Large Scale Software Building with CMake in ATLAS

IOP Conf. Series: Journal of Physics: Conf. Series **898** (2017) 072010 doi:10.1088/1742-6596/898/7/072010

Large Scale Software Building with CMake in ATLAS

J Elmsheuser[1], A Krasznahorkay[2], E Obreshkov[3], A Undrus[1] on behalf of the ATLAS Collaboration

[1] Brookhaven National Laboratory, USA
[2] CERN, Switzerland
[3] University of Texas, Arlington, USA

E-mail: Attila.Krasznahorkay@cern.ch

Abstract. The offline software of the ATLAS experiment at the Large Hadron Collider (LHC) serves as the platform for detector data reconstruction, simulation and analysis. It is also used in the detector's trigger system to select LHC collision events during data taking. The ATLAS offline software consists of several million lines of C++ and Python code organized in a modular design of more than 2000 specialized packages. Because of different workflows, many stable numbered releases are in parallel production use. To accommodate specific workflow requests, software patches with modified libraries are distributed on top of existing software releases on a daily basis. The different ATLAS software applications also require a flexible build system that strongly supports unit and integration tests. Within the last year this build system was migrated to CMake.

A CMake configuration has been developed that allows one to easily set up and build the above mentioned software packages. This also makes it possible to develop and test new and modified packages on top of existing releases. The system also allows one to detect and execute partial rebuilds of the release based on single package changes. The build system makes use of CPack for building RPM packages out of the software releases, and CTest for running unit and integration tests.

We report on the migration and integration of the ATLAS software to CMake and show working examples of this large scale project in production.

1. Introduction

The ATLAS experiment's [1] offline software is responsible for reconstructing, simulating and analysing petabytes of data collected by the detector at the Large Hadron Collider. It is composed of multiple "projects" that are each built up out of many "packages".

As the software is built out of millions of lines of C++ and Python code, developers cannot build the entire thing when developing software in the collaboration. They are instead just building a selected list of packages against numbered releases, or nightly builds of the full software release, overriding just the libraries/executables/etc. provided by the package from the ones found in the "base release".

The software is mainly distributed to the users through CVMFS [2], but it is also important to be able to install the software on an offline computing node with as little effort as possible.

ATLAS, together with LHCb, used the CMT [3] build system for building its offline software until recently, but the experiment has been looking for a build system with much wider community support to replace CMT since a few years. Finally at the beginning of 2016 the

build of the software release meant to reconstruct and simulate data in 2017 and beyond has been switched over to be built by CMake [4].

2. Software organisation

ATLAS's software is broken up into $\mathcal{O}(2000)$ "packages", which can provide:

- Shared libraries to be used by other packages;
- Module libraries used by the software framework to load software components;
- Executables performing specialised tasks;
- Scripts, configuration files, etc. to be installed alongside the build targets of the package.

A package declares explicitly which other packages it requires for its own build, what software products it needs to build, and which already existing files it needs to install.

A software "project" is just a collection of packages, built in an order consistent with the dependency declarations in the package configurations. The projects themselves depend on each other similarly to how packages do. The packages in a certain project are allowed to depend only upon packages that are in a project that their project depends on. To demonstrate the highest project complexity used, the project configuration of the software used for reconstructing events in ATLAS is shown in Figure 1.

Figure 1. Projects and their relationships in the ATLAS offline software. "LCG" represents the externals provided by the LCG software release [5], the yellow boxes represent ATLAS online software built with their own configuration, and Gaudi [6] is built using its own CMake configuration.

3. The implementation

While the concept of packages is taken from the CMT build system, we wanted to keep the same structure with CMake as well. We used CMake's subdirectory concept to keep the configuration of the CMT packages separated with CMake. In this setup we have a master `CMakeLists.txt` file describing the project, which needs to:

- Set up some basic properties of the project, like its name and version;
- Find the base project(s) that it depends on;
- Find all subdirectories in the source tree that have a `CMakeLists.txt` file in them, and include them as subdirectories of the sofware project;
- Set up all variables necessary for using CTest and CPack with the project.

All of the common code used by the project and package configuration code is collected into two special packages in the source tree:

- `AtlasCMake` provides a large number of functions and macros all sharing the `atlas_` prefix, which help in performing common tasks in the project and package configuration files;
- `AtlasLCG` collects the code necessary for finding all non-ATLAS-specific externals for the project. It provides `FindFoo.cmake` CMake modules for finding these externals, and setting up dependencies on the RPM packages of these externals for the RPM package built from the ATLAS project.

3.1. Package configuration

The configuration of an average package may look something like the following.

```
# The name of the package:
atlas_subdir( ExamplePackage )

# The packages that this package depends on:
atlas_depends_on_subdirs(
   PUBLIC
   Control/AthenaKernel
   PRIVATE
   Control/CxxUtils )

# External packages needed for the build:
find_package( ROOT COMPONENTS Core Hist )
find_package( Boost COMPONENTS regex )

# Build a shared library:
atlas_add_library( ExampleLibrary ExamplePackage/*.h src/*.cxx
   PUBLIC_HEADERS ExamplePackage
   INCLUDE_DIRS ${ROOT_INCLUDE_DIRS}
   PRIVATE_INCLUDE_DIRS ${Boost_INCLUDE_DIRS}
   LINK_LIBRARIES ${ROOT_LIBRARIES} AthenaKernel
   PRIVATE_LINK_LIBRARIES ${Boost_LIBRARIES} CxxUtils }

# Build an application:
atlas_add_executable( ExampleApp util/ExampleApp.cxx
   LINK_LIBRARIES ExampleLibrary )

# Add tests for the package:
atlas_add_test( ExampleLibrary_test
   SOURCES test/ExampleLibrary_test.cxx
   LINK_LIBRARIES ExampleLibrary )
atlas_add_test( ExampleApp_test
   SCRIPT test/ExampleApp_test.sh )

# Install files from the package:
atlas_install_python_modules( python/*.py )
```

The `atlas_` prefixed functions take care of setting up the build and installation of components according to some common rules used in the build. They also take care of declaring the shared libraries to be exported, so that child projects can make use of them.

Every package declares two helper targets:

- `Package_PkgName` builds all library and executable targets in the package;
- `Package_PkgName_tests` builds all of the test executables declared in the package.

They are meant to help during development, when the user set up the build of many packages at the same time.

3.2. Project configuration

As discussed earlier, the project's main configuration is done in its main `CMakeLists.txt` file, which is constructed in the following way:

```
# The minimum required CMake version:
cmake_minimum_required( VERSION 3.2 FATAL_ERROR )

# Find this project's main dependency. To pick up its CMake configuration:
find_package( AtlasEvent )

# Set up the flags for CTest:
atlas_ctest_setup()

# Set up the project, and all of the packages in it:
atlas_project( AtlasReconstruction 21.0.5
   USE AtlasEvent 21.0.5
   FORTRAN )

# Set up the flags for CPack:
atlas_cpack_setup()
```

By finding the base project, the ATLAS CMake helper code from the base project is included into the current project, making it capable of using the `atlas_` prefixed functions.

The `atlas_project` function is responsible for doing the heavy lifting in the project's configuration. It:

- Sets up all general build flags for the project, including the output directories for all file types inside of the build directory;
- Looks up all of the packages in the source directory tree, and includes them with CMake's built-in `add_subdirectory` call;
- After having included all packages, it includes the shared libraries from its base project(s) as imported targets.

3.2.1. Usage of exported targets The build system has to allow us to patch a release by building a package on top of the release that the release itself has also built. Jobs executed in this environment would pick up libraries/executables/installed files from the patched package.

In order to do this every project exports all of its shared libraries during the project's installation, adding the name of the project as a prefix to the exported library's name. When calling `find_package` on the project, all of these imported targets become visible to the configuration already. Package developers are strongly advised against using such imported targets directly, though, since a given package should not have to know what other packages it

is compiled together with in the same project. They should only refer to shared libraries that they need for the build without any project name prefixes.

To make the build work, the `atlas_project` function calls `find_package` on the base project after having processed all packages, in the following way:

```
find_package( AtlasEvent 21.0.5 COMPONENTS INCLUDE QUIET )
```

What the base project does in this case is that it loops over all of its imported targets, and checks if a target with the same name is already declared in the current project. If it is, the project assumes that its library is being re-built in the current project, and lets that target be used by every other component in the project. If a target with that name is not defined, then it makes a copy of that imported target, removing the project name prefix from its name.

3.3. Runtime environment setup

The ATLAS offline software uses a large number of software products that are not developed by ATLAS, and are not provided by the operating system used for the compilation. Some of these externals we build as part of the offline software, but most of them we pick up for the build from custom locations.

Most of the used externals are picked up from software bundles provided by the LCG project. CMake finds those externals for the build using the `AtlasLCG` code described earlier.

Unfortunately CMake does not provide a built-in way for setting up a custom environment for the built/installed project, as its base assumption is that the build results should work without any special setup on the build host. This is however not true in our case. We have to ensure on our own that libraries and executables that the offline software used during the build are available in the runtime environment of the software.

This is done by the build generating simple shell scripts for setting up the necessary runtime environment. At the end of the project configuration we iterate over all external packages that were found during the configuration of the project, and generate a setup script that saves information on how to extend the environment for running our software.

In order to simplify the environment setup when building multiple projects, these setup scripts know how to find the setup scripts of the current project's base project. The user only needs to use the script from the highest project that he/she wants to use, and the script takes care of executing the setup from all of the base projects of that project as well.

3.4. Making the projects relocatable

As the projects must be installable in any location, we need to be careful with setting up both the CMake configuration files generated during the CMake installation, and the environment setup scripts generated by our private code. Neither of them can hold any absolute path names in order to make the projects relocatable.

As we have full control over the generation of the environment setup scripts, all that we do is to define a small number of environment variables that, when defined before using the setup script, direct the code to the correct directory to set up the externals from.

Making the CMake project files generated during the installation step is a bit more difficult. Since the file generated by CMake cannot be made relocatable out of the box, we use a custom script that processes the CMake generated code, and makes it relocatable. We make use of the following formalism in the CMake configuration:

```
install( SCRIPT ${CMAKE_BINARY_DIR}/atlas_export_sanitizer.cmake )
```

4. Performance

One of the goals of the build system migration was to speed up the build of the ATLAS offline software, and streamline the installation procedure of the software after it was built. CMake helps with this, as it can parallelize the build of independent components very efficiently.

The build machine monitoring system tells us that the build makes quite efficient use of the resources on the build machines. See Figure 2.

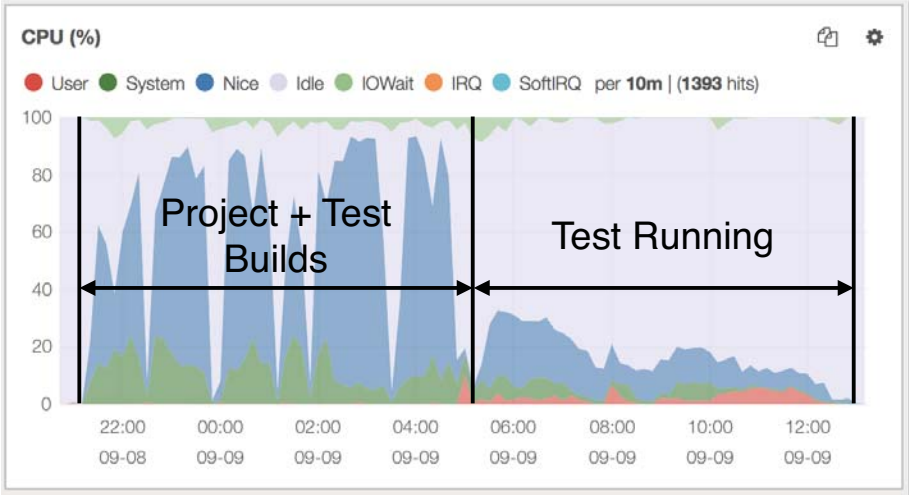

Figure 2. The CPU utilisation of the build machine while building the full ATLAS offline software.

In this project configuration, which we kept for a transition period between the build systems, the build times became just a little shorter. The single process execution steps necessary between the individual projects can be seen as clear downtimes during the build in Figure 2.

In order to simplify the offline software builds of ATLAS, we are now merging most of the projects into a single one called *Athena*, which project's code will be stored in a single Git repository. In such a setup, on a fast build machine, it is possible to build the full (single) project in $\mathcal{O}(3)$ hours, hitting the original performance goal of the build system migration.

5. Conclusions

In an effort to streamline its software build procedures, and use a system in common with many High Energy Physics Computing projects, ATLAS has switched to using CMake to describe the build procedures of its offline/simulation/trigger/analysis software. We are now in the final stages of validating the new, CMake built offline software of the experiment for the 2017 data taking.

The new build system, together with migrating the experiment's software to Git, makes the development procedures followed by the ATLAS software developers much more in common with practices used in the software development community on the whole.

The new system has so far met the requirements of the experiment both in features provided, and performance. And so it is expected to be kept for LHC's Run 3 and beyond.

References

[1] The ATLAS Collaboration 2008 *JINST* **3** S08003
[2] Blomer J *et al* 2015 The evolution of global scale filesystems for scientific software distribution *Computing in Science and Engineering* **17(6)** 61-71
[3] Arnault C 2001 Experiencing CMT in software production of large and complex projects *Proc. Int. Conf. on Computing in High Energy and Nuclear Physics* CHEP 2001 (Bejing, China)

[4] Martin K and Hoffman B 2007 An open source approach to developing software in a small organization *IEEE Software* **24** Number 1
[5] *http://cern.ch/lcgsoft*
[6] Clemencic M *et al* 2010 Recent developments in the LHCb software framework Gaudi *J. Phys.: Conf. Ser.* **219** 042006