ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THBPL05

THE ELT LINUX DEVELOPMENT ENVIRONMENT

F. Pellegrin[*], C. Rosenquist,
European Southern Observatory, Garching bei München, 85748, Germany

Abstract

The Extremely Large Telescope (ELT) [1] is a 39-metre ground-based telescope being built by ESO [2]. It will be the largest optical/near-infrared telescope in the world and first light is foreseen for 2024.

The overall ELT Linux development environment will be presented with an in-depth presentation of its core, the *waf* [3] build system, and the customizations that ESO is currently developing.

The ELT software development for telescopes and instruments poses many challenges to cover the different needs of such a complex system: a variety of technologies, Java, C/C++ and Python as programming languages, Qt5 [4] as the GUI toolkit, communication frameworks such as OPCUA [5], DDS [6] and ZeroMQ [7], the interaction with entities such as PLCs and real-time hardware, and users, in-house and not, looking at new usage patterns. All this optimized to be on time for the first light.

To meet these requirements, a set of tools was selected for the development toolkit. Its content ranges from an IDE, to compilers, interpreters, analysis and debugging tools for the various languages and operations. At the heart of the toolkit lies the modern build framework waf: a versatile tool written in Python selected due to its multiple language support and high performance.

ELT SOFTWARE NEEDS AND CHALLENGES

While choosing the software technologies that the new ESO telescope will use, many factors were taken into account, for example:

- Construction time and lifetime of the project; from the project start to the first light almost a decade will pass by and the operational time is estimated to be at least 30 years. Technology-wise these are very long timespans.
- Different scope; some parts of the new telescope require very fast, real-time, either computation hungry or low-level, operations while others are used for post-processing or data management where timing is not strict but high level abstraction may be required.
- Different developer base; the ELT project will be prepared with efforts of ESO engineers, external contractors and consortia of scientific institutes.

Given these factors a series of basic requirement are therefore set on the software technologies to be used:

- Must be maintainable in the long term and as accommodating as possible to new future developments and requests
- The software tools available must be able to cope with both low-level; real-time near hardware, and high-level; user facing or data abstraction situations depending on the scope of the software component being developed. Certainly it is difficult to find a single solution for all these so different needs. This may mean that a selection of complementary tools should be offered.
- The software environment must take into account a distributed and non-homogenous user base, working at different premises and with different knowledge bases and goals. Trying to simplify common use cases should be an important asset, while not limiting the possibilities for advanced users.

The current plan is to develop most of the software, with the exception of some PLC based development, on the Linux operating system. Where specifically needed the Linux real-time extensions [8] will be used and possibly specific lower level network throughput optimization libraries may be used to satisfy both timing and bandwidth requirements.

The main programming languages selected for the project are C/C++ chosen for its high performance and low level capabilities, Java for its higher level of abstraction and Python as a versatile scripting language that offers high productivity. Language standards requirements had been set to relatively new standards: C++11, Java 8 and Python 3.x are the current baseline.

The graphical user interface toolkit selected is Qt5, giving the possibility to build advanced, portable and performant interfaces in both C++ and Python.

Network communication will be IPv4 based and on a higher level the aim is to support multiple application level protocols to increase the expandability and interoperability of the project with different communication patterns and features that may be needed by a specific project feature implementation. While a general abstraction layer is planned to be developed to ease the integration, it is planned that OPCUA, DDS, ZeroMQ as well as an internal UDP based protocol will all be part of the project and extensively used.

A big challenge for the development environment is to try to make all these different technologies work together and in a harmonious way. While it may be easier to find optimized domain specific build systems, visual

[*] fpellegr@eso.org

development environment and other tools for each language, it is much harder to try to find one that fits all and therefore optimizes the overall ELT development experience as a whole.

THE DEVELOPMENT ENVIRONMENT

The selected base distribution for the Linux development environment is CentOS [9]. CentOS is a community maintained enterprise distribution based on the Red Hat Enterprise Linux [10]. The usage of an enterprise focused distribution guarantees solidly tested packages, long term support and updates. The drawback of this choice is that the latest version of technologies or packages may arrive relatively late or have to be introduced separately, but these problems can be addressed with usage of technologies like containers that permit creation of customized isolated environments that works on top of a stable base.

Compatibility with RPM [11] packaging format and a vast userbase provide on the other side accessibility to many additional packages or features, for example the CERN [12] maintained real-time kernel extensions just to mention one of the most important ones. Internal know-how, due to use of same or similar distribution also for other ESO projects, is also an important asset.

Relatively recent standards chosen for the programming languages calls for relatively recent tools. On the C/C++ side the availability of the Red Hat Developer Toolset [13] via the Software Collections [14] provides a complete and updated toolset of the GNU Compiler Collection [15] and other support tools, such as the GNU debugger [16], profiling tools such as Valgrind [17] and gcov [18] and system debugging tools such as strace [19] and SystemTap [20]. For unit test creation the googletest [21] framework was selected. Static analysis tools such for code checking and syntax checking are currently Cppcheck [22] and cpplint [23] but with great interest on the LLVM [24] collection and specifically Clang [25] compiler tools for code checking and formatting, as well as to the evolution of this complete compiler package.

While Python 3.x is now present in most of the recent distributions, even if coexisting with Python 2.7.x for compatibility with some system applications, to guarantee an updated and fully customized list of modules, ranging from communication to big data management and from scientific calculation to number crunching, the Anaconda Python distribution [26] was selected. The usage of such a distribution gives the possibility to prototype early and share code with the current ESO Very Large Telescope [27] software that adopted it recently as well. Unit testing preparation is natively supported using the unittest [28] module and users are allowed to write tests also using the doctest [29] format, both being then executed and managed by a test runner such as Nosetests [30]. Code style and error checking is also supported by the standard tool pylint [31].

Finally, for Java 8 support the OpenJDK [32] implementation, now mature and as widely adopted as the closed source counterparts, was selected. Unit testing preparation is supported with the additional free software framework TestNG [33] with the addition of Mockito [34]. Code style and error checking are done respectively using Checkstyle [35] and FindBugs [36].

Another important decision to try to maximize the productivity of the developers was which integrated development environment, IDE, to propose and try to fully support in the development environment. While each of the languages used may have some specific very optimized IDE, and it will still nevertheless come up to a matter of personal preferences. The best choice given the range of different languages, and as a consequence tools, was the Eclipse Foundation product, Eclipse IDE [37]. This IDE, with a couple of very well maintained plugins, supports all the languages and related tools described.

The common cross language tool for inline documentation was the Doxygen [38] package, supporting, in worse case with minimal additional filters, all the languages requested and producing standard HTML based online documentation.

As the GUI toolkit the Qt5 framework was chosen over a series of other candidates. The framework provides high quality and performant graphical widgets that can be used in control systems and other parts of the telescope. This choice of course narrows the UI programming to C++ and Python, as Java support is almost non-existent, and calls for usage of specific Qt tools to manage code generation and graphical resources used.

BUILD SYSTEMS

One big challenge given the three different languages and the user interface framework, plus of course other language independent resources such as configuration files or media files, was to try to find the best build system solution.

A first idea was based on having multiple build systems, specific per language, and then somehow orchestrate their execution. But this would make the environment quite chaotic, require knowledge and maintenance of different build scripts. Further it would preclude having a global knowledge of dependencies and therefore the possibility for full support for mixed language software modules, reliable incremental builds and full parallelization of the build tasks which was a requirement.

C/C++ projects are usually built with systems such as GNU Make [39], possibly automatically created by a system such as autotools [40] for the configuration stage, or CMake [41], with some exceptions on more recent systems such as Meson [42] or Bazel [43]. Qt5 based projects usually start from the Qt specific qmake [44] tool. The Java world is split with various solutions such as Ant [45], Maven [46] and Gradle [47]. Python has distutils [48] and setuptools [49]. Each of the tools listed is mostly focused for that specific language, making it difficult to apply to another language: so while building C files with Ant or Java files with CMake is technically possible, it is awkward and highly inefficient.

Other important requirements for the build system are:

- efficiency and parallelization,
- automatic dependencies management,

- off-tree builds,
- ease of integration with present and future external tools like code generators or test runners,
- possibility to define a multiple input to multiple output relations, an important problem with e.g. GNU Make et al.,
- presence of a build command concept to allow execution of independent build steps, e.g. documentation generation or code linting and
- built in logging and debugging support.

THE WAF BUILD SYSTEM

The waf framework is an open source project started in 2005 whose goal is to abstract specific language needs, managing everything as generic builds or support tasks, keeping focus on portability and speed of execution.

Specifically, waf already has extensive support for the three languages needed for the ELT, and in addition a long list of other ones, support for the Qt5 framework peculiarities, Eclipse IDE project generation, test unit running and many other interesting features.

All the configuration files are written in Python, not a custom domain specific language such as CMake, and therefore all build scripts, extensions and customizations can take advantage of having a full-fledged modern programming language, extended with waf build framework concepts, as the base. Being based on Python, with support from versions 2.5.x to modern 3.6.x, it can run on many different platforms that provide a Python interpreter, most notably Linux, Windows and MacOS.

Dependencies are automatically tracked internally in the project or supported easily from external sources using as input the information supplied by the standard pkg-config [50] configuration metainformation tool.

Build decisions are made by comparing hashes instead of the traditional timestamp method that e.g. GNU make uses. The hashes are created using inputs from input source files, the resulting build environment, the Python build functions and other variables. This means that it does not suffer from timestamp related issues like clock skew on NFS mounts and will not perform redundant build steps. For example, the modification of a simple comment in a C++ source file will recompile the source file but will not re-link any target as the object file remains the same.

While waf is a less known tool and has a smaller user base compared to other bigger names, it has still some very important projects using it, for example:

- Samba [51], the standard Windows interoperability suite for Linux
- RTEMS [52], an open-source real-time operating system
- NS3 [53], discrete event network simulator
- Ardour [54], multiplatform DAW suite
- NTPsec [55], a secure implementation of the NTP protocol

The waf build framework is also used by gaming companies: projects where a lot of custom actions, different languages and different pre-processing tools have to interact and be coordinated to create the final result.

The waf developers community is quite active and releases are done regularly.

To mitigate the risk associated with a smaller user base, the project can be forked in worst case and maintained by ESO as the code base is not very big[†].

[†] The core codebase of waf is roughly 5000 source lines of code (SLOC), which roughly compares to the features of GNU Make which has about 30000 SLOC. This does not include some features provided by waf where autotools is typically used instead.

WAF USAGE AND EXAMPLES

The waf framework is configured using build scripts, named *wscript*s, which are Python scripts in which classes and functions are used to interact with the underlying waf framework. Usually they can be more than one in a project, nested on various subprojects and can be executed recursively. Although it is important to note that waf build scripts are not handled by respawning another instance of *waf* but by executing the additional files and building up the whole internal dependency structure, the recursive Make problem [56] is not an issue.

All the phases of the project, from configuration, to build, to testing, to installation, to distribution, are defined in this file and any phase defined is then invoked on the command line as a waf command that effectively executes the operation.

Let's try to introduce waf with a couple of concrete examples for the practical ELT use cases.

```
VERSION='0.0.1'
APPNAME='cxx_test'

top = '.'
out = 'build'

def options(opt):
    opt.load('compiler_cxx')

def configure(conf):
    conf.load('compiler_cxx')
    conf.check(header_name='stdio.h', features='cxx')

def build(bld):
    bld.shlib(source='a.cpp', target='alib')
    bld.program(source='m.cpp', target='app', use='alib')
    bld.stlib(source='b.cpp', target='foo')
```

The simple mentioned example defines a C++ project which builds a shared library (shlib), a program using it (the *use* directive defines the dependency) and a static library (stlib). In the configuration step external dependencies for a C++ compiler and an additional header are defined.

With this small script and by invoking waf on the command line with the configure, build, install and dist commands we can effectively manage the whole application lifecycle: the configuration will find on the system the required dependencies of the project, the build will execute the build steps when needed, the install will copy the files to a predefined destination tree using standard Unix conventions for file types that can be otherwise overridden and the dist will create a customizable package that can be easily delivered. Of course each step can be greatly customized and additional steps can be easily added writing Python code.

```
fede@eelt /tmp/a $ waf configure
Setting top to                          : /tmp/a
Setting out to                          : /tmp/a/build
Checking for 'g++' (C++ compiler)       : /usr/bin/g++
Checking for header stdio.h             : yes
'configure' finished successfully (0.116s)

fede@eelt /tmp/a $ waf build
Waf: Entering directory `/tmp/a/build'
[1/6] Compiling a.cpp
[2/6] Compiling m.cpp
[3/6] Compiling b.cpp
[4/6] Linking build/libalib.so
[5/6] Linking build/libfoo.a
[6/6] Linking build/app
Waf: Leaving directory `/tmp/a/build'
'build' finished successfully (0.288s)
```

In a similar fashion also defining a Java based project is very easy, adding classic Java concepts to the recipe:

```
def configure(conf):
    conf.load('java')
    conf.check_java_class('java.io.FileOutputStream')
    conf.env.CLASSPATH_NNN = ['aaaa.jar']

def build(bld):
    bld(features   = 'javac jar javadoc',
        srcdir     = 'src/',
        outdir     = 'src',
        compat     = '1.6',
        sourcepath = ['src', 'sup'],
        classpath  = ['.', '..'],
        basedir    = 'src',
        destfile   = 'foo.jar',
        use        = 'NNN',
        javadoc_package = ['com.meow' ],
        javadoc_output  = 'javadoc',
    )
```

In this case the configuration phase defined that the Java compiler and tools will be searched, together with a check on the existence of a specific class and definition of a custom classpath. In the build phase directories to use are defined, destination archive to generate, usage of the custom classpath and how to create the documentation.

Going even higher level if we want to implement a Qt5 Python based application, with the support of automatic Qt5 UI, user interface definition, files and resources conversion, we would see something like the following example:

```
def options(opt):
    opt.load('python pyqt5')

def configure(conf):
    conf.load('python pyqt5')
    conf.check_python_version((3,5,0))

def build(bld):
    bld(features="py pyqt5",
        source="src/test.py src/gui.ui",
        install_path="${PREFIX}/play/",
        install_from="src/")
    bld(features="pyqt5", source="sampleRes.qrc")
```

In this example Python and the Qt5 bindings, both PyQt [57] and PySide [58] are supported and will be scanned at configure phase, are requested along with a minimal version of Python. In the build section a script and a Qt UI file are defined to be processed and installed in custom positions along with a Qt5 resource file, which will be transformed by the appropriate tool into a Python class whenever one of the referred resources will change in the source tree.

New commands can be added to the framework by just adding them as Python functions to the wscript:

```
def hello(ctx):
  print('hello world')
```

And since it is Python anything can be done depending on the user's needs.

Custom rules for target generation can be easily written:

```
def build(ctx):
  ctx(rule='tac ${SRC} > ${TGT}', source='foo.txt',
            target='bar.txt')
```

Or more generically an extension based rule:

```
TaskGen.declare_chain(
  name    = 'sample1',
  rule    = 'codegen ${SRC} ${TGT}',
  ext_in  = '.in',
  ext_out = ['.a1', '.a2'],
)
```

In this case, when registered, any file with the .in extension will be run through the *codegen* utility to generate both .a1 and .a2 files.

From a developer point of view, it is also important to notice that waf has a lot of debugging aides and tools built in, giving the possibility with the *zones* to display filtered information useful for bug hunting, for example the internal tasks generated, dependencies or commands executed.

THE WTOOLS EXTENSIONS

Although waf build scripts can be simple and readable as shown in the previous section, an important aspect is also to make build scripts easy to maintain, simple as possible for developers that are unfamiliar with waf and to allow new feature roll out without changing the build scripts or waf itself. The ESO waf extension *wtools* was created to permit this while still allowing developers to make full use of waf if needed.

The simplest use of wtools reduces the build scripts to a single line that specify the primary artefact and only those attributes where the defaults are unsuitable. Currently the following primary artefacts are supported:

- C/C++ program, shared and static library,
- Python program and package,
- Qt5 C++ or Python program and
- Java JAR packages.

For example, a C/C++ program may look like this:

```
from wtools.module import declare_cprogram
declare_cprogram(target=”foo”, use=”bar”)
```

By declaring the software module type, a C/C++ program in the example, the wtools extension will create not just the primary artefact but also an associated unit test, create the linting command, installation instructions and more. It does this by populating the wscript with definitions expected by waf using Python’s introspection capabilities. It can be principally thought of as a parameterized expansion macro that results in definitions for the different waf commands in the wscript.

By also levering strong conventions on how software modules are structured wtools knows where to find source files, headers, unit tests etc. using ant-like pattern matching. The result is something akin to the following for the build command:

```
def build(ctx):
  bld.auto_cprogram(target=”foo”, use=”bar”)
```

The *auto_cprogram* symbol is a method registered by wtools in waf which when invoked takes care of creating the standard artefacts as required, including the C/C++ program primary artefact and secondary artefacts like unit tests or Qt resources.

Not all use cases can be foreseen or supported in wtools so by using this method indirection allows the developer to provide their own implementation of the build command to customize the behaviour while still make use of the automatic artefact creation.

For example, if the same sources need to be used to build two flavours of a program using different compilation flags the developer can still use the wtools-registered method auto_cprogram to leverage the automatic artefact declarations while allow room for attribute customizations:

```
from wtools.module import declare_custom

def build(ctx):
  ctx.auto_cprogram(target=“targetA”, name=”targetA”,
    defines=”A=1”)
  ctx.auto_cprogram(target=“targetB”, name=”targetB”,
    defines=”B=1”)

declare_custom(provides=[“targetA”, “targetB”],
depends=[])
```

By declaring the module to be custom with *declare_custom*, the developer can still inform wtools about what the software module provides and depends on. Using this information wtools automatically finds unresolved dependencies in the project during the configuration phase and tries to resolve them on the system. For the moment dependency resolution with pkg-config is supported, but more will likely follow.

If necessary, the developer can still opt out of wtools completely and write their own wscripts. The disadvantage of this is that this wscript would then have to be maintained by hand to e.g. make use of new features or update to support newer versions of waf. These are otherwise things that can be implemented centrally in wtools and automatically rolled out to all other modules.

ADOPTION AND FUTURE WORK

A first version of the ELT Linux development environment described is already now in use by early adopters: developers, ESO personnel and external companies. The feedback from real usage is now a very important asset to help improve and extend it constantly.

Similarly, the development environment is also already in use to execute automatic tasks and quality reporting on the continuous integration system, based on the Jenkins [59] open source automation server, and used as a solid base for the ELT testing framework to execute in. All these components together will constitute the ECM, the ELT Control Model, a small scale but representative subset of the ELT telescope control system that will give the possibility to test and validate the software at ESO site in Europe before it is installed on the final setup.

The current feeling is that further requested improvements to the build framework can be, compared for example with the previous VLT experience based on GNU Make, added and managed in a much easier and flexible way, thanks to the versatility of the powerful Python language and the infrastructure offered by the waf package and wtools layer. The centralized management of features given by wtools makes it easier to roll out improvements without the need of modification on the user level. The debugging and study of malfunctions with the integrated facilities is notably easier than the bare-bone facilities provided by GNU Make.

Nevertheless, with the full-fledged adoption of the environment, when all the software activities will be on-going, we realize that a committed team has to be allocated to follow its evolution and, especially to help new users to adopt it, being a very different technology from the ones previously used. In this direction it is important to mention that appropriate documentation, planned internal presentations and a growing list of ready to run examples are top priorities.

One of the future important decisions currently being under investigation and prototyping, for which the ELT Linux development environment will have to be adapted, is to deal with the deployment of the prepared artefacts. This includes technologies related to containers and container orchestration, generation of independent packages that can be easily deployed and distributed avoiding dependency hell situations and solutions for process management and supervision.

REFERENCES

[1] Extremely Large Telescope, https://www.eso.org/public/teles-instr/elt
[2] European Southern Observatory, https://www.eso.org
[3] waf, https://waf.io
[4] Qt, https://www.qt.io
[5] OPC-UA, https://opcfoundation.org/about/opc-technologies/opc-ua
[6] DDS, http://portals.omg.org/dds
[7] ZeroMQ, http://zeromq.org
[8] Real-Time Linux, https://rt.wiki.kernel.org
[9] CentOS, https://www.centos.org
[10] Red Hat Enterprise Linux, https://www.redhat.com/it/technologies/linux-platforms/enterprise-linux
[11] RPM, http://rpm.org
[12] CERN, https://home.cern
[13] Red Hat Developer Toolset, https://developers.redhat.com/products/developertoolset/overview
[14] Software Collections, https://www.softwarecollections.org
[15] GNU Compiler Collection, https://gcc.gnu.org
[16] GNU Debugger, http://www.gnu.org/software/gdb
[17] valgrind, http://valgrind.org
[18] gcov, https://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html
[19] strace, https://strace.io
[20] SystemTap, https://sourceware.org/systemtap
[21] googletest, https://github.com/google/googletest
[22] Cppcheck, http://cppcheck.sourceforge.net
[23] cpplint, https://github.com/cpplint/cpplint
[24] LLVM, https://llvm.org
[25] Clang, https://clang.llvm.org
[26] Anaconda, https://docs.continuum.io/anaconda
[27] ESO Very Large Telescope , http://www.eso.org/public/teles-instr/paranal-observatory/vlt
[28] unittest, https://docs.python.org/3/library/unittest.html
[29] doctest, https://docs.python.org/3/library/doctest.html
[30] Nosetests, https://nose.readthedocs.io
[31] pylint, https://www.pylint.org
[32] OpenJDK, http://openjdk.java.net
[33] TestNG, http://testng.org
[34] Mockito, http://site.mockito.org
[35] Checkstyle, http://checkstyle.sourceforge.net
[36] FindBugs, http://findbugs.sourceforge.net
[37] Eclipse, https://eclipse.org
[38] Doxygen, http://www.stack.nl/~dimitri/doxygen
[39] GNU Make, https://www.gnu.org/software/make
[40] Autotools, http://www.gnu.org/software/automake
[41] CMake, https://cmake.org
[42] Meson, http://mesonbuild.com
[43] Bazel, https://bazel.build
[44] qmake, http://doc.qt.io/qt-5/qmake-manual.html
[45] Apache Ant, https://ant.apache.org
[46] Apache Maven, https://maven.apache.org
[47] Gradle, https://gradle.org
[48] distutils, https://docs.python.org/3/distutils
[49] setuptools, https://github.com/pypa/setuptools
[50] pkg-config, https://www.freedesktop.org/wiki/Software/pkg-config
[51] Samba, https://www.samba.org
[52] RTEMS, https://www.rtems.org
[53] NS3, https://www.nsnam.org
[54] Ardour, https://ardour.org
[55] ntpsec, https://www.ntpsec.org
[56] P.A. Miller, "Recursive Make Considered Harmful'', *AUUGN Journal of AUUG Inc.*, vol. 19. No. 1, pp. 14-25, 1998.
[57] PyQt, https://riverbankcomputing.com/software/pyqt
[58] PySide, https://wiki.qt.io/PySide
[59] Jenkins, https://jenkins.io/