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–NN(U), LA–12–NNNNN.

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 C++ on all current ASC platforms;
3. support for multiple build project types (**Makefiles**, **XCode**, **Eclipse**, etc.)
4. on-demand and automated unit and regression testing;
5. the ability to support multiple code projects;
6. support for external vendors;
7. support for explicit template instantiation;
8. support for multiple languages, but the recommended langues are C and C++.
9. support for C++ on all current ASC platforms;
10. C and C++ coding must conform strictly to issued ISO standards.
11. extensibility;
12. low-cost on developers to add new packages, code, and tests;
13. adoption of selected sections of the GNU coding standard [1].
14. support for on-demand and automated regression testing with web-based dashboard presentation.
15. 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.

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 tells how to get up and moving without working through the manual. 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 Draco build system. Specifically, we will show both 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, but the recommended languages are C and C++.
- support for C++ on all current ASC platforms;
- C and C++ coding must conform strictly to issued ISO standards.
    1. C++03, also known as the ISO/IEC 14882:1998 standard amended by the 2003 technical corrigendum, ISO/IEC 14882:2003.
    2. C99, also known as the ANSI C ISO/IEC 9899:1999 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;
- adoption of selected sections of the GNU coding standard [1] .

This last requirement has evolved a 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

TABLE 1.1. Typefaces used throughout the text.

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

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 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
Kelly Thompson	kgt@lanl.gov	CCS–2
Jae Chang	jhchang@lanl.gov	CCS–2

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.

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).

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
Jeff Densmore	jdd@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 and 4	
Component Developers	Chap. 2, 3 and 5	Chap. 4 and 6
System Developers	Chap. 2, 5 and 6	Chap. 3

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

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	http://tf.lanl.gov
Bug Tracker	http://tf.lanl.gov
Regression files	`ccscs8://home/regress/cmake_draco` and `hpc://usr/projects/jayenne/regress`
Regression Dashboard	http://coder.lanl.gov/cdash

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.

1.5. Adminstration

1.5.1. Important locations. Draco make 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 notificiations. 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. Dracodevelopers are encouraged to visit the dashboard regularrly 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 § 3.6 for examples on configuring and building Draco. We also recommend a review of Ref. [12] for developers who are new to 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 § 2.2.3. 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 C++ library; however, other language support is not precluded in Draco. In particular, Draco has plans to support ISO_C_BINDING interfacing between C++ and F90. A more general type of automatic type-interfacing between C++ and F90 [13] 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 [14] and generic programming [15] 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 and made use of **autoconf** [8], **gmake** [9], and **gm4** [16] 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** [17,18]. In addition, an archived email list is available to submit design plans and discussion between team members. Also, **Valgrind** [19], **BullseyeCoverage** [20] and **CLOC** [21] 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 1998 ANSI Standard C++ [22]. Draco does not currently support the 2011 ANSI C++ Standard [23] because LANL standard compiler's do not support the new standard yet. As soon

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

Tool	Required	Recommended	Optional
Configuration	**CMake**-2.8.6+	**CTest**	**CDash**
C++ compiler	Any standards compliant C++ compiler	g++ 4.5	Intel 10-12, PGI 11
F90 compiler	optional	gfortran 4.5+	Intel 10-12, PGI-11
Build tool	any of ...	**gmake**	Eclipse CDT, XCode, Visual Studio
Scripting language	**CMake**	PYTHON	PERL
Version control	**SVN**		**Git**
Dynamic Analysis	optional	**Valgrind**	
Code Coverage	optional	**BullseyeCoverage**	
Lines-of-Code	optional	**CLOC**	
Bug Tracking	optional	**TeamForge**	ChangeLog, text files, email
Documentation	optional	LaTeX typesetting tool, **DOxygen**	

as the required compilers support the new standard, Draco will allow code to be written that conforms to the 2011 standard. Currently, any C++ compiler that conforms to the 1998 standard will compile Draco. Portland Group, Intel and GNU C++ are regularly used for building on ASC hardware. Currently, there is no F90 code in Draco; although, we expect that to change in the future. (The build system does look for a F90 compiler during the configuration step so that a compatible version of LAPACK can be located and tested with the selected linker.) Additional languages required by Draco are PYTHON and PERL. Draco expects these scripting languages to be located in a directory that is included int he 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 mainly a development tool and is not required to configure, build, or use Draco. Also note that the GNU set of compilers is 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.

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 and C++;
3. support for C++ on all current ASC platforms;
4. C and C++ coding must conform strictly to issued ISO standards [22] :
   (a) C++03, also known as the ISO/IEC 14882:1998 standard amended by the 2003 technical corrigendum, ISO/IEC 14882:2003;
   (b) C99, also known as the ANSI C ISO/IEC 9899:1999 standard;
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, 24] presentation of regression results;
8. support for **Valgrind** [19] dynamic analysis integrated into **CDash** presentation;
9. the ability to support multiple code projects;

10. extensible support for external vendors;
11. support for explicit template instantiation;
12. extensibility;
13. low-cost on developers to add new packages, code, and tests;
14. adoption of selected sections of the GNU coding standard [1] .

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 presently composed of C++ code, we expect multiple languages (in particular Fortran) to be supported for interfacing and numerical optimization. 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.

C++ source code in Draco must conform to the the ISO/IEC 14882:1998 standard amended by the 2003 technical corrigendum, ISO/IEC 14882:2003. 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 is 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 [24] 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 other 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) [25]. 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 remain 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.

The final requirement is that the Draco build system should continue to adhere to the GNU Coding Standard [1] as much as is reasonable. Obviously, 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). A summary of the pertinent parts of the GNU standard is given in § 2.2.3

2.2.3. The GNU Build Model. A full description of the GNU Coding Standards is beyond the scope of this text. Interested readers are referred to Ref. [1] for more information. However, a brief summary of the pertinent aspects of the coding standard is useful here. As we have previously mentioned, the Draco build system corresponds to the guidelines set forth in the GNU standard. The relevant parts of the GNU standard that affect Draco are the sections pertaining to documentation and program release. We shall look at these in turn.

The GNU standard specifies the following requirements on release documentation:

1. provide a manual describing the system using **texinfo**, this can refer to other documentation;
2. a `ChangeLog` file should contain a log of changes made to the product between different releases;
3. man pages are optional.

Draco does not presently have man pages, and no attempts are underway to make them. The other requirements are met with the following exceptions: Draco uses LaTeX and **DOxygen** for its documents and contains more than one manual due to its size and multiple uses.

2.2.4. 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

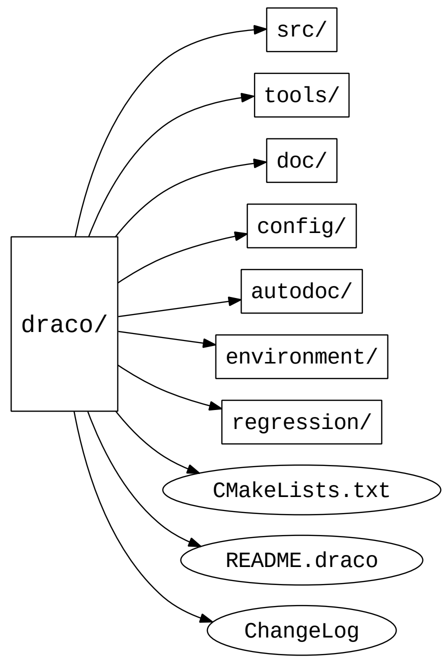


FIGURE 2.1. The Draco source tree. Directories are in boxes and files are in ellipses. The subdirectories are shown in Fig. 2.2.

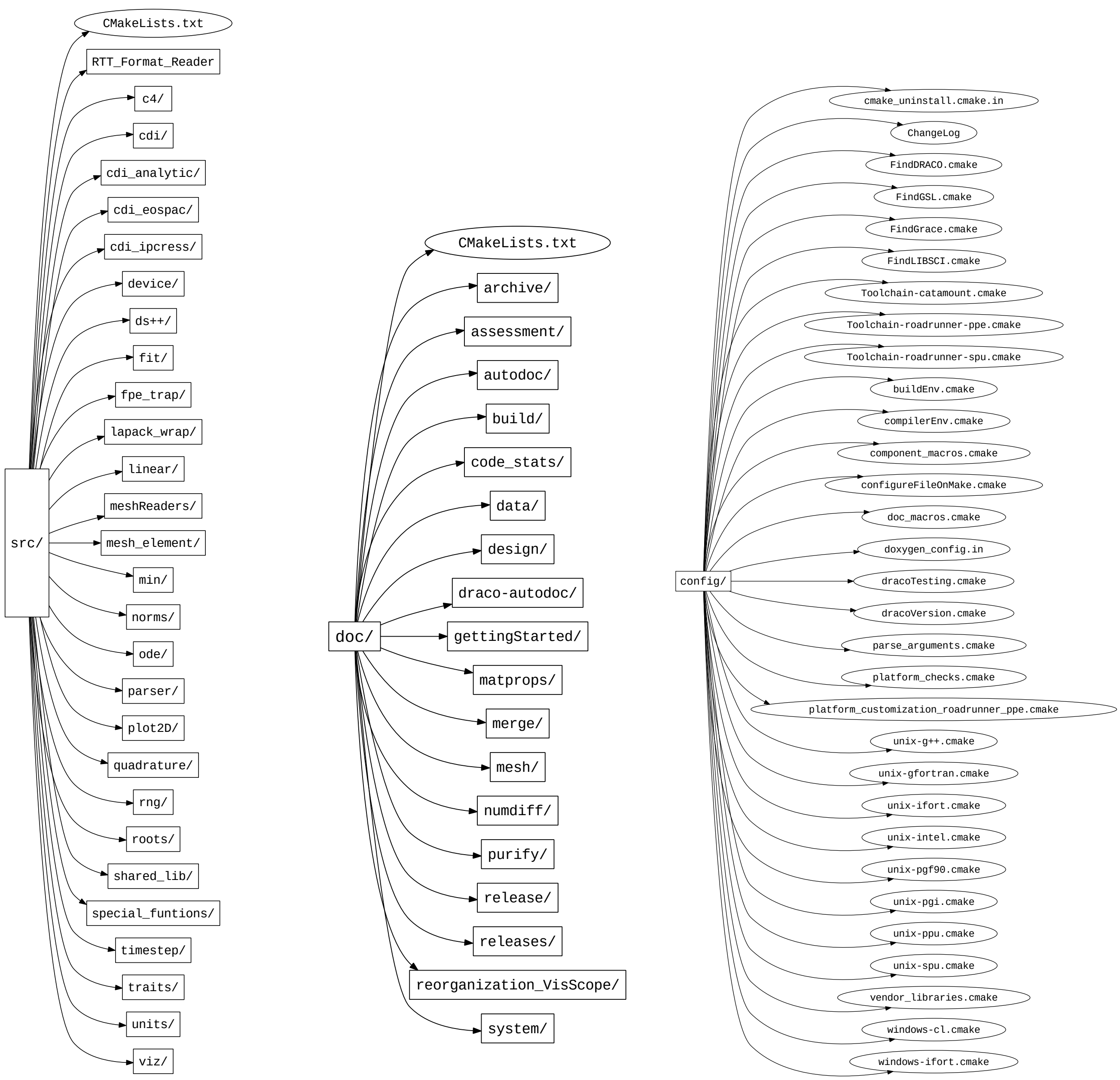
   - test - run the unit tests
   - uninstall - remove files copied as a result of the install target
5. The DBS standardizes additional Makefile targets:
   - Lib_<component>
   - Lib_<component>_test
   - Ut_<component>_<testname>_exe

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 exists several files generated or needed by **CMake** and **CTest**. The files `CTestConfig.cmake` and `CTestCustom.cmake` are read by **CTest** and specify the 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



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

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

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.

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.

---

[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.

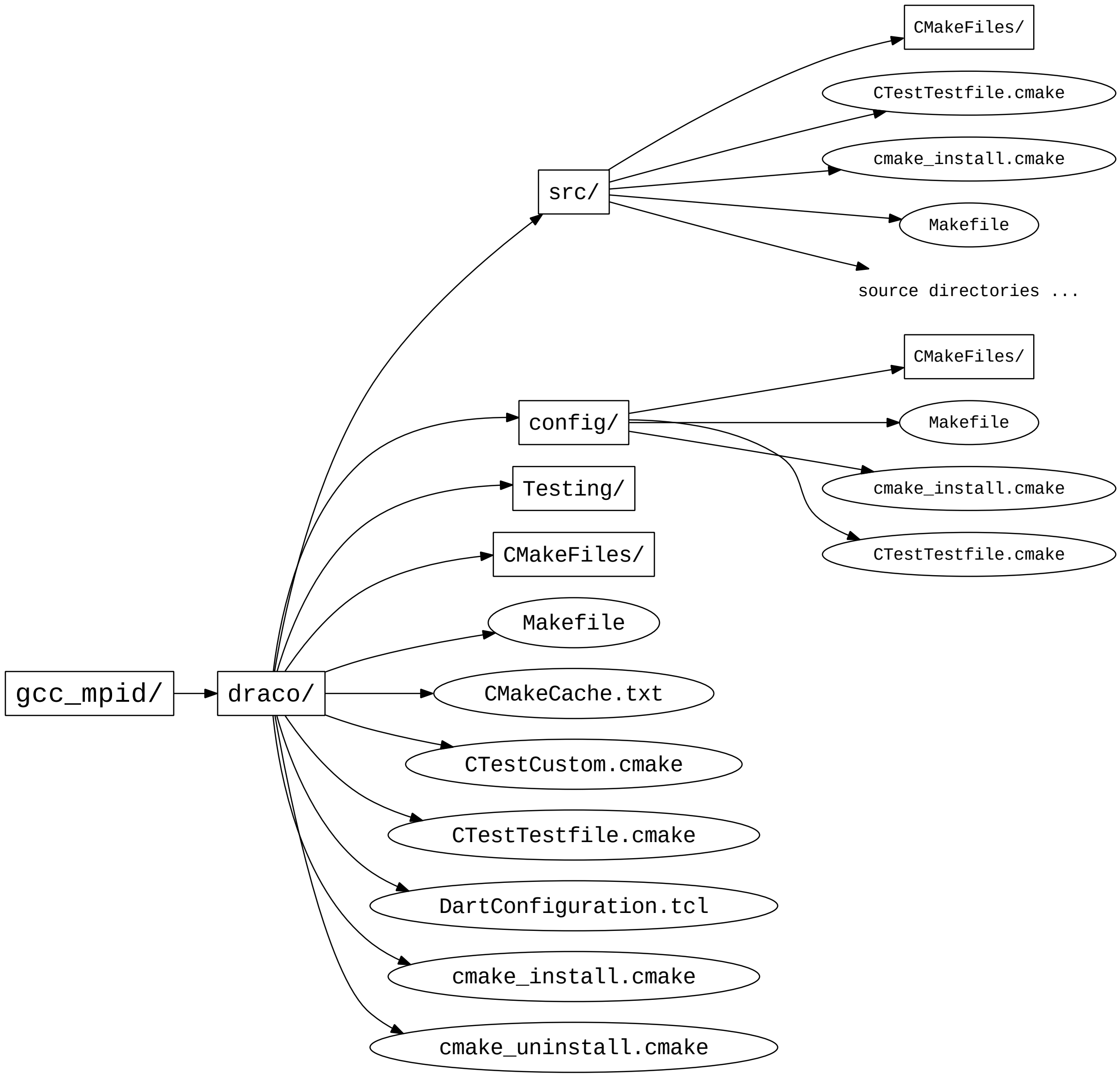


FIGURE 2.3. The Draco build tree after running **CMake** for a Makefile-based project. The `source directories` are shown in Fig. 2.2a. The `doc directories` are shown in Fig. 2.2b. The binary tree for Eclipse CDT4 - Unix Makefile based projects will match this layout but will also include two additional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

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. 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.

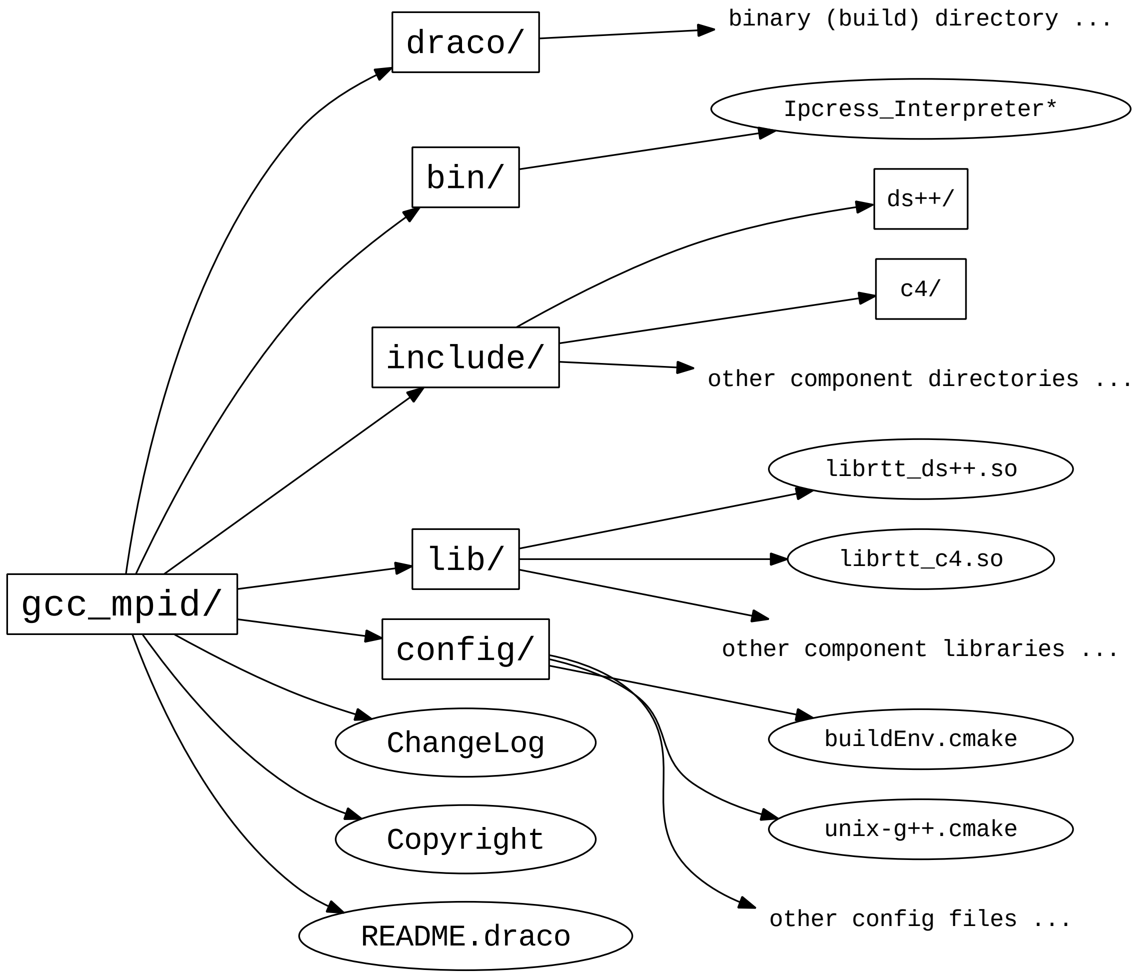


FIGURE 2.4. The Draco build tree after running `make install` for a Makefile-based project. The *install* target creates the `lib/`, `bin/`, `include/` and `config` directories under the *target* directory. The target directory is specified at *configure* time by setting the **CMake** variable `CMAKE_INSTALL_PREFIX`.

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 on the link line.

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`. This component-level diagram includes dependencies for unit tests. Dotted lines represent dependencies that are only required for building the unit tests. 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 using Draco on 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. 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.

Some components in Draco require external vendor support. Additionally, some configuration options in Draco require external vendor support. Table 3.2 lists all of the present components in Draco and the vendors that are required to build those components. Also, because Draco is based on the concepts of levelized design as stated above, 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.

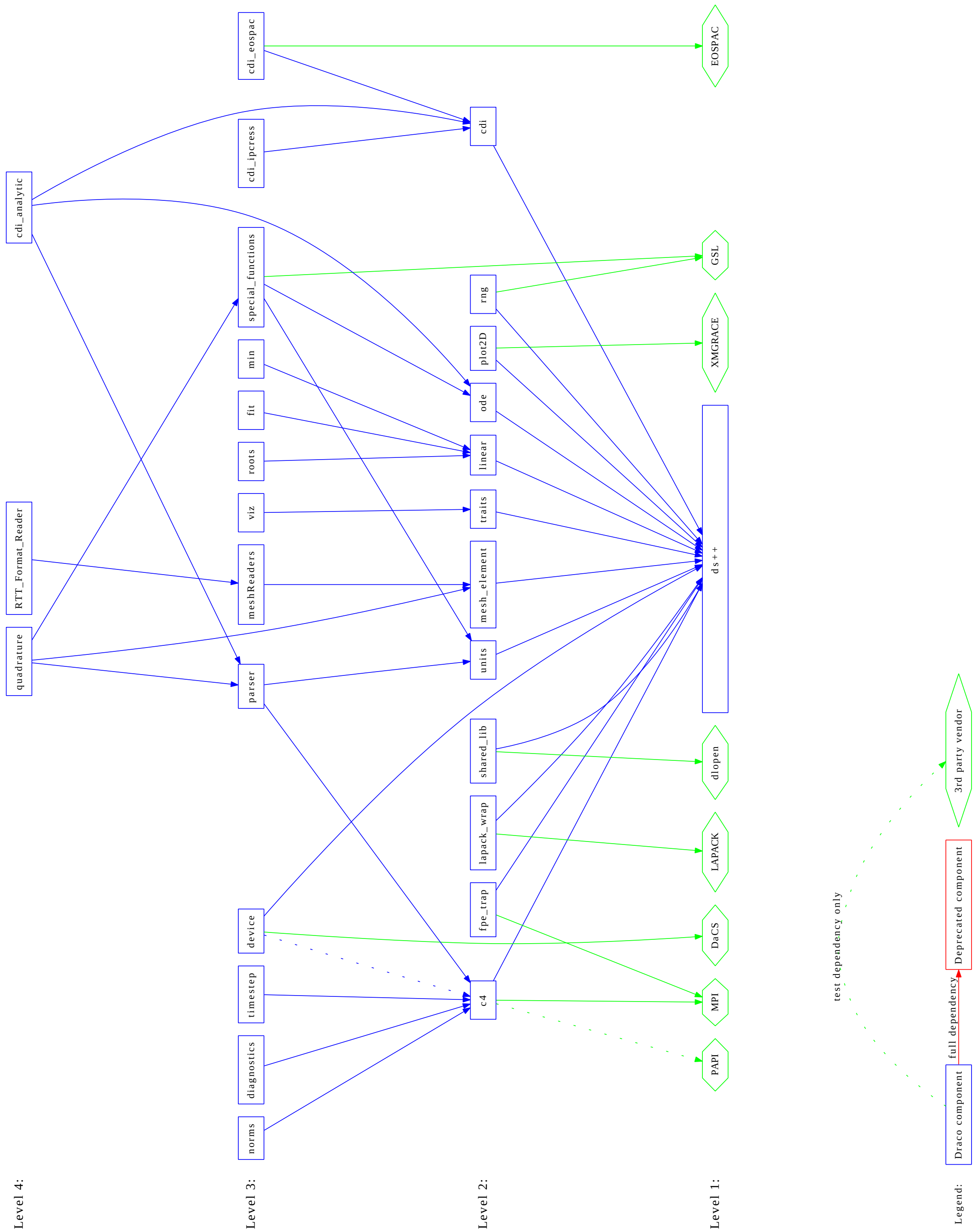


FIGURE 3.1. Component-level diagram for Draco.

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	-	-	-
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++	-	-
CDI_IPCRESS	3	CDI	DS++	-
CDI_EOSPAC	3	CDI	DS++	-
DEVICE	3	DS++	-	C4
DIAGNOSTICS	3	C4	DS++	-
FIT	3	LINEAR	DS++	-
MESHREADERS	3	MESH_ELEMENT	DS++	-
MIN	3	LINEAR	DS++	-
NORMS	3	C4	DS++	-
PARSER	3	C4, UNITS	DS++	-
ROOTS	3	LINEAR	DS++	-
SPECIAL_FUNCTIONS	3	ODE, UNITS	DS++	-
TIMESTEP	3	C4	DS++	-
VIZ	3	TRAITS	DS++	-
CDI_ANALYTIC	4	PARSER, ODE, CDI	C4, DS++, UNITS	-
QUADRATURE	4	PARSER, MESH_ELEMENT, SPECIAL_FUNCTIONS	C4, DS++, UNITS, ODE	-

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 Table 3.1 we see that QUADRATURE is dependent on the Draco packages SPECIAL_FUNCTIONS, PARSER, MESH_ELEMENT, ODE, DS++, UNITS, and C4. From Table 3.2 we see that ODE 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_parser -lrtt_special_functions -lrtt_c4 -lrtt_units \
-lrtt_mesh_element -lrtt_ode -lrtt_ds++ \
-lgsl -lgslcblas \
-lmpi_cxx -lmpi -lopen-rte -lopen-pal
```

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

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	yes
SHARED_LIB	2	dlopen	yes
TRAITS	2	-	-
UNITS	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
QUADRATURE	4	GSL	yes
		MPI	no
RTT_FORMAT_READER	4	-	-

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 `$LD_LIBRARY_PATH`. Thus, even though QUADRATURE does not explicitly depend on MPI or GSL, it uses Draco packages (C4 and SPECIAL_FUNCTIONS) that do require these libraries. In practice, the Draco build system attempts to find and use full paths to vendor and Draco component libraries so that `$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`}.

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,

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
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
SPECIAL_FUNCTIONS	yes	yes	no
TIMESTEP	yes	yes	no
VIZ	yes	yes	no
CDI_ANALYTIC	yes	yes	no
QUADRATURE	yes	yes	no
RTT_FORMAT_READER	yes	yes	no

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 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 linux_mpid
$ cd linux_mpid
$ mkdir 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 additional 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`. We need to 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/draco/`) 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 a `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 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.

[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`.

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

```
# CMakeCache.txt template for Draco
# $Id: compile.tex 6535 2012-04-16 23:09:43Z 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/kgt/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 (0-7)?
DRACO_DBC_LEVEL:STRING=7

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

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

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)
```

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

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

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)
```

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/`.

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 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

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

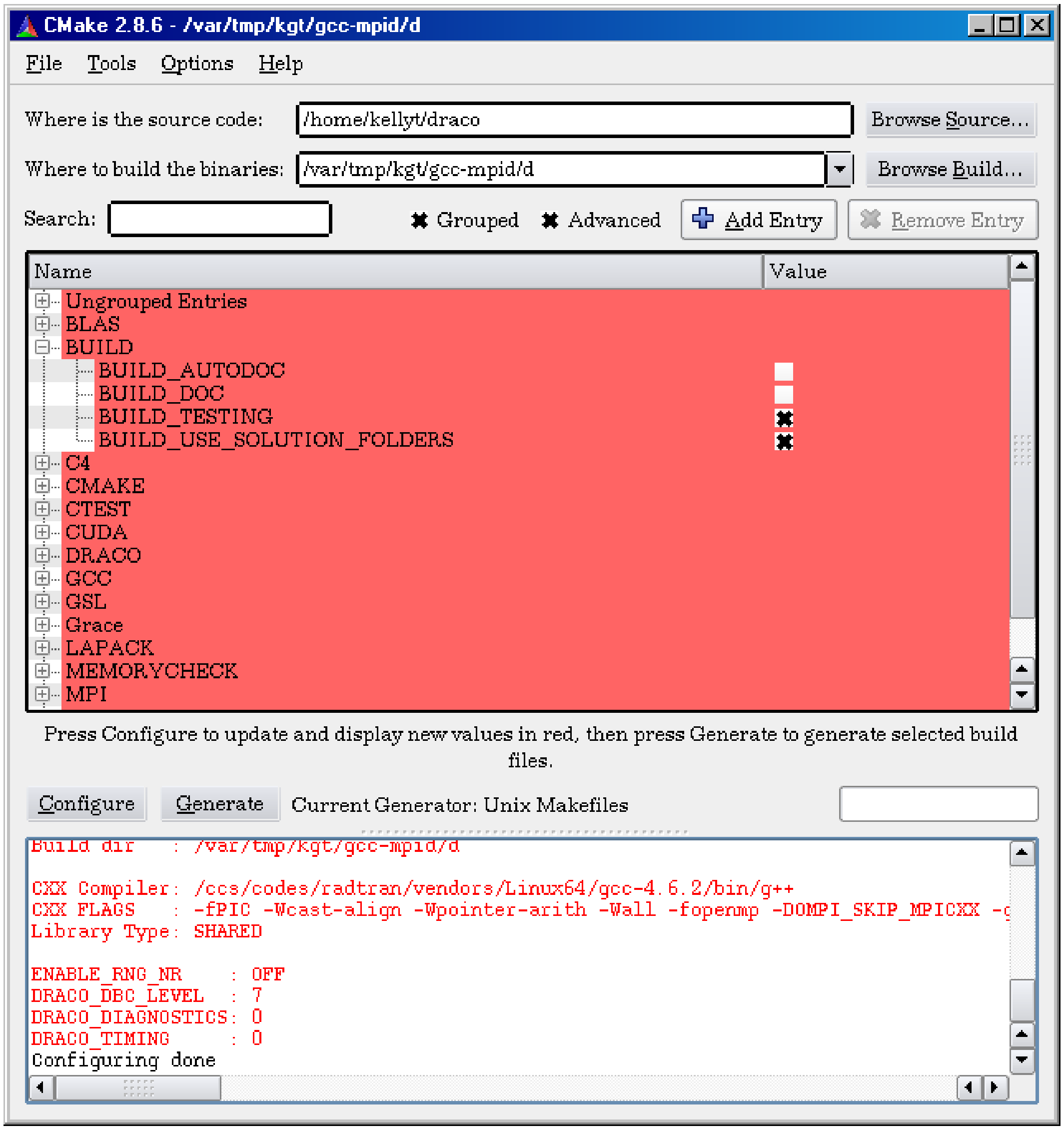


FIGURE 3.2. CMake GUI after populating cache with default values.

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;

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

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_AUTODOC	OFF	Enables discovery of Doxygen and the autodoc build target
BUILD_TESTING	ON	Allows developers to omit configuring for and building code found in test directories (disables **CTest** features)
BUILD_USE_SOLUTION_FOLDERS	ON	Only available for Visual Studio and X-Code generators; makes each Draco component a solution folder
CMAKE_VERBOSE_MAKEFILE[a]	OFF	Forces the make process to echo all commands to the screen during the build process
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.

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 but 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 it 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.

We note that if an optional vendor is not found by the build system then the packages that depend on these vendors will not build. Draco knows what packages require what vendors, so optional vendor libraries that are missing in a particular distribution will have no effect on the rest of the packages.

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.

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*

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	0-7	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.

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-catamount.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.

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 7 (highest). The value is a bit mask similar to that used

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	BLA_STATIC	Look for archive (static) LAPACK and BLAS libraries. Default is ON.
	BLA_VENDOR	Use a particular type of LAPACK installation like ATLAS or Intel10_64lp_gf_sequential[b].
	ENV{LD_LIBRARY_PATH}	To help the build system find LAPACK, ensure that the library location is appended to this environment variable.
	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.
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]To obtain a list of support installations of LAPACK, see the documentation for **CMake**'s `FindLAPACK.cmake` module (try `'cmake --help-module FindLAPACK'`).

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

by the UNIX command `chmod`, `+1` turns on `Require`, `+2` turns on `Check` and `+4` turns on compEnsure. If all options are activated, the Design-by-Contract is 7. 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 highest 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

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`

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`

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`

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.

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` 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 `'-ansi -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 § refsec:examples.

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*.

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 [22] 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. [26].

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/config` 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.

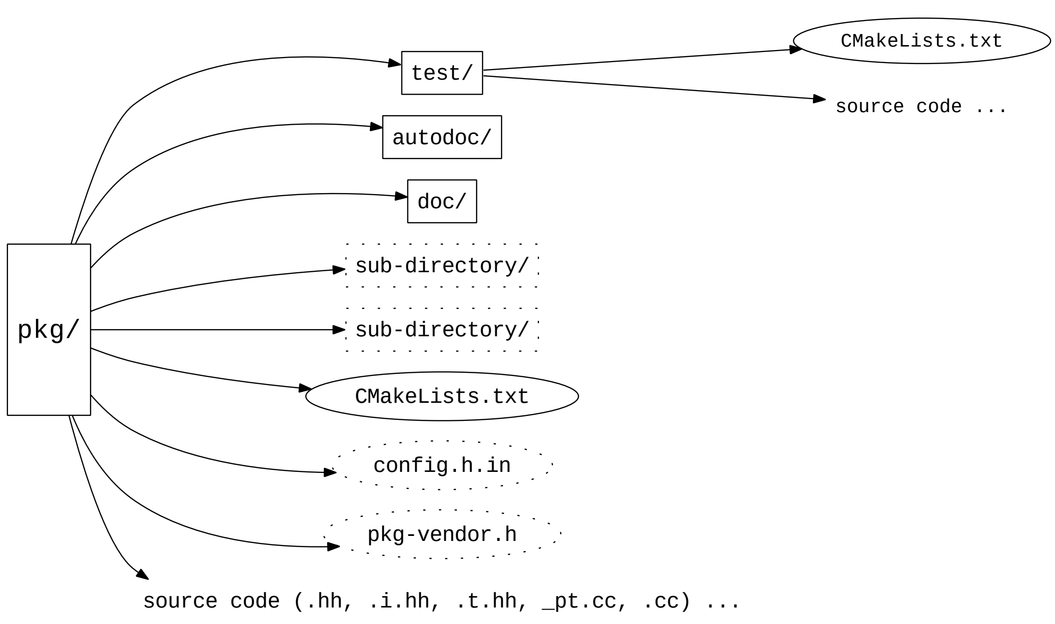


FIGURE 5.1. Standard package directory configuration in Draco.

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.
	`vendor_libraries`	controlling macro that looks for requested vendor libraries by calling `find_package()`.
Toolchain files	`Toolchain-catamount`	allow use of **CMake**'s cross-compile feature to aid the build system configuration for Catamount-like systems.
	`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	`buildEnv`	establish top-level defaults like `DRACO_DBC_LEVEL`
	`compilerEnv`	controls compiler discovery and calls appropriate compiler flag setup routines
	`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`.

Continued on next page

Type	Configuration File	Description
	`platform_checks`	macros that probe the local system for available features, headers, etc.
	`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
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.

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.

`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 5.3 lists all of the usable macros in a Draco `CMakeLists.txt` file. Customizing a `CMakeLists.txt` script is explained in § 5.3.

`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.

5.3. Customized Packages

Section not complete.

```
cmake_minimum_required(VERSION 2.6)
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( Lib_quadrature ${PROJECT_NAME} "${sources}" )
add_dependencies( Lib_quadrature
   Lib_units
   Lib_special_functions
   Lib_ode )

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

install( TARGETS Lib_quadrature 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.

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 2.6 )`	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`.
`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( Lib_quadrature ${PROJECT_NAME} "${sources}" )`	yes	Generate a library from sources, `${sources}`, whose name is based on `${PROJECT_NAME}` and has the build target key `Lib_quadrature`
`add_dependencies( Lib_quadrature Lib_special_functions Lib_ode )`	yes	The build target `Lib_quadrature` must be linked against build targets (libraries) Lib_quadrature, Lib_special_functions and Lib_ode.
Section 5: Install commands		
`install( TARGETS Lib_quadrature 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.
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.

CHAPTER 6

Extending the Draco Build System

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

A.2. LAPACK

A.3. GSL, GNU Scientific Library

A.4. CUDA

A.5. DaCS

A.6. XMGRACE

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

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

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

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

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

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

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

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

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

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

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

[23] ISO/IEC, International Standard, "Programming languages-C++," Tech. Rep. 14882:2011, American National Standard Institute, New York, Sept. 2011.

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

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

[26] 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/.

Index