A C/C++ BUILD SYSTEM BASED ON MAVEN FOR THE LHC CONTROLS SYSTEM

J. Nguyen Xuan, B. Copy, CERN, Geneva, Switzerland
M. Dönszelmann, Bogazici University, Istanbul, Turkey

Abstract

The CERN accelerator controls system, mainly written in Java and C/C++, consists nowadays of 50 projects and 150 active developers. The controls group has decided to unify the development process and standards (e.g. project layout) using Apache Maven and Sonatype Nexus. Maven is the de-facto build tool for Java, it deals with versioning and dependency management, whereas Nexus is a repository manager. C/C++ developers were struggling to keep their dependencies on other CERN projects, as no versioning was applied, the libraries have to be compiled and available for several platforms and architectures, and finally there was no dependency management mechanism. This results in very complex Makefiles which were difficult to maintain. Even if Maven is primarily designed for Java, a plugin (Maven NAR) adapts the build process for native programming languages for different operating systems and platforms. However C/C++ developers were not keen to abandon their current Makefiles. Hence our approach was to combine the best of the two worlds: NAR/Nexus and Makefiles. Maven NAR manages the dependencies, the versioning and creates a file with the linker and compiler options to include the dependencies. The Makefiles carry the build process to generate the binaries. Finally the resulting artifacts (binaries, header files, metadata) are versioned and stored in a central Nexus repository. Early experiments were conducted in the scope of the controls group's Testbed. Some existing projects have been successfully converted to this solution and some starting projects use this implementation.

MAVEN

Motivation for Maven

The BE/CO group is migrating from an in-house build tool to Maven [1], the industry standards for managing Java projects. This tool is widely used among the Java community by about 60% of developers. More than 100'000 open source projects are available on Maven Central repository, millions of connections happen per year. In addition, Maven can be used seamlessly with many tools for testing, source management, issue tracking, continuous integration, and so on. Since 2003, the BE/CO Java team have been putting a lot of efforts into software quality and reuse of resources in cooperation with other departments [2].

As the C/C++ developers want to improve quality and testing among their projects, the opportunity of unifying the build tools has been taken so they could profit from the same infrastructure, tools and experience from the Java development teams.

Maven Basics

Maven's key features are dependency management and versioning. It tends to enforce standards by providing a uniform build cycle with well defined succinct steps. In short, Maven takes the sources code, resolves and downloads the needed dependencies, builds the files and publishes the output to a repository. The output is called an artifact; it can be a jar, zip, war, etc...

Maven is plugin based and therefore can be easily extended with plugins. The latter can be provided by the Maven core developers, contributors, or one can write its own one. Plugins cover a very wide range of functionalities and have various purposes, such as generating docs, downloading specific artifacts, etc...

As shown in Fig. 1, a project is represented by a XML file called pom.xml (Project Object Model). Its main two parts are project information for the current project and dependencies information.

```
<project>
    <groupId>cern.testbed</groupId>
    <artifactId>testbed</artifactId>
    <version>1.0.0</version>
    <description>...</description>
    <url>http://...</url>

    <dependencies>
        <dependency>
            <groupId>cern.japc</groupId>
            <artifactId>japc-ext-cmwrda</artifactId>
            <version>[2.0.0,3.0.0)</version>
        </dependency>
        <dependency>
            <groupId>cern.japc</groupId>
            <artifactId>japc</artifactId>
            <version>2.11.0</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.8.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Figure 1: Simple pom.xml example.

THE NAR PLUGIN

Motivation, JNI and C Libraries

Maven handles the build steps for Java programs very well. Since Java allows the coupling to C and C++ programs it seems logical to include the step of compilation and linking of JNI (Java Native Interface) [3] modules as part of the Maven build steps. To extend Maven to handle native (C, C++, objective-C, etc...)

languages the NAR (Native ARchive) plugin [4] was written. How this plugin handles the compilation of JNI and other native modules for different platforms as well as their distribution and their usage of Maven's dependency mechanism is explained below. The NAR plugin consists of multiple sub plugins each of which takes part in the NAR lifecycle.

NAR Lifecycle

As mentioned earlier, to build any product Maven runs through a sequence of build steps. These steps are defined as a build lifecycle and directly associated to a packaging definition. By default Maven comes with packagings/lifecycles for jar, war, ear and some more. To enhance the standard build steps in the jar lifecycle with native compilation and linking, extra steps (in bold) where defined in the NAR lifecycle. The packaging for this lifecycle is NAR and a simplified version is given below:

- nar-download
- nar-unpack
- compile, nar-javah, nar-compile
- nar-testCompile
- test, nar-test
- nar-package, jar
- nar-integration-test
- install
- deploy

Native Sources and Headers

Maven assumes standardization for its plugins. The sources and header files for native parts of the code need to be stored in predefined places, which can be redefined if necessary. Header files and c files are not stored in the same location to make it easier to distribute the headers without the sources. These locations are used by the nar-compile plugin but also by the nar-javah plugin which runs the javah compiler. All generated output is stored, as usual in Maven, in subdirectories of the target directory.

AOL and Properties

To distinguish different platforms, operating systems and compiler/linkers the NAR plugin uses an AOL (Architecture Operating system Linker) qualifier. This qualifier looks like "i386-Linux-gcc" on a Linux i386 platform with gcc, but could be further extended in the form i386-Linux63-gcc4.1. The qualifier is used to handle different distributions as well as for selection of options for different compilers. Options and other flags are specified in an aol.properties file which sits next to the pom.xml file. A property file makes more sense than trying to put everything into profiles in the pom.xml as the number of platform/compiler combinations can be fairly large. Properties are stored in the aol.properties file for specific AOLs in dotted notation, for instance:

x86.Windows.msvc.cpp.compiler=msvc
i386.Linux.g++.c.options=-Wall -Wno-long-long

Compilation and Testing

For Java the maven-compiler-plugin handles the flags such as debug, optimization and others. We chose to use the cpptasks library [5] as this library unifies flags, options and linker strategies across different platforms and compilers/linkers. Based on the AOL a compiler, linker and their default flags are retrieved from the aol.properties file. These can be overridden by a project specific aol.properties file. The nar-compile plugin handles the native compilation/linking phase for which the cpptasks library was extended to use multiple cores in parallel to speed up compilation.

The nar-test plugin runs unit tests against the created JNI or standalone library. The test plugin makes sure that all libraries can be found.

The NAR Format and its Attached Artifacts

To package created libraries, executable and object files for re-use by others the NAR plugin uses its own format, the NAR file. A NAR file is no more than a standard jar file containing object files, executables, libraries and or header files. Files are stored in a directory structure that includes the AOL specifier but also reflects if libraries are static or dynamic. In its unpacked form the NAR compiler plugin is able to pick up header files and refer to libraries. This is important if some other package depends on a NAR library.

Three artifacts are produced under normal conditions: a standard NAR containing all Java class files if there are any as part of the project, a *-noarch* NAR file containing all non-architecture (non-AOL) specific files, such as header files and a *-<aol>* NAR file containing AOL specific files such as libraries. The first NAR file contains a property file that refers to the other two, which is used by the nar-download plugin, see below. The two latter NAR files are attached artifacts to the first. As can be seen this set of NAR files is a native equivalent for Java's single jar file.

The NAR files are split up in *-noarch* and multiple *-<aol>* files to make generating them easier and to download only the ones one need for a developer on a particular platform.

Distribution, Install and Deploy

The NAR plugin relies on the standard maven-install and maven-deploy plugins to install and deploy the primary NAR artifact and its attached artifacts. Any mechanism of caching such as the use of Sonatype Nexus [6] works transparently. Any Maven repository server will just store NAR files and their attached artifacts as another type of packaging.

Dependencies on other NAR Libraries

The reason for creating NAR files is so one can make other projects depend on them. These projects need to be also of the NAR packaging type and can then declare NAR dependencies. Any dependency declaration will initiate a download of the primary NAR artifact by Maven

itself. This primary NAR artifact is stored in the local repository.

Downloading and Unpacking Locally

Once a primary artifact of a dependency is downloaded, it is inspected for the above mentioned property file to see what attached artifacts need to be downloaded. In the normal case a *-noarch* and a *-<aol>* artifact will be downloaded by the nar-download plugin and stored in the local repository. As these NAR files as such are no use to any compiler, they will be unpacked by the nar-unpack plugin in a subdirectory of the current projects target directory (the latest version of Maven supports concurrent access to the local repository directory, so unpacking can also be done there in the future, thereby sharing artifacts and gaining space). The unpacked NAR files reveal the header files and libraries of the dependency and can thus be used the nar-compile plugin. Include paths and library paths will be set up automatically.

Cross Talk with other Systems

Other systems exists to build native (and even Java) code. The NAR plugin tries to be open and to integrate with those systems. One can for instance fairly easily call "configure", "autoconf" and "automake" of the GNU build system [7], or just call "make" to build libraries the usual way and use the NAR files purely for distribution and dependencies, as is explained below.

EXTENDING THE NAR PLUGIN

At CERN

At CERN, some projects were successfully converted to Maven NAR, but it resulted in big XML configurations. Indeed to simply add a compilation flag, about fifteen XML lines are needed whereas only one line is required with Makefiles, thus C/C++ developers were not keen to abandon the flexibility of their current Makefiles. In addition to that, CERN projects rely on cross-compilation, which is not covered by Maven NAR out of the box.

Design

Therefore it was essential to modify the NAR plugin according to our needs. A hybrid solution has been favoured: separate the build tasks between Maven NAR and Makefiles. Maven NAR takes care of the dependency management and versioning and Makefiles are in charge of the compilation process. NAR lifecycle has been modified so the goal nar-compile calls a Makefile instead of calling a compiler. Then binaries are expected to be generated and to be published to a binary repository. In our organization we use Sonatype Nexus for Java and decided to reuse it for C/C++ projects.

Finally, in order to support our cross-compilation infrastructure used by our Makefiles, it was necessary to modify cpp-tasks to add our compilers.

Implementation

As shown in Fig. 2, several steps are needed to build a C/C++ project with the customized Maven NAR. The following paragraphs describe each step along with its detailed implementation.

Makefile Generation Phase

The usual nar-download and nar-unpack phases are run, but in addition after those NAR will generate a Makefile with compiler and linker options. This Makefile contains only the dependency information for a specific platform, therefore the chosen naming convention is *Makefile.dep.<aol>*.

Compilation Phase

Maven NAR simply execute the command “make MAVEN_BUILD=true”. By convention, a Makefile needs to be present next to the pom.xml and its default target has to compile the source code. It also needs to include the Makefile.dep previously generated and use the defined macros from it.

The output binaries have to be placed in specific folders. The agreed standard is to put the library in *build/lib/<aol>* and includes in *build/include*. In BE/CO, we tend to enforce platforms independent headers, but some teams required platform specific headers. In this case, these headers go in *build/include/<aol>* and the Makefile generator will add an extra –I flag accordingly.

Packaging Phase

Files will place in the right folders so nar-package can do its job. Thanks to the directory conventions, NAR knows where to pick up what.

Deployment Phase

In Maven terminology, deployment means publishing on a server to make it available to other developers. This phase has not been altered from the official NAR plugin.

Usage Example

At CERN, C/C++ developers are used to define a macro called CPU to define the target platform. Instead of typing the whole target platform, shortcuts such as *L865* or *ppc4* are used. The same shortcuts were kept when invoking Maven commands, but these shortcuts are expanded to the AOL standards as i386-SLC5-gpp.

BENEFITS AND APPLICATION

Benefits

Since the dependencies information is separated and automatically generated, the Makefiles are simplified, the developers do not need to be concerned about dependency management and versioning anymore, and ultimately they can keep their habits with their Makefiles.

The previous implementation using pure Makefiles remains compatible with Maven NAR. Makefiles are called with the flag *MAVEN_BUILD=true* from Maven

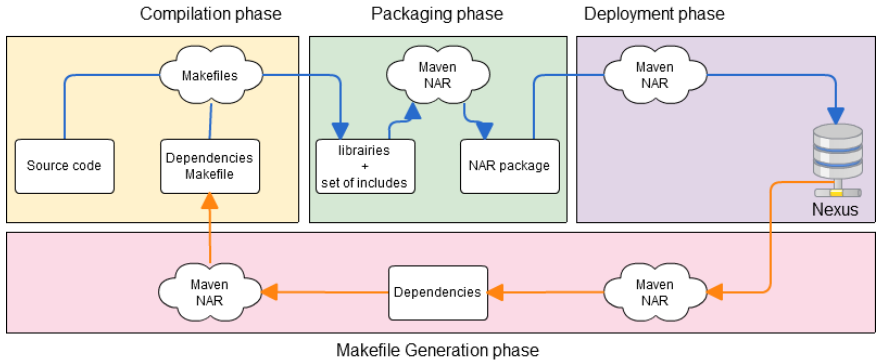

Figure 2: Extension phases.

NAR, thus it is known when a build is processed by Maven or pure Makefiles, some conditions can thus be added in order to include files accordingly.

Standards are also enforced, as explained earlier; Maven NAR needs to know where to pick up the different binaries (executables, libraries, headers). Directories need to have a well-defined structure with standardized names, as well as the files.

In addition of the build tool, the software development/release/deployment process and binary repository are also unified between programming languages. Java developers can easily switch to C/C++ and invoke the same commands through Maven to achieve the same goals. The build lifecycles are almost identical, first the code is compiled, then the unit tests are compiled and run, finally the product gets packaged.

Sharing C/C++ projects across and outside CERN becomes easy with Sonatype Nexus, CERN developers just needs to point to the binary repository and collaborators can proxy it and use the artifacts.

A CI (Continuous Integration) server takes care of building the projects with the bleeding edge source code from our SVN trunk. If a commit breaks a project, it will be immediately spotted.

Applications

The Maven NAR plugin is especially suitable to simplify cross-platform Java C/C++ build processes.

As an example, the build process of the CERN Data Interchange Protocol (DIP), a platform independent middleware protocol, was updated early 2010 to move from two entirely separate build systems relying on a series of manual steps and environment variables configuration into a unified Maven based build.

Because DIP is available both as a C++ and Java API, for three different platforms (Windows 32 bits, Linux SLC5 32 bits and SLC5 64 bits), a grand total of six builds, executed manually, had to be coordinated to assemble a complete distribution release.

Besides reducing the associated maintenance overhead, relying on a Maven build also helped to:

- Integrate more seamlessly the Java and C++ APIs (through JUnit testing)
- xDistribute its various components (Header files, static and dynamic libraries, auto generated documentation, associated development tools).

Since DIP is used by many projects at CERN, distributing it in NAR format also greatly simplified reuse for all CERN Maven based projects.

NEXT STEPS

Unit Tests

There are some improvements which can be done at the test phase. Instead of calling binaries which returns a code exit, we would like to integrate a testing framework to ease the writing of tests. Google C++ Testing Framework is a good candidate [8], to be used with Gcov [9] which offers code coverage. The goal is to generate full reports similar to the JUnit one, which will be displayed in our continuous integration server.

Merge Back with the Official Maven NAR

The CERN Maven NAR version embed functionalities specific to CERN, but most of the used methodology and chosen convention are standards in the C/C++ community. These changes need to be generalized and integrated back into the main version of Maven NAR in order to be able to profit from a community.

REFERENCES

[1] Apache Maven, http://maven.apache.org/
[2] B. Copy, M. Mettaelae, Agile Development and Dependency Management for Industrial Control Systems, WEPKS001, Proceedings of ICALEPCS'11, Grenoble, France.
[3] JNI, http://en.wikipedia.org/wiki/Java_Native_Interface
[4] Maven Nar plugin, http://duns.github.com/maven-nar-plugin/
[5] Cpptasks, http://ant-contrib.sourceforge.net/cpptasks/index.html
[6] Sonatype Nexus, http://nexus.sonatype.org/
[7] GNU Build System, http://en.wikipedia.org/wiki/GNU_build_system
[8] Google C++ Testing Framework, http://code.google.com/p/googletest
[9] Gcov, http://gcc.gnu.org/onlinedocs/gcc/Gcov.html