EasyBuild: Building Software With Ease.

Kenneth Hoste, Jens Timmerman, Stijn De Weirdt, Andy Georges
DICT
Ghent University
Krijgslaan 281, S9
9000 Gent, BELGIUM
E-mail: {kenneth.hoste, jens.timmerman, stijn.deweirdt, andy.georges}@ugent.be

***Abstract*—**

Building and installing numerous software packages is often a tedious, repetitive, error-prone and time-consuming task. There are various tools and package managers that aim to automate and simplify this task, yet they too have their quirks and are often quite different from each other. Moreover, they require system administrators to redo the same work with every upgrade of the system, its core components, or when new versions of software packages are released.

In this work, we present *EasyBuild*, a build framework written in Python that aims to support the various install procedures used by the vast collection of software packages that are typically installed in an HPC environment with a large number of (quite different) user profiles. EasyBuild provides a framework on top of existing tools, making classical build procedures trivial while at the same time providing support for simplifying complex custom installation procedure. The former are supported out-of-the-box, while the latter require little more effort than defining an *easyblock* – as a Python class definition adhering to the provided interface. The easyblocks allow the sharing of real software install procedures, which is a very valuable asset when the same software packages are used at multiple HPC sites.

EasyBuild fully supports using different compiler toolkits, and installing multiple versions of a software package alongside each other. It suffices to provide an *easyconfig* to specify the software package, version, compiler and libraries, and build parameters to deploy a custom built software package. For allowing the co-existence of different versions of a software package, EasyBuild relies on *environment modules*.

Our framework aims to take away the burden of looking up, understanding and implementing install procedures for various software packages over and over again. It simplifies the task of HPC site system administrators to keep their software installations consistent and up to date while catering to widely different user profiles.

I. Introduction

HPC software environments can be quite diverse. Some environments have a rather limited number of installed software packages they offer to end users, while others offer support to users with very diverse needs in terms of the software packages they use and which they thus require to be installed on the system.

Unfortunately, not all (scientific) software packages build and install according to the same procedure or using the same tools. This makes the task of installing the diversity of software required by end users time-consuming and error-prone. For every software package, documentation that describes this (often very software-specific) install procedure has to be read and understood thoroughly. If some automation is desired for installing a package – for example on mulitiple clusters – a script that implements the procedure needs to be written.

Of course, the problem of implementing software install procedures is not new. Most UNIX-like systems such as Linux and BSD based distributions employ some package manager for deploying new software packages and for maintaining the installation. Most package managers have some custom format to detail how a software package should be built and on which packages it depends on. For example, RedHat-based systems use `rpm`/`yum`, with RPM `spec` files for specifying build details. Debian-based systems offer a similar `dpkg` based manager, *BSD have ports, etc. While all of these make the maintenance of software environments simpler by supporting easy upgrade paths, automatic dependency resolution, etc., there are several shortcomings for maintaining one or more installations of scientific software.

There are several important differences between scientific software packages and other software. We found installation procedures for them may suffer from any of the following shortcomings.

- It is (often) incomplete. For example, only compilation in the source directory is supported and it is a hassle to actually install the executables, libraries and include files, etc.
- It is very non-standard, i.e., far more involved than a sequence of configure, make and install steps. The reason is that the installation procedure is often interactive, requiring action on the part of the administrator during the configuration and installation.
- It uses custom-built scripts instead of a set of standard tools such as `configure`, `make`, `cmake`, etc.
- Quite often, system-specific parameters such as the compiler commands or the list of library dependencies (e.g., BLAS/LAPACK, MPI, ...) are hard-coded in the configuration scripts and files or in the installation scripts.

On top of this, commercial scientific software packages often come with binary – again, potentially interactive – installers.

It is precisely because of these shortcomings that package managers do not to handle a lot of scientific software well; moreover these software packages users would like to have for their experiments are often not available as RPMs or their equivalent for other package managers. Originally, we tried fitting the install procedures for a couple of scientific software packages into RPM `spec` files, but we quickly realised that

such traditional package managers are ill suited for resolving the issues we faced.

The problem is complicated further in an HPC software environment where users also have several typical requirements that are not (well) handled in traditional package managers or build tools. Scientists need to have particular builds, i.e., versions built with a specific compiler toolchain, of software packages available for an extensive period of time – preferably indefinitely. The reason is quite straightforward: they should be able to reproduce[1] or extend their previously obtained results whenever the need arises. When they start (new) experiments, researchers often desire to use to the latest and greatest version of the software. Furthermore, they like to be able to experiment with various builds of a particular software package with different compilers or libraries, for evaluating performance and correctness. Although package managers are good at keeping software up to date and taking care of dependencies, to the best of our knowledge, most do not support these other requirements well.

In short, to maintain the software in an HPC environment, we require a tool that:

- is able to build and install scientific software in a flexible, reproducible and robust way,
- supports the co-existence of multiple installed versions or builds of a particular software package,
- takes care of handling dependencies between software packages to be installed,
- allows sharing of implementations of software install procedures between sites.

A tool that meets these criteria has several advantages: (i) it reduces the effort on behalf on the system administrator when the result from earlier installations can be re-used in a simple way, and (ii) it enables forming a community to tackle the software maintaining problem in a collective manner.

Since we were unable to find a tool that matches these requirements, we began the development of EasyBuild, a modular build and install framework for software, written in *Python* [Python Software Foundation(1990)].

Originally, EasyBuild only supported a couple of software packages which feature custom build procedures, but quickly grew to become a very important part of the system administration toolchain at HPC-UGent[2]. Today, EasyBuild has support for over 250 scientific software packages – it is being developed and improved continuously. It allows us to limit the amount of time and effort required to install and update end-user software packages, by investing time once to implement the install procedure in the EasyBuild framework. Subsequent builds of new versions of a software package or builds that are using different parameters can usually be obtained with very little effort, thereby saving lots of time and manpower.

In April 2012, after more than three years of in-house development, we have started releasing EasyBuild as a free and open source (GPLv2 license) tool[3]. We are currently in the process of porting code for all supported scientific software packages from our legacy version to the publicly available open source version.

This paper aims to show the potential EasyBuild has for system administrators and for user support teams at other HPC sites and to encourage them to provide feedback and to add their ideas for further developing this framework, as well as to use it at their sites for handling end-user software installation requests.

The remainder of this paper presents an overview of EasyBuild in Section II, highlights its main features in Section III, discusses how it can be used to easily install software that is already supported in Section IV, and explains how to add support for additional (scientific) software packages in Section V. Finally, we compare our framework to related tools in Section VI.

II. EASYBUILD OVERVIEW

The EasyBuild framework consists of (i) a collection of Python modules grouped in five packages implementing classes that provide the functionality required by various software install procedures, (ii) a collection of *easyblocks*, each of which is a Python module implementing support for a (group of) software packages, and (iii) scripts to perform the actual building, installation and testing. We present a detailed overview of the complete EasyBuild design in Section II-A.

A so-called *easyconfig* file (`.eb`) specifies (i) the software package and its version that should be built, (ii) the *compiler toolkit* (see section II-C) and the build parameters to use, and (iii) lists which (versions of) dependencies are required. Easyconfig files are discussed in detail in Section II-B.

By supplying an easyconfig file, EasyBuild is able to determine how the various steps that collectively make up the software install procedure should be performed. Assuming that it knows about how to build the desired software package, EasyBuild will perform said steps on after the other, thus executing the install procedure as dictated by the easyblock. An overview of the step-wise install procedure that EasyBuild follows is presented in Section II-D.

After building and installing the desired software package according to the specifications given, a *module file* is produced, both as a simple way of giving the user access to the built software, and as a tool for EasyBuild to use during subsequent builds of (other) software packages, e.g., when resolving dependencies. EasyBuild relies heavily on the *environment modules* tool [Furlani and Osel(1996a)], [Furlani and Osel(1996b)], which is covered in Section II-E.

A. Design

In this section, we present to design of EasyBuild as illustrated in Figure 1.

We first give an overview of the various packages that make up our framework.

[1]Reproducibility of results is good scientific practice.

[2]http://www.ugent.be/hpc

[3]http://github.com/hpcugent/easybuild

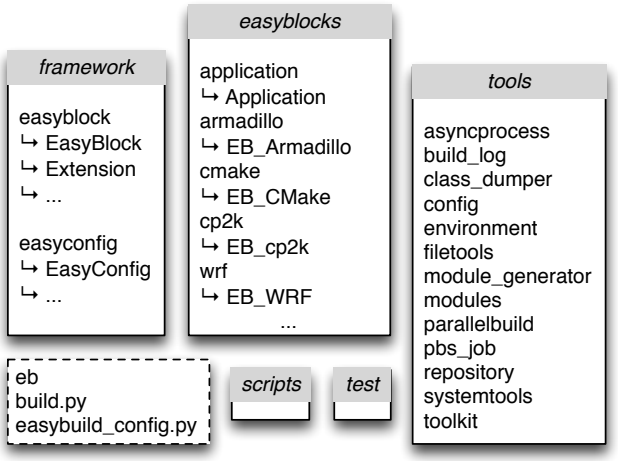

Fig. 1: Overview of the EasyBuild design.

- The *framework* package hosts the core of EasyBuild, i.e., the `easyblock` and the `easyconfig` modules.
- Modules that build on this base infrastructure by providing subclasses of a generic `EasyBlock` class and thus implement support for actual install procedures are hosted in the *easyblocks* package.
- Supporting functionality for these modules is provided by the *tools* package, which for example offers interfaces to and wrappers for common shell commands through Python functions.
- The *scripts* and *test* packages host stand-alone scripts that are useful during EasyBuild development and a unit tests for EasyBuild, respectively. Next to this, the `easybuild_config.py` file provides a sensible default configuration file for EasyBuild, while `build.py` is the main EasyBuild script. The `eb` command is a handy wrapper around `build.py` that automatically sets the correct `PYTHONPATH`, such that all EasyBuild packages, modules and classes are accessible to the Python runtime environment.

In the next sections, we provide further details about the most important aspects of the EasyBuild design.

1) the `easyblock` module: The most significant part of this module is the `EasyBlock` class. This class implements generic support for software install procedures, and serves as an abstract class that should be subclassed to obtain an easyblock that describes the install procedure for a particular (group of) software package(s). The `Extension` class that accompanies `EasyBlock` provides generic support for installing software extensions – for example, Python packages, R libraries, Perl modules, etc. Such extensions can be installed in two ways. First, as a part of the installed base software package they extend. Second, completely separate from this software package, in which case the extensions have their own dedicated module file (see section II-E).

Andy: TOT HIER

2) the easyblocks package: This provides a collection of easyblocks implemented as Python modules, each of which providing support for a particular software package, or a group of software packages. For example, the `cp2k` module provides the `EB_CP2K` Python class, which implements the install procedure for the molecular simulation software CP2K. Likewise, the `EB_Armadillo` and `EB_WRF` Python classes shown in Figure 1 provide support for the expected software packages.

All classes in easyblock modules are named according to a class name encoding scheme, which we have put into place to cope with names of software packages that do not map to valid Python class names directly, for example `python-meep` or `7zip`. To avoid potential name clashes with existing functionality, we also prefix all class names with `EB_`.

The `application` module is an example of a more generic easyblock. The `EB_Application` class it hosts implements the commonly used `configure`, `make`, `make install` install procedure, allowing to specify custom options to the `configure` and `make` commands. For software packages that use this well-known install procedure, it is likely no dedicated easyblock needs to be implemented, since the `application` easyblock already provides support for them.

Similarly, EasyBuild already provides generic support for binary software packages that should simply be unpacked in the installation directory via the `EB_Binary` class in the `binary` easyblock, and for software packages that replace `configure` with `cmake` for build configuration through `EB_CMake`, among others.

Figure 2 shows the hierarchy for the mentioned easyblock classes. All classes derive from the generic `EasyBlock` class, albeit direct or indirect. Classes that implement custom support for particular software package, like `EB_CP2K` or `EB_WRF`, or implement support that is shared by groups of software packages, like `EB_Application`, derive directly from `EasyBlock`. Others, like `EB_CMake`, are further customisations of already existing easyblocks; for `EB_CMake` specifically, only the build configuration step differs from the install procedure implemented by `EB_Application`. In turn, they too can be used as a base for implementing further custom support, as is the case with `EB_Armadillo`, which slightly modifies the `EB_CMake` install procedure with configuration parameters and dependency checks that are Armadillo-specific.

3) the `easyconfig` module: The functionality in this module, which mainly consist of an `EasyConfig` class with appropriate instance methods, takes care of processing the easyconfig files supplied to EasyBuild. After parsing the easyconfig files and creating `EasyConfig` instances, EasyBuild is able to obtain the required information to set up and run the install procedure according to the given specifications. We will discuss easyconfig files in details in section II-B.

4) the tools package: The modules provided in this package form the backbone of EasyBuild. We briefly clarify the goal of the available modules:

- `asyncprocess`: implements support for interacting

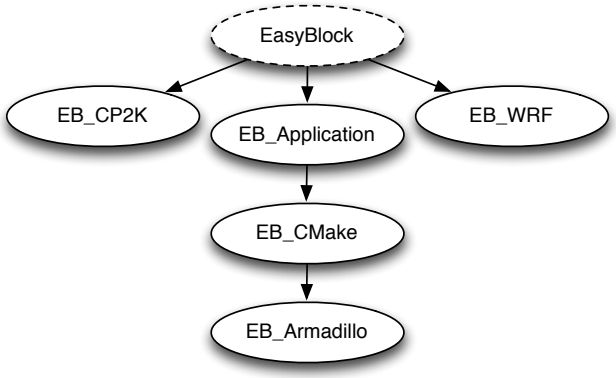

Fig. 2: Schematic of the hierarchical organisation of selected easyblocks, all deriving from the `EasyBlock` class.

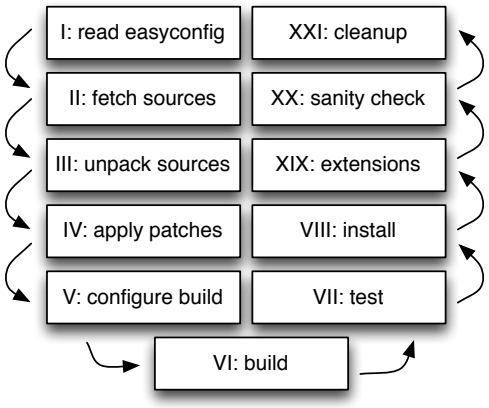

Fig. 3: Installation procedure broken down into steps as performed by EasyBuild.

Python subprocesses; this is the base for supporting interactive installers
- `build_log`: provides a custom logger class `EasyBuildLog` and accompanying log utility functions, along with a custom error class `EasyBuildError`
- `class_dumper`: offers functionality for supporting the EasyBuild command line option to dump a class/easyblock hierarchy overview
- `config`: functions for parsing and querying the EasyBuild configuration file
- `environment`: utility functions for keeping track of changes to the session environment; the `set` function provided by this module is used throughout EasyBuild to set or update environment variables
- `filetools`: large module with various wrapper functions for shell commands; a couple of noteworthy examples are `extract_file` for extracting source files with a command determined by the file extension, `apply_patch` for applying patch files and automatically determining patch levels, `run_cmd` and `run_cmd_qa` for running (interactive) shell commands, etc.
- `module_generator`: an interface module for producing environment module files (see section II-E)
- `modules`: a Python API for environment modules
- `parallelbuild`: utility functions for running multiple builds in parallel
- `pbs_job`: a Python API for PBS/Torque, to submit builds as jobs on a cluster
- `repository`: interfaces for file, SVN or git easyconfig repositories
- `systemtools`: operating system specific functionality, e.g., determining the number of processor cores available or getting system information
- `toolkit`: support for compiler toolkits, see section II-C

Together, these Python packages and modules form the modular software installation framework that we call EasyBuild. The subsequent sections discuss further important aspects of the inner working of EasyBuild.

B. Easyconfig files

C. Compiler toolkits

D. Step-wise install procedure

E. Environment modules

III. Features

(Kenneth, Andy): Nifty features, like:
- *keeping track up build logs, adding used easyconfigs to repository*
- *robot: automatic dependency resolution*
- *support for interactive installers (e.g. WRF)*
- *building in parallel on a cluster with a single command*
- *regression testing, both unit tests and building software*

IV. Using EasyBuild

Briefly discuss two running examples, e.g., one simple configure/make/make install and one with a lot of dependencies. Provide the dependency graph in a figure. Can we autmagically generate these, e.g. using dot? Give details about the effort it took to port. Preferably, we have something for which we support multiple versions, so we can then detail how much less effort the next versions took.

(Kenneth, Andy): Examples, from simple to complex (needs trimming)

- *gzip as example of default build procedure (configure/make/make install) (SHORT!)*
- *building and installing Python packages (e.g. FIAT or ScientificPython)*
- *Python package with CMake configuration and custom options (UFC)*
- *fully custom build procedure (e.g. WRF)*
- *example with complex dependency graph (DOLFIN)*

V. Adding support for additional software

VI. Related Work

see wiki

VII. Conclusion

Acknowledgments

References

[Furlani and Osel(1996a)] J. L. Furlani and P. W. Osel. Environment modules. `http://modules.sourceforge.net/`, 1996a.

[Furlani and Osel(1996b)] J. L. Furlani and P. W. Osel. Abstract yourself with modules. In *Proceedings of the 10th USENIX conference on System administration*, LISA '96, pages 193–204, Berkeley, CA, USA, 1996b. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1029824.1029858.

[Python Software Foundation(1990)] Python Software Foundation. Python Programming Language. `http://python.org/`, 1990.