EasyBuild: Building Software With Ease.

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

***Abstract*—Maintaining a collection of software installations for a diverse user base can be a tedious, repetitive, error-prone and time-consuming task. Because most end-user software packages for an HPC environment are not readily available in existing OS package managers, they require significant extra effort from the user support team. Reducing this effort would free up a large amount of time for tackling more urgent tasks.**

In this work, we present *EasyBuild*, a software installation framework written in Python that aims to support the various installation procedures used by the vast collection of software packages that are typically installed in an HPC environment – catering to widely different user profiles. It is built on top of existing tools, and provides support for well-established installation procedures. Supporting customised installation procedures requires little effort, and sharing implementations of installation procedures becomes very easy. Installing software packages that are supported can be done by issuing a single command, even if dependencies are not available yet.

Hence, it simplifies the task of HPC site support teams, and even allows end-users to keep their software installations consistent and up to date.

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 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 often required by end-users time-consuming and error-prone [2], [13].

Of course, the problem of implementing software installation procedures is not new. Most UNIX-like systems such as Linux and BSD based distributions employ a package manager – closely coupled to the distribution – for deploying new software packages and for maintaining the installation. Most package managers have some custom format to define how a software package should be built and which packages it depends on. For example, RedHat-based systems use `rpm` packages and `yum` to install them, where the build specification is detailed in the RPM `spec` files. SUSE, Debian-based systems, BSD derivatives, etc. have similar tools. 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. This is mostly due to the differences between scientific software and software that is provided as a part of an operating system defined and controlled by software maintainers. In the scientific community, typically fewer resources are spent on the maintenance of build procedures; almost all of the effort is put into the development (and testing) of the code. We found the following issues with installation procedures for the scientific software packages we support at the UGent HPC site[1].

- *Incompleteness*. 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.
- *Non-standard procedure*. Installation procedures are often far more involved than a sequence of configure, build and install steps. For example, the installation procedure can be interactive, i.e., requiring human intervention during the configuration and installation.
- *Custom-built scripts*. On various occasions, custom shell scripts need to be used, as opposed to a set of standard tools such as `configure`, `make`, `cmake`, etc.
- *Hard-coded parameters*. System-specific parameters such as the compiler commands or the list of libraries, e.g., BLAS/LAPACK, MPI, etc., are hard-coded in the configuration files or in the installation scripts.

Commercial software packages, which are commonly used in various scientific research domains, rarely follow any standard. They all provide their own procedures, and thus also suffer from similar problems.

The problem is aggravated further in an HPC software installation environment where users typically have several requirements that are not (well) handled by traditional package managers or build tools. Scientists need to have particular builds – versions built with a specific compiler toolchain – of software packages available for an extensive period of time, preferably indefinitely. [2] Additionally, researchers often desire to use to the latest and greatest version of the software when they start (new) experiments. Often, they like to experiment with various builds of a particular software package using different compilers or libraries, for evaluating both 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.

Due to these shortcomings, scientific software packages are getting less support from distributions and their package manager maintainers – detailed customisation is time-consuming and difficult; package managers rarely provide sufficient flexibility for dealing with this.

In short, to maintain the software in an HPC environment, a tool is required that offers:

- *Flexibility*. There are many different installation procedures and they should be supported with minimal effort. This results in a tool that is able to build and install software in a flexible, reproducible and robust way.
- *Co-existence of versions*. In principle, installed software should never have to be removed. Hence, different versions of builds must be able to be installed independently of each other.

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

[2] The reason is quite straightforward: they should be able to reproduce – this is good scientific practice – or extend their previously obtained results whenever the need arises.

- *Dependency handling.* Many critical software packages in an HPC environment have dependencies on each other, e.g., numerical libraries such as BLAS, LAPACK, etc., but also cooperative stacks of software applications – see for example the WRF dependency graph discussed in Section IV. Handling these dependencies is an aspect that is traditionally handled well by most package managers. Such automatic dependency resolution significantly simplifies the maintenance of a collection of software installations, and is thus indispensable in any relevant framework.
- *Sharing implementations of installation procedures.* Although usually the installation procedure for software packages is (well) documented, a lot of work is duplicated among user support teams: (i) digging through the documentation, (ii) following the installation procedures, and likely (iii) scripting the installation in some way. To reduce this inefficient approach, it should be easy to share implementations of software installation procedures with others. First and foremost, this requires a modular plug-and-play enabled infrastructure, to which support for any particular software package can be added with minimal effort. This should be independent of the package source, whether it is self-written or obtained from others. Note that package managers also allow sharing of installation procedures, but they are more rigid than what an HPC site typically requires.

A tool that meets these criteria has several advantages: (i) it reduces the effort on behalf of the user support team when the result from earlier installations can be reproduced in a simple way, and (ii) it enables forming a community to tackle the software maintenance problem in a collective manner.

Because we were unable to find another tool that matches the requirements listed above, we started the development of EasyBuild, a modular build and install framework for software, written in *Python* [11]. This framework replaced our earlier effort of deploying through custom RPM `spec` files using a traditional package manager, which was ill suited for addressing the issues we faced and mostly resulted in a number of large, hardly maintainable shell scripts for expressing customisation.

While EasyBuild originally supported just a few software packages that featured custom build procedures, it quickly grew to become a very important part of the user support tools used by the HPC UGent team. 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 installation procedure in an easyblock, see Section II-E. Subsequent builds of new versions of a software package or builds that are using different parameters can usually be obtained with very little or no effort, thereby saving lots of time and manpower.

In April 2012, after more than three years of in-house development, we have released EasyBuild on GitHub[3] as open source software under the GPLv2 license. It is also available from the Python Package Index (PyPi)[4]. We are currently in the process of migrating all supported software packages from our legacy version to the publicly available version. This paper presents EasyBuild version 1.0, which was released in November 2012, and is considered to be the first stable and robust public release of this framework.

With this paper, we want to show the potential EasyBuild has for both user support teams and for end-users, given the targeted ease-of-use and lack for a need of admin rights. Our aim is twofold: (i) to encourage the use of EasyBuild, and (ii) to receive feedback and ideas to further improve this framework, increasing its usefulness for tackling the issues HPC sites have when maintaining software installations. EasyBuild is structured in a very modular way, simplifying collaboration and allowing the HPC community to provide support for new software packages. Contributing is as easy as forking the repository on GitHub and filing a pull request so we can incorporate the contributions in the framework.

The remainder of this paper presents an overview of EasyBuild in Section II, highlights its main features in Section III, and discusses a software installation use case in Section IV. Finally, we compare our framework to related tools in Section V, and conclude in Section VI.

II. EASYBUILD OVERVIEW

In this section, we give a detailed overview of EasyBuild by presenting its basic usage and configuration, listing the required dependencies, and discussing the design.

A. Basic usage and configuration

The basic usage of EasyBuild is simple: run the `eb` command with the appropriate arguments. Usually, the path to the easyconfig files, see Section II-D, should be specified explicitly. Several command-line options for `eb` are available, e.g., `--debug` to enable debug mode, `--robot` to enable the the automatic dependency resolver and supply a path to it, etc. A full list of options can be obtained by running `eb --help`.

Before EasyBuild is used the first time, a simple configuration file containing plain Python code should be created. There, fixed-name variables are defined that specify (i) the paths where the temporary log files are stored, (ii) the paths of the build, source and installation directories, (iii) the format of the log file names, and (iv) the path to the easyconfig files repository – see Section III-B. The location of this configuration file can be provided to the `eb` command with the `--config` option or by setting the `EASYBUILDCONFIG` environment variable.

EasyBuild comes with a default configuration file `easybuild_config.py` that uses the `$HOME/.local/easybuild` directory as a prefix for the build, source and installation directories.

B. Dependencies

EasyBuild only has two direct dependencies: Python and environment modules. We use Python version 2.x, where the version should be at least 2.4. The reason is quite straightforward: on the UGent HPC clusters we run Scientific Linux 5.x and 6.x, where the system Python versions are 2.4 and 2.6, respectively.

The environment modules software package is a well-known tool in the HPC community. Through simple text files, referred to as *environment modules*, an easy-to-use interface can be offered to users to prepare their session environment for using a particular software package. Environment modules describe the changes to the session environment that are necessary for a piece of software to work correctly. They can append to the `PATH` and `LD_LIBRARY_PATH` environment variables, such that binaries and shared libraries are readily available.

[3]http://hpcugent.github.com/easybuild/
[4]http://pypi.python.org/pypi/easybuild/

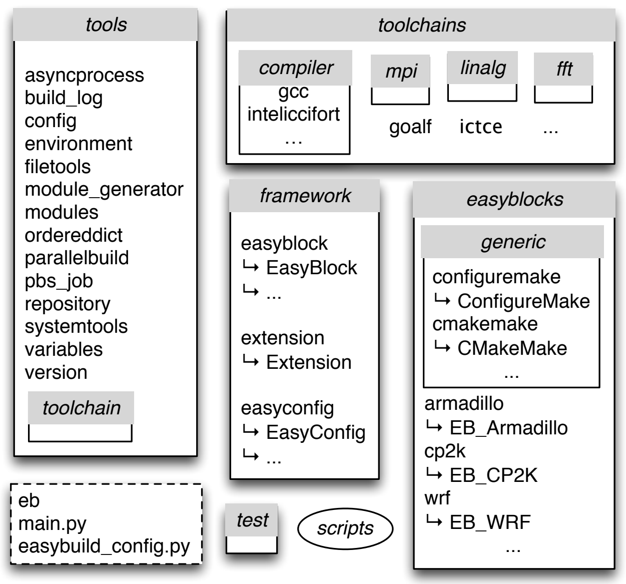

Fig. 1: Overview of the EasyBuild design.

EasyBuild heavily relies on environment modules in a number of ways, making the environment modules software package an important prerequisite. EasyBuild automatically generates environment modules for every software package it installs, thereby relieving the user having to manually create appropriate module files. Furthermore, it relies on the set of available environment modules for obtaining information about installed software packages and their versions, and for resolving dependencies.

By generating an environment module for every completed installation, EasyBuild allows for keeping different versions and/or builds of software packages side-by-side, without affecting each other. Next to providing access to the installed package for end-users, these modules also help EasyBuild to locate the software during subsequent installations of other dependent software packages, i.e., for resolving dependencies – as will be explained in more detail later on.

C. High-level design

We first give a high-level overview of the EasyBuild design before diving into the details in the next sections, see Figure 1.

EasyBuild consists of (i) the *framework* Python package containing several modules that form the core of the tool, (ii) the *easyblocks* package providing the easyblocks that implement specific installation procedures that can be used to install one or more software packages (see Section II-E), (iii) the *tools* package providing a set of tools that offer supporting functionality, (iv) the *toolchains* package providing support for compiler toolchains, (v) the `eb` command, the main script and a default configuration file, and (vi) a unit testing framework in the *test* package and some useful stand-alone scripts. Next to this, a collection of *easyconfig* files for specifying installation parameters is available, see Section II-D.

From an easyconfig file and a matching easyblock, EasyBuild is able to determine what is required by the various steps that form the installation procedure and how they should be performed, see Section II-H.

The EasyBuild design is purposely very modular: easyblocks that provide support for new software packages can be easily plugged in without modifications to the existing code. If easyblocks are available in the Python search path, EasyBuild will find them and use them when appropriate. Likewise, new compiler toolchain definitions and support for additional compilers or libraries can be plugged in to be a part of toolchains (see Section II-G).

D. Easyconfig files

Essentially, easyconfig files are text files with the `.eb` file name extension that contain a description in Python code format. They specify the software package that should be installed along with a number of parameters to steer the build and installation procedure, including dependencies. For the sake of space, we only discuss the most important parameters here; a complete list of available parameters[5] can be obtained by running '`eb --avail-easyconfig-params`'.

There are several mandatory parameters: (i) `name` denotes the software package name, (ii) `version` denotes the specific version of the package, (iii) `toolchain` denotes the compiler toolchain, specified as a dictionary with the name and the version, (iv) `homepage` denotes the URL of the software package's website, and (v) `description`. The last two parameters are only used to include some documentation in the environment module that is generated upon successful completion of the installation.

The following are noteworthy optional parameters.

`easyblock` specifies the custom easyblock that provides the build and installation procedure, see Section II-E. `sources` specifies the source[6] files that should be present at any of the predetermined locations – if not, EasyBuild will try and download them. Software packages specified in the `dependencies` parameter have a twofold effect. First, before building the packages, EasyBuild makes sure that the environment modules for all dependencies are available and loaded. If this fails, EasyBuild can recursively build the required dependencies, see Section III-C. Note that automatic dependency resolution is not enabled by default. Second, they are added in the generated environment module file, such that all dependencies are correctly loaded in the environment. This makes sure the environment for the user or for subsequent EasyBuild runs is set up correctly and the installed software can be found. `sanity_check_paths` specifies a dictionary of files and directories that should be present after the software package files have been installed.

Next to these parameters, it is also possible to specify configuration options, compiler flags, optimisation levels, etc.

The *framework* package contains the `easyconfig` module with the `EasyConfig` class that processes the easyconfig files at run time. For each easyconfig file supplied to EasyBuild, an `EasyConfig` instance will be created. This allows our framework to obtain the information required to set up and run the installation procedure according to the given specifications.

An example easyconfig file for the WRF software package is discussed in Section IV.

E. Easyblocks

[5]Documentation for all easyconfig parameters is provided at https://github.com/hpcugent/easybuild/wiki/Easyconfig-files.

[6]Note that *source* does not mean source code, it can also refer to a binary installer or binary package

As mentioned above, an easyblock defines the build and installation procedure that can be used by one or more software packages. The *framework* Python package provides support for implementing easyblocks through the EasyBlock class which resides in the easyblock module. This class implements generic support for software installation procedures. It serves as the base class that should be sub-classed to obtain an easyblock that describes the installation procedure for a particular (group of) software package(s).

The extension module provides support for installation procedures of software packages extensions, e.g., Python packages, R libraries, Perl modules, etc., via the Extension class. Such extensions can be installed in two ways: (i) using Extension, as a part of the installed base software package they extend, and (ii) completely separately from the base software package. In the former case, the extensions are listed in the exts_list parameter of the software package's easyconfig file. In the latter case, they are specified in their own easyconfig and treated as a stand-alone software package with the base software as a dependency, and with their own dedicated environment module files.

The easyblocks themselves are placed into a separate Python package, aptly named *easyblocks*. Each easyblock is implemented as a Python module. For example, the cp2k module provides the EB_CP2K Python class, which implements the installation procedure for the molecular simulation software CP2K. Likewise, the EB_Armadillo and EB_WRF Python classes shown in Figure 1 provide support for the corresponding software packages.

All the classes in the easyblock modules for specific software packages are named according to a fixed class name encoding scheme. This allows us to adequately cope with names of software packages that do not directly map to valid Python class names, for example python-meep or 7zip. To avoid potential name clashes with existing functionality, we prefix all class names of easyblocks bound to a particular software package with EB_.

Next to software package-specific easyblocks, EasyBuild offers a number of generic easyblocks in the *generic* subpackage of the *easyblocks* package. We briefly discuss a couple of these shown in Figure 1. The configuremake module defines the ConfigureMake class, that implements the commonly used GNU configure, make, make install installation procedure. This class allows specifying custom options to the configure and make commands. Software packages that use this well-known installation procedure likely do not require a dedicated easyblock to be implemented, because the configuremake easyblock already provides support for them. Another example of a generic easyblock is the CMakeMake class from the cmakemake module, which supports software packages that use cmake instead of configure for their build configuration.

Figure 2 shows the hierarchy for the easyblock classes discussed above. The generic EasyBlock class sits at the root of the class hierarchy. Classes that implement custom support for a particular software package directly sub-class this hierarchy root, for example EB_CP2K and EB_WRF. Others implement an installation scheme that can be used by multiple software packages, such as ConfigureMake. Further customisation of already supported installation procedures is implemented by sub-classing existing easyblocks. An example is CMakeMake where only the build configuration step differs from its parent class ConfigureMake. All classes that offer support for a particular software package sub-class one or more of the

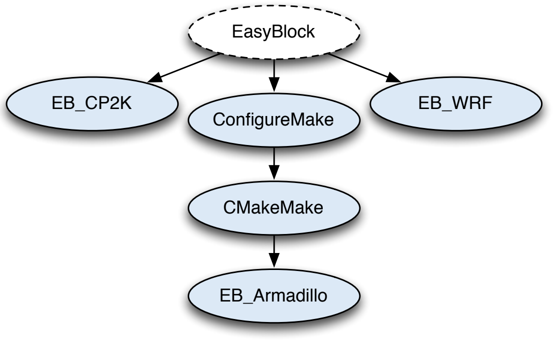

Fig. 2: Schematic of the hierarchical organisation of selected easyblocks, all deriving from the EasyBlock class.

generic easyblock classes. In Figure 2, this is illustrated by EB_Armadillo, which slightly modifies the CMakeMake installation procedure with configuration parameters and dependency checks that are specific to Armadillo.

By organising the implementation of software install procedures in isolated Python modules as is done with the easyblocks, sharing them becomes particularly easy – one only needs to provide the Python module. Making easyblocks available to EasyBuild amounts to extending the *easyblocks* package, allowing the Python modules to be found in the Python search path. We feel this is an important feature of the EasyBuild framework.

The EasyBuild framework provides a very flexible interface for implementing software install procedures. Not only does it provide a lot of useful functionality required when installing software, it also allows to easily plug in easyblocks thereby adding support for new software packages and to build on existing easyblocks by extending or customising them.

A concrete example of an easyblock implementation for the WRF software package is shown in Section IV.

F. Scripts, tests and tools

The main EasyBuild script is main.py. A handy wrapper script named eb that searches for main.py in the Python runtime search path is also provided, and is generally used as a command line tool for EasyBuild.

Additionally, there are several stand-alone Python scripts available that are useful during EasyBuild development, next to Python package *test* for running unit tests, but these fall outside the scope of this paper.

The *tools* package provides the backbone modules of the EasyBuild framework. We will briefly highlight some functionality provided there. More details are provided later in dedicated sections.

Two important elements in the *tools* package are the filetools module and the *toolchain* package. The filetools module provides various wrapper functions for shell commands. A couple of noteworthy functions are (i) extract_file for extracting source files with a command determined by the file extension, (ii) apply_patch for applying patch files and automatically determining patch levels, and (iii) run_cmd and run_cmd_qa for running (interactive) shell commands. The *toolchain* package provides the necessary support for compiler toolchains, see Section II-G.

Python modules providing an interface for communicating with the environment modules tool and with the PBS cluster

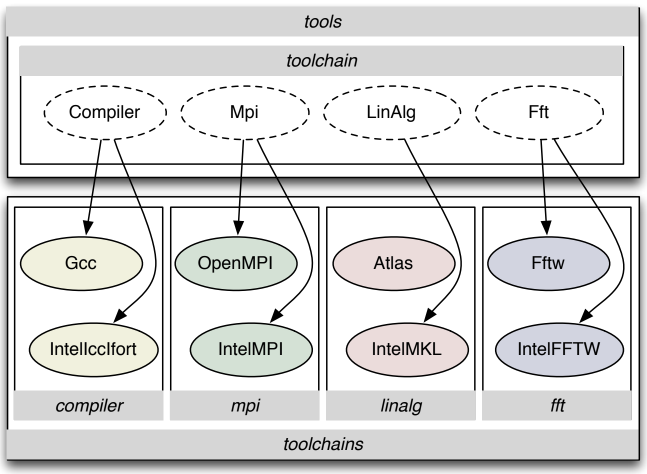

Fig. 3: Modular design of support for compiler toolchains.

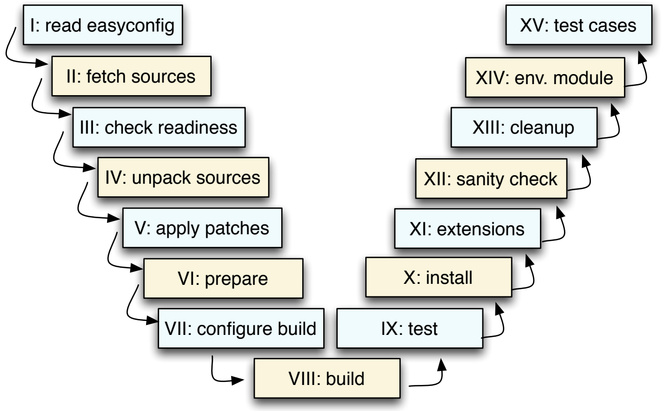

Fig. 4: Installation procedure broken down into steps, as performed by EasyBuild.

resource manager are also available in the *tools* package.

G. Compiler toolchains

All software packages that are supported by EasyBuild and for which source code is available are compiled from source before they are installed. This is preferred over using readily available binary packages in the case of scientific software. This way, EasyBuild maintains complete control over the compiler and the libraries that provide low-level functionality, e.g., MPI, linear algebra support like BLAS, LAPACK, etc. As a consequence, system-specific optimisations can be used and the overall performance of the installed binaries can be increased. In EasyBuild, a (set of) compiler(s) and accompanying libraries are grouped together in a *compiler toolchain*. For every software installation procedure, a compiler toolchain should be provided in the obligatory easyconfig `toolchain` parameter.

The support for custom compiler toolchains is organised in a very modular manner, see Figure 3. Essentially, any compiler toolchain may be defined if EasyBuild supports all of its constituent parts. The framework already provides generic classes for compilers and different library types (MPI, linear algebra and FFT) in the *tools.toolchain* package. It suffices to define the specifics of the toolchain elements in custom classes derived from these generic classes, thus providing the information required by EasyBuild for using the compiler and for accessing its associated libraries in a particular compiler toolchain. Definitions of compiler toolchains and their constituent parts are provided in the *toolchains* package (see Figure 1). By making additional modules available in the *toolchains* package, custom toolchains can be provided without any changes required to the EasyBuild codebase, again highlighting the modular design of the EasyBuild framework.

An example of a custom compiler toolchain is *goalf* which entirely comprises open-source tools: the GCC compiler suite, and the OpenMPI, ATLAS, BLACS, ScaLAPACK and FFTW libraries. To implement this, `Gcc` sub-classes the generic `Compiler` class. It supplies EasyBuild with e.g., the commands to run the GNU C, C++ and Fortran compilers, (`gcc`, `g++` and `gfortran`, respectively). Similarly, it states how to enable OpenMP support (`-fopenmp`), how to control floating-point precision, etc.

Next to *goalf*, other compiler toolchains can be defined. They may comprise Intel-provided compilers and libraries (e.g., the `ictce` toolchain) or they can be based on another compiler suite with the AMD Core Math Library (ACML), etc.

A toolchain should be installed like any other software package. This requires an easyconfig file that lists the name and version along with the dependencies. The install procedure is provided by the `Toolchain` easyblock.

In a few special cases – when building a compiler for use in a custom toolchain or when installing binary software packages – it is necessary to use a *dummy* toolchain. This amounts to setting both name and version to `dummy` in the easyconfig `toolchain` parameter.

In practice, the first thing to do in an HPC environment to build and install a custom toolchain that is then subsequently used to deploy the rest of the end-user software packages and their dependencies. This allows the low-level libraries to remain in place – recall that typically, software is never removed; using OS supplied compilers and libraries may break the installation should there be an upgrade of the HPC systems.

H. Step-wise installation procedure

In this section, we briefly discuss the step-wise installation procedure that EasyBuild follows, see Figure 4. Most of the steps can be tailored to a particular (group of) software package(s) by providing a custom easyblock that sub-classes either the generic `EasyBlock` class or one of the existing easyblocks, see Section IV-A for an example.

Figure 4 should be mostly self-explanatory: the step-wise installation procedure consists of reading the easyconfig file, obtaining the source files, checking whether everything is ready to start the build (modules for dependencies are loaded, build directory is present, etc.), unpacking the sources, applying patches, preparing for build by setting up the toolchain, configuring and running the build process, running the provided software test suite (if any), installing the software, adding extensions if specified, performing a sanity check, cleaning up any temporary files or directories, and finishing with the generation of an environment module. If any user-defined test cases are specified in the easyconfig file, they are run after completing the install procedure. This way, they validate and/or benchmark the installed software as it will be used by the user. If any of these steps fail, EasyBuild reports relevant information about the failure in the installation log file, and throws an `EasyBuildError` which results in a termination of the installation procedure.

We limit this discussion to highlighting a couple of interesting aspects of the installation procedure. For a detailed description of each step, we refer to the EasyBuild documentation [7].

If no easyblock is specified through the `easyblock` parameter in the easyconfig and if EasyBuild is unable to locate a suitable easyblock based on the software package name, it will fall back to the `ConfigureMake` easyblock by default. This easyblock implements the GNU `configure`, `make`, `make install` installation procedure.

If for some reason, EasyBuild is unable to locate a source file that is listed in the easyconfig, it tries to download it from the URLs provided in the `source_urls` parameter. Should this fail as well, the installation procedure is terminated and an appropriate message is written to the installation log.

During the unpack step, EasyBuild determines the command for unpacking the sources from the source file name extension. Similarly, when patches are applied in the next step, the patch level is determined automatically, unless it was hard-coded in the easyconfig.

If the software package does not provide a way to move built files to their target installation location, EasyBuild allows building inside the installation location. This can easily be specified in the constructor of the easyblock class, by setting the instance variable `build_in_install_dir` to `True` (see Section IV-A).

After the software has been installed, a sanity check is performed. This check comprises making sure that a predefined list of files and (non-empty) directories is available in the installation directory. This check aims to catch cases where the installer fails to return a non-zero exit code when it is terminated prematurely, which causes EasyBuild to mistakenly assume all is well. For example, we saw installation procedures running to completion without actually producing any of the expected binaries or libraries in the target installation directory.

By default, the sanity check makes sure that the `bin` and `lib` sub-directories of the target installation location are non-empty. Custom sanity checking can be triggered by specifying a list of files and directories in the `sanity_check_paths` easyconfig parameter or by adding a default sanity check in the easyblock. If any of these files or directories cannot be found, EasyBuild assumes that the installation procedure has at least partially failed.

After a successful installation procedure, an environment module corresponding to the installed software package is created. This allows to easily set up the shell session environment for running the installed software at a later point. Usually, this involves adjusting some standard system environment variables such as `PATH` for binaries, `LD_LIBRARY_PATH` for libraries required at runtime, etc.

EasyBuild also defines a couple of custom environment variables in each environment module it generates. For example, the `EBROOTWRF` and `EBVERSIONWRF` environment variables are set in the WRF environment module. These variables are useful for both EasyBuild when resolving dependencies and for end-users who load the module. This allows them to determine the installation path of the software, e.g., for accessing examples, header files or libraries that were installed. The generated environment module also includes the documentation provided by the `homepage` and `description` easyconfig parameters.

Finally, any user-supplied tests cases are considered – these are specified in the `tests` easyconfig parameter. While a test suite provided by the software package is primarily intended to test the correctness of (certain aspects of) the software, the user test cases have a different purpose. They aim to test the installed software package as it would be used by an end-user or by EasyBuild in subsequent installation runs of dependent packages. However, these tests can also be used as benchmarks to evaluate the performance, the correctness and the accuracy of the installed software.

III. Features

We now discuss the key features of EasyBuild.

A. *Keeping track of build logs*

EasyBuild thoroughly keeps track of the executed installation procedure via the custom logger class `EasyBuildLog`. Easyblocks can (and should) produce informative or debug log messages, that describe the actions being performed. The log messages produced by the EasyBuild framework are intertwined in the a single log file, and supply sufficient information should issues arise during any step of the installation procedure and when the software packages are being used.

The installation log is stored in a sub-directory named `easybuild` of the install directory for future reference, along with a copy of the used easyconfig file.

B. *Archiving easyconfig files*

Easyconfig files that led to a successful installation procedure are archived in an easyconfig repository. This can be either a regular directory, or an Subversion or Git repository, as specified in the EasyBuild configuration file.

The original easyconfig file is augmented with extra information before it is stored in the archive, e.g., a comment mentioning the EasyBuild version that was used, the git commit ID if it can be determined, a list of build statistics specified in the `build_stats` easyconfig parameter, etc.

C. *Automatic dependency resolution*

One of the key features is automatic dependency resolution, which is indispensable for a software installation framework as mentioned earlier. This feature is referred to as the EasyBuild *robot*.

Recall that the easyconfig `dependencies` are checked by verifying they are available through environment modules – generated by EasyBuild. If the robot feature is turned on, EasyBuild automatically tries to locate easyconfig files for missing dependencies, including the compiler toolchain. These easyconfigs are then installed in a hierarchical order, allowing to bootstrap a complete installation. Paths where the robot should look for the easyconfig files can be provided on the command line as arguments to the `--robot` option. For any supported software package, a dependency graph can be generated using the `--dep-graph` option.

Automatic dependency resolution is a very useful and powerful tool when installing large sets of software packages at once, e.g. when setting up a new system, or when installing a software package with a large and/or complex dependency graph.

D. Support for interactive installers

Some scientific packages use an interactive installation procedure. Moreover, there often is no alternative that allows providing a completed configuration file to allow the installation to proceed autonomously. This is clearly an undesirable feature for a framework that focuses on automating software installation and maintenance.

EasyBuild addresses this issue through the `run_cmd_qa` function which supports a Q&A mechanism for handling interactive installers. It suffices to provide a Python dictionary that maps regular expression patterns for questions to the correct answer strings or actions – for example, hitting return to continue – for the interactive installer to run to completion autonomously. A concrete example is given in Section IV-A for the WRF software package.

E. Installing multiple software packages in parallel

When multiple software packages are being installed at the same time, either because they belong to a set of (independent) software packages or because they are independent parts of a dependency graph, it is often desirable to be able to run the installation procedures in parallel, provided that the required resources are available and that the dependencies between the software packages do not prevent it.

Through the `parallel_build` module in the *tools* package, EasyBuild provides support for running installation procedures in parallel. This is done by submitting jobs to a resource manager like Torque/PBS, while setting dependencies between jobs if necessary, to make sure that installations are performed in the correct order. To enable this feature, the command line option `--job` should be used.

F. Regression and unit testing

Unit tests for EasyBuild are available in the *test* package. They can be used to check whether the functionality provided by the framework and covered by these tests was not broken during development. Running the suite of EasyBuild unit tests is done using `python -m easybuild.test.suite`.

The unit tests are run by a Jenkins continuous integration server on a regular basis, i.e., on every commit to the `develop` branch of the EasyBuild git repository as well as before every release. The Jenkins instance is located at https://jenkins1.ugent.be/view/EasyBuild/.

Additionally, the entire set of example easyconfigs files that are provided as a part of EasyBuild is installed in a dedicated and pristine installation path by means of the regression test. This way, the installation procedures for all the supported software packages are tested, as well as their interoperability. This is important for both compiler toolchains and for software packages that have a non-empty set of dependencies. The results of these regression tests will also be available through the EasyBuild Jenkins project.

IV. Use case: installing WRF

To illustrate the use of EasyBuild, we discuss the installation of the WRF weather modeling software package. WRF is a particularly interesting use case, because its installation procedure is by no means standard. However, actually installing WRF using its easyconfig file is trivial. It suffices to run the following command:

```
eb WRF-3.4.eb --robot
```

Using the `--robot` option without an argument requires that easyconfig files for any WRF dependencies (see Figure 5) are provided by EasyBuild. In this use case, we assume that EasyBuild already provides these easyconfig files and has the support required for installing these dependencies. Although additional command line parameters can be specified, for example, `--debug` to obtain detailed debug information as the installation procedure progresses, or `--job` to perform the installation through PBS jobs, they are not required to successfully build WRF – or any other software package for that matter.

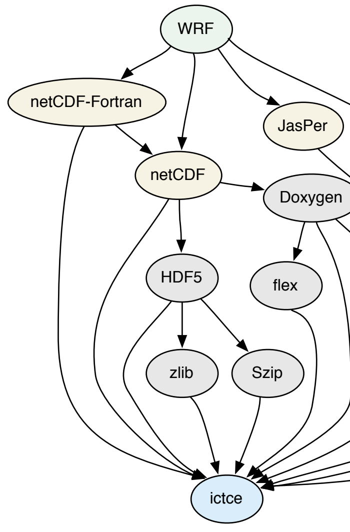

Fig. 5: Dependency graph for the WRF software package. The arrows indicate a depends-on relationship.

After successfully completing the installation procedure, EasyBuild creates an environment module named (by default) `WRF/3.4-ictce-4.0.6`. This name is derived from the `name` and `toolchain` parameters in the easyconfig file.

We now discuss how EasyBuild can be augmented to support installing this software package.

The WRF source provides a configuration script called `configure`. This script, however, is an interactive script that generates a configuration file `configure.wrf` rather than a classical GNU configure script. On top of this, said generated configuration file should be edited to gain full control of the build parameters, e.g., compiler commands, optimisation options, etc. Compilation is done using the `compile` script, which wraps around the `make` command to perform some extra custom tasks.

Note that the WRF installation procedure does not feature an actual installation step, the sources must be unpacked in the installation directory. Thus, the `compile` script must be run there as well[7].

Besides this, WRF has a rather complex dependency graph, see Figure 5. Assuming that EasyBuild provides the support for all dependencies in this graph, the automatic dependency resolution allows to install WRF using a single simple command, as shown above.

In the following two sections we discuss how to add support for the WRF install procedure, and how to subsequently use it to obtain a custom installation of WRF.

[7]The WRF easyblock could copy the binaries, libraries and other fields required at runtime to the installation directory after running the `compile` script, but this is not implemented at this point. We do welcome contributions to EasyBuild.

A. Implementing the WRF easyblock

To add support for installing WRF to EasyBuild, a Python module named `wrf.py` should be created in the *easyblocks* package. This module provides a class named `EB_WRF` that inherits from the generic base easyblock (`EasyBlock`).

The `EB_WRF` class implements the WRF installation procedure, as shown in Listing 1. We only show the relevant lines in the listed code, otherwise the easyblock does not fit on a single page[8]. The complete code is available from the EasyBuild GitHub repository[9]. Note that this example is rather involved due to the complexity of the WRF installation procedure. Typically, a new easyblock should be less complex and contain far less Python code. Nonetheless, this example shows that even non-standard complex install procedures can easily be captured in a custom easyblock.

At the top of the Python module `wrf.py`, we import the required Python modules and EasyBuild functionality. Next to useful functions such as `run_cmd_qa`, this also comprises the generic `EasyBlock` which `EB_WRF` sub-classes.

The `__init__` constructor is customised to specify that WRF should be built in the installation directory. This only needs to be stated here; EasyBuild will use this information during the installation process.

Through the static `extra_options` method, the `buildtype` custom easyconfig parameter for WRF is added. It is marked as mandatory, and will be enforced as such.

The configuration step is implemented in the `configure_step` method, and consists of three parts. First, several required preparations are made (lines 23–40):

- The environment variables for the dependencies are set with the `env.set` function. For netCDF and netCDF-Fortran, this is done with the `set_netcdf_env_vars` function provided by the netCDF easyblock.
- The `WRFIO_NCD_LARGE_FILE_SUPPORT` environment variable is set to indicate that WRF should be built with support for large files.
- The `Config_new.pl` Perl script – which is the actual interactive configure script – is patched using the `patch_perl_script_autoflush` function provided by EasyBuild to enable auto-flushing. This is required to run it correctly in an autonomous way.
- The `buildtype` easyconfig parameter is checked, to make sure a valid value was supplied.

Now that the environment is properly set up, the configure script is executed (see lines 43–51), using the `run_cmd_qa` function provided by EasyBuild. A number of question patterns and matching answers are supplied to allow the interactive configure script to autonomously run to completion. Should the script crash for some reason or if an answer cannot be supplied to a question being asked, an `EasyBuildError` will be thrown, halting the installation procedure.

Note that for one of the questions posed by the configure script, the answer is being determined dynamically (see line 49). The desired answer for selecting the Intel compiler toolkit is determined by EasyBuild from the matching option line.

[8]Omissions include docstrings and comments, the required custom implementations of the `sanity_check_step`, `make_module_req_guess` and `make_module_extra` functions, the definition of the `test` function that runs the test suite included with WRF, support for building WRF with others compilers, e.g., GCC, custom easyconfig parameters that are irrelevant to this discussion, etc.

[9]The easyblock detailing the WRF install procedure can be found at https://github.com/hpcugent/easybuild-easyblocks/blob/develop/easybuild/easyblocks/w/wrf.py

Listing 2: Example easyconfig file for WRF (`WRF-3.4.eb`)

```
name = 'WRF'
version = '3.4'

homepage = 'http://www.wrf-model.org'
description = 'Weather Research and Forecasting'

tcver = '3.2.2.u3'
toolchain = {'name': 'ictce','version': tcver}
toolchainopts = {'opt': False, 'optarch': False}

sources = ['%sV%s.TAR.gz' % (name, version)]

patches = [
    'WRF_parallel_build_fix.patch',
    'WRF-3.4_known_problems.patch',
    'WRF_tests_limit-runtimes.patch',
    'WRF_netCDF-Fortran_separate_path.patch']

dependencies = [('JasPer', '1.900.1'),
                ('netCDF', '4.2'),
                ('netCDF-Fortran', '4.2')]

buildtype = 'dmpar'
```

Finally, the generated configuration file – `configure.wrf` – is patched using the `fileinput` and `re` Python modules, to ensure that the correct compiler settings are being used (see lines 54–64).

The appropriate toolchain module has set the environment variables such as `CC`, `F90`, `MPICC`, etc. These can be used in an easyblock without having to know the exact compiler commands. Similarly, the default optimisation settings in the WRF configure file can also be overridden to use the settings provided by EasyBuild, but due to space limitations this has been omitted in Listing 1.

The next step is to perform the compilation procedure using the `compile` wrapper script. This is implemented in the `build_step` method. The `-j N` option is passed to `compile`, enabling parallel compilation. The actual value is obtained through the `parallel` easyconfig parameter. The `compile` script is run three times: once to build WRF and twice to build two test cases. This ensures that both the `ideal.exe` and `real.exe` tools are also compiled.

As mentioned above, the WRF installation procedure does not feature an actual installation step (the build is performed in the installation directory), so the `install_step` method is defined as empty using the `pass` statement.

B. Installing WRF using an easyconfig file

The easyconfig file shown in Listing 2 specifies the version of the WRF software package that should be built along with the build parameters. Some optional easyconfig parameters are used here to steer the installation procedure. The `toolchainopts` parameter indicates that aggressive compiler optimisations should be avoided. The `patches` parameter defines the list of patches that need to be applied in order to correctly build WRF with the Intel compiler toolchain `ictce`. The custom easyconfig parameter `buildtype` is also defined, as required by the WRF easyblock. This easyconfig file can then be used to install the WRF software package as was shown above.

This example illustrated how EasyBuild can be used in practice. Usually, after a one-time effort implementing the installation procedure for a particular software package as an easyblock, custom installations of that software package can

Listing 1: Shortened version of the easyblock implementation for WRF (`wrf.py`).

```python
import fileinput, os, re, sys

import easybuild.tools.environment as env
from easybuild.easyblocks.netcdf import set_netcdf_env_vars
from easybuild.framework.easyblock import EasyBlock
from easybuild.framework.easyconfig import MANDATORY
from easybuild.tools.filetools import patch_perl_script_autoflush, run_cmd, run_cmd_qa
from easybuild.tools.modules import get_software_root

class EB_WRF(EasyBlock):

  def __init__(self, *args, **kwargs):
    super(EB_WRF, self).__init__(*args, **kwargs)
    self.build_in_installdir = True

  @staticmethod
  def extra_options():
    extra_vars = [('buildtype', [None, "Type of build (e.g., dmpar, dm+sm).", MANDATORY])]
    return EasyBlock.extra_options(extra_vars)

  def configure_step(self):
    # prepare to configure
    set_netcdf_env_vars(self.log)

    jasper = get_software_root('JasPer')
    jasperlibdir = os.path.join(jasper, "lib")
    if jasper:
      env.setvar('JASPERINC', os.path.join(jasper, "include"))
      env.setvar('JASPERLIB', jasperlibdir)

    env.setvar('WRFIO_NCD_LARGE_FILE_SUPPORT', '1')

    patch_perl_script_autoflush(os.path.join("arch", "Config_new.pl"))

    known_build_types = ['serial', 'smpar', 'dmpar', 'dm+sm']
    self.parallel_build_types = ["dmpar", "smpar", "dm+sm"]
    bt = self.cfg['buildtype']

    if not bt in known_build_types:
      self.log.error("Unknown build type: '%s' (supported: %s)" % (bt, known_build_types))

    # run configure script
    bt_option = "Linux x86_64 i486 i586 i686, ifort compiler with icc"
    bt_question = "\s*(?P<nr>[0-9]+).\s*%s\s*\(%s\)" % (bt_option, bt)

    cmd = "./configure"
    qa = {"(1=basic, 2=preset moves, 3=vortex following) [default 1]:": "1",
          "(0=no nesting, 1=basic, 2=preset moves, 3=vortex following) [default 0]:": "0"}
    std_qa = {r"%s.*\n(.*\n)*Enter selection\s*\[[0-9]+-[0-9]+\]\s*:" % bt_question: "%(nr)s"}

    run_cmd_qa(cmd, qa, no_qa=[], std_qa=std_qa, log_all=True, simple=True)

    # patch configure.wrf
    cfgfile = 'configure.wrf'

    comps = {
             'SCC': os.getenv('CC'), 'SFC': os.getenv('F90'),
             'CCOMP': os.getenv('CC'), 'DM_FC': os.getenv('MPIF90'),
             'DM_CC': "%s -DMPI2_SUPPORT" % os.getenv('MPICC'),
            }
    for line in fileinput.input(cfgfile, inplace=1, backup='.orig.comps'):
      for (k, v) in comps.items():
        line = re.sub(r"^(%s\s*=\s*).*$" % k, r"\1 %s" % v, line)
      sys.stdout.write(line)

  def build_step(self):
    # build WRF using the compile script
    par = self.cfg['parallel']
    cmd = "./compile -j %d wrf" % par
    run_cmd(cmd, log_all=True, simple=True, log_output=True)

    # build two test cases to produce ideal.exe and real.exe
    for test in ["em_real", "em_b_wave"]:
      cmd = "./compile -j %d %s" % (par, test)
      run_cmd(cmd, log_all=True, simple=True, log_output=True)

  def install_step(self):
    pass
```

then be performed relatively easily. Additionally, easyblocks that are sufficiently general can be re-used for supporting other software packages with similar install procedures. Sharing implementations of install procedures is quite simple as well: usually it suffices to provide the module that implements the custom easyblock.

V. Related Work

There are various other frameworks or tools that have similar goals as EasyBuild. However, none of them have full support for the required features discussed in Section I.

The *Ports* frameworks [5] provide so-called Port files, which are basically `Makefiles` together with patches to easily build software packages. This framework is used by FreeBSD, NetBSD, OpenBSD, as well as re-implemented by Gentoo (Portage), Arch (Arch Build System), CRUX and for OS X by MacPorts [9]. These Ports frameworks all share the goal of easily sharing known installation procedures. However, they are all very operating system dependent, have no support for installing different software builds alongside each other, and most of them only have support for one specific compiler, i.e., the installed system compiler.

Compile [10] is the compilation system used in GoboLinux. It has support for over 10,000 so-called recipes. Compile only supports GCC, and has a built-in system that resembles environment modules. The recipes are bash scripts, and thus a lot harder to maintain than a Python framework.

The *Red Hat software collections* [12] are lists of software that can be installed alongside each other by extending the `spec` files with some extra macros. Scriptlets are used to achieve similar features to environment modules. Because `spec` files are being used, this system is again very operating system dependent, as less flexible compared to EasyBuild.

Rocks [1] provides bash scripts and `Makefiles` named *Rolls* to build software on HPC clusters, e.g. the UCSD Triton cluster [15]. The same concerns apply as with the other projects, mainly w.r.t. maintenance and flexibility.

Homebrew [6] is a package manager for OS X, that installs packages in dedicated directories. Although it does not readily provide environment modules, this should be easy to support. Homebrew is very much tied to OS X, and thus is a lot less flexible than EasyBuild.

Slawinska et al. present a so-called *system-call virtual machine (SCVM)* based approach to make software installations more portable [14]. This approach involves intercepting system calls through `strace` and adding directives to build scripts in a custom language. For now this framework is limited to a proof-of-concept, and seems to be significantly harder to maintain.

Finally, there are some frequently used installation frameworks for Python packages, such as *Buildout* [16] and *Setuptools* [3]. However, these tools focus on installing Python packages and applications, whereas we require the ability to install any application, irrespective of the language it is written in. *SCons* [8] on the other hand does allow installing general software packages. Unlike EasyBuild, *SCons* essentially requires (re-)writing the actual makefiles in Python, rather than allow using the existing makefiles. As such, EasyBuild is more general, and requires less work as it is able to re-use the existing installation scripts offered by the target software packages.

VI. Conclusion

This paper presented the EasyBuild Python framework for installing software packages. We outlined its major features, discussed the design of the framework, and showed how it can be used for installing (scientific) software packages so end-users can easily access them for their experiments. Through its modular design, EasyBuild allows adding custom installation procedures for new software packages or support for new compiler toolchains with minimal effort. Software installations are performed in isolated directories. This effectively supports installing various versions of a software package side-by-side. It also minimises the potential effect of operating system updates on the installed end-user software. Sharing implementations of software installation procedures, the so-called easyblocks, is particularly easy.

EasyBuild is open source software; we aim to make it a community effort, thereby reducing the time required by a user support team for installing the diverse set of software HPC sites may require. We hope this framework is useful for the community at large, including end-users. Merging support for over 250 software packages from the in-house to the public version is ongoing.

Acknowledgments

The authors would like to thank Dries Verdegem, Pieter De Baets, Toon Willems, Wouter Depypere, Luis Fernando Muñoz Mejías, Fotis Georgatos and Cédric Laczny for their valuable discussions and contributions.

Kenneth Hoste, Jens Timmerman, Andy Georges and Stijn De Weirdt are supported by the Unit ICT Infrastructure of Ghent University and the Flemish Supercomputer Centre (VSC).

This work was carried out using the STEVIN Supercomputer Infrastructure at Ghent University, funded by Ghent University, the Flemish Supercomputer Centre (VSC), the Hercules Foundation and the Flemish Government – department EWI.

References

[1] Rock Clusters. Rocks - Open Soure Toolkit for Real and Virtual Clusters. http://www.rocksclusters.org/roll-documentation/base/5.5/, 2012.
[2] P. F. Dubois, T. Epperly, G. Kumfert. Why Johnny Can't Build Computing in Science and Engineering, Vol. 5(5), pp. 83–88, 2003.
[3] P.J. Eby Setuptools and easy_install. http://pypi.python.org/pypi/setuptools.
[4] J.L. Furlani, P.W. Osel. Abstract yourself with modules. In *LISA*, 1996, pp. 193–204.
[5] The FreeBSD Project, FreeBSD Ports. http://www.freebsd.org/ports, 2000.
[6] M. Howell. Homebrew, the missing package manager for OS X. http://mxcl.github.com/homebrew/, 2012.
[7] HPC UGent. EasyBuild documentation. http://github.com/hpcugent/easybuild/wiki, 2012.
[8] S. Knight SCons User Guide. http://www.scons.org/doc/production/PDF/scons-user.pdf, 2010.
[9] MacPorts. The MacPorts Project. http://www.freshports.org, 2002.
[10] H. Muhammad. The ideas behind Compile. http://www.gobolinux.org/index.php?page=doc/articles/compile, 2003.
[11] Python Software Foundation. Python Programming Language. http://www.python.org, 1990.
[12] Red Hat. Red Hat Developer Toolset 1.0 – Software Collections Guide. https://access.redhat.com/knowledge/docs/en-US/Red_Hat_Developer_Toolset/1/html/Software_Collections_Guide/index.html, 2012.
[13] M. Slawinska. Enhancing Portability of HPC Applications across High-end Computing Platforms. In *IPDPS*, 2007, pp. 1–8.
[14] M. Slawinska, J. Slawinski, V. Sunderam. A Practical, SCVM-based Approach to Enhance Portability and Adaptability of HPC Application Build Systems In *IMECS*, 2012.
[15] UCSD. Triton Resource - Build Your Own Cluster http://tritonresource.sdsc.edu/build_own.php, 2012.
[16] Zope Foundation Buildout. http://www.buildout.org, 2009.