ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV041

MODERNISATION OF THE TOOLCHAIN AND CONTINUOUS INTEGRATION OF FRONT-END COMPUTER SOFTWARE AT CERN

P. Mantion†, S. Deghaye, L. Fiszer, F.Irannejad, J. Lauener, M. Voelkle
CERN, 1211 Geneva 23, Switzerland

Abstract

Building C++ software for low-level computers requires carefully tested frameworks and libraries. The major difficulties in building C++ software are to ensure that the artifacts are compatible with the target system's (OS, Application Binary Interface), and to ensure that transitive dependency libraries are compatible when linked together. Thus developers/maintainers must be provided with efficient tooling for friction-less workflows: standardisation of the project description and build, automatic CI, flexible development environment. The open-source community with services like Github and Gitlab have set high expectations with regards to developer user experience. This paper describes how we leveraged Conan and CMake to standardise the build of C++ projects, avoid the "dependency hell" and provide an easy way to distribute C++ packages. A CI system orchestrated by Jenkins and based on automatic job definition and in-source, versioned, configuration has been implemented. The developer experience is further enhanced by wrapping the common flows (compile, test, release) into a command line tool, which also helps transitioning from the legacy build system (legacy makefiles, SVN).

INTRODUCTION

Front-End Computer Software Development at CERN

CERN's Front-End Computers (FECs) are disk-less computer crates which host electronic cards connected on a back-plane. The software running on these computers typically uses a framework such as Front-End Software Architecture (FESA) [1,2] to interface with:

- The upper layer of the control system (settings management, timing, network (Remote Device Access (RDA3) protocol [3]), logging, post-mortem, machine protection, etc.)
- The cards' driver (C library) to drive the equipment.
- The OS (Linux, CentOS 7 with RT kernel) and framework (FESA) are designed to provide near real-time execution of the tasks through scheduling, thread priorities, and optimisation: this is a strong reason (amongst others), for using a performance-oriented language like C++ to build the software.

The production FECs run on CentOS 7, so the software must be built for that target, ensuring compatibility with the system's libraries (especially libc) and ABI (changes in ABI for *C++11 support*).

Software built using the FESA framework consists of an executable that is statically linked against the framework's libraries (versioned headers and *.a* archive files). The framework libraries themselves depend on a collection of middleware libraries provided by different teams across different groups.

The FESA framework is mostly used by equipment developers who are not full-time software engineers. As such, the framework providers aim to offer tooling that promotes best practices (e.g. source code versioning, releasing, tagging) and minimises human errors.

NEED FOR MODERNISATION

After almost two decades of building C++ software with Makefiles, a well-deserved modernisation was needed. A Continuous-Integration (CI) solution, based on a shared central Bamboo Server instance, was put in place almost ten years ago. A general move away from Bamboo to Jenkins or Gitlab CI for Controls software also needed to be taken into consideration. Additional objectives were to ensure that the new solution provides a better dependency management and ensure a smooth transition for our users.

Dependency and Toolchain Management

The correct execution of FEC software requires the binaries to be built using consistent versions of the dependencies. At the lowest level, this means that versions of the dependent libraries should be both binary and functionally compatible. However, a complex dependency graph means it is not easy to ensure, especially if no compilation/linking errors are raised at build time. Dependency management entails two aspects:

1. Knowing where to find/store artifacts (header / library files) from the build system.
2. Being able to check the consistency of dependencies and versions in the dependency graph.

Beyond ensuring production software is built correctly, strong dependency management is very useful to the developer. When working with local copies of a sub-set of a dependency graph, developers are one step away from so-called "dependency hell". Without automatic dependency management developers need to ensure that all libraries used locally are compatible, which entails editing makefiles and building/rebuilding lots of dependencies before having a working setup. Compilation time in such cases is not negligible. The process is also error prone, often requiring to re-build several dependencies after each correction. Figure 1 highlights the difficulty of modifying dependencies by hand by showing the complexity of the dependency graph for a representative example of FEC software.

† pierre.mantion@cern.ch

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV041

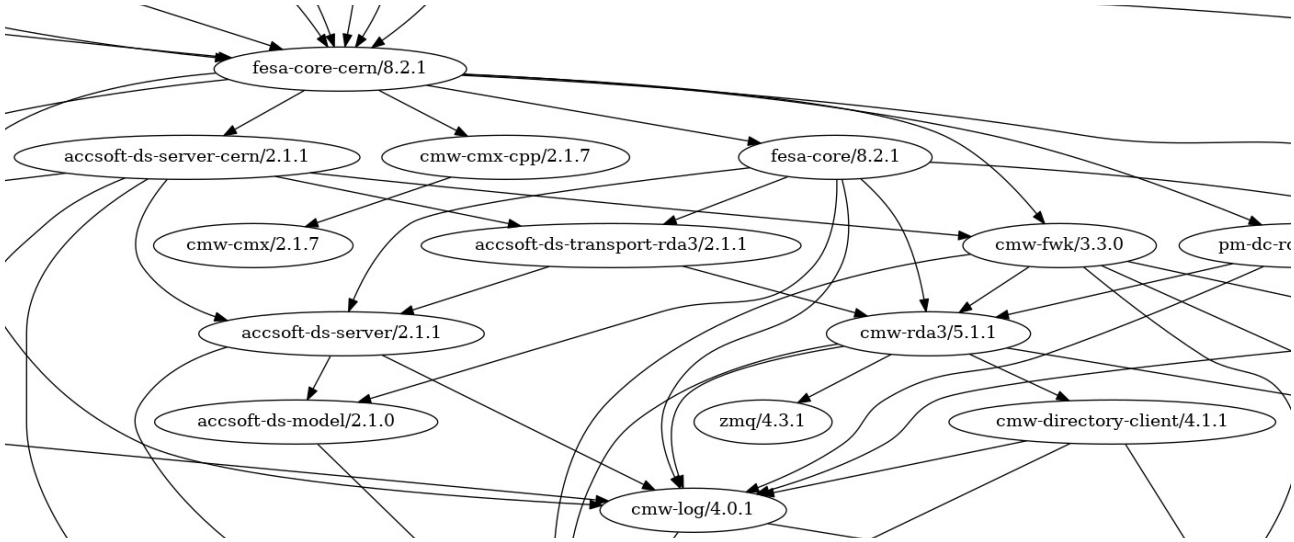

Figure 1: example of dependency graph for a FEC software (cropped)

Binary compatibility of the dependency libraries requires building all dependencies with compatible toolchains (compiler tools and settings, including libc and *C++*11 ABI [4]). The archive file (.a) artifacts of dependencies can be used by the software that is including them without many checks regarding compatibility.

New *C++* standards require more recent compilers, but the ABI must still be compatible with the runtime environment. The CentOS7 toolchain (gcc 4.8, "old" ABI [4]) is the current standard at CERN, but production software can also be released using a more recent compiler (gcc 7). Since CentOS 7 was released several years ago (2014) a transition to a new supported toolchain must be anticipated to ease equipment groups migration efforts and ensure smooth operations.

The rest of this paper will describe how Conan [5] and CMake [6] can be leveraged to meet the aforementioned challenges and objectives, and improve developer experience.

Introduction of CMake as a Modern Build Tool

While Make is still a standard way of building software, it is a fairly low-level tool. CMake brings deeper domain knowledge of C/C++ projects, is extremely popular and very well supported by IDEs, and has a permissive BSD license.

"CMake is an open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice." – https://cmake.org

As a standard tool with good public documentation, CMake is favourable over the legacy in-house makefile hierarchy for a multitude of reasons:

- Better public documentation means less user support for FESA framework providers.
- Better integration with modern and powerful IDEs such as CLion increases developer productivity and satisfaction.
- The experience gained working with CMake is more valuable for trainees and developers in terms of transferable skills beyond CERN.

A Command-Line Helper

Although aiming to increase the ease and quality of development, transitioning to new tools like CMake and Conan represents a challenge, especially for part-time developers, where adaptation and practice are required to master the new tools. To minimise this challenge a command line (CLI) tool, named Codeine is provided to simplify and standardise workflows e.g:

- Create a new project with a standard file layout.
- Wrap complex commands.
- Provide safe defaults (CentOS 7 compiler toolchain).
- enforce a consistent release flow (versioning, build, tag VCS, release location, structure of released artifacts).

As such, Codeine can help the transition to a new compiler toolchain by providing setting profiles and migration helpers.

Renovation of the CI / Testbed

Tooling that can ensure a correct and reproducible build by design does nothing to help avoid functional failures. Testing the software is critical and making the test feedback loop as short as possible increases development efficiency. Having convenient tooling to setup and run tests helps ensure that tests are written, ran and results checked. Continuous Integration (CI) has been a standard practice in software development for a long time, and supporting tooling for it is evolving.

CERN's front-end software is well tested by an extensive collection of tests built over years, automatically executed from an ageing Bamboo platform [7]. Besides obsolescence of this platform (licence renewal, end of life support from Atlassian [8]), new

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV041

paradigms such as storing test job configuration together with the versioned source code have appeared and make developers' lives easier, by automating the configuration of the test plans / jobs. This approach saves thousands of clicks in the Bamboo interface to configure the dozens of CI jobs.

MODERNIZATION OF THE DEVELOPMENT ENVIRONMENT

Legacy Environment

The FEC software development environment at CERN (Figure 2) is tightly coupled to a Network File System (NFS) and to Virtual Machines (VPCs) that are setup for FEC development. The typical developer connects to their VPC and runs a dedicated distribution of the Eclipse IDE to develop. The NFS is mounted automatically on the developer VPCs. The development tools and dependencies (IDE, makefiles, dependent libraries) are either hosted on NFS, or depend on data or binaries hosted there.

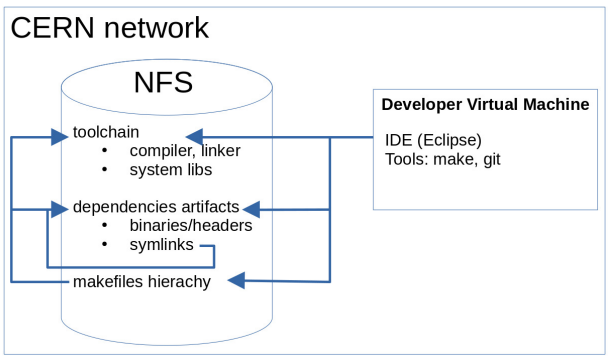

Figure 2: High-level view of the legacy development environment

The in-house makefiles provide build and release functionality, which is possible by knowing the layout of the dependency artifacts and the toolchain location on NFS. It knows the base path for artifacts (headers/libs) at build time, as well as where to store an artifact upon release. There is no enforcement of dependency consistency, nor of toolchain/compiler settings. As it predates Git, it only supports SVN.

This setup makes it easy to provide a functional development environment to the developer, but at the cost of flexibility:

- The development machine requires access to NFS, which is accessible only from the CERN network.
- Connecting to the development VPC is inconvenient (VNC, ssh -X), especially when the network access is with reduced bandwidth (e.g. teleworking), or when using multiple screens (VNC limitations).
- The custom layout of the dependencies folder on NFS makes it incompatible with most standard tools (IDEs, build systems) without extra integration work and project metadata duplication.
- The need to access NFS makes offline development or CI on the cloud either impossible or very inefficient.
- The Eclipse IDE for C++ is falling behind other C++ IDEs in terms of features and ergonomics (e.g. CLion, QtCreator, VSCode)

Modernizing the developer's experience entails keeping the ease of access of the development environment, while ensuring the continuity and quality of production artifacts. Regarding the development tools, that means providing more flexibility (e.g. offline work, local toolchain, choice of modern IDE, etc.). On the CI side, it means an easy initial setup and no redundant configuration tasks, as well as support for modern development workflows based on branches and merge requests.

Modernization of the Development Tools

Replacing the makefile hierarchy with CMake brings standardisation to what was an in-house custom project structure. Moreover most C++ IDEs have good support for CMake projects, thus enabling much better IDE support out of the box.

Alongside CMake, Conan is used to alleviate other pain points of the legacy setup.

"Conan is a MIT-licensed, Open Source package manager for C and C++ development, allowing development teams to easily and efficiently manage their packages and dependencies across platforms and build systems." - https://conan.io

- Conan describes package dependencies precisely in its recipe file (a text or python file called *'conanfile.[txt/py]'*), and is able to check the consistency of the versions in the dependency graph, thus avoiding the "dependency hell".
- Conan enforces strong but configurable versioning of binary packages. The binary package's ID includes a hash of compiler settings to ensure binary compatibility of the produced artifacts
- Through the given metadata, Conan provides convenient ways to either fail a build if a dependency cannot be satisfied (wrong version, wrong toolchain/compiler settings), or build the dependency from source
- Conan can package binary artifacts and share them through Conan repositories. Binary packages can be downloaded and build artifacts uploaded via HTTP(S) with appropriate authentication and authorization. Conan repositories are implemented by various products, including Artifactory, Gitlab, Conan-server, Nexus. With no need for NFS access anymore, cloud and offline builds are much easier.
- Setting up a local build environment for a project with arbitrary dependencies is as simple as running "conan install"

Renovation of the CI and Testbed

The decision to move away from Bamboo was triggered by the need to migrate to a newer version, and to

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV041

renew the licenses. Migration would have been a significant effort that already justified considering other platforms. Two options were evaluated, both already used at CERN and both without extra license fees: Jenkins (open-source) and Gitlab CI (deployed globally by the CERN IT department).

The legacy version of Bamboo in use so far required many slow interactions with the user interface to configure jobs and it had limited support for merge requests and branches. Therefore, the renovation was also an occasion to benefit from features brought by more modern tools. Below is a list of features expected from a CI platform:

- In-code configuration (.gitlab-ci.yml / JenkinsFile) and auto-configuration (zero-click) of the CI jobs
- Support for complex pipelines (serial and parallel stages, optional stages)
- Automatic support for branches and merge requests
- Support for triggers (an upstream jobs triggers the build of a downstream dependent job)
- Support for custom agents/executors
- Access rights management, integration with CERN's SSO
- Support for branch heavy workflows (e.g. libA on branch b1 will automatically pick up dependencies on the b1 branch if it exists)

IMPLEMENTATION STORIES, LESSONS LEARNT

The Bamboo to Jenkins Story

In terms of features, both Gitlab CI and Jenkins offer all basic features of a CI platform. What tipped the balance in favour of Jenkins was the fact that it is open-source, and not dependent on licenses from Gitlab. By using an open-source product we limit the risk of having features moved behind a higher-priced licence tier in the future. However, deploying dedicated Jenkins instances means additional work to build and maintain them versus the IT-managed Gitlab CI platform. Gitlab integrates vertically a lot of features that could be useful but which increase vendor lock-in (e.g. docker/Conan registries). The documentation of Gitlab is more complete and much better structured than that of Jenkins, especially regarding the domain specific language (DSL) to define pipelines. The Jenkins ecosystem is made of a lot of plugins. Support, documentation and compatibility of plugins can be a challenge. In practice, these are not blocking points, since most configuration is done once, and experience is shared within the team. Additionally it was decided to keep most of the CI jobs as simple shell command invocations, which is very portable across CI systems in case it is needed to migrate again in the future. This reduces vendor lock-in, and facilitates running jobs manually for testing or debugging.

The Jenkins master is deployed as an OpenShift pod, whose configuration is completely described as code. Deployment on OpenShift means no need to deploy/maintain hardware. Configuration as code mitigates the lack of durability of pods on OpenShift, as well as making it easy to reuse (e.g. for another project/team) and easy to bring back up in case of major failure. The main challenge here was to find how to serialize the configuration (Configuration as Code) of Jenkins and its plugins, because the format and commands to use are barely documented, requiring long searches for help on the internet/stackoverflow [9], as well as a lot of trial and error.

Beyond the basic Jenkins installation, the most important plugin is the Gitlab Branch Source plugin [10] that allows to simply point to a Gitlab folder from Jenkins, and auto-configures the CI jobs for all repositories, branches and merge requests in that Gitlab folder.

Codeine: CLI helper

Codeine (a play on words between "code" and the painkiller drug) is an idea to make development painless, especially considering two types of software developers:

- Framework and tooling providers
- Equipment developers

The distinction is often blurred depending if the focus of the team is on software engineering or more direct operational commitments. The goal with Codeine is to make the former's life easier, while letting the latter benefit from technical improvements with no/minimal impact to their work. Considering the ageing makefile-based build framework, the original idea behind Codeine was to wrap and abstract the steps of software development in the tool (create empty project, build, test, release) to be able to evolve the underlying implementation bricks with no impact on developer's workflow and habits. Codeine is based on a file describing the project, called *'product.xml'* which, in a first approach, contains metadata about the project: name, version, VCS repository, type (library or executable).

The first Codeine implementation re-used the custom makefiles to perform operations on the project (e.g. build, release). This approach helped consolidate the knowledge about existing workflows: inputs, outputs, dependencies. It also facilitated the migration of repositories from SVN to Git, based on the metadata of the *'product.xml'*. By implementing Git repository support in Codeine, the risky and complicated task of adding sub-par Git support to the legacy makefiles was avoided. After working on implementing the artifact release flow in Codeine, further metadata was added to the *'product.xml'* to structure the released artifacts (e.g. does the artifact require extra files beyond the binary to be delivered? symbolic links defined?). Metadata identifying the dependencies of the product were also added to *'product.xml'* in anticipation of better and/or automated dependency management. In summary, this version of Codeine fully supported the legacy makefiles framework, additional support for Git repositories, and a structured release process.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV041

A First Proof-of-Concept

Building on the Codeine base described above, a proof of concept was launched to integrate modern tools into the development workflow and validate the feasibility of using Conan to manage dependencies and CMake as a more modern build system.

From the outset Codeine was intended to support the transition between the makefile-based build system to a more modern one. It was anticipated that the existing *'product.xml'* content should be sufficient, or require only minor adaptation, however it was soon discovered that this was not the case. The first *'product.xml'* change required was to include an element to tell Codeine which build system it should use. This was considered a necessary hole in the abstraction layer of Codeine.

Having decided on trying out Conan to manage dependencies, the proof of concept used Conan through Codeine to build most of the framework development team's libraries. The first step was learning about Conan usage, while the second milestone was letting Conan know about Codeine's metadata, by using Conan's mechanism to extend python conanfiles.

Since Conan is not a build system for C++, but rather generates configuration for one, and CMake, among other advantages, has good support from Conan, it was decided to use CMake from Conan to build libraries. This makes for a complex flow: Codeine calls Conan, which uses a Codeine extension (python) to understand the *'product.xml'*, and generate files for CMake, which in-turn is used to build the library.

Although complex, this proof of concept already brought several advantages:

- Strong dependency and toolchain management thanks to Conan.
- Standard dependency declaration
- Standard artifact packaging, ready to be consumed by a downstream project (using Conan).
- Conan's remote features to upload/fetch packages from an artifact repository, in turn enabling jobs isolated from the legacy NFS repository to fetch artifacts through HTTP.

However this proof of concept also showed that abstracting over both a loose legacy system and a stricter modern one is very hard. Several complications to Codeine were implemented to support both systems, in an imperfect way. Furthermore, the generated CMake files were complex and did not provide the extensive IDE support expected (e.g. QtCreator supports CMake based projects, but was not able to parse the generated CMake files satisfactorily)

Lessons learnt:

- A floating period transitioning back and forth between the legacy and new system is to be avoided
- Good abstractions are hard, especially over vast and/or vague domains such as "developer experience"
- Retaining legacy comptibility takes a lot of effort, with little reward
- Conan brings a lot of structure to a project with its strict dependency management, however, this does not fit nicely with the loose structure of the legacy makefiles, which harmed the goal of Codeine being a perfect abstraction over the build system.
- More abstraction layers make debugging harder (why/where did a build fail: in Codeine? in CMake? in Conan?)

Looking Forward: Second Proof-of-Concept

Given the experience of the Conan + Codeine + CMake proof-of-concept, a next iteration is foreseen, with aims to reduce complexity, and avoid the leaky abstraction problem posed by Codeine's *'product.xml'*.

Instead of Codeine being a complete wrapper over the development workflow, it will be re-focused on the parts missing from other tools: creation of new empty projects, and managing the release process (e.g. ensuring code is committed, pushed, that the code repository is tagged with the release version, etc.). This reduced scope of Codeine will allow to simplify the usage of tools like Conan and CMake, making the most of them without being limited by the incomplete abstraction that Codeine's *'product.xml'* tried to be. The advantage of using publicly available and documented tools instead of Codeine-as-a-wrapper are:

- More flexibility to the developer.
- Less support for the providers of Codeine.
- More publicly available support for the tools (e.g. Conan's and CMake's documentation, experts on StackOverflow [9]).

CONCLUSION

Software engineering best practices evolve and improve with time, as do automation and tooling to support developers. Embracing such evolution in CERN's FEC software development environment brings added value, but given the variety of constraints,the modernisation is both technically and organisationally challenging. Forced migrations such as the CI/Testbed are a good opportunity for improvements. Modernisation is a continuous iterative process, and while changes are disruptive and require stakeholder buy-in, a flexible architecture allows for iterative changes to be rolled-out progressively.

Beyond developer productivity and satisfaction, having a modern development environment is also more attractive to potential candidates, for whom experience with industry standard tools is more valuable.

REFERENCES

[1] M. Arruat *et al.*, "Front-end software architecture", in *Proc. ICALEPCS'07*, Knoxville, Tennessee, USA, Oct. 2007, paper WOPA04, pp. 310-312. https://jacow.org/ica07/PAPERS/WOPA04.PDF

[2] A. Guerrero *et al.*, "CERN front-end software architecture for accelerator controls", in *Proc.ICALEPCS'03*, Gyeongju, Korea, Oct. 2003, paper WE612, pp. 342-344. https://jacow.org/ica03/PAPERS/WE612.PDF

[3] J. Lauener and W. Sliwinski, "How to design & implement a modern communication middleware based on ZeroMQ", in

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV041

Proc. ICALEPCS'17, Barcelona, Spain, Oct. 2017, pp. 45–51. doi:10.18429/JACoW-ICALEPCS2017-MOBPL05

[4] GCC Dual ABI, https://gcc.gnu.org/onlinedocs/libstdc++/manual/using_dual_abi.html

[5] Conan Package Manager, https://conan.io

[6] CMake, https://cmake.org

[7] Bamboo, https://www.atlassian.com/software/bamboo

[8] Attlassian Server end of support, https://www.atlassian.com/migration/journey-to-cloud

[9] Stack Overflow, https://stackoverflow.com/

[10] Gitlab Branch Source plugin, https://plugins.jenkins.io/gitlab-branchsource