/-ExtraRepositoriesList.cmake` which is used to construct the TriBITS meta-project. There are no explicit dependencies between TriBITS Repositories. Instead, the dependencies between repositories are defined implicitly through the dependencies of the TriBITS packages in those repositories. Some of the repositories like `VERAView` contain stand-alone packages that do not depend on packages in other repositories but are built along with the other VERA packages in the single VERA `cmake` meta-build for convenience.

The set of VERA TriBITS Repositories that make up VERA and some of their dependencies are shown in Figure 1. The aggregation of TriBITS Repositories and Packages into the VERA TriBITS Meta-Project is depicted in Figure 2. All of the software integrated into VERA uses TriBITS as their native build and test system with the exception of Dakota and MOOSE/Bison. In the case of Dakota, it is wrapped using the DakotaExt TriBITS Package to allow the Dakota executables to be built and installed. The DakotaExt package has a dependency on the Trilinos package Teuchos but that is a minor complication. The situation with MOOSE/Bison is more complex. The MOOSEExt TriBITS package wraps the native MOOSE/Bison build which uses the libmesh Autotools system to configure the code and then a customized Makefile system is used to build the MOOSE/Bison software. To complicate matters, the MOOSE Makefile build system only allows in-source builds while the VERA CMake system only

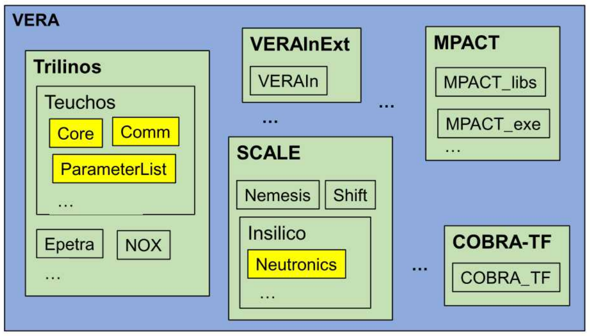


Fig. 2. **Aggregation of VERA TriBITS Repositories, Packages and SubPackages into the TriBITS Meta-Project**: This shows the TriBITS Repositories `Trilinos`, `VERAInExt`, `SCALE`, `MPACT`, and `COBRA-TF` and some TriBITS Packages and SubPackages inside each of these repositories. For example, it shows the Trilinos Package `Teuchos` which contains the SubPackages `Core`, `Comm`, and `ParameterList`. Downstream packages refer to these as `TeuchosCore`, `TeuchosComm`, etc. in their `Dependencies.cmake` files.

allows out-of-source builds. To reconcile this incompatibility, the MOOSEExt TriBITS wrapper package automatically copies the MOOSE/Bison/libmesh source to the build directory where it is configured and built. The generated libraries are then exported for downstream TriBITS packages to use like in `Tiamat`. The MOOSEExt wrapper package can also automatically detect when source files or upstream dependencies have changed that require MOOSE/Bison to be reconfigured and rebuilt from scratch. This maintains an automated and robust rebuild process but also creates a more complex and inefficient development environment for VERA developers and complicates upgrades of MOOSE/Bison into VERA. It is estimated that developer costs for upgrading MOOSE/Bison into VERA were higher than for all of the other software repositories integrated into VERA combined, largely due to the inconsistencies between the incompatible build systems.

One of the reasons why TriBITS was so quickly adopted by the different teams and software efforts was that it not only allowed the different components to be easily integrated into the single multi-physics VERA meta-build, but it also allowed them to maintain their own software projects independent from CASL and VERA using the same build and test system, just by rearranging the existing TriBITS Repositories into different TriBITS Projects. This is depicted in Figure 3 for the Trilinos, COBRA-TF, SCALE, and MPACT projects.

Another area where the TriBITS framework has proven useful is in driving automated testing supporting development and deployments. For this, the `checkin-test.py` tool is used to drive pre-push asynchronous continuous integration testing before pushing to the 'master' branch (using the fast-running `BASIC` test suite) using the standard VERA driver script `checking-test-vera.sh --do-all --push` for a single build configuration. Since asynchronous integration can lead to broken code on the 'master' branch (because two or more incompatible independent branches can be tested and pushed at the same time), a post-push continuous integration build (running a slightly larger and more expensive `CONTINUOUS` test suite) is run which posts to the VERA CDash project and triggers emails sent to developers if there are any build or test failures. That way, developers can react immediately if such a case occurs. (Breakages of the `CONTINUOUS` VERA test suite only occurred a few times a year on average and they were usually addressed that same day.) Finally, a set of nightly builds are run for different build configurations (e.g. debug and release, shared and static libraries) and for a set of `HEAVY` tests for one fully optimized build. (The `HEAVY` test category was added to TriBITS to support VERA and the VERA `HEAVY` test suite takes an entire 24 hour day on a fast 128 core node with 256G of memory.) These test categories are subsets of each other. That is, the `HEAVY` test suite contains all of the `NIGHTLY` tests which contain all of the `CONTINUOUS` tests which contain all of the `BASIC` tests. Without having standardized test categories, it is very difficult to drive automated testing of a large number of independently developed software components.

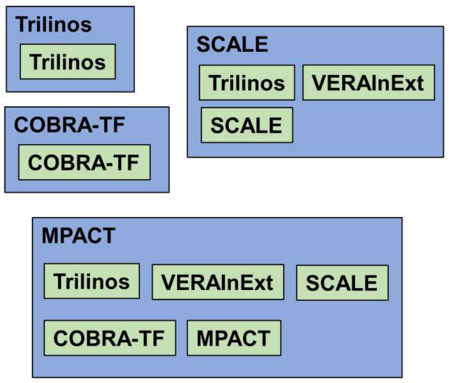
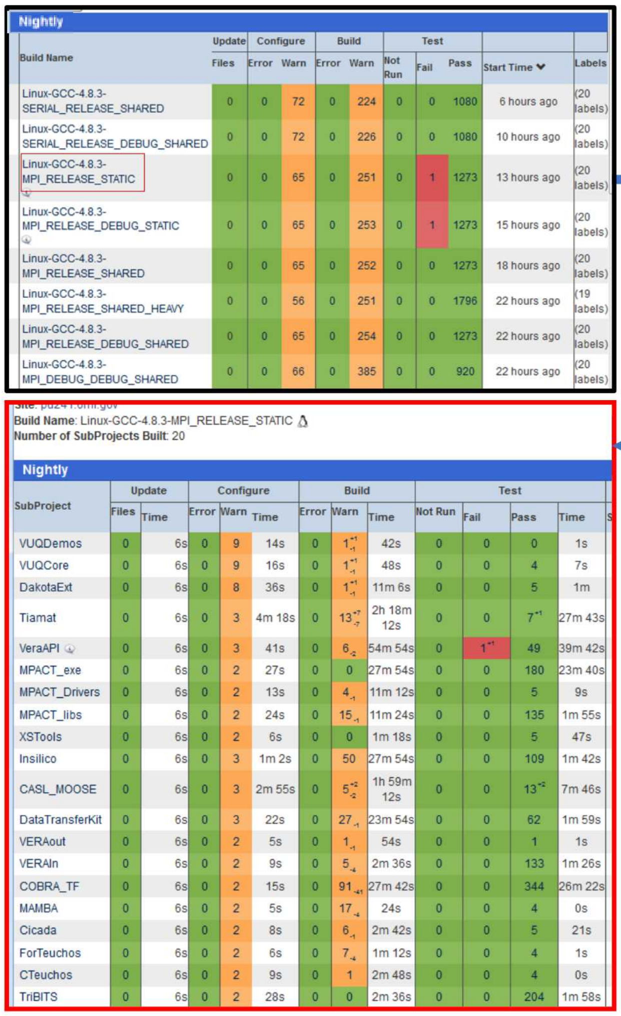


Fig. 3. **Rearrangement of the TriBITS Repositories into different TriBITS Projects**: This shows the same TriBITS Repositories and TriBITS Packages from Figure 2 rearranged into different TriBITS Projects. In the case of `Trilinos` and `COBRA-TF`, one can just clone their Git repositories and directly configure them as stand-alone CMake projects (where the TriBITS Repository and TriBITS Project live in the same base directory). `SCALE` and `MPACT` use some other TriBITS Repositories as well as their own TriBITS Packages to implement their own applications and projects independent from VERA.

Figure 4 shows the CDash dashboard for the VERA nightly builds from around 2016 showing the package-by-package breakdown for one of those builds. This shows the usage of the CDash SubProjects feature where a subset of TriBITS packages that make up VERA are specifically selected as meta-project packages from the different TriBITS repositories and the tests and examples are enabled and run for this subset of packages. In this version of VERA, only 20 of the over 90 enabled top-level TriBITS packages are included in this subset of tested packages.

Finally, another area where the TriBITS package-based framework approach has proven useful with VERA is in creating distributions of the software. VERA puts out several


Nightly

Build Name	Update	Configure		Build		Test				
	Files	Error	Warn	Error	Warn	Not Run	Fail	Pass	Start Time	Labels
Linux-GCC-4.8.3-SERIAL_RELEASE_SHARED	0	0	72	0	224	0	0	1080	6 hours ago	(20 labels)
Linux-GCC-4.8.3-SERIAL_RELEASE_DEBUG_SHARED	0	0	72	0	226	0	0	1080	10 hours ago	(20 labels)
Linux-GCC-4.8.3-MPI_RELEASE_STATIC	0	0	65	0	251	0	1	1273	13 hours ago	(20 labels)
Linux-GCC-4.8.3-MPI_RELEASE_DEBUG_STATIC	0	0	65	0	253	0	1	1273	15 hours ago	(20 labels)
Linux-GCC-4.8.3-MPI_RELEASE_SHARED	0	0	65	0	252	0	0	1273	18 hours ago	(20 labels)
Linux-GCC-4.8.3-MPI_RELEASE_SHARED_HEAVY	0	0	56	0	251	0	0	1796	22 hours ago	(19 labels)
Linux-GCC-4.8.3-MPI_RELEASE_DEBUG_SHARED	0	0	65	0	254	0	0	1273	22 hours ago	(20 labels)
Linux-GCC-4.8.3-MPI_DEBUG_DEBUG_SHARED	0	0	66	0	385	0	0	920	22 hours ago	(20 labels)

Site: pu241.ornl.gov
Build Name: Linux-GCC-4.8.3-MPI_RELEASE_STATIC
Number of SubProjects Built: 20

Nightly

SubProject	Update		Configure			Build			Test			
	Files	Time	Error	Warn	Time	Error	Warn	Time	Not Run	Fail	Pass	Time
VUQDemos	0	6s	0	9	14s	0	1^{+1}_{-1}	42s	0	0	0	1s
VUQCore	0	6s	0	9	16s	0	1^{+1}_{-1}	48s	0	0	4	7s
DakotaExt	0	6s	0	8	36s	0	1^{+1}_{-1}	11m 6s	0	0	5	1m
Tiamat	0	6s	0	3	4m 18s	0	13^{+7}_{-7}	2h 18m 12s	0	0	7^{+1}	27m 43s
VeraAPI	0	6s	0	3	41s	0	6_{-2}	54m 54s	0	1^{+1}	49	39m 42s
MPACT_exe	0	6s	0	2	27s	0	0	27m 54s	0	0	180	23m 40s
MPACT_Drivers	0	6s	0	2	13s	0	4_{-1}	11m 12s	0	0	5	9s
MPACT_libs	0	6s	0	2	24s	0	15_{-1}	11m 24s	0	0	135	1m 55s
XSTools	0	6s	0	2	6s	0	0	1m 18s	0	0	5	47s
Insilico	0	6s	0	3	1m 2s	0	50	27m 54s	0	0	109	1m 42s
CASL_MOOSE	0	6s	0	3	2m 55s	0	5^{+2}_{-2}	1h 59m 12s	0	0	13^{+2}	7m 46s
DataTransferKit	0	6s	0	3	22s	0	27_{-1}	23m 54s	0	0	62	1m 59s
VERAout	0	6s	0	2	5s	0	1_{-1}	54s	0	0	1	1s
VERAIn	0	6s	0	2	9s	0	5_{-4}	2m 36s	0	0	133	1m 26s
COBRA_TF	0	6s	0	2	15s	0	91_{-41}	27m 42s	0	0	344	26m 22s
MAMBA	0	6s	0	2	5s	0	17_{-4}	24s	0	0	4	0s
Cicada	0	6s	0	2	8s	0	6_{-1}	2m 42s	0	0	5	21s
ForTeuchos	0	6s	0	2	6s	0	7_{-4}	1m 12s	0	0	4	1s
CTeuchos	0	6s	0	2	9s	0	1	2m 48s	0	0	4	0s
TriBITS	0	6s	0	2	28s	0	0	2m 36s	0	0	204	1m 58s

Fig. 4. **VERA CDash Builds and Package-by-Package Detail**: Top shows summary of nightly VERA builds for different build cases. Top summary line for build `LINUX-GCC-4.8.3-MPI_RELEASE_STATIC` shows 1 failing test and 1273 passing tests. Clicking the build name `LINUX-GCC-4.8.3-MPI_RELEASE_STATIC` goes to the package-by-package breakdown page shown at bottom. Shows that the failing test is in package `VeraAPI`. Package-by-package breakdown includes build and tests results and test run-times for each package.

different distributions of the software which contain different subsets of repositories and packages. For example, several of the repositories are currently not released in any of the distributions of VERA which include `MAMBA`, `Cicada`, `VERAView` and `VUQDemos`. These repositories are explicitly excluded from distribution source tarballs of VERA. In addition, any packages that are not enabled in the configuration process are automatically excluded from source tarball distributions by the TriBITS system. (TriBITS knows all of the packages and will automatically exclude the source directories for packages that are not enabled when generating the source tarball distributions.) And the TriBITS package dependency system will automatically disable downstream packages when upstream packages are missing when the reduced untarred source is built on the target system (or in the Docker distribution container). The configurations of future VERA distributions are also tested every day by creating a source tarball off the 'master' branch, untarring it, doing the configure, build and running the full `NIGHTLY` test suite, then running the install and finally running some installed "smoke tests" to ensure the VERA executables and supporting files have been installed correctly. These TriBITS features and daily deployment testing processes significantly reduce the costs and overhead of putting out new releases of VERA.

SUMMARY

The creation and adoption of the TriBITS framework by the CASL VERA project allowed for the efficient development and integration of many software development efforts from many different groups and institutions at a fraction of the cost of traditional approaches typically employed in the CSE community. By using a common systematic approach to building, testing, and deploying software, the infrastructure and integration costs were kept to a minimum. Comparing the costs of the integration efforts over the 10 year history of CASL for those codes that natively used TriBITS as their build and testing system to the codes that did not, provided a contrast and motivation for using a homogeneous framework such as TriBITS for driving such efforts. As post-CASL VERA maintenance will likely only include those codes that use TriBITS as their native build and test system, the infrastructure and integration costs for VERA are likely to remain comparatively low for years to come.

ACKNOWLEDGMENTS

This material is based upon work supported by a Department of Energy Consortium for the Simulation of Light-water Reactors.

REFERENCES

1. J. A. TURNER, K. CLARNO, M. SIEGER, R. BARTLETT, B. COLLINS, R. PAWLOWSKI, R. SCHMIDT, and R. SUMMERS, "The Virtual Environment for Reactor Applications (VERA): Design and architecture," *Journal of Computational Physics*, **326**, 544 – 568 (2016).
2. R. MARTIN, *Agile Software Development (Principles, Patterns, and Practices)*, Prentice Hall (2003).
3. P. DUVALL and ET. AL., *Continuous Integration*, Addison Wesley (2007).
4. C. SCOTT, *Professional CMake: A Practical Guide, 5th Edition)*, Crascit.com (2019).
5. K. BECK, *Extreme Programming (Second Edition)*, Addison Wesley (2005).
6. M. HEROUX and ET. AL., "An overview of the Trilinos project," *ACM TOMS* (2005).