The Draco Build System

K.G. Thompson[a]

T.M. Evans[b]

R.M. Roberts[c]

Author address:

[a]CCS–2, MS D409, Los Alamos National Security, LLC, Los Alamos, NM 87545
E-mail address: kgt@lanl.gov

[b]currently at ORNL

[c]D–6, MS F609, Los Alamos National Security, LLC, Los Alamos, NM 87545
E-mail address: rsqrd@lanl.gov

LANL Report Designation: CCS–2:12–43(U).

ABSTRACT. The Draco build system is designed to facilitate both development and usage on multiple platforms and using multiple build tools. Originally, the build system adopted much of the GNU Coding Standards [1] for software packages. However, as of 2011, the build system no longer relies on the GNU **autotools** and it supports non-Linux build platforms. While the design of the Draco build system is moving away from strict conformance to the GNU Coding Standards, it has been designed according to the following list of requirements:

1. support for simultaneous, multiple configurations (Release, Debug, etc);
2. support for multiple programming languages. The primary language is C++ but support for for C and Fortran is required for all current ASC platforms.
3. support for multiple build project types (**Makefiles**, **XCode**, **Eclipse**, etc.)
4. the ability to support multiple code projects;
5. support for external vendors;
6. support for explicit template instantiation;
7. C and C++ coding must conform strictly to issued ISO standards.
8. extensibility;
9. low-cost on developers to add new packages, code, and tests;
10. support for on-demand and automated regression testing with web-based dashboard presentation.
11. preconfigured development environment for ASC platforms (`bashrc/cshrc`, `elisp`, vendors, extensions to the **module** command, etc.)

These requirements plus additional features such as threaded/parallel building, heterogeneous architectures and vendor support have been included in the Draco build system.

The build system primarily uses three CMake tools [2], **CMake**, **CTest**and **CDash**. Version control of the Draco source is performed by **SVN** [3]. These tools are freely available from Kitware and Tigris.org, respectively.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

In this chapter we will examine the design contraints of the Draco build system (DBS). We will introduce some definitions of terms and typographical conventions that will be employed throughout the text. Additionally, the organization of the manual will be described. For those who wish to get started right away, § 1.6 details how to get up and moving without working through the entire manual. Also, a detailed developer manual is processed with **Doxygen** [4] and is maintained with the Draco source code.

1.1. Purpose

The purpose of this document is to describe the DBS. Specifically, we will show both Draco users and Draco developers how to:

- configure Draco for different platforms, build types and options;
- compile Draco components;
- compile and link programs that use Draco;
- add components to Draco;
- add new support options to Draco;
- generate developer documentation via **Doxygen**.

Thus, this manual is an invaluable reference to those who work in, or with, Draco.

As a review, we restate the Draco mission statement [5]:

> Draco *is a comprehensive, radiation transport framework that provides key, reusable components for serial and parallel computational physics codes.*

To meet these requirements Draco uses modern software engineering concepts including object-oriented, generic design [6], multi-environment build systems, and service libraries based on levelized component designs [7]. The build system described in this manual allows Draco to satisfy its mission statement and enforces the concept of levelized component design.

The Draco build system has been carefully designed. In particular, we had several requirements that the build system should satisfy. These requirements are:

- support for simultaneous, multiple configurations (Release, Debug, etc);
- support for multiple languages. The recommended language is C++, but support for C and Fortran is provided on all current ASC platforms.
- C and C++ coding must conform strictly to issued ISO standards.
    1. C++11, also known as the ISO/IEC 14882:2011.
    2. C99, also known as the ANSI C ISO/IEC 9899:1999 standard
    3. C++11, also known as the ISO/IEC 14882:2011 standard, was published in September 2011 and is supported by the Draco Build System. Because compiler support for the new standard is immature, the build system will automatically disable support for 2011 features if the selected compiler is unable to support basic features of the updated standard.
- support for multiple build project types (**Makefiles**, **XCode**, **Eclipse**, etc.)
- on-demand and automated unit and regression testing;
- the ability to support multiple code projects;
- support for external vendors;
- support for explicit template instantiation;
- extensibility;
- low-cost on developers to add new packages, code, and tests;

Formerly, the DBS implemented the GNU coding standard [1] , however, this requirement has evolved somewhat as the supported platforms and build systems scope has grown to include **XCode** and **Eclipse** on non-Linux platforms. The use of the **CMake** suite of tools preculdes the use of GNU autotools , **autoconf** and **automake** [8]. It also precludes the use of **make** [9] for non-Makefile based build projects (e.g.: **XCode**, **Eclipse CDT**). More detail will be given in Chap. 2.

1.2. Definitions and Conventions

Before continuing we shall clarify the terminology and typeface conventions that will be employed throughout the remainder of this manual. The definitions that we use here are for convenience. They are not to be interpreted as an "universal standard." They are simply used to make sure that the concepts illucidated within this manual have a common point of reference.

A *product* is anything that is produced from a source code tree [10]. A *system* is a code, or a group of codes, that persist over time [11]. A *project* is an undertaking that has a definite beginning and ending date, and it produces a product. A *package* is one component of a system (package and components are used interchangeably). Packages normally reside in a single directory in the source code tree; although, that directory may have subdirectories. However, packages are sometimes used to refer to larger units. For example, a code package may be a system that contains many components. In this case, package has macro (system level) and micro (system-component level) connotations.

Table 1.1 show the typefaces that we will employ throughout the text to better distinguish certain

TABLE 1.1. Typefaces used throughout the text.

code systems (Draco)
PACKAGES (DS++)
`files` (`Makefile`)
variables (`draco/src/`*pkg*`/`)
software programs (**gmake**)
languages (C++)

elements. In general, anything that exists on a computer screen (directory trees, files, etc) is typefaced using `typewriter` font. Files are distinguished in the standard UNIX way by appending the following symbols after the name, `*` for executables, `/` for directories, and @ for links. Computer screen prompts are represented by the `$` symbol.

1.3. Draco Build System Support and Procurement

Questions about procuring a copy of the DBS or its use can be directed to:

Name	Email	Group	Team
Kelly Thompson	kgt@lanl.gov	CCS–2	Jayenne
Jae Chang	jhchang@lanl.gov	CCS–2	Capsaicin

Additional information is available on the Draco **TeamForge** site, https://tf.lanl.gov/draco. Also note that the Draco team maintains a a UNIX file sharing group, `draco`. Developers wishing to access Draco source code or development environment files must be members of this file sharing group. Membership is controlled by the current super-users of the group as listed on https://register.lanl.gov which should correspond to the individuals listed in the table above.

1.4. Manual Organization

This manual is written for three basic groups: (a) Draco users, (b) Draco component developers, and (c) Draco system developers. The manual is organized around these three groups. There is an additional group consisting of developers who plan to use Draco as a model for their own code systems. For this group the entire Draco Build System Manual is of interest.

TABLE 1.2. Roster of the Draco package developer group.

Name	Email	Group
Kelly Thompson*	kgt@lanl.gov	CCS–2
Jae Chang	jhchang@lanl.gov	CCS–2
Allan Wollaber	wollaber@lanl.gov	CCS–2
Gabe Rockefeller	gaber@lanl.gov	CCS–2
Kent Budge	kgbudge@lanl.gov	CCS–2
Jim Warsa	warsa@lanl.gov,	CCS–2
Rob Lowrie	lowrie@lanl.gov	CCS–2
Todd Urbatsch	tmonster@lanl.gov	CCS–2

*Draco project leader.

TABLE 1.3. Roster of the Draco systems developer group.

Name	Email	Group
Kelly Thompson	kgt@lanl.gov	CCS–2
Allan Wollaber	wollaber@lanl.gov	CCS–2
Jae Chang	jhchang@lanl.gov	CCS–2

TABLE 1.4. Chapters targeted for each group of Draco developers.

Group	Recommended	Optional
Users	Chap. 2, 3, 4 and 7	
Component Developers	Chap. 2, 3, 5 and 7	Chap. 4 and 6
System Developers	Chap. 2, 5, 6 and 7	Chap. 3

The Draco users group consists of clients who use Draco components in some form. The primary interest of this group is configuring and compiling Draco so that it meets their product's needs. The relationship between this group and Draco can be very close (CCS–2 code teams) or very distant (XCP-1 code teams).

The Draco components developers group contains people who write packages in Draco. The primary interest of this group is configuring, compiling, and adding new packages to Draco. The final group, the Draco system developers group, are those who maintain the Draco infrastructure, including the build system. This group is concerned with maintaining the integrity and stability of Draco as a whole unit. Tables 1.2 and 1.3 gives a current list of the Draco package and system developers.

Table 1.4 lists the recommended chapters for review by each Draco developer group. Chapter 2 gives an overview of the Draco source tree and build system. A complete listing of the Draco source tree is included in this chapter. This chapter is useful for all three groups that are associated with Draco.

Chapter 3 describes how to configure and compile the Draco system. Included in this chapter are detailed descriptions of all the Draco configure options. Chapter 4 shows how to emulate the Draco build model and functionality in external code systems that use Draco. This chapter is geared to code teams that heavily use Draco, and, thus, they may gain advantages by using the Draco build model. These chapters target the Draco users and Draco package developers groups.

Chapter 5 shows how to add new component packages to Draco. In this chapter detailed instructions are given that show how to add a new package directory, test directory, and, to a lesser extent, build options. The intended audience for this chapter is the Draco package developers, and, to a lesser extent, Draco system developers will use this material.

Finally, Chap. 6 shows how to extend the Draco build system. This chapter focuses on adding new configure options and new language support. In general, this chapter shows how the **CMake** files are used

Table 1.5. File and Internet locations used by Draco

Item	Location
Repository	`svn+ssh://ccscs8.lanl.gov/ccs/codes/radtran/svn/`
Archival storage	`HPSS://hpss/jayenne/`
Wiki[a]	http://tf.lanl.gov
Bug Tracker[a]	http://tf.lanl.gov
Regression files	`ccscs8://home/regress/cmake_draco` and `hpc://usr/projects/jayenne/regress`
Regression Dashboard	http://coder.lanl.gov/cdash

[a]The Draco team is considering adoption of Redmine for Wiki and Tracker support. The location of this tool is `http://coder.lanl.gov/redmine`

Table 1.6. Suggested email lists for Draco developers

List Name	Purpose
`draco`	General Draco related discussion, including **SVN** commit messages.
`jayenne`	General Jayenne related discussion, including **SVN** commit messages for ClubIMC, Wedgehog and Milagro.
`capsaicin`	General Capsaicin related discussion, including **SVN** commit messages.

and work. This chapter is primarily intended for Draco system developers; however, some content in this chapter is necessary for Draco package developers.

1.5. Adminstration

1.5.1. Important locations. Draco makes use of NFS-based and web-based tools. Locations of Draco related files and services are provided in Table 1.5.

1.5.2. Mailing list and commit notifications. The Draco team maintains a mailing list used for announcements and general Draco discussion. To subscribe to this list, send an email to `listmanager@listserv.lanl.gov` with the body `'subscribe draco'`. For more information about mailing lists at LANL you can send an email to the same address with the message `'help'`. Some list management functions can also be completed by visiting `https://register.lanl.gov`. If you are not sure if you are subscribed or not you can send the command `'which'` to `listmanger@listserv.lanl.gov` to see what LANL lists you are subscribed to. Table 1.6 lists a few other mailing lists that may be useful for Draco developers to subscribe to.

1.5.3. Regression Dashboard. Draco maintains a regression dashboard at http://coder.lanl.gov/cdash. Draco developers are encouraged to visit the dashboard regularly to view the nightly reports. You may need to contact one of the Draco group super-users (see Sec. 1.3) to gain access to the the dashboard. You can configure the dashboard to send email for various situations like failing unit tests.

1.5.4. Issue Tracking. Draco uses the LANL **Team Forge** bug tracking system to manage issue, bug and feature tracking. **Team Forge** can be accessed by opening a web browser from the Yellow network (i.e.: inside the LANL firewall) to `https://tf.lanl.gov`. You are encouraged to browse the list of known bugs/issues of the project you are working on.

1.6. Quick Start

Many Draco users will undoubtedly be familiar with **CMake** and **make**. These users can progress directly to Chapter 7 and § 3.6 for examples on configuring and building Draco.

CHAPTER 2

The Draco Model

This chapter presents an overview of the Draco build model, architecture and source. We present, in detail, the requirements for the Draco build system in § 2.2.2. Because the Draco build model was originally designed to conform to the GNU coding standard, a brief summary of GNU requirements is given in § 2.2. Finally, this chapter concludes with a description of the Draco source tree and files that are created during configuring and building.

2.1. Overview of Draco

Documentation describing the purpose and capabilities of packages within Draco is beyond the scope of this text. However, a brief summary of Draco is pertinent to this discussion. Draco is a component library for computational radiation transport. Draco is primarily a suite of C++ libraries; however, other language support is not precluded in Draco and full language support is provided for both C and Fortran. In particular, Draco supports ISO_C_BINDING interfacing between C++ and F90. A more general type of automatic type-interfacing between C++ and F90 [12] was implemented to support Dante code, but was abandoned in favor of a simpler interface paradigm.

The products of Draco are individual component libraries that provide reusable services geared towards radiation transport applications. For example, Draco provides random number generators that may be used by a Monte Carlo radiation transport solver. Draco also provides access to opacity models and an angular quadrature component that can be used by deterministic radiation transport solvers. In addition to components designed for radiation transport, Draco provides several service packages including DS++, a data structures library that contains numeric containers, smart pointers, and assertions, and C4, a communications library, among others.

Draco is designed using object-oriented [13] and generic programming [14] philosophies. Foremost among these notions are levelized design , Design-by-Contract™, and the generic concept-model idea. Other software engineering methods are employed for quality control including regression testing, automatic documentation, code profiling, and design and code reviews.

2.2. Overview of the Draco Build Model

2.2.1. Software Requirements. Originally, the Draco build system was designed according to the GNU coding standard [1] and made use of **autoconf** [8], **gmake** [9] , and **gm4** [15] for configuring and building components. In 2011, this requirement was altered as the development team chose to replace the aging build system with a **CMake**-based system [2]. In addition, Draco version control is performed by **SVN** [3].

Additional software is used for performing quality control. Regression testing is handled by **CMake**'s **CTest** and **CDash** tools. Bugs are tracked using **TeamForge** [16,17]. In addition, an archived email list is available to submit design plans and discussion between team members. Also, **Valgrind** [18], **BullseyeCoverage** [19] and **CLOC** [20] play important roles in the quality control process. The build system supports processing code and tests through **Valgrind** on all platforms for performing dynamic memory and cache analysis. It supports the use of **BullseyeCoverage** for analyzing C++ code coverage metrics (function and condition/decision branch coverage) and it uses **CLOC** for tracking total lines of source code. All of this information is collected nightly by the regression system and published on the Draco regression dashboard at `http://coder.lanl.gov/cdash`.

The Draco build system contains support for multiple language environments. However, the primary language in use at present is the 2011 ISO Standard C++ [21] . Currently, any C++ compiler that conforms to the 2011 C++ standard will compile Draco. Intel and GNU C++ are regularly used for building on ASC hardware. Additional languages required by Draco are PYTHON and PERL. Draco expects these scripting languages to be located in a directory that is included in the developer's `PATH`, and the build system checks for this. A suite of tools that can typeset LaTeX sources is also necessary for compiling much of the documentation that comes with Draco.

With the exception of **BullseyeCoverage** and the listed commercial compilers, all of the aforementioned software products are freely available. Note that **BullseyeCoverage** is an analysis tool and is not required to configure, build, or use Draco. Also note that the GNU set of compilers are freely available and using them allows Draco to be built without the use of any commercial software.

The build system checks for the presence of each of aforementioned software products; thus, as long as the software is in the user's path, the configuration will succeed. Finally, Draco utilizes a number of vendor libraries, depending upon configuration, that must be installed on the system in which Draco resides. Detail on these packages and their configuration options is given in Chap. 3.

Table 2.1. Required and optional tools for configuring and building Draco.

Tool	Required	Recommended	Optional
Configuration	**CMake**-2.8.8+	**CTest**	**CDash**
C++ compiler	Any standards compliant C++ compiler	g++ 4.7+	Intel 12+
F90 compiler	Must support the 2003 standard	gfortran 4.7+	Intel 12+
Build tool	any of ...	**gmake**	Eclipse CDT, XCode
Scripting language	**CMake**	PYTHON	PERL
Version control	**SVN**		**Git**
Dynamic Analysis	optional	**Valgrind**	
Code Coverage	optional	**BullseyeCoverage**	
Lines-of-Code	optional	**CLOC**	
Bug Tracking[a]	optional	**TeamForge**	ChangeLog, text files, email
Documentation	optional	LaTeX typesetting tool, **DOxygen**, text files	

[a] The project management software Redmine is currently under evaluation by the Draco board.

2.2.2. Build System Requirements. In § 1.1 the requirements that guided the development of the Draco build system were summarized. In this section, we shall take an expanded look at the complete list of requirements. The list of requirements for the Draco build system is:

1. support for simultaneous, multiple configurations (Release, Debug, Scalar, Parallel, etc);
2. support for multiple programming languages, while preferring C, C++ and Fortran;
3. support for C++ on all current ASC platforms;
4. C and C++ coding must conform strictly to issued ISO standards [21] :
   (a) C++11, also known as the ISO/IEC 14882:2011 standard
   (b) C99, also known as the ANSI C ISO/IEC 9899:1999 standard;
   (c) support for explicit template instantiation;
5. support for multiple build project types (**Makefiles** [9], **XCode**, **Eclipse**, etc.)
6. on-demand and automated unit and regression testing;
7. support for **CDash** [2, 22] presentation of regression results;
8. support for **Valgrind** [18] dynamic analysis integrated into **CDash** presentation;
9. the ability to support multiple code projects;

10. extensible support for external vendors;
11. extensibility;
12. low-cost on developers to add new packages, code, and tests;

We will analyze each of these requirements in turn. First, from a development standpoint, having multiple configurations at the same time is a must. This feature is required because certain tools work better on certain platforms. Additionally, certain tools work better in certain environments. For example, we often require both scalar and parallel versions of the code for profiling and testing. The Draco build systems allow each configuration, and the products it produces, to exist in a unique directory. Thus, builds are not performed in the source code tree; they are done in a user-specified directory. More detail is given on multiple configurations and builds in § 2.3 and Chap. 3.

Draco is not a single language system. Although Draco is primarily composed of C++ code, portions of Draco are written in C and Fortran . The Draco build system is general and is not restricted to single language support. Supporting multiple languages is an essential requirement because Draco customers utilize many frameworks and languages. Note that difficulty in modifying the build system to use of other compiled languages depends heavily on whether **CMake** supports the extra programming language. As of this writing, **CMake** only supports C, C++ and Fortran as primary languages.

C++ source code in Draco must conform to the the ISO/IEC 14882:2011 standard. This ensures that C++ sources will compile by any standards compliant C++ compiler. The DBS activates many compiler warning flags for Debug builds so that developers are notified if their code deviates from the standard. In particular, code that does not compile cleaning using ASC standard compilers on target ASC platforms must be brought into compliance or be removed from the Draco system.

In recent years the need to support non-command line integrated development environments, IDE, has become increasingly important. With the conversion of the build system from **autotools** to **CMake** in 2011, this has become a build system requirement. The primary build tool continues to be **Makefiles** but support for the **Eclipse CDT** IDE on Linux and for the **XCode** IDE on OS/X are also supported.

The next three requirements are aimed at Draco system and package developers and are essential for developing high quality software. Regression and unit testing are an important part of the Draco quality assurance program. Draco developers must have the ability to run a test suite to determine if local changes adversely impact other parts of the Draco system. The daily testing of Draco components also ensure that commits to one part of the library do not adversely affect other components. These daily tests are run on multiple platforms and may catch coding issues that only appear on specific hardware/software combinations. The results from nightly testing must be published to the Draco Dashboard [22] and email sent to developer who request nightly updates. Additionally, code coverage metrics by **BullseyeCoverage**, dynamic analysis (via **Valgrind**) and lines-of-code metrics must also be collected nightly so that the time evolution of issues can be tracked and to allow the Draco development team the data needed to target specific improvements. Such improvements might be the addition of unit tests so that more baseline code is checked in the nightly tests or the elimination of memory errors.

In addition to being self supportive, the Draco build system will be designed to be exported to other code projects. This requirement includes the ability of other code projects to use build system scripts found in Draco and the ability to find and link against Draco code and vendor code known by by Draco. The Capsaicin and Jayenne code projects both employ the Draco build system and also link to Draco component libraries.

As a suite of scientific simulation tools, Draco components often need to link against vendor software to provide specialized capabilities. For example, most of Draco is designed to be run in parallel under MPI. MPI is a vendor tool supported by Draco. The build system can detect the local availability of MPI and will adjust build parameters based on the flavor and version of MPI. For example, if MPI is not found on the local system, the build system automatically switches to `SCALAR` mode and does not attempt to link libraries and executables against the MPI libraries. While Draco can be built without MPI, it is assumed to be a required vendor in most cases. The GNU Scientific Library is also considered to be a required vendor for Draco as it provides random number generation features and many linear algebra functions. Optional vendors for Draco include BLAS, LAPACK, ScaLAPACK, BLACS, Trilinos and xmgrace among others. Extending the Draco build system to support additional vendors is straight forward and transparent.

A guiding principle of the Draco build system is explicit template instantiation. We have found that this provides a more robust and efficient build system compared to the environment where the compiler is allowed to instantiate template classes and functions *automatically*. The essence of explicit instantiation is that the package developer determines what templated classes are instantiated (and when they are instantiated) [23]. Draco does not implicitly instantiate template classes and functions. Draco has rules on how template classes and functions are explicitly instantiated. These are listed in Chap. 5. Additional information for clients that use Draco class and function templates is listed in § 4.1.

Another guiding principle of the Draco suite of components is the concept that active features are constantly changing. New features are regularly added to Draco and deprecated features are removed periodically. It is imperative that the DBS flexibly support these changes so that the developer cost of adding or removing components remains low. This is accomplished through the encapsulation paradigm and the levelized component design of Draco. The DBS itself must remain extensible as new requirements for the build system come into play.

Finally, it should be noted that the DBS adhered to the GNU Coding Standard [1] until the adoption of **CMake** and non-Makefile based build systems is not supported by this standard and we have made a deliberate decision to make the build system more flexible at the cost of standardization. By formalizing our build model on an accepted standard, we reduce the overhead associated with maintaining the system. In the case of the **CMake**-based build system, we make heavy use of the features provided by **CMake** and avoid the use of custom build scripts except for cases that are absolutely necessary (i.e.: python scripts for running application codes and checking the results against gold standards).

2.2.3. The CMake Build Model. The GNU standard specifies many constraints on build and make systems including a mandate to use `configure*` scripts and standardized configure options and `Makefile` targets. The **CMake**-based Draco build system no longer makes any attempt to support these requirements. Instead, the following build system features must be supported:

1. **CMake** will be used to configure code products;
2. Each directory in the build system will have a `CMakeLists.txt` file that describes local build targets, platform checks, and local testing instructions.
3. **CMake** guarantees certain names (e.g.: `PROJECT_NAME`, `PROJECT_BINARY_DIR`, `CMAKE_<LANG>_FLAGS, MPI_FOUND, etc.`) to exist in the build. A more complete list can be obtained by running '`cmake --help-variable-list`.'
4. **CMake** guarantees certain build project targets will be defined:
   - `all` - generate all normal targets
   - `clean` - remove generated files (libraries, object files, binaries, etc.)
   - `Experimental` - run the build and tests and submit the results to the dashboard.
   - `install` - copy built files to the `CMAKE_INSTALL_PREFIX` directory
   - `test` - run the unit tests
   - `uninstall` - remove files copied as a result of the install target
   - A complete list of build targets can be generated by using the command '`make help`.'
5. The DBS standardizes additional Makefile targets:
   - Lib_<*component*>
   - Lib_<*component*>_test
   - Ut_<*component*>_<*testname*>_exe

Because **CMake** does not restrict the build project to **Makefiles**, we don't speak explicitly about *Makefile targets* and instead use the term *build targets*. In general, the Draco build system attempts to use the same variable naming conventions as **CMake**.

2.3. The Draco Source Code Tree

2.3.1. Draco Source Tree. In this section we give an overview of the Draco source tree. The Draco source tree is illustrated in Fig. 2.1. The subdirectories are in Figs. 2.2a through c. Note that these figures show the complete source tree. Under Draco there exist several files generated or needed by **CMake** and **CTest**. The files `CTestConfig.cmake` and `CTestCustom.cmake` are read by **CTest** and specify the

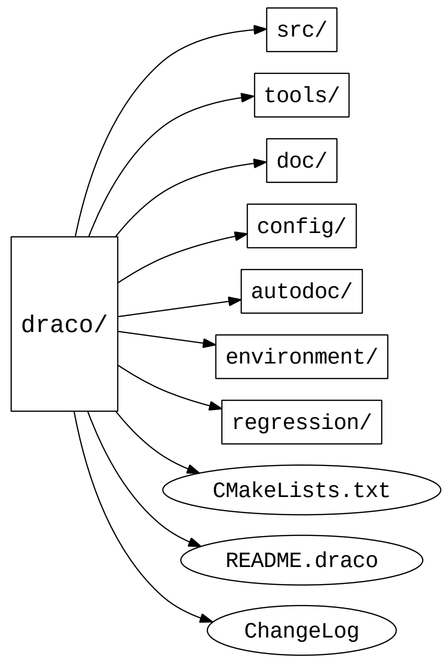


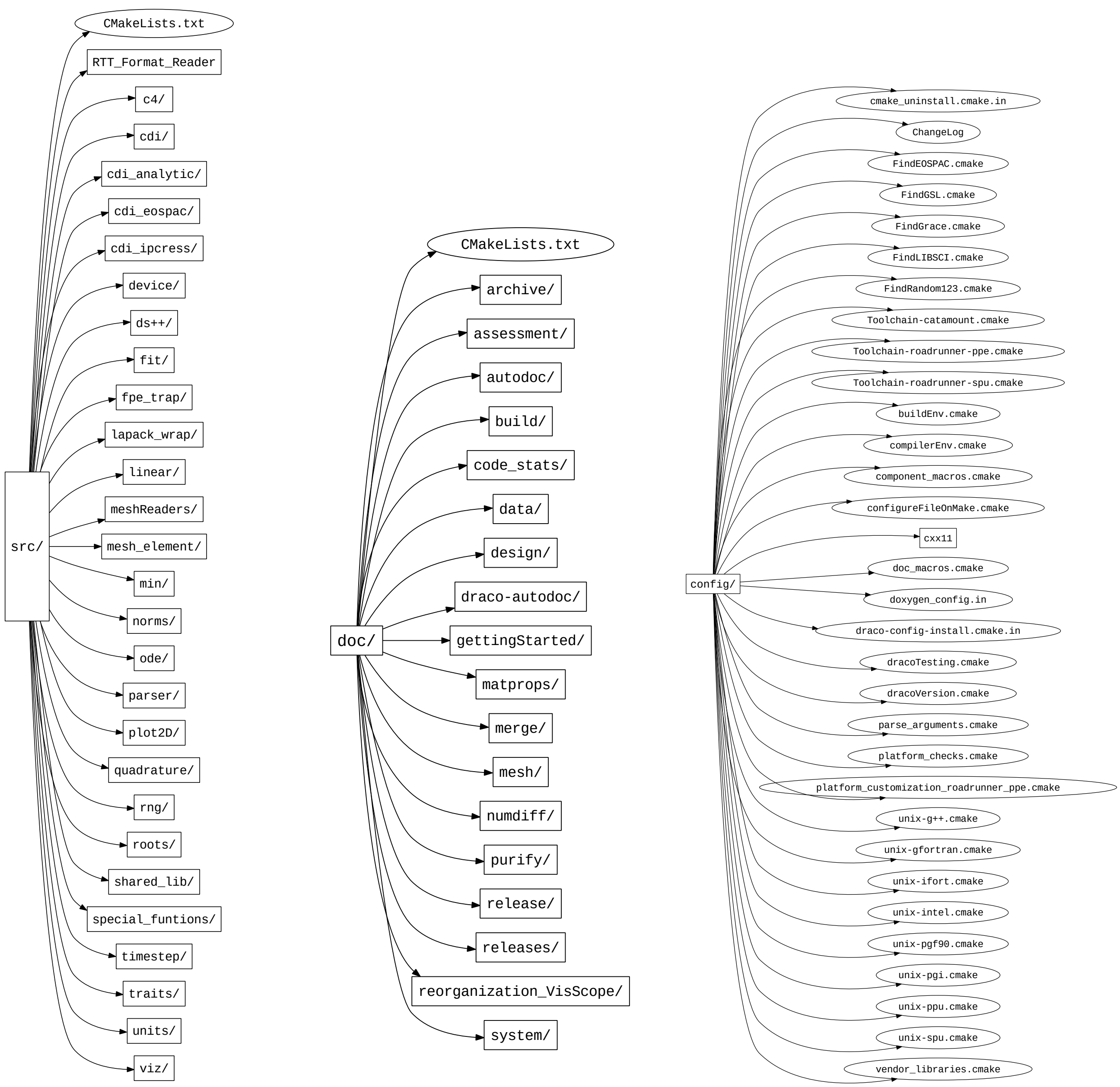
FIGURE 2.1. The Draco source tree. Directories are in boxes and files are in ellipses. The subdirectories are shown in Fig. 2.2.

location of the Draco dashboard and configuration setup for testing. These settings include a list of source to omit from code coverage, tests to omit from dynamic analysis, warnings that should be ignored, etc. The `CMakeLists.txt` file is the primary build system configuration file parsed by **CMake** in order to generate the build project (i.e.: Makefiles, project files, etc.). The `CMakeCache.txt` file is a template that developers can copy to their build sandbox and update initial build configuration settings before the first invocation of **CMake**. This file is not required by the Draco build system and is provided only as an aid for developers. The `*.cmake` files from the `config` directory contains macro tests for configuring Draco. Descriptions of these files are given in Chap. 6. Finally, the files `Copyright`, `ChangeLog`, and `README.draco` are descriptive ASCII files.

In the `src/` directory there is a `CMakeLists.txt` file that instructs **CMake** to setup the selected project generator, configure compiler options, run platform checks, locate and check vendor libraries, and tells **CMake** to descend into each of the component directories for individual configuration. This file instructs **CMake** to establish most of the build parameters and the master structure and inter-component dependencies. The encapsulation and levelization of packages is controlled at this level.

In addition to the source code (`.cc`, `.hh`, and `.h` files), each component directory contains two build system files: A `CMakeLists.txt` file controls the specific build instructions for the individual component including a list of sources to be compiled into a library. The `config.h.in` may not be provided with every component, but when it is provided it encapsulates preprocessor macro settings that are needed by the component. Ref [8] provides some background information on how the `config.h.in` should be used. Detailed discussions about how these files are used in Draco are given in Chaps. 3 and 5.

Finally, each directory under `src/` may contain source subdirectories, `doc/` subdirectories and `autodoc/` directories and should contain a `test/` directory. The `test/` directory holds component tests for the package. Details on how to compile the test directory using **CTest** are given in Chap. 3. Directions showing how to format different test directories are given in Chap. 5.



(A) `src/` (B) `doc/` (C) `config/`

Figure 2.2. Subdirectories under `draco/`. Directories are in boxes and files are in ellipses.

2.3.2. Binary Directory Trees. The Draco source is normally compiled in a separate, user-generated directory tree called the *target* or *binary* directory tree[1]. In this standard build mode, no files are actually generated in the Draco source tree. The Draco build system also supports building from within the source tree, but this mode of development is not recommended except in special circumstances (e.g: **Eclipse** based projects work better when the source and binary tree are collocated).

To configure Draco, the user checks out a version of the source from **SVN**. The checkout location is known as the *source* directory. In the *binary* directory tree, the user runs **CMake** with appropriate configure options. These options can be provided (1) on the command line, (2) through the **CMake** graphical interface via `ccmake` or `cmake-gui` or (3) by providing options in a `CMakeCache.txt` file located in the *binary* directory. **CMake** will generate a directory tree that is parallel to the Draco source tree with the appropriate project and configuration files (e.g.: `Makefile`, `config.h`, `draco.sln`, `.project`, etc.). Once the project files have been generated by **CMake**, the compilation step can be initiated. For a *Unix Makefiles* based project this is accomplished by running **gmake** from within the *binary* directory. The product of the compilation step is the generation of component libraries and executables by the selected compilers. For non-*Makefile* based projects, the `all` or `ALL_BUILD` project target should be selected to begin compilation.

For example, consider a parallel (MPI), debug configuration of Draco on a x86_64 Linux platform using the **gcc** compiler suite. The user might create a *target* directory called `gcc_mpid` and a *build* directory under the target directory called `draco`. After running **CMake** with the appropriate options[2], the directory structure illustrated in Fig. 2.3 is generated. Inside of each component directory are Makefiles and configuration files generated by **CMake**. Object (`.o`) files are stored in the `CMakeFiles/` directory along with specialized build and dependency instructions (`depend.make`, `flags.make`, `link.txt`, etc.)

After building the project (possibly by running **make**), the generated libraries and binary files will show up in the component directories. The Draco build system is able to execute the compile step in parallel, utilizing all of the available local cores. For Unix Makefile based project, the option `-j N` should be given to **make** where the value of N is set to be the number of available cores on the local machine[3].

The generated files can be installed to the *target* directory by building the `install` target. For development, it is a common practice to set the *install* location to be the platform *target* directory. This allows the generated libraries, headers and executable files to be stored under an appropriately named directory like `gcc_mpid` or `intel10_openmpi145_rwdi` (Version 10 Intel compilers, OpenMPI version 1.4.5 and ReleaseWithDebInfo build model). Running the `install` target will add the `lib/`, `bin/`, `include/` and `config` directories under the *target* directory as shown in Figure 2.4.

In many cases, Draco based software projects will be configured and compiled alongside Draco. In this case, a binary directory for the client (e.g.: ClubIMC or Capsaicin) might be found parallel to the `draco` binary directory (see Fig. 2.4) and the the client products (headers, libraries, etc.) are installed into the same target directory used by Draco. Details on how to configure and compile Draco are found in Chap. 3, § 3.3 and § 3.4.

2.4. Coding Conventions

Draco does not adhere strictly to any code convention for formatting, but it is recommended that developers attempt to follow the styles already used in existing source code. The following subsections list specific code conventions that are required by Draco.

2.4.1. File Extensions. Table 2.2 lists file extension standards used by Draco along with a description of the expected content type.

---

[1]The GNU model normally uses the term *target* directory while the **CMake** community uses the term *binary* directory, e.g.: `PROJECT_BINARY_DIR`.

[2]By default, the build system will configure for a debug build. If MPI can be found on the local system, the build system will automatically enable parallel (MPI) features. The compiler set is chosen based on the value of environment variables `CC`, `CXX` and `FC`. If these variables are not set the build system will use whatever compiler it can find. For this example, the configuration command is '`cmake -DCMAKE_INSTALL_PREFIX=.. $draco_src_dir`'

[3]Some developers have reported good results when requesting 50% more jobs than cores. For example, for a machine that has 8 cores, the command `make -j 12` has worked well.

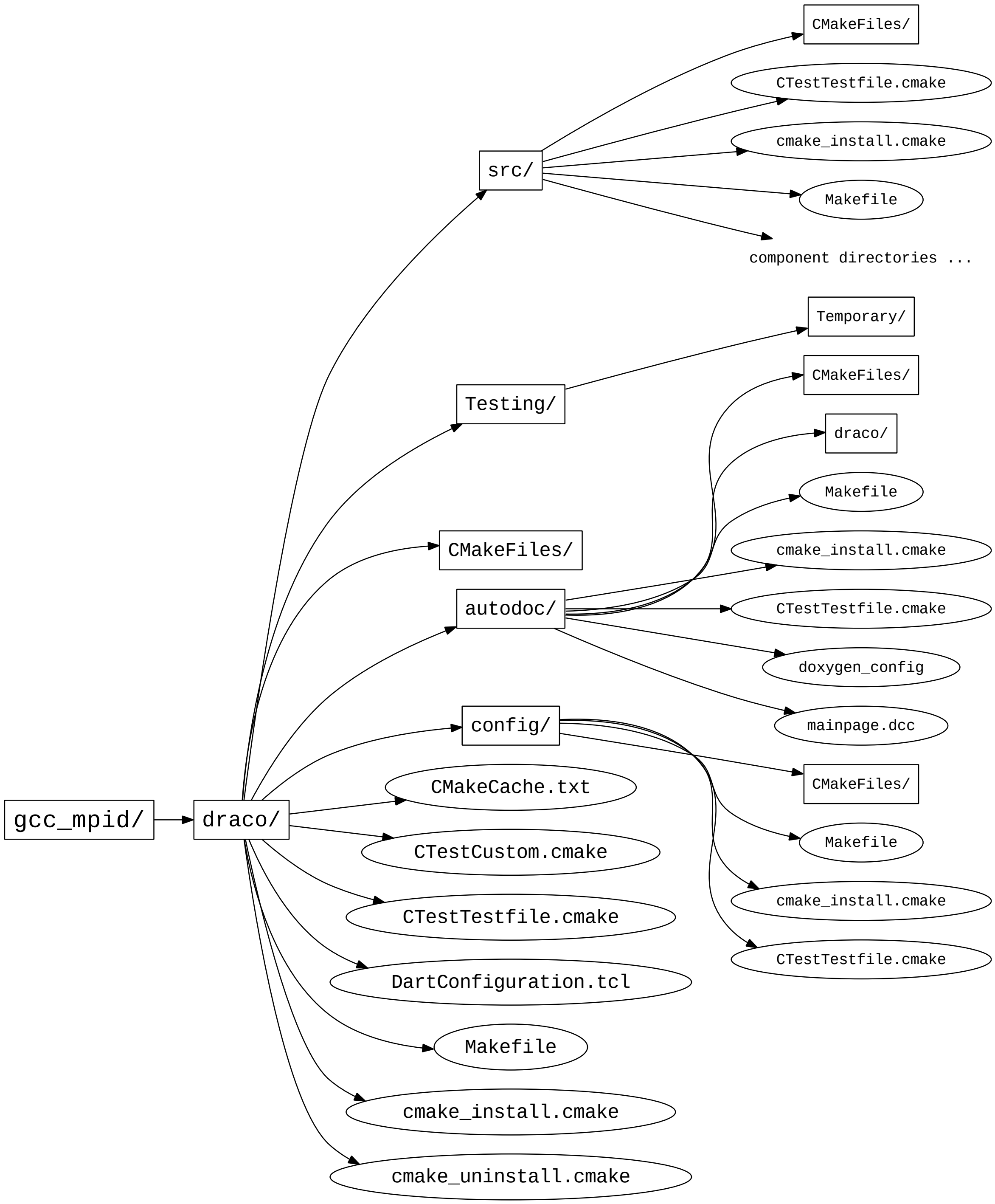


FIGURE 2.3. The Draco build tree after running **CMake** for a Makefile-based project. The `source directories` are shown in Fig. 2.2a. The `doc directories` are shown in Fig. 2.2b. The binary tree for Eclipse CDT4 - Unix Makefile based projects will match this layout but will also include two additional dot files: `.cproject` and `.project`.

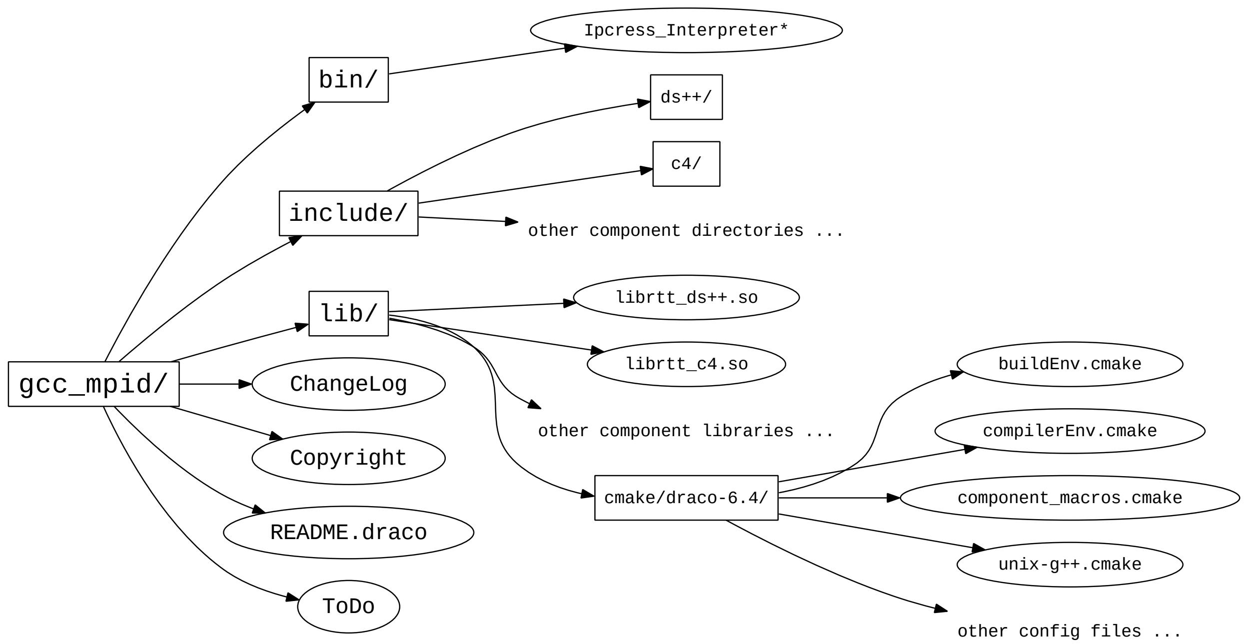


FIGURE 2.4. The Draco install tree after running `make install` for a Makefile-based project. The *install* target creates the `lib/`, `bin/` and `include/` directories under the *target* directory. The target directory is specified at *configure* time by setting the **CMake** variable `CMAKE_INSTALL_PREFIX`.

2.4.2. Tabs and editor width. Draco developers have found that code is most cleanly viewed on various editors if spaces are used in place of tabs and the maximum column width is restricted to 80 columns. The Draco-mode for **Emacs** captures the C++ indention style informally adopted by Draco. The basic indentation is 4 spaces except for namespace bocks where no indentation is used. Additional indentation may be used for case labels and multi-line math statements are normally aligned to improve readability. The class block keywords `public` and `private` are only indented 2 spaces. Comment blocks should use C++ style for C++ sources and C-style for C sources and should employ **DOxygen** [4] markup. Fortran sources should use free-format and start in in column zero.

2.5. Summary

In this chapter we have summarized the basic structure of the Draco build system. We have illustrated the requirements for the Draco build system. In the following chapters we will elaborate on the details of how to configure, build, and test Draco installations.

TABLE 2.2. Draco Coding Conventions: Standard File Extensions

File Extension	Description	Details
`.h`	C99 header file	Rarely used in Draco except for wrapping calls to vendor functions.
`.c`	C99 implementation file	Rarely used in Draco except for wrapping calls to vendor functions.
`.hh`	C++ declarations file	Common file that provide class and function declarations, including templated functions and classes.
`.i.hh`	C++ inline implementation file	C++ implementation instructions that could be placed directly in the `.hh` file. These implementations are for inlined functions or implicitly instantiated template functions. This file is always included directly by the `.hh` file after all declarations have been made.
`.t.hh`	C++ template implementation file	C++ implementation instructions for explicitly instantiated functions. This file always includes the associated `.hh` file and is only included by an explicit instantiation file(`_pt.cc`).
`.cc`	C++ implementation file	C++ implementation instructions for functions declared in the `.hh` file that are not to be inlined and are not template functions.
`_pt.cc`	C++ explicit instantiation file	C++ explicit instantiations of templated classes and functions found in the associated declarations file.
`.in`	Files that must be processed	Usually found as `config.h.in` that is processed into `config.h` and provides preprocessor definitions used by the local component.
`.cmake`	**CMake** build system file	Additional build system commands that have been extracted form `CMakeLists.txt` files to improve reuse via cmake-macros.
`.f90`	Fortran source file	Fortran files in draco are assumed to be free-format and Fortran2003 compliant.

CHAPTER 3

Configuring and Compiling Draco

This chapter describes how to configure and build Draco. All configure options will be illuminated in detail. After reading this chapter the user and/or developer will know how to build Draco on multiple platforms, for various build project systems (**Makefiles**, **Eclipse**, **XCode**, etc.) and for different options. In addition, the user will know how to build multiple versions of Draco simultaneously. To illucidate the concepts about Draco dependencies, configuration options, and build targets, § 3.6 provides several examples that show how to build Draco for various configurations.

3.1. Draco Dependencies

As mentioned in § 2.1, Draco is based on the concept of levelized design [7] . A component-level diagram is shown in Fig. 3.1. By following the dependency lines of this diagram, one can determine the exact dependencies required by each component in Draco. Thus, to compile a component static library, all of the dependencies, both explicit and implicit, must be included when linking.

In addition to the direct component dependencies illustrated in Fig. 3.1, the *pkg*`/test/` directory may require additional components for its compartmentalized unit tests. For example, `device` does not explicitly require `c4`, but `c4` and `MPI` are required for compiling the unit tests for `device`. Unit test dependencies are included in this component-level diagram and dependencies unique to the tests are represented by dotted lines. The policy in Draco is that test directories are responsible for their own template instantiations in `*_pt.cc` files. Directions showing how to include test directory dependencies are given in Chap 5.

When linking Draco into an external product, all component dependent libraries must be included. The list of necessary libraries for linking a package is given in Table 3.1. This list of dependencies is also captured in the installed file `lib/cmake/draco-6.4/draco-config.cmake`. In summary, when a Draco component is used in an external code, all libraries listed under Draco Package Dependencies (Explicit and Implicit) in Table 3.1 must be included on the link line. The packages listed under Draco Package Test Dependencies do not have to be linked. We note that Draco packages know both their package dependencies and their package test dependencies (Fig. 3.1). The external Draco client must be aware that when using a Draco package all of the packages dependencies (not package test dependencies) must be included.

Draco requires support (vendor) libraries in order to build specific components (e.g.: C4 requires **MPI**). Additionally, some optional features in Draco can only be enabled if optional external vendor libraries are available to the local build (e.g.: LAPACK_WRAP needs **LAPACK**). Table 3.2 lists all of the present components in Draco and the vendors that are required to build those components. Because Draco is based on the concepts of levelized design, each component in Draco may have dependencies on lower level Draco components. In these cases only the dependencies of the specified package are of interest. For example, the PARSER component itself requires no external vendors; however, PARSER requires C4 that does require an external vendors (MPI). The build system knows the Draco dependencies of each component; however, the external user (client) must be aware of vendor requirements in the component dependencies. If a required vendor library is not found by the Draco build system, that component will be omitted from the configuration. For example, if LAPACK is not found on the current machine, the Draco build system will not attempt to configure or compile LAPACK_WRAP. Some dependencies, like PAPI are only used for profiling and if activated anywhere, must be activated everywhere. Tables 3.1 and 3.2 can be used to determine what Draco components and what vendor libraries must be included when linking to a specific Draco components. Details on how to configure components with certain vendor options are given in § 3.3.5. Information on linking to specific libraries is given in Appendix A.

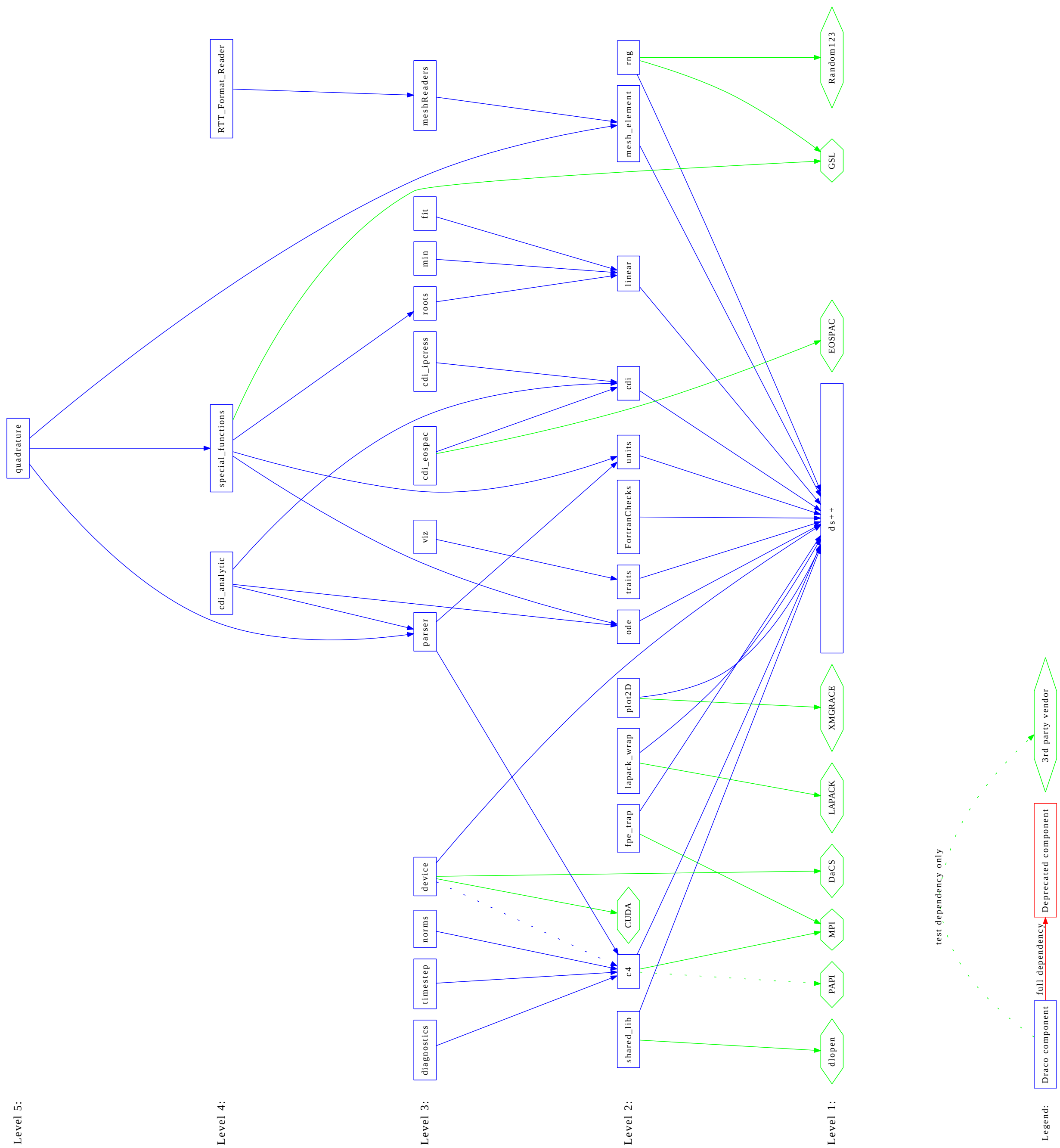


FIGURE 3.1. Component-level diagram for Draco.

To summarize the preceding we will employ a simple example. Let us assume that a user wishes to use the QUADRATURE package. Additionally, this configuration will be a parallel configuration using MPI[1]. From

[1]C4 is Draco's communication package and determines if the resulting code will be scalar or parallel.

TABLE 3.1. Listing of Draco component dependencies. The Draco component dependencies are the sum of the explicit and implicit dependencies. For general package use, only components listed under Draco Component Dependencies (Explicit and Implicit) are required. The components listed under Draco Component Test Dependencies are only required for testing.

Draco Package	Level	Draco Explicit Package Dependencies	Draco Implicit Package Dependencies	Draco Package Test Dependencies
DS++	1	-	-	-
FORTRANCHECKS	1	-	-	-
C4	2	DS++	-	-
CDI	2	DS++	-	-
FPE_TRAP	2	DS++	-	-
LAPACK_WRAP	2	DS++	-	-
LINEAR	2	DS++	-	-
MESH_ELEMENT	2	DS++	-	-
ODE	2	DS++	-	-
PLOT2D	2	DS++	-	-
RNG	2	DS++	-	-
SHARED_LIB	2	DS++	-	-
TRAITS	2	DS++	-	-
UNITS	2	DS++	-	-
DIAGNOSTICS	3	C4	DS++	-
FIT	3	LINEAR	DS++	-
MESHREADERS	3	MESH_ELEMENT	DS++	-
NORMS	3	C4	DS++	-
MIN	3	LINEAR	DS++	-
PARSER	3	C4, UNITS	DS++	-
ROOTS	3	LINEAR	DS++	-
TIMESTEP	3	C4	DS++	-
VIZ	3	TRAITS	DS++	-
CDI_IPCRESS	3	CDI	DS++	-
CDI_EOSPAC	3	CDI	DS++	-
DEVICE	3	DS++	-	C4
SPECIAL_FUNCTIONS	4	ROOTS, ODE, UNITS	LINEAR, DS++	-
CDI_ANALYTIC	4	PARSER, ODE, CDI	C4, DS++, UNITS	-
QUADRATURE	5	PARSER, MESH_ELEMENT, SPECIAL_FUNCTIONS	C4, DS++, UNITS, ODE, ROOTS LINEAR	-

Table 3.1 we see that QUADRATURE is directly dependent on the Draco packages SPECIAL_FUNCTIONS, PARSER and MESH_ELEMENT. Its implicitly dependent on ODE, DS++, UNITS, ROOTS, LINEAR and C4. From Table 3.2 we see that SPECIAL_FUNCTIONS requires the GSL library and C4 requires the MPI library. Accordingly, for a UNIX Makefile-based build, the following libraries must be included on the link line:

```
-lrtt_quadrature -lrtt_special_functions -lrtt_parser -lrtt_roots -lrtt_c4 \
-lrtt_units -lrtt_mesh_element -lrtt_ode -lrtt_linear -lrtt_ds++ \
-lgsl -lgslcblas \
-lmpi_cxx -lmpi -lopen-rte -lopen-pal
```

where `-lgsl -lgslcblas` are the GSL libraries (see § A.3) and `-lmpi_cxx -lmpi -lopen-rte -lopen-pal` are the MPI libraries (see § A.1). Additional system libraries may also be required (e.g.: `-ldl -lnsl -lutil -lm`). This example assumes that all of the following library locations are defined in `$LD_LIBRARY_PATH`.

TABLE 3.2. Vendors required by packages in Draco. Implicit are the C++ Standard Template Library (STL), the C Standard Library, and compiler libraries (`libF77`). Systems libraries (`sys/time.h`) are also not included.

Package	Level	Vendor Options	Required
DS++	1	-	-
C4	2	MPI, OpenMP	no
		PAPI	no
CDI	2	-	-
FPE_TRAP	2	MPI	no
LAPACK_WRAP	2	LAPACK, BLAS	yes
LINEAR	2	-	-
MESH_ELEMENT	2	-	-
ODE	2	-	-
PLOT2D	2	XM Grace	yes
RNG	2	GSL, Random123	yes
SHARED_LIB	2	dlopen	yes
TRAITS	2	-	-
UNITS	2	-	-
FortranChecks	2	-	-
CDI_EOSPAC	3	EOSPAC	yes
CDI_IPCRESS	3	-	-
DEVICE	3	DaCS or CUDA	yes
		MPI	no
DIAGNOSTICS	3	MPI	no
FIT	3	-	-
meshReaders	3	-	-
MIN	3	-	-
NORMS	3	MPI	no
PARSER	3	MPI	no
ROOTS	3	-	-
SPECIAL_FUNCTIONS	3	GSL	yes
TIMESTEP	3	MPI	no
VIZ	3	-	-
CDI_ANALYTIC	4	MPI	no
RTT_Format_Reader	4	-	-
QUADRATURE	5	GSL	yes
		MPI	no

Thus, even though QUADRATURE does not explicitly depend on MPI or GSL, it uses Draco packages (C4 and SPECIAL_FUNCTIONS) that do require these libraries. In practice, the Draco build system attempts to find and use full paths to vendor and Draco component libraries so that `$LD_LIBRARY_PATH` does not need to be manipulated by the developer or software user. The build system also locates and sets all of the libraries required or each vendor so that a Draco component or client only needs to specify that it should link to a set of libraries provided by the **CMake** variable `${`*${VENDOR}*`_LIBRARIES}`. We note that the installed file `lib/cmake/draco-6.4/draco-config.cmake` captures all of the interdependency and vendor dependencies shown here. This file is a list of *exported* **CMake** targets for Draco and can be used by **CMake** to *import* all of the Draco dependencies and targets into another build system.

TABLE 3.3. Products for Draco packages.

Package	include/	lib/	bin/
DS++	yes	yes	no
C4	yes	yes	no
CDI	yes	yes	no
FPE_TRAP	yes	yes	no
LAPACK_WRAP	yes	no	no
LINEAR	yes	yes	no
MESH_ELEMENT	yes	yes	no
ODE	yes	yes	no
PLOT2D	yes	yes	no
RNG	yes	yes	no
SHARED_LIB	yes	yes	no
TRAITS	yes	no	no
UNITS	yes	yes	no
FortranChecks	no	no	no
CDI_EOSPAC	yes	yes	no
CDI_IPCRESS	yes	yes	yes
DEVICE	yes	yes	no
DIAGNOSTICS	yes	yes	no
FIT	yes	yes	no
meshReaders	yes	yes	no
MIN	yes	yes	no
NORMS	yes	yes	no
PARSER	yes	yes	no
ROOTS	yes	yes	no
TIMESTEP	yes	yes	no
VIZ	yes	yes	no
CDI_ANALYTIC	yes	yes	no
RTT_Format_Reader	yes	yes	no
SPECIAL_FUNCTIONS	yes	yes	no
QUADRATURE	yes	yes	no

3.2. Draco Package Products

In Chap. 2 we insinuated that each Draco component provides a library. For example, QUADRATURE provides `librtt_quadrature.a(.so)` on UNIX systems. This is not entirely accurate. Some Draco packages, TRAITS is an example, consist only of header files and do not provide a compiled library. Users need to be aware of this fact when linking Draco components. If a Draco client uses the VIZ package, which requires TRAITS, linking `-lrtt_traits` is incorrect because TRAITS does not produce a library. Table 3.3 lists the Draco packages and their products. Users must only link against those products that make a library. Note also that package executables produced under the *pkg*`/test/` directories are not considered executable products.

3.3. Configuring Draco

We have previously mentioned that building Draco is a two-step process. First, Draco must be configured for a particular build configuration and build tool. Second, Draco is built using particular build targets. In this section we shall concentrate on configuring Draco. Note that many details about **CMake** and the Draco **CMake** macros are glossed over in this treatment. Interested readers are referred to Ref. [2] for more information. Examples that illustrate the concepts described in this section are given in § 3.6.

3.3.1. Preparing the target and binary directories. Running **CMake** is straightforward; however, setting up the target directories where various builds will take place require some consideration. In § 2.3 we described how the source tree is not necessarily where the build takes place. In fact, we advise that builds be performed in a location separate from the source tree[2]. Through this method multiple builds (debug, optimized) can be performed simultaneously using the same source files. Additionally, the source tree will not be cluttered by build-file remnants such as object dependency files.

Before running **CMake** to configure your build directory, we set up target and binary directory trees. Note that this directory can be the same as the source directory; although, we do not recommend this strategy. The target directory name should be descriptive of the particular build that is being performed. For example to build a debug version on a Linux platform with MPI support, one might make a directory entitled `linux_mpid/`. Thus, for Unix systems the user enters

```
$ mkdir -p linux_mpid/draco
```

Similar directory creation processes should be used on other platforms. Note that we do not require a `draco/` binary directory under the target directory. However, this strategy does alleviate the complexity when using Draco with other code systems. If the user is planning on using a product that emulates the Draco build system, such as Capsaicin, ClubIMC or Milagro, then parallel binary directories should be added for each product

```
$ cd linux_mpid
$ mkdir capsaicin
$ mkdir clubimc
$ mkdir milagro
```

Details on using the Draco build model in external products are reserved until Chap. 4.

3.3.2. Running CMake from the command line. The next step is to run the **CMake** command line tool inside the Draco binary directory. If you prefer, you can run the interactive **CMake** tools described in § 3.3.3 and § 3.3.4. We assume that the Draco source tree lives at `$draco_home`. To generate a build project in the binary dreictory, run **CMake** from the Draco binary directory, `linux_mpid/draco/` and provide the location of the Draco source tree as a **CMake** argument. Most importantly, we need to set the install directory, denoted by `CMAKE_INSTALL_PREFIX` on the **CMake** command line[3]. To set a build parameter on the **CMake** command line we use the **CMake** option `-D`. An example **CMake** configuration command for a UNIX Makefile configuration is illustrated here:

```
$ cd linux_mpid/draco
$ cmake -DCMAKE_INSTALL_PREFIX=.. $draco_home
```

The first option sets the install location to the target directory, `linux_mpid/`. The second option provides the source location of Draco and the controlling `CMakeLists.txt` file. Note that additional options for **CMake** are simply appended to the command-line ahead of the source location. Thus, to force the creation of static libraries for the Draco build, we would enter

```
$ cmake  -DCMAKE_INSTALL_PREFIX=.. -DDRACO_LIBRARY_TYPE=STATIC $draco_home
```

More details on the configuration options are given in § 3.3.5.

Running **CMake** in the `draco/` binary directory within the target directory (`linux_mpid/`) produces a directory tree under the `draco/` subdirectory that is parallel to the Draco source directory tree. This directory structure is illustrated in Fig. 2.3. Note that we could have run **CMake** under the target top-level directory (`linux_mpid/`). If we had proceeded with this strategy the contents of the `draco/` target subdirectory would be moved up one level.

One final point should be mentioned about `CMAKE_INSTALL_PREFIX`. **CMake**'s default location for the installation directory is `/usr/local/` on UNIX and `%ProgramFiles%` on Windows, but the build system will modify this value to point to an `install` subdirectory located beneath the Draco binary diretory (e.g.: `linux_mpid/draco/install`) if the developer does not provide another location. Thus, the user must enter

[2]The exception to this advice is when the target build tool is Eclipse CDT where there are advantages to using a *within-source-tree build*, namely, better support for **SVN** through the Eclipse IDE.

[3]This can also be set in the file `linux_mpid/draco/CMakeCache.txt`, or in the GUI interface provided by `ccmake` or `cmake-gui`.

a value for CMAKE_INSTALL_PREFIX either on the command line or via the **CMake** GUI explicitly to override the default. The suggested method is to set CMAKE_INSTALL_PREFIX to the target directory (linux_mpid/ in this example). This will allow multiple targets to be built simultaneously without risk of name collisions.

If the build configuration requires many options, the developer may choose to create a CMakeCache.txt file in the Draco binary directory before running **CMake** for the first time. All build options can set in the CMakeCache.txt file so that the configuration command only requires the location of the sources:

```
$ cd linux_mpid/draco
$ ls
CMakeCache.txt
$ cmake $draco_home
```

A sample CMakeCache.txt file is provided in the root Draco source directory. The contents of this file are also provided in Listing 3.1. Developers can copy this file to the Draco binary directory and modify, comment out or add options as needed before running **CMake** for the first time.

One final note about running **CMake** from the command line is the option to choose a project *Generator*. On UNIX systems, the default generator is Unix Makefiles, but Eclipse CDT4 - Unix Makefiles is also supported. To select an alternative generator, use the -G command line argument for **CMake**. The full list of available generators can be obtained by running 'cmake --help'.

3.3.3. Running CMake interactively from the command line. **CMake** can be run in an interactive environment by running ccmake from command line. This configuration mode is similar to the the method described above in § 3.3.2. To start the interactive configure session, navigate to the binary directory and run ccmake

```
$ cd linux_mpid/draco
$ ccmake $draco_home
```

This will start an interactive configure session that will look similar the screenshot shown in Listing 3.2. Selecting interactive command [c] (followed by [e] to exit the output review screen) will populate the configure environment with default values (be sure to turn caps lock off) resulting in something similar to what is shown in Listing 3.3. To modify the target location navigate to the CMAKE_INSTALL_PREFIX line and press enter to edit the path location. ccmake navigation and editing commands can be found by selecting the [h] option. Similarly, the build can be configured to generate static libraries by editing the DRACO_LIBRARY_TYPE value and setting it to STATIC. Once the build settings are complete, select the [c] (configure) option two more times to complete the configuration process and review the new values. To generate the controlling build project files (e.g.: Makefiles), select the [g] (generate) option. The most common build features are shown in the default ccmake environment. In some cases, the developer may need to edit an *advanced* build variable. The list of all variables can be viewed by the [t] (toggle) advanced values option.

3.3.4. Running CMake interactively through the GUI. **CMake** can be run in an interactive graphical user interface (GUI) by running cmake-gui either from the command line for by selecting the tool from the operating system's toolbar (this tool may not be available for all systems). This configuration mode is similar to the the methods described above in § 3.3.2-3.3.3. This tool can be started from any directory because the source and binary locations must always be provided manually, when using the GUI tool. As in the previous section, after providing the source and binary directory locations, select the *Configure* button to populate the Cache with default values. Before the configuration begins, the GUI will request that you select a *Generator*. This discussions assumes that you have selected Unix Makefiles as the generator. After the initial configure, the GUI should appear similar to Fig. 3.2. As in § 3.3.3, edit the values as needed and rerun the *configure* and *generate* options to generate the desired build project.

3.3.5. Configuration Options. Because Draco has many packages it must support many configurations. Additionally, some of these options can be matrixed. For example, Draco can be configured for 64-bit or 32-bit machines, scalar or parallel, with shared libraries or archived libaries, and so forth. The options that one gives during the **CMake** configure step specify most of these options. Also, the build system

Listing 3.1. A sample `CMakeCache.txt` file.

```
# CMakeCache.txt template for Draco
# $Id: compile.tex 6642 2012-07-05 14:53:27Z kellyt $

# Instructions.
# 1. Copy this file to your build directory as CMakeCache.txt.
# 2. Review and update all values in this file.
# 3. From the build directory run 'cmake /full/path/to/source'
# 4. make
# 5. ctest
# 6. make install

# ------------------------------------------------------------
# You must set these values for your build:
# ------------------------------------------------------------

# Location where 'make install' will copy files to.
# CMAKE_INSTALL_PREFIX:PATH=c:/Release-x64/draco
CMAKE_INSTALL_PREFIX:PATH=/var/tmp/$ENV{USER}/cmake/gcc/mpid/t

# VENDOR_DIR:PATH=$ENV{VENDOR_DIR}
# Windows: k:/vendors/x64-Windows
VENDOR_DIR:PATH=/ccs/codes/radtran/vendors/Linux64

# CMAKE_BUILD_TYPE == { Release, Debug, RelWithDebInfo, MinSizeRel }
CMAKE_BUILD_TYPE:STRING=Debug

# CMAKE_GENERATOR == { NMake Makefiles, Unix Makefiles,
#                      Visual Studio 9 2008, Visual Studio 9 2008 Win64 }
CMAKE_GENERATOR:STRING=Unix Makefiles

# ------------------------------------------------------------
# Review these additional settings
# ------------------------------------------------------------

# Should we compile the tests?
BUILD_TESTING:BOOL=ON

# C4 communication mode (SCALAR or MPI)
DRACO_C4:STRING=MPI

# Design-by-Contract (0-15)?
DRACO_DBC_LEVEL:STRING=7

# Keyword for creating new libraries (STATIC or SHARED).
DRACO_LIBRARY_TYPE:STRING=SHARED

# Build the tests also (default: ON)
BUILD_TESTING:BOOL=ON

# Tell the build system to print lots of extra information about its
# state (default: OFF)
DBS_PRINT_DEBUG_INFO:BOOL=OFF

# If the compiler is GCC, optional debug modes can be selected:
GCC_ENABLE_ALL_WARNINGS:BOOL=OFF
GCC_ENABLE_GLIBCXX_DEBUG:BOOL=OFF
```

has built-in intelligence that will try to make the right choice if incomplete listings for various options are specified.

Listing 3.2. The `ccmake` screen prior to running configure.

```
                                                   Page 0 of 1
 EMPTY CACHE


EMPTY CACHE:
Press [enter] to edit option                                        CMake Version 2.8.8-rc1
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Listing 3.3. The `ccmake` screen after the initial configure command.

```
                                                   Page 1 of 1
 BUILD_AUTODOC                     *OFF
 BUILD_DOC                         *OFF
 BUILD_TESTING                     *ON
 BUILD_USE_SOLUTION_FOLDERS        *ON
 CMAKE_BUILD_TYPE                  *Debug
 CMAKE_INSTALL_PREFIX              */var/tmp/gcc-mpid/d/install
 DRACO_C4                          *MPI
 DRACO_DBC_LEVEL                   *7
 DRACO_DIAGNOSTICS                 *0
 DRACO_LIBRARY_TYPE                *SHARED
 DRACO_TIMING                      *0
 DRACO_VERSION                     *6.3
 DRACO_VERSION_FULL                *6.3.20120412
 ENABLE_RNG_NR                     *OFF
 GCC_ENABLE_ALL_WARNINGS           *OFF
 GCC_ENABLE_GLIBCXX_DEBUG          *OFF
 GSL_FOUND                         *ON
 NUMDIFF                           */ccs/codes/radtran/vendors/numdiff-5.2.1/bin/numdiff
 USE_OPENMP                        *ON
 VENDOR_DIR                        *

BUILD_AUTODOC: OFF
Press [enter] to edit option                                        CMake Version 2.8.8-rc1
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

For the standard set of **CMake** options, see Ref. [2]. The full set of configure options may be examined by running the **CMake** interactive sessions (`ccmake` or `cmake-gui`). Some options may only appear on specific systems, after specific vendor installations are discovered, or for specific *generators*. This is the reason that you may need to run *configure* more than once when using the interactive versions of **CMake**. As mentioned in the previous section, the built-in configure variable, `CMAKE_INSTALL_PREFIX`, should be set explicitly (usually to the target directory) by the user to avoid installation of Draco components in `/usr/local/`. Alternately, the developer may consider the use of **CMake**'s `DESTDIR` feature (See CMake FAQ).

Configuration options come in four forms: `FILEPATH`, `PATH`, `STRING` and `BOOL`[4]. Draco policy is to name `BOOL` options prefixed with `ENABLE_` or `USE_`, although there are some exceptions to this policy and this policy is not adopted by **CMake** built-in variables. Other variable names should be prefixed with a name to provide context. This provides a sorted in list the `CMakeCache.txt` file and in the `ccmake` interface and it allows groupings to be collapsed in the GUI. This policy of using prefix context strings is a **CMake** and

[4]There are actually variable types in **CMake**. The 5th type is `INTERNAL`, but this type is not available for user manipulation.

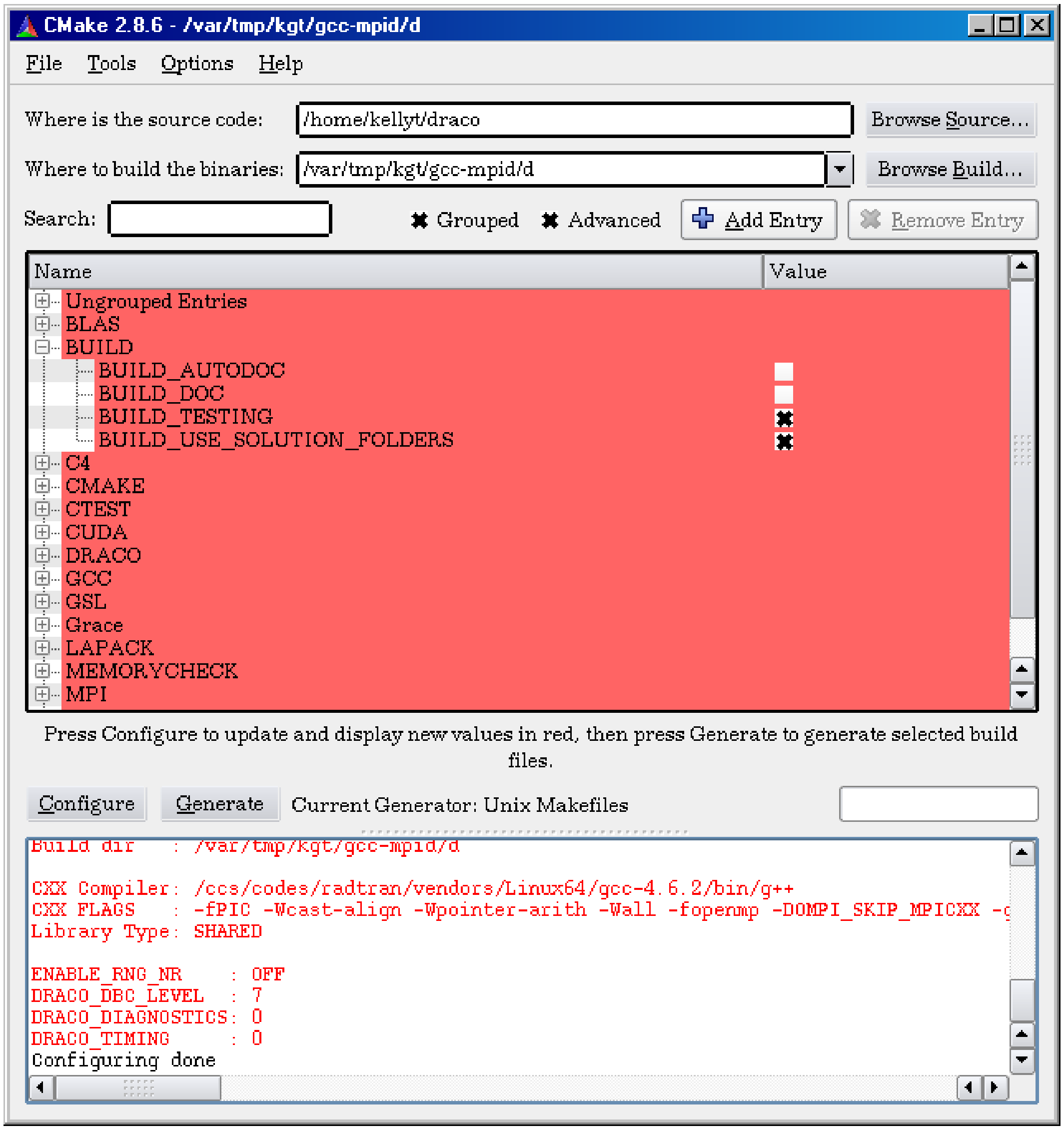


FIGURE 3.2. CMake GUI after populating cache with default values.

a Draco policy standard. Table 3.4 lists the complete set of `BOOL` switches that are unique to Draco. To turn off a switch the user has two options, `-DENABLE_SWITCH=OFF` or run an interactive **CMake** session and toggle the value. See Ref. [2] for more details concerning **CMake** variables.

The `PATH`, `FILEPATH` and `STRING` **CMake** configure options take actual string arguments. Table 3.5 lists commonly used argument based configure options for Draco. Some of these are restricted to values provided by a drop down list in the GUI. Notice that the build system will automatically populate all fields with default values so that Draco can be configured without supplying values for every possible feature. For example, the following two configurations are equivalent because the default value for `DRACO_C4` is `MPI` if MPI can be found on the local system.

```
$ cmake -DDRACO_C4=MPI $draco_home
$ cmake $draco_home
```

In each case, the C4 package is configured for parallel operation with MPI. In the first case an explicit argument is given. In the second case the default value for `DRACO_C4` is used. Not all cases have the same defaults for options and arguments. An example is the `MPI_LIBRARY` option. The default is the value

TABLE 3.4. List of BOOL options that are unique to Draco.

Option	Default Value	Description
ENABLE_RNG_NR	OFF	Selects the non-reproducible random number generator feature
USE_OPENMP	ON	Disables OpenMP pragmas when compiling Draco sources
BUILD_TESTING	ON	Allows developers to omit configuring for and building code found in test directories (disables **CTest** features)
CMAKE_VERBOSE_MAKEFILE[a]	OFF	Forces the make process to echo all commands to the screen during the build process
DBS_PRINT_DEBUG_INFO	OFF	Extremely verbose mode for **CMake**Ȧll state and variables are printed during **CMake** execution.
GCC_ENABLE_ALL_WARNINGS	OFF	When using gcc, enable more compiler warning features
GCC_ENABLE_GLIBCXX_DEBUG	OFF	When using gcc, use alternate glibc library that provides bounds checking and more safety features

[a]this option is provided by **CMake** and is not unique to Draco. It is provided here because it is a commonly used feature.

returned from the **CMake** built-in function `find_package(MPI)`. For more information about configuring and auto-discovery of vendor software see § 3.3.5.1 below.

3.3.5.1. *Vendor Libraries.* Draco uses external vendors whenever possible to reduce the ammount of code development required by Draco package developers. The following vendors are used by Draco:

- MPI communication library, see § A.1;
- GSL, GNU Scientific Library, provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting, see § A.3;
- LAPACK and BLAS provide optimized linear algebra algorithms, see § A.2;
- CUDA provides support for running threads on Graphic Processing Units, GPUs, see § A.4;
- DACS provides support for running threads on IBM cell processing units, see § A.5;
- XMGRACE provides a 2D plotting capability, see § A.6;
- RANDOM123 provides random numbers, see § A.8;

Vendors are accessed through the Draco build system via automatic discovery. Table 3.6 lists the environment and build system variables that can be manipulated by the developer to alter the discovery process. Tables 3.4 and 3.5 list additional controls that can manipulate how each vendor is used in Draco. Details on how to use these variables are given below.

Draco vendor libraries are of two types; *required* and *optional*. The type classification for each vendor is found in Table 3.2. Draco treats vendors according to the following rules:

1. Required vendor libraries are on by default and the configuration will **fail** if the libraries cannot be located;
2. Optional vendor libraries are on by default but the configuration will **pass** if the libraries cannot be located.

For example, GSL is a required vendor. If GSL cannot be found, the project will not be configured and cannot be built. If the optional vendor XMGRACE cannot be found, the configuration will be successful, but the Draco component PLOT2D will be omitted because it requires XMGRACE. Finally, if the optional vendor MPI cannot be found the configuration will be sucessful, but the C4 component will be built with the `SCALAR` option instead of `MPI`. Review Table 3.1 to determine what components may be omitted when vendor libraries are not found. This concept applies to all vendor libraries.

TABLE 3.5. List of value based options that are unique to Draco.

Option	Valid Arguments	Default Value	Description
`NUMDIFF`	`FILEPATH` to numdiff	automatic discovery	tool used by testing system for comparing output to gold standard output.
`VENDOR_DIR`	`PATH`	empty	location used by build system for auto discovery of vendor software.
`DRACO_C4`	MPI or SCALAR	MPI[a]	see § 3.3.5.1
`DRACO_DBC_LEVEL`	0-15	7[b]	see § 3.3.5.3
`DRACO_DIAGNOSTICS`	0-7	0	see § 3.3.5.4
`DRACO_LIBRARY_TYPE`	STATIC or SHARED	SHARED	toggle compilation of archive or shared object (DLL) libraries
`DRACO_TIMING`	0-2	0	see § 3.3.5.4
`DRACO_VERSION`	`STRING`	hard coded[c]	string that represents the current version of Draco. This is embedded into the installed Draco products.
`DRACO_VERSION_FULL`	`STRING`	hard coded[c]	string that represents the current version of Draco. This is embedded into the installed Draco products.
`CMAKE_BUILD_TYPE`	Debug or Release or RelWithDebInfo or MinSizeRel	Debug	choose type build type; default compiler flags are triggered based on this selection.
`CMAKE_CXX_COMPILER`	`FILEPATH` to compiler	`$ENVCXX`	C++ compiler chosen for build.
`CMAKE_C_COMPILER`	`FILEPATH` to compiler	`$ENVCC`	C compiler chosen for build.
`CMAKE_Fortran_COMPILER`	`FILEPATH` to compiler	`$ENVFC`	Fortran compiler chosen for build.
`CMAKE_INSTALL_PREFIX`	`PATH`	`$BINARY_DIR /target`	location for installing Draco(libraries, headers, executables, etc).
VENDOR_VARIABLE	varies	automatic discovery	configuration of vendors is done automatically by the Draco build system. See § 3.3.5.1.

[a]MPI if MPI can be found, otherwise SCALAR.

[b]The default is 7 for DEBUG builds and 0 for RELEASE (optimized) builds.

[c]The version string is built from hard coded values `DRACO_VERSION_MAJOR` and `DRACO_VERSION_MINOR` hard coded in the top level `CMakeLists.txt`. The patch revision number is set to the configure date for development builds. Scripts used for releases set the patch version manually.

3.3.5.2. *C4 Package Options.* C4 is Draco's parallel communication package. It uses MPI to perform message passing operations. Therefore, C4 is intimately connected to the MPI vendor. The **CMake** variable `DRACO_C4` determines how C4 should be configured. The default option and implied argument is `DRACO_C4=MPI` when an MPI installation is located by the build system. However, if C4 is set to `SCALAR` then that vendor will be turned off with their implied argument settings. Of course, the implied arguments can be overridden by using the MPI build system variables from Table 3.6. In summary, if C4 is set to `mpi`, those libraries will be turned on with all default settings. However, any of the defaults can be changed by using the MPI vendor tags that are listed in Table 3.6.

TABLE 3.6. Environment and build system variables used to specify vendors in Draco. See Tables 3.4 and 3.5 for variable defaults.

Vendor	Variable	Details
MPI[a]	`ENV{PATH}`	The build system looks for `mpirun` in the current `PATH`.
	`MPIEXEC`	The full path to the `mpirun` program. This may need to be set manually if the program has a non standard name like `aprun`.
	`MPIEXEC_NUMPROC_FLAG`	The string used to specify then number of processors to use. Defaults to `'-np'`.

	`MPI_C_LIBRARIES`
	`MPI_CXX_LIBRARIES`
	`MPI_Fortran_LIBRARIES`
		Manually specify the full paths to MPI libraries.

	`MPI_C_INCLUDE_PATH`
	`MPI_CXX_INCLUDE_PATH`
	`MPI_Fortran_INCLUDE_PATH`
		Manually specify the full path to the MPI include directory.

LAPACK[b]	`ENV{LAPACK_LIB_DIR}`	To load a specific LAPACK, ensure this variable is set to the desired location.
GSL	`ENV{GSL_INC_DIR}`	Help the build system find the desired installation by setting this environment variable.
	`ENV{GSL_LIB_DIR}`	Help the build system find the desired installation by setting this environment variable.
EOSPAC	`ENV{EOSPAC_INC_DIR}`	Help the build system find the desired installation by setting this environment variable.
	`ENV{EOSPAC_LIB_DIR}`	Help the build system find the desired installation by setting this environment variable.
RANDOM123	`ENV{RANDOM123_INC_DIR}`	Help the build system find the desired installation by setting this environment variable.
XMGRACE	`ENV{GRACE_INC_DIR}`	Help the build system find the desired installation by setting this environment variable.
	`ENV{GRACE_LIB_DIR}`	Help the build system find the desired installation by setting this environment variable.
CUDA[c]	`ENV{PATH}`	The build system looks for `nvcc` in the current `PATH`.
	`CUDA_NVCC_FLAGS`	Modify the `nvcc` compiler flags. Default is `'-arch=sm_21'`.
	`CUDA_TOOLKIT_ROOT_DIR`	If the build system cannot find `nvcc`, the developer must set this location to enable CUDA.
	`CUDA_BIN_PATH`	To use a non-standard location, set this before running **CMake**.

[a] Run `'cmake --help-module FindMPI'` for more details on the discovery process for MPI.

[b] As of Draco-6_5_0, the Draco build system imports the LAPACK targets via the **CMake** imported targets feature.

[c] Run `'cmake --help-module FindCUDA'` for more details on the discovery process for CUDA.

Some MPI installations for ASC hardware are not fully supported by **CMake**'s built in `FindMPI.cmake` routines. This is the case for *Cielito* and *Cielo*. For these systems the Draco build system employs a *toolchain* file to aid in the selection of appropriate compilers and MPI environment variables. To configure Draco on these systems use a command similar to

```
$ cmake -DCMAKE_TOOLCHAIN_FILE=$draco_home/config/Toolchain-roadrunner-spu.cmake \
        $draco_home
```

The `CMAKE_TOOLCHAIN_FILE` command line argument should appear first in the list of arguments provided to **CMake**. It is recommended that developers review the `Toolchain-catamount.cmake` file to observe how the compilers and MPI libraries are set before compiling on either of these systems.

TABLE 3.7. DBC support in Draco.

DBC Setting	DBC Functions
0	`None`
1	`Require, Remember`
2	`Check, Remember`
3	`Check, Require, Remember`
4	`Ensure, Remember`
5	`Ensure, Require, Remember`
6	`Ensure, Check, Remember`
7	`Ensure, Check, Require, Remember`
8	`None`
9	`Require, Remember, NoThrow`
10	`Check, Remember, NoThrow`
11	`Check, Require, Remember, NoThrow`
12	`Ensure, Remember, NoThrow`
13	`Ensure, Require, Remember, NoThrow`
14	`Ensure, Check, Remember, NoThrow`
15	`Ensure, Check, Require, Remember, NoThrow`

TABLE 3.8. Diagnostics support in Draco.

Diagnostic Setting	Diagnostic level description
0	`all off`
1	`low cost diagnostics enabled`
2	`moderate cost diagnostics enabled`
3	`moderate and low cost diagnostics enabled`
4	`high cost diagnostics enabled`
5	`high and low cost diagnostics enabled`
6	`high and moderate cost diagnostics enabled`
7	`all diagnostics enabled`

3.3.5.3. *Design-by-Contract.* The `DRACO_DBC_LEVEL` variable controls Draco's Design-by-Contract (DBC) machinery. DBC support ranges from `0` (lowest) to `15` (highest). The value is a bit mask similar to that used by the UNIX command `chmod`, `+1` turns on `Require`, `+2` turns on `Check`, `+4` turns on `Ensure` and `+8` disables exception throwing. If all options are activated, the Design-by-Contract is 15. Normal Debug builds use `DRACO_DBC_LEVEL=7` where all Design-by-Contract checks are active and exceptions are thrown when a contract is violated. Table 3.7 shows the DBC level for various settings of `DRACO_DBC_LEVEL`. If this option is not explicitly set by the developer (`DRACO_DBC_LEVEL` is not defined) then the DS++ package automatically sets DBC to `7`, its most aggressive setting, for Debug configrations. For Release configurations, the DBC will be defaulted to `0`, no DBC checking. For more information on the DS++ package DBC and assertion components, see the DS++ source documentation.

3.3.5.4. *Diagnostics.* The `DRACO_DIAGNOSTICS` and `DRACO_TIMING` build variables control Draco's DIAGNOSTIC machinery. The purpose of this component is allow other Draco components to collect and report diagnostic data during runtime. When `DRACO_DIAGNOSTICS` feature is turned off, the inserted diagnostic code does not cause any performance penalty because it is a compile time feature. The same is true for `DRACO_TIMING` which focuses on profiling and reporting perfomance timing statistics. The allowed values for each of these build variables are bit masks as explained in § 3.3.5.3. Tables 3.8 and 3.9 provide a description for variouls settings.

TABLE 3.9. Timing diagnostic support in Draco.

Timing Diagnostic Setting	Timing diagnostic functions
0	`all off`
1	`TIMER, TIMER_START, TIMER_STOP and TIMER_RECORD available`
2	`all functions available, including TIMER_REPORT`

3.3.5.5. *Optimization.* The optimization flags for the `CXX`, `CC` and `FC` compilers have default values established based on the compiler vendor and the selected build type (`Release`, `Debug`, etc.). These flags are established in Draco build system's configuration files `config/`*arch_compiler_vendor*`.cmake` (e.g.: `config/unix-g++.cmake` or `config/windows-cl.cmake`). In general, `Release` configurations will use optimization flags like `-O3 -funroll-loops -march=native` and `Debug` configurations will include debug symbols and no optimization, `-g -O0`. The `RelWithDebInfo` configuration uses a mixture of flags trying to produce an optimized configuration that still has the debug symbols. Draco policy to keep the source code as close to the language standard as possible. To aid the developer, the `Debug` configurations impose compiler flags that will increase the warning level and verbosity during the compilation. For example, when using the g++ compiler the flags `'-std=c++11 -pedantic -Wcast-align -Wpointer-arith -Wall'` are used for `Debug` configurations. In general, `Release` builds use the most aggressive optimizations that provide reliable and consistent results.

3.4. Building Draco

After configuration, building Draco is mostly straightforward. For UNIX Makefile build configurations, one simply enters the binary directory, or a component's binary subdirectory, and runs **gmake**. The Draco Makefiles include all of the standard targets provided by **CMake**. For more detail, see ref. [2]. The most commonly used targets are `all` and `install`. The Draco build system takes full advantage of multi-core architectures allowing multithreaded compilation of Draco. To take advantage of this features use the `'-j N'` option of **gmake**. The recommended value for the number of concurrent threads, `N`, is 50% oversubscription of the number of available cores (i.e.: 24 for a 16-core machine). Examples of various builds are reserved until § 3.6.

For other build environments like Eclipse or XCode, **CMake** provides a solution configuration that can be loaded into the IDE. Use the build environment's normal methods for compiling the `ALL_BUILD` target.

3.4.1. Building and Installing. Building and installing Draco is specific to each generated project type. The following subsections provide details for the most commonly used development environments.

3.4.1.1. *Unix Makefiles.* To build and install Draco simply enter the *target*`/draco/` binary directory and run `'gmake -j'`. At this level, **gmake** will enter each subdirectory under *target*`/draco/` and do a full build. The default targets in subdirectories under *target*`/draco/` are the same as at the top level. It should be noted that the default target, `all`, does not run unit tests or install Draco libraries or headers. You must run `'make -j install'` to tell the build system to copy the installable artifacts to the prefix directory.

3.4.1.2. *Eclipse.* To be completed later.

3.4.1.3. *XCode.* To be completed later.

3.4.1.4. *Visual Studio.* To be completed later.

3.4.1.5. *Running the Tests.* Each Draco component provides a full suite of unit tests that demonstrate and check the component algorithm's capabilities. To run the tests, run **CTest** from any location in the binary directory. If **CTest** is run from the top level, all unit tests will be run. If run from a component subdirectory, only the tests for that component will be executed. The Draco build system knows how to run the unit tests in parallel taking advantage of all available hardware resources. It is recommended that the **CTest** command be issued with the `'-j N'` option, where `N` is the number of concurrent threads that should be used. For testing purposes, it is better to avoid over-subscription of the machine's hardware.

CTest provides many options for running tests: selecting a subset of tests to run; running with different output verbosity, etc. The developer should review the **CTest** documentation found at Ref. [2] and by using the 'ctest --help' command. In particular, the -VV options selects full verbosity for tests and the -R option selects all tests whose names match a provided regular expression. It is Draco policy that tests names will provide both the component name and the number of MPI ranks used (if any) in the test name. This policy allows the developer to run all C4 tests by using the command

```
$ ctest -R c4
```

or all 4 processor MPI tests could be selected by the command:

```
$ ctest -R _4
```

A list of available test can be obtained using the -N option to **CTest**.

3.4.1.6. *Additional Observations and Features.* At each target directory level the Draco build system knows all of the component dependencies so the developer can start the build at any place in the binary tree. For example, when compiling from the C4 component directory, the build system will check to see if the DS++ library has been compiled. If not, then the DS++ library will be built before compilation of C4 sources begins. Even in this situation the build system remains fully aware of threading and it is recommended that a parallel build process be performed unless the developer is trying to debug a build system error. This aspect of the build system is a feature for Draco developers. It allows the developer to only compile or recompile sources that are required for building the desired target. One drawback is that other components in parallel directories may be modified during a targeted compile and the developer should remain aware of these dependencies as illustrated in Fig. 3.1 and are listed in Table 3.1.

An additional feature of the Draco build system is that Draco will automatically rerun the **CMake** configuration step if any of the configuration system files have been modified. Thus, if `CMakeLists.txt` or any of the files from the `config` source directory are changed then the build process will first reconfigure the entire project.

3.4.2. Build Targets. A detailed discussion of all the build targets provided by **CMake** [2] is beyond the scope of this text. What follows is a brief description of the build targets in Draco and what operations they perform.

- `all`: (default) build all products at the current level and in all subdirectories. If configuration files have been modified, rerun **CMake** to reconfigure the project before compilation begins.
- `install`: build all products at the current level and in all subdirectories; then install the products in the locations specified by `CMAKE_INSTALL_PREFIX`. All build products are compiled before any are installed.
- `check`: build all products at the current level and in all subdirectories; then run **CTest** to execute all unit tests. This target does not install any products (this behavior is different than older versions of Draco).
- `clean`: clean the compiled files (`*.o`, libraries and executables) from the target sub-directories.
- `rebuild_cache`: rerun the **CMake** configure process and regenerate all project files (e.g.: `Makefiles`, `config.h`, etc.).
- `edit_cache`: run the **CMake** editor to allow the developer to edit the configuration variables.
- `test`: run **CTest** to execute all unit tests.
- `Experimental`: configure, compile, run the tests and submit the results to the Draco **CDash** dashboard.
- `Lib_`*pkg*: Compile the library for Draco component *pkg*.
- `Ut_`*pkg*`_`*test*`_exe`: Compile the unit test executable for test *test* for the Draco component *pkg*.
- `help`: provides a list of available targets.

These targets have been designed to satisfy the needs of users, who perform one-time global builds, and developers, who perform multiple local builds.

3.5. Recommended Practices

Although Draco can be configured and built in any number of ways, we have a set of "standard" recommended practices that are followed by Draco team members. This methodology for configuring and building Draco is summarized in the following steps:

1. Checkout a version of Draco from **SVN**; the location of which is `$draco_home`.

2. Make a target directory that appropriately describes the configuration options; we call this directory *target*/.
3. Make a `draco/` binary subdirectory under the target directory, ie. *target*/`draco/`.
4. Run **CMake** in *target*/`draco/` with the appropriate options for this configuration. Set `-DCMAKE_INSTALL_PREFIX=`*target*/. Thus, the configure line is:

   ```
   $ cmake -DCMAKE_INSTALL_PREFIX=‘pwd‘/.. options $draco_home
   ```

5. Run **gmake** in *target*/`draco/`;

   ```
   $ gmake -j install
   ```

   This step will build and install all of the Draco products from each subdirectory under *target*/`draco/`. The headers will be installed in *target*/`include/`. The libraries will be installed in *target*/`lib/`. And the executables will be installed in *target*/`bin/`.

This procedure simplifies adding an external code system that uses, and is based on, Draco. For example, ClubIMC uses Draco as a build-model template. Thus, we can add a *target*/`clubimc` directory and configure, build, and install ClubIMC in the same location as Draco products. Details on this process are given in § 4.2.

3.6. Examples

To illucidate some of the concepts that we have described in this chapter, we proceed to show some configuration and build examples. The following examples give a cross section of the processes that Draco users and developers will use.

EXAMPLE 3.1. Build a scalar version of Draco on a Linux platform using the Makefile generator. GSL libraries are found in `$LD_LIBRARY_PATH`. The Draco source directory is `/usr/tmp/joe/draco`. We want Draco installed in `/usr/local/draco`.

SOLUTION TO 3.1. First, we need to make a target directory. In § 3.3.1 we advised not to use the source directory as the build directory. We will follow this policy and make our target directory `draco_target`:

```
$ cd /usr/local/draco
$ mkdir draco_target
```

Note that we could have created a directory named `draco/` instead of `draco_target/`. We used a different name to illustrate the independence of the target-build directory. Now, we configure Draco according to the specification in Ex. 3.1. This configuration is scalar so we must specify alternate settings for C4. Additionally, all of the required vendors are located in default locations.

```
$ cd draco_target
$ cmake -DCMAKE_INSTALL_PREFIX=‘pwd‘/.. -DDRACO_C4=SCALAR \
        /usr/tmp/joe/draco/draco_config
```

All other defaults are used except the explicit setting for `DRACO_C4`. Finally, we want to do a build and install of all Draco products; thus, according to § 3.4, we must enter

```
$ gmake -j install
```

Generally, it is better run the build (`all` target and then run the unit tests via **CTest** before running the `install` target. This gives the develper a chance to ensure that all tests pass before the build is installed.

```
$ make -j
$ ctest -j
$ make -j install
```

□

Example 3.1 is straightforward. We will now give a series of examples and solutions that involve more detailed configurations and builds. At this point, we will only show the steps in the solution procedure. The details about each step can be inferred from § 3.3 and 3.4.

EXAMPLE 3.2. Build a version of Draco with MPI. Additionally, this build takes place on a Linux platform with OpenMPI and GSL loaded as a modules.

SOLUTION TO 3.2. Before proceeding to the solution, we note that the Draco build system knows about the OpenMPI module. Thus, setting the include and library paths for MPI is not required.

```
$ cd /usr/local/draco
$ mkdir draco
$ cd draco
$ cmake -DCMAKE_INSTALL_PREFIX=‘pwd‘/.. /usr/tmp/joe/draco ..
$ make -j
$ ctest -j
$ make -j install
```

□

EXAMPLE 3.3. Build a version of Draco with MPI and optimization set to level 3. Turn off all DBC support. Use the mpich version of MPI that is installed in `/usr/local/`. GSL is in `/usr/local/gsl`. This could be considered a production version of Draco.

SOLUTION TO 3.3. We will proceed in a slightly different manner than the previous examples. Here we will use environment variables to determine the location of GSL. We assume the **BASH** shell is in use.

```
$ export GSL_INC_DIR=/usr/local/gsl/include
$ export GSL_LIB_DIR=/usr/local/gsl/lib
$ cd /usr/local/draco
$ mkdir draco
$ cd draco
$ cmake -DCMAKE_INSTALL_PREFIX=‘pwd‘/.. -DCMAKE_BUILD_TYPE=RELEASE /usr/tmp/joe/draco
$ make -j
$ ctest -j
$ make -j install
```

The MPI setup is automatic assuming that `mpirun` for mpich is available from the environment variable `PATH`. Setting the `CMAKE_BUILD_TYPE=RELEASE` sets the optimization level to 3 and turns off DBC. If we had wanted to keep the DBC turned on for the optimized build, we would need to provide an additional argument to **CMake** `'-DDRACO_DBC_LEVEL=7'`. □

3.7. Summary

We have given a tutorial on how to configure and build Draco. The component and vendor dependencies in Draco have been listed in § 3.1. Details on configuring and building Draco have been given in § 3.3 and § 3.4. We have tied these concepts together with several examples in § 3.6.

CHAPTER 4

Using the Draco Build Model in External Codes

This chapter illustrates how to use the Draco and the Draco build model in external code systems. One of the advantages of Draco is that it is independent from its clients. Thus, one may use Draco without having any direct connections to its build system. All that is required is linking to the Draco libraries that one wishes to use. Details on how to use Draco as a client are given in § 4.1.

Code systems that use Draco heavily may find benefits in emulating the Draco build model. This prevents these systems from having to define all of Draco's dependencies. By using the Draco build system they get the correct compile and link-line options automatically. We discuss how to use the Draco build system in external codes in § 4.2.

4.1. Using Draco in External Codes

As mentioned in the previous section, Draco and external clients are separate entities. Thus, any build system that the external client desires is acceptable. This can range from a simple "compile-script" to a detailed **autoconf-gmake** or **CMake**-based build system. Describing all possible build systems that use Draco is beyond the scope of this, or any, text. However, we will point out some useful items that should be considered when using Draco as an external client.

First, clients of Draco should follow the Draco practice of setting include paths to the Draco include directory specified by `CMAKE_INSTALL_PREFIX`. Thus, headers should be included in source code using

```
#include "pkg/header.hh"
```

For example, if a client wishes to use the DS++ smart pointer class then the client source code should contain the following:

```
#include "ds++/SP.hh"
```

The Draco headers are included on the compile and link-lines with the following statement

```
-I /usr/local/draco/include
```

where `/usr/local/draco` is the Draco installation location. By following this convention the client will avoid name clashes among Draco packages.

Draco clients must remember to include all dependencies for a particular Draco package. These dependencies are both implicit and explicit for Draco packages and vendors. Tables 3.1 and 3.2 can be used to determine the full list of dependencies for a particular Draco package. Additionally, clients must remember to include the same vendor installations as the ones supplied to Draco. For example, if Draco used an OpenMPI version of MPI then the client should use the same vendor and version when linking.

As stated in § 2.1, Draco uses the generic programming archetype. Thus, many classes and functions in Draco are templates and are not compiled into libraries. Draco does not support implicit instantiation [21] of template classes and functions. Thus, the user must provide explicit instantiation source code for Draco template components. Draco template code is stored in `.t.hh` and `i.hh` files. These files are installed in the `include/` directory along with the rest of the Draco package headers. Specific information on the generic programming approach used in Draco is given in ref. [24].

4.2. Emulating the Draco Build System

The Draco build system can be emulated at varying levels ranging from full to minor. We will describe a method for using the Draco build system directly. If this method is used the Draco build system requires little or no modification in the external system. External code systems that utilize the Draco build system in this manner are Capsaicin, ClubIMC, Milagro, and Wedgehog.

The most direct method of incorporating the Draco build system into an external product is to mimic the structure of the Draco source tree. This process has three steps:

1. Setup the external code source tree in the same manner as Draco; setup a top-level directory, a `src/` directory, a `doc/` directory, and a `pkg_config/` directory. Components of the code should be placed in `src/`*pkg*`/` directories.
2. build system macros that are specific to the external code or replace Draco specific versions should be placed in the `pkg_config/` directory.
3. When configuring, set the `CMAKE_INSTALL_PREFIX` tag to the same location as the installed Draco components.

If these steps are followed, then the external code system should properly attach Draco libraries and find Draco headers. However, the external code developer should be aware that external code components will be installed in the same location as Draco components.

The Draco design allows both the Draco build system and its installed components to be used in an external code system. In this model, Draco must be installed in an accessible location. The external code system can load the Draco build system by adding the `$draco_install/lib/cmake/draco-6.4/` directory to the `CMAKE_MODULE_PATH` allowing the external code system to use any or all of the Draco build system.

Each package subdirectory in the external code system must have a `CMakeLists.txt` file as described in § 5.1. The contents of these files are described in § 5.2. These files will be very similar to the `CMakeLists.txt` files found in Draco. The fundamental difference is that the external code will add its own package dependencies to the existing Draco dependencies. Chapters 5 and 6 go into greater detail on package design in the Draco build system.

Additionally, the external code may require vendors that are not supported by Draco. Thus, vendor discover and setup in addition to those shown in Table 3.6 may need to be defined. Defining vendors using Draco-like methodology is described in Chap. 6.

On a final note, any deviations from the Draco build model in an external code system are perfectly acceptable. The external code client and Draco are independent entities. In many cases, exactly emulating Draco is the most straightforward way of incorporating Draco into an external code package.

4.3. Summary

We have given directions on how to use Draco in an external code system. In § 4.2 we have shown how to directly use Draco in a code system that makes heavy use of the Draco component library. In § 4.1 we have given some pointers to codes that are Draco clients but simply want to link to Draco components without using the Draco build system.

CHAPTER 5

Adding a Component to Draco

5.1. Overview

New Draco components should be added in subdirectories under `draco/src/`. Each Draco package may have its own additional subdirectories under `draco/src/`*pkg*. Figure 5.1 illustrates a representative package directory. A component directory should conform to following guidelines:

- each component directory should have a `test/` subdirectory that holds component test code, these tests are also used to verify package builds as described in § 3.4. Most unit tests should use the features provided by the `ds++/ScalarUnitTest` or `c4/ParallelUnitTest` helper classes,
- each component should have `autodoc/` and `doc/` subdirectories. The `autodoc/` directory should provide at least one file, *pkg/*`.dcc.in`, that provides basic information about the component that can be included in the compiled HTML autodoc for Draco,
- all subdirectories in the package should have the same configuration and build options,
- the component should use as many of the **CMake** macros defined by the Draco build system as possible to avoid duplicate code,
- special configuration requirements for a package may be added to that package's `CMakeLists.txt` file.

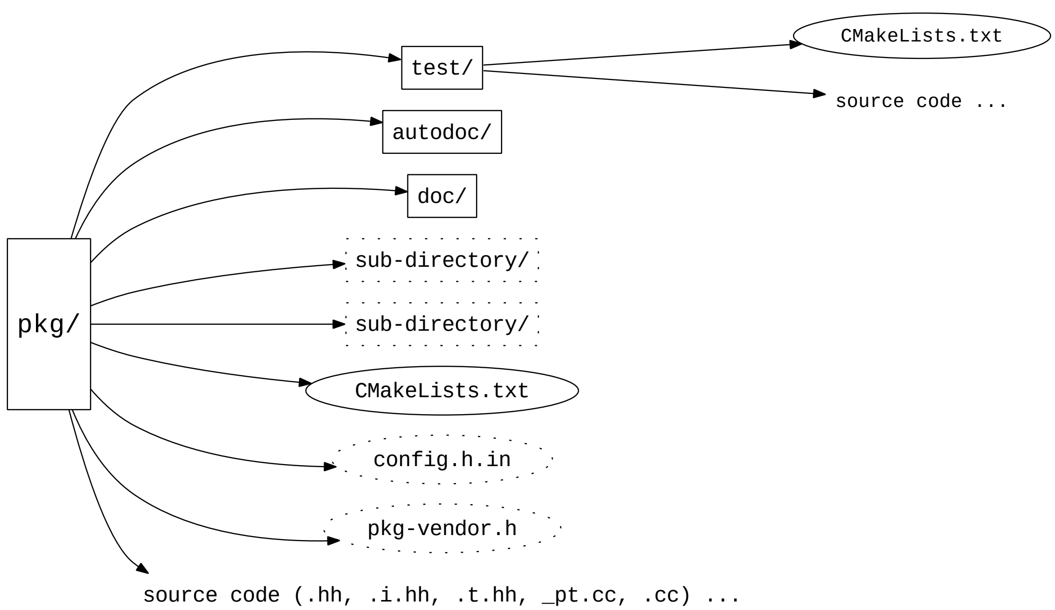


FIGURE 5.1. Standard package directory configuration in Draco.

In general, most packages will be able to use another component's `CMakeLists.txt` and `autodoc/`*pkg*`.dcc.in` as a templates. Customization of component `CMakeLists.txt` files is treated in § 5.3.

TABLE 5.1. Draco build system package files.

Package Configuration Files	Description
`CMakeLists.txt`	file contains cmake instructions for generating directly local project files (e.g.: Makefiles).
`config.h.in`	package specific environment configuration file that will be processed into `${PROJECT_BINARY_DIR}`/*pkg*/`config.h`
pkg-*vendor*.`h`	package-specific vendor include headers

At a minimum, each package requires a `CMakeLists.txt`. To set configuration options on a package-by-package basis, the files `config.h.in` and *pkg*-*vendor*.`h`, may also be required. Table 5.1 lists all of the possible configuration files that can be found in a Draco package. All of these files are explained in § 5.2.

Draco does not provide templates for package-level `CMakeLists.txt` and `config.h.in` files, but the contents of these files are straight forward for most cases and the developer can use an existing component's `CMakeLists.txt` and `config.h.in` files as templates.

Draco macros are defined in **CMake** configuration files located at `draco/config/`. These macros are used by **CMake** to generate the build logic and platform checks that go into the generated project files for the selected build type (e.g: Makefiles). The macros are divided into separate files to provide appropriate groupings as presented in Table 5.1.

TABLE 5.2. Draco configuration macro files.

Type	Configuration File	Description
Vendor support	`FindGSL`	use **CMake**'s `find_package_handle_standard_args` to locate and register settings for GSL.
	`FindGrace`	use **CMake**'s `find_package_handle_standard_args` to locate and register settings for XM Grace.
	`FindLIBSCI`	use **CMake**'s `find_package_handle_standard_args` to locate and register settings for Cray's LibSCI (an optimized LAPACK replacement.
	`FindEOSPAC`	use **CMake**'s `find_package_handle_standard_args` to locate and register settings for libeospac6 (access Sesame physical data).
	`FindRandom123`	use **CMake**'s `find_package_handle_standard_args` to locate and register settings for the Random123 random number generator.
	`vendor_libraries`	controlling macro that looks for requested vendor libraries by calling `find_package()`.
Toolchain files	`Toolchain-roadrunner-ppe`	allow use of **CMake**'s cross-compile feature to aid the build system configuration for Roadrunner cell front end.
	`Toolchain-roadrunner-spu`	allow use of **CMake**'s cross-compile feature to aid the build system configuration for Roadrunner cell back end.
Primary configuration	`apple-clang`	sets compiler flags for the clang and clang++ compilers on OS/X
	`buildEnv`	establish top-level defaults like `DRACO_DBC_LEVEL`
	`compilerEnv`	controls compiler discovery and calls appropriate compiler flag setup routines

Continued on next page

Type	Configuration File	Description
	`dracoVersion`	establishes the Draco version tag that is embedded in the code
	`dracoTesting`	establishes the `check` build target. Test registration macros are provided in `component_macros.cmake`.
	`unix-g++`	sets compiler flags for GNU C and C++
	`unix-gfortran`	sets compiler flags for GNU Fortran
	`unix-ifort`	sets compiler flags for Intel Fortran
	`unix-intel`	sets compiler flags for Intel C and C++
	`unix-pgf90`	sets compiler flags for PGI Fortran
	`unix-pgi`	sets compiler flags for PGI C and C++
	`unix-ppu`	sets compiler flags for GNU C and C++ on PPC PPU architectures
	`unix-spu`	sets compiler flags for GNU C and C++ on PPC SPU architectures
	`windows-cl`	sets compiler flags for Microsoft C and C++
	`windows-ifort`	sets compiler flags for Intel Fortran on Windows
Platform checks	`platform_checks`	macros that probe the local system for available features, headers, etc.
	`cxx11/`	macros and tests for checking C++11 feature availability
Draco build system debug features	`debug_macros`	provides **CMake** macros that can be used to debug the build system
Component configuration	`component_macros`	provides build system macros (`add_component_library`, `add_scalar_test`, etc.) for use in component level `CMakeLists.txt` files.
Documentation configuration	`doc_macros`	these macros simply the generation of a `CMakeLists.txt` file needed for generating documentation from LaTeX sources.
	`doxygen_config.in`	this file will be processed if `BUILD_AUTODOC=ON` and contains the configuration settings for **DOxygen** processing of source code to generate HTML developer documentation.
General helper macros	`parse_arguments`	a helper program that simplifies argument processing for **CMake** macro definition.
	`cmake_uninstall.cmake.in`	this template is processed by the build system to keep track of generated files that can be uninstalled.
	`configureFileOnMake`	this script can be used by an `add_custom_command` to generate files/scripts/etc. on the fly.
	`draco-config-install`	provides support for **CMake**'s exported targets and dependencies. This file is processed and installed as `draco-config.cmake`.

In general, Draco package developers need only be concerned with *Component configuration* set of macros. The remaining macros are the the domain of Draco system developers and are described in Chap. 6.

Most component directories use the standardized `CMakeLists.txt`. Simple modifications to the standard component and test `CMakeLists.txt` is achieved inserting **CMake** scripting, including specialized Draco build system configuration macros, directly into the local `CMakeLists.txt`. The use of these files and macros is summarized in § 5.2.

In summary, each package has a `test/` directory for component tests and an `autodoc/` directory for documentation that can be generated by **DOxygen**. Additional subdirectories that contain package components may be included. All package subdirectories are configured using the same options. Also, components may use a unique scripting commands from within `CMakeLists.txt` if they require special functionality that does not exist in the standard `CMakeLists.txt` file. We will now turn our attention to a more detailed description of the configure files.

5.2. Package Files

In this section we give expanded descriptions of the default package-dependent files listed in Table 5.1. We will not go into great detail about the **CMake** macros that are defined in the Draco system. That discussion is reserved until Chap. 6. We will concentrate primarily on the three file-types that are found in each package directory: `CMakeLists.txt`, `config.h.in`, and *pkg*`-vendor.h`. We reserve a discussion of makefile customization until § 5.3.

pkg`/CMakeLists.txt`: The `CMakeLists.txt` provides all of the build instructions needed to generate a set of build instructions for the local file scope. In the case of a Unix Makefiles build project, the generated Makefiles are created based on the instructions provided in `CMakeLists.txt`. We will step through a standard package `CMakeLists.txt` script to learn how to properly instruct **CMake** to generate a component level build project. An example is the QUADRATURE `CMakeLists.txt` script illustrated in Listing 5.1.

Notice that the `CMakeLists.txt` file has seven basic sections. Within these sections there are both required and optional macros. Table 5.2 lists all of the usable macros in a Draco `CMakeLists.txt` file. Customizing a `CMakeLists.txt` script is explained in § 5.3.

pkg`/test/CMakeLists`: This `CMakeLists.txt` provides all of the build and execution instructions needed to generate and execute unit tests.

`src/CMakeLists`: This `CMakeLists.txt` provides all of the build instructions concerning compiler setup, vendor setup and specifies what components to include in the current build.

`config.h.in`: The `config.h.in` file contains `#define` and other cpp macros needed to build the package. By isolating macros to the `config.h.in` file, compile line bloat is drastically reduced. Additionally, each package's macro requirements are isolated from other packages. A symptom of placing `-D`*option* on the compile line is that these definitions tend to get probagated throughout the build cycle.

TABLE 5.3. Macros used by the `CMakeLists.txt` files. Macros that require arguments are indicated by `()` following the macro name.

Macro	Required	Description
Section 1: Project declaration		
`cmake_minimum_required( VERSION 2.8 )`	yes	states that this file uses features of **CMake** that were not introduced until version 2.6. If an older version of **CMake** is used, a fatal error will be thrown.
`project( quadrature CXX )`	yes	This command registers the component name (must be unique within the Draco project) as the **CMake** project name and sets the source code language.
Section 2: Source code registration		
`file( GLOB sources *.cc )`	no	This regular expression command selects all `*.cc` files and assigns them to the list `$sources`.
`file( GLOB headers *.hh )`	no	This regular expression command selects all `*.hh` files and assigns them to the list `$headers`.

Continued on next page

Macro	Required	Description
`if( MSVC_IDE ... )`	no	This if-block appends all of the header files to the list of C++ sources if the project generator is an IDE where we want easy navigation to both sources and headers.
Section 3: Include directives		
`include_directories( ${PROJECT_SOURCE_DIR} ... )`	no	This command instructs the build system to look in the provided list of directories to satisfy include directives found in the source code. For Unix Makefiles, this command results in `-I`*dir*/ on each compile line. The command uses **CMake** variables that contain the appropriate paths. In this context, the quotes are important.
Section 4: Compile directives		
`add_component_library( TARGET Lib_quadrature TARGET_DEPS Lib_special_functions;Lib_parser;Lib_mesh_element LIBRARY_NAME ${PROJECT_NAME} SOURCES "${sources}" )`	yes	Generate a library from sources, `${sources}`, whose name is based on `${PROJECT_NAME}` and has the build target key `Lib_quadrature`. The build target `Lib_quadrature` must be linked against build targets (libraries) `Lib_special_functions`, `Lib_parser` and `Lib_mesh_element`.
Section 5: Install commands		
`install( TARGETS Lib_quadrature EXPORT draco-targets DESTINATION lib )`	no	The file represented by the build target `Lib_quadrature` (the component library) is to be installed into the `${CMAKE_INSTALL_PREFIX}/lib` directory.
`install( FILES ${headers} DESTINATION include/quadrature )`	no	The files represented by the **CMake** variable `${{headers}}` are to be installed into the `${CMAKE_INSTALL_PREFIX}/include/quadrature` directory. This target should be included in the list of exported targets that can be picked up by another **CMake**-based build system.
Section 6: Unit Tests		
`if( BUILD_TESTING ) ...`	yes	This logic block instructs **CMake** to include the test directory when generating build project unless the developer has explicitly set `BUILD_TESTING=OFF`.
Section 7: Autodoc		
`process_autodoc_pages()`	no	If this package provides **DOxygen** documentation, process the source files when instructed to build the `autodoc` build target.

5.3. Customized Packages

This is an advanced topic and requires detailed knowledge of both **CMake** and the Draco build system. If needed, this section can be provided in a future revision of this document.

In the future these common issues may need to be addressed with detail:

5.3.1. Platform Checks. See **CMake** documentation and existing build files.

5.3.2. Generator executables. To generating a tool that is used to generate a source file see **CMake** documentation (FAQ).

5.3.3. Complicated dependencies. To generating a tool that is used to generate a source file see **CMake** documentation (FAQ).

```
cmake_minimum_required(VERSION 2.8)
project( quadrature CXX )

# ------------------------------------------------------------------------------ #
# Source files
# ------------------------------------------------------------------------------ #

#file( GLOB template_implementations *.t.hh *.i.hh )
file( GLOB sources *.cc )
#file( GLOB explicit_instantiations *_pt.cc )
file( GLOB headers *.hh )
#list( REMOVE_ITEM headers ${template_implementations} )

# Make the header files available in the IDE.
if( MSVC_IDE OR ${CMAKE_GENERATOR} MATCHES Xcode )
   list( APPEND sources ${headers} )
endif()

# ------------------------------------------------------------------------------ #
# Directories to search for include directives
# ------------------------------------------------------------------------------ #

include_directories( ${PROJECT_SOURCE_DIR}       # sources
                     ${draco_src_dir_SOURCE_DIR} # ds++ header files
                     ${dsxx_BINARY_DIR}          # ds++/config.h
                     ${GSL_INCLUDE_DIRS}
                     ${MPI_INCLUDE_PATH}
)

# ------------------------------------------------------------------------------ #
# Build package library
# ------------------------------------------------------------------------------ #

add_component_library(
   TARGET       Lib_quadrature
   TARGET_DEPS  "Lib_special_functions;Lib_parser;Lib_mesh_element"
   LIBRARY_NAME ${PROJECT_NAME}
   SOURCES      "${sources}" )

# ------------------------------------------------------------------------------ #
# Installation instructions
# ------------------------------------------------------------------------------ #

install( TARGETS Lib_quadrature EXPORT draco-targets DESTINATION lib )
install( FILES ${headers} DESTINATION include/quadrature )

# ------------------------------------------------------------------------------ #
# Unit tests
# ------------------------------------------------------------------------------ #

if( BUILD_TESTING )
 add_subdirectory( test )
endif()


# ------------------------------------------------------------------------------ #
# Autodoc
# ------------------------------------------------------------------------------ #

process_autodoc_pages()
```

LISTING 5.1. `CMakeLists.txt` file for the QUADRATURE package.

CHAPTER 6

Extending the Draco Build System

This chapter should cover the advanced topic of documenting, maintaining and writing new build system (**CMake**) macros. However, covering these items in detail appears to have a limited return-on-investment and will be omitted from the build document in this iteration.

CHAPTER 7

Quick Start

This chapter presents the new Draco user with a *quick start guide* to developing applications and libraries in the Draco environment. This guide consists of a simplified description and specific examples that demonstrate how to check out, configure and build the Draco code library. We will also cover building some of Draco's clients including ClubIMC, Wedgehog, Milagro, and Capsaicin. In addition to building these libraries and clients, we will also cover setting up the shell environment, accessing recommended scripts and tools, and using third party vendors (e.g.: **GSL** [25] or **OpenMPI** [26]). We will also point the reader to other Draco related documents where more detailed information can be located.

This chapter is designed to give the new user an introduction to working in the Draco environment. For a more in depth description of the Draco code development philosophy, framework, and build system please read the remainder of this document and Refs. [4, 5, 27–31]

This chapter originally appeared as a stand-alone memo [32].

7.1. Local Tools

7.1.1. Draco File Sharing Group. The Draco team maintains a Unix file sharing group named `draco`. You must be a member of this group to access Draco source code or development environment files. Membership is controlled by the current super-users as listed on `register.lanl.gov`.

7.1.2. Important Locations. Important file and web locations for Draco are listed in Table 1.5.

7.1.3. Mailing List. The Draco team maintains a mailing list used for announcements and general Draco discussion. To subscribe to this list, send an email to `listmanager@listserv.lanl.gov` with the body "subscribe draco". For more information about mailing lists at LANL you can send an email to the same address with the message "help". Some list management functions can also be completed by visiting `register.lanl.gov`.

If you are not sure if you are subscribed or not you can send the command "which" to `listmanger@listserv.lanl.gov` to see what LANL lists you are subscribed to. Table 1.6 below lists a few other mailing lists that may be useful for CCS–2 developers to subscribe to.

7.1.4. SVN commit notifications. The Draco effort uses **SVN** for revision control. When changes are committed to the repository email is sent to team member who have requested the notification. These notifications are managed by the LANL listserver. Table 1.6 summarizes the active lists. Developers are encouraged to subscribe to lists corresponding to projects that they are involved in.

7.1.5. Nightly regression reports. The Draco effort uses **CDash** for reporting nightly regression results. Draco developers are encouraged to visit the dashboard (`coder.lanl.gov/cdash`) regularly to view the nightly reports. Developers are also encouraged to register their email address with the dashboard to receive email reports of regression failures or warnings.

7.1.6. Issue Tracking. Draco officially uses the LANS Team Forge issue tracking system to manage issue, bug and feature tracking. Team Forge can be accessed by opening a web browser from the Yellow network (i.e.: inside the LANL firewall) to `tf.lanl.gov`.

Starting Q4CY12, many key developers have started using CCS-2's Redmine project management console to manage issues, bugs, features, releases, etc. The Redmine console can be accessed by opening a web browser from the Yellow network to `coder.lanl.gov/redmine`. If you need an account please contact someone who is in the Draco systems developer group (see Table 1.3).

You are encouraged to browse the list of known bugs/issues of the project you are working on.

7.1.7. C++. The Draco code library is primary written in C++. The team recommends that the GNU g++ compiler (v. 4.7+[1]) be used for development of the Draco code library. Recent versions of g++ have shown excellent adherence to the C++ standard and very good cross-platform support.

LANL ASC projects rely on Intel-12.1+ compiler for released code and Draco must also work with this tool set. Code added to the Draco repository must also compile without warnings using under both GNU and Intel compiler suites. This requirement is checked by the nightly regressions. Draco will also compile under Mac XCode, but this is not a requirement.

7.1.8. Build tools. For Draco to be correctly configured you must use **CMake** and **CTest** [2] version 2.8.8[2] or later. Many machines at LANL provide **CMake**, but not 2.8.8+. Thus, **CMake** is provided as a part of the Draco development environment for supported LANL machines.

Additionally, **svn**, **python** and **numdiff** are also required for successful execution of the unit test suite. The Draco development environment (i.e.: **modules**) provides access to these tools on platforms that do not provide them natively.

7.1.9. Documentation.

7.1.9.1. *Doxygen.* If you want to use the automatic documentation features of Draco [4] then you will need to have both **DOxygen** and LaTeX configured for your system. On the CCS–2 LAN, **DOxygen** has been fully configured and can be used without any additional user setup.

7.1.9.2. *LaTeX.* Many Draco documents found in the `doc` directory are generated using LaTeX. Typically, the developer can run `'make pdf'` from the document source directory to generate a PDF for viewing. If you want to build LaTeX documentation from a Windows PC, we recommend the use of TeXworks' MikTeX. For proper compilation of LaTeX sources into PDF you may need to set the environment variables shown in Table 7.1.

TABLE 7.1. Environment variables for comilation of LaTeX sources

Variable	Location	Description
`BIBINPUTS`	`${draco}/environment/bibfiles`	Location of bibliographic entry files.
`BSTINPUTS`	`${draco}/environment/bibtex`	Location of biliographic style files.
`TEXINPUTS`	`${draco}/environment/latex`	Location of document style files.

7.1.10. Instrumentation and Memory Profiling.

7.1.10.1. *CTest.* When a test is failing, one of the first things to do is to run it by hand outside of the **CTest** framework. The execution command and full output of a test can be retrieved by running **CTest** with the *verbose* option. For example, `ctest -VV -R regex_for_test_name`. Once the default execution command is known, it is easy to run the test manually or modify the command so as to run it by under totalview, valgrind or some other tool.

7.1.10.2. *Valgrind.* If **valgrind** is available on the local system, it may be used to check debug Draco executables (including unit tests) for memory issues. There are no special build system settings required for **valgrind** instrumentation. An example of typical use is shown here:

```
valgrind -q --tool=memcheck --leak-check=full --show-reachable=yes --num-callers=50 \
   /ccs/codes/mpi/openmpi/Linux64/1.3.3/bin/mpiexec -np 1 <my_executable> <arg1> \
   --suppressions="$HOME/draco/regression/valgrind_suppress.txt"
```

Dynamic Analysis (i.e.: **valgrind**) results are provided by the nightly regression scripts and can be viewed from the Draco dashboard at `coder.lanl.gov/cdash/index.php?project=Draco`. Developers can generate and post a valgrind results to the `Experimental` section of the dashboard by issuing the commands `'make ExperimentalMemCheck; make ExperimentalSubmit'` from a Draco build directory.

[1]gfortran-4.7.0 was the first version that provided complete support for `ISO_C_BINDING` used by Draco to interface between Fortran and C++.

[2]**CMake** version 2.8.8 introduced support for exported and imported build system targets used by Draco and client projects

7.1.10.3. *Bullseye Code Coverage.* Code coverage statistics for Draco and client codes are provided by using the Bullseye Code Coverage tool [19] on the regression dashboard at `coder.lanl.gov/cdash/index.php?project=Draco`.

Developers can generate their own coverage data (section to be completed later)

7.1.10.4. *Lines of Code.* Lines-of-code statistics Draco and client codes are provided on the nightly regression scripts and can be viewed from the Draco dashboard at `coder.lanl.gov/cdash/index.php?project=Draco` by clicking on the document icon next to the `Linux64_gcc_Debug_Cov` build name. These metrics can be generated manually by running the **cloc** script found in the `regression` subdirectory.

7.1.10.5. *Purify.* Purify is not currently supported by the build system.

7.1.10.6. *Assure.* Assure is not currently supported by the build system.

7.1.10.7. *Totalview.* Newer versions of Totalview provide memory checking and memory profiling tools. Totalview instrumentation occurs at run time and the Draco build system does not need to provide any special support.

7.1.11. Debuggers.

7.1.11.1. *Totalview.* The Totalview debugger is available on the CCS–2 LAN and on the ICN LANs. It is the recommended debugger. You may need to use the **module** command to load the newest version into your current working environment. Totalview can be used to attach to a running process or a new process can be started and directed to run under the Totalview environment. To start an MPI job under total view, use a command similar to:

```
mpirun -tv -np 4 <path/to/exe> -a <arguments>
```

or

```
totalview mpirun -a -np 4 <path/to/exe> -a <arguments>
```

7.1.11.2. *gdb.* The **gdb** debugger is the only option on some specialized platforms such as the PPC node of Roadrunner.

7.1.11.3. *Debugging with print statements.* This old standby is always supported.

7.1.11.4. *XCode/Eclipse.* Some developers have reported success with debuging from within these Integrated Development Environments. Details of this style of debugging is beyond the scope of this document.

7.1.12. Emacs. Draco does not require anyone to use **Emacs**. However, many of the Draco developers use **Emacs** as their primary editor. Custom templates and functions for **Emacs** that should help programmers write code that conforms to the Draco standard format have been developed and are provided as part of the Draco developer environment.

This additional functionality can be incorporated into your **Emacs** customization by adding a few short *elisp* code snipits to your `~/.emacs` file. Review the `README.elisp` script found in the `draco/environment/elisp` directory for details.

7.1.13. Recommended Reading. A list of recommended texts is shown below in Table 7.2. These texts are also listed in the bibliography.

TABLE 7.2. Recommended reading for Draco developers

Area	Suggested Sections	Reference
New C++ programmers	Chapters 1-15 provide an excellent introduction to programming with C++ and the STL	[33]
	Whole book	[34]
	Whole book	[35]
Experienced C++ programmers	Chapters 1-10	[23]

7.2. Setting up the User's Environment

The Draco build system assumes that the user has configured his or her environment so that particular tools may be used. This document assumes that we are setting up a Linux PC that is on the CCS–2 LAN. In some parts of this document specific information is provided about other platforms, but in general these instructions will only work for a CCS–2 Linux PC. This section discusses these tools and how to configure your environment to make them work properly with the Draco build system [27].

7.2.1. Bash. The recommended shell for Draco development is **bash**. All of the following examples assume that the user is running **bash** as his or her primary shell. If you are not using **bash** the syntax shown in all of this documents examples may be slightly different. On CCS–2 Linux machines, **bash** is the default shell. It is not the default shell for the ICN machines. If you are running on one of these machines and would like to make **bash** start automatically, you can make the request to the consultants by calling 5-4444 or emailing `consult@lanl.gov`.

7.2.2. SVN. **SVN** is used exclusively for version control in Draco and its clients (Capsaicin, Milagro, etc.). In order to work with these code-packages you must have access to the CCS–2 **SVN** repository. To simplify **SVN** usage syntax we recommend that your have the `SVN_EDITOR` environment variable set in your shell If you are running **bash**, these can be set in your `.bash_profile`. If you are running csh or tcsh then you should have the equivalent coding using csh syntax in your `.cshrc`.

The `SVN_EDITOR` environment variable should be set to your preferred editor(e.g. `setenv SVN_EDITOR emacs`). Most Draco developers use **Emacs** and it is the recommended editor for new users because Draco specific functionality has been added to this editor [28]. If you do not set this environment variable, Linux systems will assume `vi`. You can also choose another editor such as **NEdit**.

7.2.3. .bash_profile and .bashrc. Section 7.12 includes a sample listing of a `.bash_profile` and `.bashrc` files that includes all of the recommended Draco developer environment setup. These changes should be implemented in your work environment before you attempt to checkout, configure or build Draco or any of its clients.

7.3. Directory Structure and Obtaining the Source Files

The Draco directory structure is discussed with some detail in Ref. 2. We will not repeat that discussion here. Draco and its clients will check out of the **SVN** repository into a *source* directory (usually in your personal workspace). You will also create one or more *build* directories where configuration and compiling will take place and unit tests will be run. Finally, you will create a directory where header files and compiled binaries and libraries will be installed.

We suggest creating a working directory in scratch space (`mkdir -p /var/tmp/work`) but you can name the directory anything you wish. The examples in this document assume that the work directory is named `/var/tmp/work`. Please substitute your own directory name if you have chosen to use a different path for your work space.

7.3.1. Draco. To obtain a copy of the Draco sources execute the following *checkout* command from within your work directory. The italicized text is representative of the shell's response to your command.

```
[/var/tmp/work/source] % svn co svn+ssh://ccscs8/ccs/codes/radtran/svn/draco/trunk draco
U draco/environment/bashrc/.bashrc
U draco/environment/bashrc/.bashrc_linux64
U draco/environment/bashrc/.bashrc_linux
...
```

This will create the directory `draco` under `/var/tmp/work`. All of the Draco source files will be placed in this directory.

7.3.2. Build Directory. Under each build directory you need to create directories for each architecture and set of configuration options you want to work with. For example, if we are building a parallel version of Draco on a Linux machine we need to create a target directory (`linux_mpid`) and then create a directory for each code under this build location.

```
[/var/tmp/work] % ls -F
source/   linux_mpid/  linux_scalar/
[/var/tmp/work] % ls -F linux_mpid
draco/ capsaicin/
[/var/tmp/work] % ls -F source
draco/ capsaicin/
```

We will configure the sources in these directories. `cmake` will be used to generate the required directory structure in the build tree and fill the tree with appropriate Makefile and header files. This document will show you how to configure, build and install build both parallel and scalar versions of Draco under Linux.

7.3.3. Target Directory. You will also need a *target* directory (aka: install or prefix directory) where object and executable files can be installed. This directory will be created automatically by the Draco make system.

You should use separate target directories for each set of configure options (i.e.: scalar vs MPI) and for each architecture you are working on. These install directories should and a one-to-one correspondence with the build directories created in section §7.3.2.

7.4. Configuring

7.4.1. Draco. Draco and its clients use **cmake** to generate a customized build project for the current architecture. The most common build project type is the *Makefile* system. Other project types include *XCode* and *Eclipse CDT*. We must generate a Draco (or one of its clients) build project to allow compilation of of the package libraries and binaries. Generation of the project is accomplished by running **CMake** from the build directory. This will populate the build tree with the subdirectories, Makefiles and header files that define up the make system. Variables in your environment or provided on the **CMake** command line modify the properties of the generated build project.

The following example will generate a Makefile-based build project that uses MPI and generates debug code:

```
[/var/tmp/work/linux_mpid/draco] % cmake
        -DCMAKE_INSTALL_PREFIX=/var/tmp/work/linux_mpid /var/tmp/work/draco
```

In this example, we have assumed that several environment variables have been set correctly in the local environment. In order for this build to use **OpenMPI** the command **mpirun** must be found in the `PATH` and the environment variables `GSL_INC_DIR` and `GSL_LIB_DIR` must point to a valid install of the GNU Scientific Library. In our case, these are set automatically through the Draco development environment's use of vendor modules loaded automatically by the `.bashrc` provided in Draco's `environment` directory.

Different builds will need to include different vendor libraries. See Ref. [27] for details about what libraries may need to be included. Additionally, some packages may have their own documentation concerning required configure options. For example, to include `plot2D` in the build project, the build system must find **xmgrace** in the local environment or variables must be set to point to the correct locations.

```
-DGRACE_INC_DIR=/ccs/opt/x86_64/grace/grace/include
-DGRACE_LIB_DIR=/ccs/opt/x86_64/grace/grace/lib
```

Alternatively, you can load the Draco module for `XMGrace` that will set these values for you automatically.

To configure a scalar version of Draco we would execute a similar command.

```
[/var/tmp/work/linux_mpid/draco] % cmake
        -DCMAKE_INSTALL_PREFIX=/var/tmp/work/linux_mpid
        -DDRACO_C4=SCALAR
        /var/tmp/work/source/draco
```

Once the configuration is complete the files required for building the Draco libraries will exist in the binary directory. The Makefiles (or project files) will know where the required system tools live, where the vendor libraries are installed and where the final Draco library files should be installed.

It should be noted that the build configuration can be modified after the initial **CMake** setup by issuing the command `'ccmake .'` from the build directory. **CMake** also provides a GUI, aptly named **cmake-gui**, that can also be used to modify the options for an existing build directory.

7.5. Compiling the Sources

7.5.1. Draco. We are now ready to compile Draco. From the Draco target directory simply run **gmake** to create the Draco libraries. You may want to run **gmake** in parallel to speed up the compilation process. Using the optional gmake argument "`-j N`" will tell **gmake** to use **N** processors for the compilation process. We recommend that **N** be set to value that is about 50% larger than the number of available cores (i.e.: 24 on a 16 core machine).

```
[/var/tmp/work/linux_mpid/draco] % gmake -j 24
```

Once the build process has finished several new directories will exist in the binary directory. If the `install` target is requested (e.g.: `make -j24 install`), the Draco library files will have been installed in the *prefix* directory (e.g.: `/var/tmp/work/linux_mpid/lib`). The C++ header files from Draco will be stored under `/var/tmp/work/linux_mpid/include`.

Before the build started each Draco package stored its configuration information and `Makefile` in its own directory under `/var/tmp/work/linux_mpid/draco/src`. After the build completes these directories will also contain dependency and object files for each package.

All of the test files will have been compiled after a standard build. See section §7.6 for details about unit tests. To avoid building the unit tests set `-DBUILD_TESTING=OFF` during the **CMake** configuration step.

7.6. Regression Tests

Draco has a complete set of regression tests included with source. If you wish to test any particular component you can run the command `ctest` from that component's build directory. For example, if you wanted to run the regression tests on the Draco *Quadrature* package you would execute the following command.

```
[/var/tmp/work/linux_mpid/draco/src/quadrature] % ctest -j 16
```

This will run the test routines and give a short report on the success or failure of the test suite. A typical test report will look something like the following output.

```
Test project /var/tmp/work/linux_mpid/draco/src/quadrature/test
    Start 1: quadrature_tstgaulag
    Start 2: quadrature_tQuadCreator
    Start 3: quadrature_tstgauleg
    Start 4: quadrature_tAxialQuadrature
    Start 5: quadrature_tstOrdinate
    Start 6: quadrature_tQuadrature
    Start 7: quadrature_tQuadServices
    Start 8: quadrature_tAngle_Operator
1/8 Test #3: quadrature_tstgauleg ............   Passed    0.06 sec
2/8 Test #2: quadrature_tQuadCreator ..........   Passed    0.07 sec
3/8 Test #7: quadrature_tQuadServices .........   Passed    0.09 sec
4/8 Test #6: quadrature_tQuadrature ...........   Passed    0.09 sec
5/8 Test #4: quadrature_tAxialQuadrature ......   Passed    0.09 sec
6/8 Test #8: quadrature_tAngle_Operator .......   Passed    0.09 sec
7/8 Test #5: quadrature_tstOrdinate ...........   Passed    0.09 sec
8/8 Test #1: quadrature_tstgaulag ............   Passed    0.10 sec

100% tests passed, 0 tests failed out of 8

Total Test time (real) =   0.18 sec
```

If you want to test all of the Draco packages simply run `ctest` from the Draco binary directory (`/var/tmp/work/linux_mpid/draco`). Similarly you can run the same command from the Capsaicin binary directory (or any other client's binary directory) to run all of the Capsaicin (client's) regression tests.

7.7. Generating the Documentation

7.7.1. Automatic Documentation. In order to generate the HTML *automatic documentation* for Draco you must have **DOxygen** configured for the system you are working on (see section §7.1.9.1 [4]). Once you are setup to use **DOxygen** and the configuration step (§7.4) has been completed you can generate the automatic documentation by executing the following command from the Draco build directory.

TABLE 7.3. Listing of Draco LaTeX documents available from the Draco **SVN** repository. Some of documents describe the build system in detail while others offer assistance in configuring the developer's environment.

Document	Ref.
The Draco Build System	[27]
Emacs Development Environment for Draco	[28]
Doxygen C++ Automatic Documentation Utility	[4]
Draco Release Policy and Procedures	[36]
Generic Programming in the Solon Interface	[24]
The Draco web site	[37]

```
[...rk/linux_mpid/draco] % make autodoc
```

This will create two new directories under `/var/tmp/work/`*`targetdir`*`/autodoc` named `html`. Hypertext versions of all of the automatically created documents are created in the `html` directory. The head file is named `index.html`.

7.7.2. Package Documentation. Some Draco packages have LaTeX documentation in the source tree. The build model for these directories has not been standardized but you will likely need to run `make pdf` to generate these documents.

7.7.3. General Draco **Related Documentation.** Additional documentation (other than the *automatic code documentation*) is available for Draco. This documentation includes research notes, technical memorandum and other potentially useful documentation. An abbreviated list of these documents in shown in Table 7.3. In addition to this list of references you may want to peruse the Draco team forge web site (`tf.lanl.gov`).

These documents are automatically checked out when you check out Draco from the **SVN** repository and are stored in the Draco source directory, `/var/tmp/work/draco/doc`.

To generate the actual postscript documentation may take some work. Some of the document directories contain Makefiles and the documents may be generated by using **gmake**. In other directories you may have to run LaTeX, **bibtex** and **dvi2ps** to obtain postscript versions of the documents.

7.8. SVN Update

Most of the Draco packages are currently under active development and they change on a regular basis. If you are developing or modifying Draco packages you should be receiving **SVN** update notifications so that you will know when other packages have been updated. (If you would like to be on the Draco mailing list send an email message to *listmanager@listman.lanl.gov* with *subscribe draco* in the body of the message.) If your package depends on another Draco component that has been modified you will need to update your version of Draco and recompile (and possibly reconfigure). To update each package to the latest revision requires that you execute the following commands. (If whole packages have been added then you may need to reconfigure as well.)

```
[/var/tmp/work/draco] % svn status -u
[/var/tmp/work/draco] % svn update
[/var/tmp/work/linux_mpid/draco] % make rebuild_cache
[/var/tmp/work/linux_mpid/draco] % make -j 20 install
[/var/tmp/work/linux_mpid/draco] % ctest -j 16
```

The `-u` flag requests **SVN** to check against the `HEAD` revision instead of looking only at local changes. A similar procedure may need to be completed for Draco client files (i.e.: execute the `svn update` command from within a client's package directory).

It may be useful to execute the command `'svn status -u'` from within a package's source directory to see what files **SVN** thinks have been updated.

7.9. Adding a New Package to Draco

Adding a package to the Draco build system is straight forward. However, the following guidelines should be followed when a package is added.

First you should verify that all of the other Draco components are up to date. You can use the "`svn update -u`" command as described in the preceding section to do this.

Now you can create a package directory beneath `/var/tmp/work/draco/src` and put your new files in this location. At a minimum, you must provide `CMakeLists.txt` and at least one source file in the component directory. You should copy the `CMakeLists.txt` from a nearby component directory as a template. Please follow the formatting style use by other Draco source code.

Once the component directory has been established, you must tell the `src/CMakeLists.txt` file to include your new component in the build by adding an appropriate `add_dir_if_exists( pgk )` instruction. Once this is done, you must run '**make rebuild_cache**' from the build directory to reconfigure the build project and then '**make**' to build your files.

When you are satisfied that your code is working as intended you need to add it to the **SVN** repository. From the Draco `src` directory execute the command "`svn add` <*package*>". This command will add your directory and all files recursively.

7.9.1. Package Documentation.

7.9.1.1. *LaTeX.* If you want to include LaTeX documentation for your new package create a new subdirectory called `doc` in the source tree under your package subdirectory.

7.9.1.2. *Doxygen.* If you want to include Doxygen documentation for your new package create a new subdirectory called `autodoc` in the source tree under your package subdirectory. Your documentation files in this subdirectory must use the file extension `.dcc`.

7.10. Troubleshooting

According to Murphy's Law things never go as smoothly as they should. The following sections outline some of the problems that Draco developers and users have encountered and their respective resolution.

7.10.1. C++ Error Messages. If you encounter a C++ error message that you don't understand, you should contact other Draco developers and search the web for help. The following web sites may be of some use:

- `cplusplus.com` - Documentation, tutorials, discussion boards for C++.
- C++ FAQ (`www.parashift.com/c++-faq-lite`) - Answers to common C++ questions.
- OpenOffice Wiki (`http://wiki.services.openoffice.org/wiki/Writing_warning-free_code`) - Writing warning-free code.

7.10.2. DYLD_LIBRARY_PATH. On Mac OSX, it may be necessary to modify your `DYLD_LIBRARY_PATH` environment variable to point to the location of some vendor libraries like CUDA or Thrust.

7.11. Sample `.bash_profile` File

This is a sample `.bash_profile` file. This file leaves room for user-defined options that you may want to modify for your own specific needs. Note that it sources `.bashrc` and that file is discussed in the next section.

```
##############################################################################
## .bash_profile
##############################################################################

# If this is an interactive shell then the environment variable $-
# should contain an "i":
case ${-} in
*i*)
   export INTERACTIVE=true
   ;;
*) # Not an interactive shell
   export INTERACTIVE=false
```

```
   ;;
esac

##---------------------------------------------------------------------------##
## Establish environment for login shell
##---------------------------------------------------------------------------##
if [ -f ~/.bashrc ]; then
   source ~/.bashrc
fi

###############################################################################
## end of .bash_profile
###############################################################################
```

7.12. Sample `.bashrc` File

This is a sample `.bashrc` file that sets all of the environment variables required by Draco. Additionally, this file leaves room for user-defined options that you may want to modify for your own specific needs.

```
## .bashrc - my bash configuration file upon bash login

#------------------------------------------------------------------------------#
# CCS-2 standard setup
#------------------------------------------------------------------------------#

export DRACO_SRC_DIR=~/draco

# If this is an interactive shell then the environment variable $-
# should contain an "i":
case ${-} in
*i*)  export INTERACTIVE=true  ;;
*) # Not an interactive shell
   export INTERACTIVE=false  ;;
esac

#------------------------------------------------------------------------------#
# Load default Draco Environment
#------------------------------------------------------------------------------#
if test -f ${DRACO_SRC_DIR}/environment/bashrc/.bashrc; then
    source ${DRACO_SRC_DIR}/environment/bashrc/.bashrc
fi

#------------------------------------------------------------------------------#
# User Customizations
#------------------------------------------------------------------------------#
if test "$INTERACTIVE" = true; then

    export SVN_EDITOR=emacs
    export LPDEST=gumibears_d

fi

##---------------------------------------------------------------------------##
## end of .bashrc
##---------------------------------------------------------------------------##
```

APPENDIX A

Vendor Libraries

As described in § 3.1, Draco uses and requires several external vendor libraries. The packages that require these vendors are listed in Table 3.2. This appendix gives additional details on the vendor libraries. Specifically included are common headers used by Draco and link-line dependencies.

A.1. MPI

MPI is the message passing interface used for most parallel codes including Draco. While Draco can be built without **MPI**, we find that most production capabilities require **MPI** and the scalar build mode becomes useful only for debugging. At the current time, Draco can use either the OpenMPI or MPICH2 flavor of **MPI**. The Draco build system should automatically detect the available **MPI** system and link against it (enabling `DRACO_C4=MPI`).

The Draco development environment does not normally provide the **MPI** vendor installation since it must be compiled to take advantage of specialized hardware on each machine (interconnects, etc). The one exception is on CCS development machines were a version is provided for testing purposes only.

MPI is discovered by the build system via a call to `vendor_libraries.cmake` which, in turn, calls the **CMake** provided `FindMPI.cmake` (use `cmake --help-module FindMPI` for detailed information). If the build system cannot find the flavor of **MPI** that you are targeting, you can try manually set the **CMake** variables `MPIEXEC` and `MPIEXEC_NUMPROC_FLAG`. This must be done on platforms like Cielo.

A.1.1. Additional Notes.

- On LANL HPC machines, you must link against a **MPI** built with compatible compilers. That is you may need to load a gcc version or an Intel version of openmpi using the module command.
- Some vendors (e.g.: **Trilinos**, **ScaLAPACK**) also use **MPI** and you should be careful to configure your code to use the same version of **MPI** as the vendor libraries.
- To link your component against MPI, modify the `target_link_libraries` command to include `MPI_LIBRARIES`.
- The MPI header directory is provided by `MPI_INCLUDE_PATH`.

A.2. LAPACK

The LAPACK libraries provide efficient implementation of linear algebra operations on shared-memory vector and parallel processors. The Draco development environment provides it's own installation of **LAPACK** that can be loaded into the developer environment through the use of the Draco modules feature.

The Draco build system uses the **CMake** provided `FindLAPACK.cmake` module (use `cmake --help-module FindLAPACK` for more details) to discover LAPACK. If the build system is finding the wrong installation of LAPACK, you can manually set `BLAS_blas_LIBRARY` and `LAPACK_lapack_LIBRARY` to the full path of these 2 libraries before running **CMake** for the first time.

Make your component depend on LAPACK by adding `LAPACK_LIBRARIES` to your `target_link_libraries` command.

A.3. GSL, GNU Scientific Library

The GSL libraries provide efficient implementation of common numerical operations like root-finding, splines, and statistics. The Draco development environment provides it's own installation of **GSL** that can be loaded into the developer environment through the use of the Draco modules feature.

The Draco build system uses its homegrown `FindGSL.cmake` module to discover GSL. If the build system is finding the wrong installation of GSL, you can manually set `GSL_INC_DIR` and `GSL_LIB_DIR` to the full paths of these two items before running **CMake** for the first time.

Make your component depend on GSL by adding `GSL_LIBRARIES` to your `target_link_libraries` command. You may also need to add `GSL_INCLUDE_DIRS` to your list of `include_directories`.

A.4. CUDA

CUDA^{TM} is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU). Support for CUDA is still experimental and you will need to review the code found in the DEVICE component, the discovery routines in `vendor_libraries.cmake` and the **CMake** module `FindCUDA` for details on using this vendor.

A.5. DaCS

DaCS is the communication library used to run code on the IBM Cell architecture (Roadrunner). Support for DaCS is mature, but no further development is planned since this architecture has been discontinued. You can glean some of the implementation features by review Draco's DEVICE component.

A.6. XMGRACE

Grace is a WYSIWYG 2D plotting tool for the X Window System. The Draco development environment provides it's own installation of **Grace** that can be loaded into the developer environment through the use of the Draco modules feature.

The Draco build system uses its homegrown `FindGrace.cmake` module to discover Grace. If the build system is finding the wrong installation of Grace, you can manually set `GRACE_INC_DIR` and `GRACE_LIB_DIR` to the full paths of these two items before running **CMake** for the first time.

Make your component depend on Grace by adding `Grace_LIBRARIES` to your `target_link_libraries` command. You may also need to add `Grace_INCLUDE_DIRS` to your list of `include_directories`.

A.7. EOSPAC

EOSPAC is a LANL provided library that provides access to Sesame data. Draco provides a module that can be loaded to set the default developer environment for building with EOSPAC. CDI_EOSPAC requires this vendor.

The Draco build system uses its homegrown `FindEOSPAC.cmake` module to discover EOSPAC. If the build system is finding the wrong installation of EOSPAC, you can manually set `EOSPAC_INC_DIR` and `EOSPAC_LIB_DIR` to the full paths of these two items before running **CMake** for the first time.

Make your component depend on EOSPAC by adding `EOSPAC_LIBRARY` to your `target_link_libraries` command. You may also need to add `EOSPAC_INCLUDE_DIRS` to your list of `include_directories`.

A.8. Random123

Random123 is a header-only random number generation implementation used by the Jayenne project. Draco provides a module that can be loaded to set the default developer environment for building with Random123. RNG requires this vendor.

The Draco build system uses its homegrown `FindRANDOM123.cmake` module to discover Random123. If the build system is finding the wrong installation of Random123, you can manually set `RANDOM123_INC_DIR` to the full path before running **CMake** for the first time.

Make your component depend on Random123 by adding `RANDOM123_INCLUDE_DIRS` to your list of `include_directories`.

Bibliography

[1] R. Stallman, *GNU Coding Standards*. Free Software Foundation, Sept. 1996.

[2] "CMake." http://www.cmake.org, 2011.

[3] C. M. P. Ben Collins-Sussman, Brian W. Fitzpatrick, "Version Control with Subversion." http://svnbook.red-bean.com/en/1.7/index.html, 2012.

[4] J. McGhee and T. Evans, "Doxygen C++ automatic documentation utility," Tech. Memo XTM:JMM-99-57(U), Los Alamos National Lab., 1999.

[5] T. Evans, "The Draco system for XTM transport code development," Research Note XTM-RN(U)-98–046, Los Alamos National Lab., 1998. LA-UR-98–5562.

[6] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, third ed., 2005.

[7] J. Lakos, *Large-Scale C++ Software Design*. Addison Wesley, 1996.

[8] D. MacKenzie, *Autoconf*. Free Software Foundation, 2.12 ed., Nov. 1996.

[9] R. Stallman and R. McGrath, *GNU Make*. Free Software Foundation, Boston, MA, May 1997. v3.77.

[10] K. Jameson, *Multi-Platform Code Management*. O'Reilly Associates, Inc., 1994.

[11] True North pks, Inc., Portland, OR, *Mastering Projects Workshop*, revision 4 ed., Jan. 1998.

[12] M. Gray, R. Roberts, and T. Evans, "Shadow object interface between F 95 and C++," *Computers in Physics*, 1999. Accepted for publication.

[13] B. Meyer, *Object-Oriented Software Construction*. ISE Inc., second ed., 1997.

[14] M. Austern, *Generic Programming and the STL*. Reading, MA: Addison-Wesley, 1999.

[15] R. Seindal, *GNU m4*. Free Software Foundation, 1.4 ed., Nov. 1994.

[16] "CollabNet TeamForge." http://www.collab.net/products/ctf/, 2012.

[17] "TeamForge at Los Alamos National Laboratory." https://tf.lanl.gov/, 2012.

[18] "Valgrind." http://valgrind.org, 2012.

[19] "BullseyeCoverage." http://www.bullseye.com, 2011.

[20] "CLOC - Count Lines Of Code." http://http://cloc.sourceforge.net/, 2012.

[21] ISO/IEC, International Standard, "Programming languages-C++," Tech. Rep. 14882, American National Standard Institute, New York, Sept. 1998.

[22] "Cdash dashboard for ccs-2." http://coder.lanl.gov/cdash, 2011.

[23] D. Vandevoorde and N. M. Josuttis, *C++ Templates: The Ccomplete Guide*. Addison-Wesley, 2002.

[24] R. Roberts, G. Furnish, M. Gray, and S. Pautz, "Generic programming in the SOLON interface," in *Proceedings of the Nuclear Explosives Code Development Conference* (G. Olson, ed.), U.S. Dept. of Energy, Oct. 1998. available through Draco under draco/doc/GenericProgInSolon/.

[25] B. Gough, *GNU Scientific Library Reference Manual*. Network Theory Ltd., 2009.

[26] "OpenMPI." In Development, http://www.openmpi.org/, 2011.

[27] T. E. K.G. Thompson and R. Roberts, "The Draco Build System," Technical Memorandum CCS-2:12-43(U), Los Alamos National Laboratory, 2012.

[28] T. Evans, "XEmacs development environment for Draco," Tech. Memo XTM:99–09(U), Los Alamos National Lab., 1998.

[29] T. Evans, "Using Purify in Draco with MPI," Tech. Memo XTM:98-099(U), Los Alamos National Lab., 1998.

[30] M. Buksas, "Using the draco autodoc system," Research Note CCS-4:04–35(U), April 2004.

[31] K. Thompson, "Draco release policy and procedures," Tech. Memo CCS–2:12–04(U), Los Alamos National Security, LLC, 2012.

[32] K. G. Thompson, "Getting Started with Draco - Help for new users and developers," Technical Memorandum CCS-2:12-16(U), Los Alamos National Laboratory, 2012.

[33] A. Koeng and B. Moo, *Accelerated C++ Practical Programming by Example*. Addison-Wesley, 2000.

[34] S. Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 2005.

[35] S. Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley, 1996.

[36] T. Evans, "Draco release policy and procedures," Tech. Memo XTM:99–36(U), Los Alamos National Lab., 1999.

[37] J. McGhee, "Draco computation physics system home page." In Development, http://www.ccs.lanl.gov/rad/htdocs/XTM/radtran/draco_www/draco/, 2001.

Index