EASY AND AUTOMATIC CMAKE BASED PROJECT GENERATION FOR LARGE-SCALE SIMULATION APPLICATIONS

Noah Wiederhold, Luca Tiedemann
DLR e.V., Institut für Flugsystemtechnik, Lilienthalpl. 7, 38108 Braunschweig

Abstract

When working with large-scale software projects, it is essential to use a meta-build system in order to maintain flexibility in terms of the build system used. At the Department of Flight Dynamics and Simulation (FDS) in the Institute of Flight Systems (FT) of German Aerospace Center (DLR), there exists several such large-scale projects, for instance in the Air Vehicle Simulator (AVES). AVES utilises the real-time simulation framework 2Simulate, which consists of multiple subprojects. Currently, 2Simulate relies on a project structure that enables to build with Visual Studio and Unix Makefiles. Migrating from such existing project structures to a meta-build system is time-consuming, and requires a coordinated design approach. To address this issue, this paper proposes the introduction of an abstraction layer to cross-platform Make (CMake) with regards to research specific requirements based on the meta-build system CMake. The general concept for this approach is based on state of the art principles from web development. The approach aims to create a uniform project standard while applying common software development best practices.

Keywords

CMake; Simulation; C++; Cross Plattform; Windows; Linux

DLR	German Aerospace Center
VS	Visual Studio
CI	Continuous Integration
IDE	Integrated Development Environment
QNX	QNX RTOS by Blackberry
FDS	Department of Flight Dynamics and Simulation
FT	Institute of Flight Systems
AVES	Air Vehicle Simulator
GTest	GoogleTest
CppLint	C++ Lint
Doxygen	Doxygen Documentation Generator
CTest	CMake Test
CMake	cross-platform Make

1. INTRODUCTION

At DLR, working with large-scale facilities, such as AVES within FT FDS, is common. AVES enables the simulation of air vehicles for research.
To this end, AVES employs a complex and interconnected software architecture consisting of multiple large-scale software projects. In this specific context, large-scale does not necessarily mean that there is a high number of lines of code, but rather that there is a complex dependency structure between many projects, regardless of their size. One of these software projects is called 2Simulate [1]. 2Simulate is a real-time simulation framework consisting of several subprojects. At present, 2Simulate projects depend on manually created Visual Studio (VS) project files or QNX RTOS by Blackberry (QNX)-specific build processes [2]. Maintaining these build systems is resource-intensive and often results in inconsistent states of the same subproject across different VS versions or platforms. As developers work on different parts of 2Simulate, the project structure of each subproject tends to be unique, as there is no complete standard for the definition of projects. This makes it challenging to add generalized testing or Continuous Integration (CI).

FIG 1. AVES

Using a dedicated build system is common practice when working with large software projects. With the increasing amount of files and dependencies, the maintainance of multible build systems for different Integrated Development Environment (IDE) versions or platforms becomes time-consuming. The adoption of meta-build systems streamlines this task to maintaining only a single build system-specific configuration. This configuration can then be used to generate the necessary project files for the build systems itself. This enables flexible workflows, as build systems can be seamlessly interchanged. Additionally, most meta-build systems enable cross-platform capabilities by design. Despite working with complex C++ projects in FDS, a meta-build system is currently missing.

2. STATE OF THE ART

In the C++ domain, there is a wide range of build systems, as well as meta-build systems available. Plain build systems provide a set of instructions to a compiler, detailing the necessary parameters to create an executable file, for example. Meta-build systems, in contrast, are designed to generate project files for a range of supported build systems. Notable meta-build systems include CMake, Autotools, GYP/GN and Meson [3] [4] [5] [6]. Of these, CMake is the most popular option [7, p.17] [8, p.15] [9, p.14]. The aforementioned meta-build systems have in common, that they offer a platform and IDE-independent way of defining building logic. This includes features such as defining targets and linker dependencies, including source code and setting compiler flags. In addition, CMake and Meson offer a way of managing dependencies, by using modules called either "FetchContent" in CMake or "wrap" in Meson [10] [11]. These methods allow loading of files or even projects at build-time. While all these features improve the overall workflow for developers, the level of complexity of meta-build systems itself is a significant challenge. For instance as of 2025 CMake consists of 1.5 million lines of code which is roughly equivalent to the size of the 2010 Linux kernel, highlighting the scope of the project [12] [13].
Nevertheless, as of 2021 46% the highest-ranking GitHub repositories use a dedicated meta-build system [14, Fig.5.1 p.28]. In the field of scientific applications, meta-build systems are a common occurrence as well. For instance, the ATLAS particle detector utilizes CMake [15].
In order to mitigate the potential disadvantages associated with meta-build system integrations, an abstraction layer is needed. This layer should conceal the complexity of the meta-build systems, thereby facilitating a smoother transition from existing build systems.

3. REQUIREMENTS

For the purposes of this paper, 2Simulate is used as an example. Given the aforementioned context, the requirements are as follows:
Cross-platform The build and generation approach shall support configuring, building and installing all defined targets on both Microsoft Windows (VS) and a supported Linux distribution using the same unified project definition without modifications.
Cross-version The solution shall allow generating project files for multiple supported IDE's / compiler versions (e.g. different VS releases) without requiring per-version maintenance inside individual projects.
Effortless testing The proposed architecture shall enable the addition and execution of unit tests via a full-fitted testing framework with at most one explicit test enable switch (e.g. a single variable) per project, and without custom per-project build logic.
Effortless CI The solution shall permit headless configuration, build and test execution (including linting and documentation targets when enabled) through a single reusable continuous integration pipeline configuration file applied unchanged across projects.
Uniform projects Every project shall conform to a specified directory layout so that working with the projects can be done in a generalized manner.
Dependency optimality The solution shall ensure that each third-party dependency is integrated exactly once within a multi-project generation, in order to avoid redundancy.

A short overview of the defined requirements is given in Table 1. Optional design drivers are depicted in Table 2.

Requirement	Description
cross-platform	support for Windows and Linux
cross-version	supports different IDE-Versions
effortless testing	straightforward tests integration
effortless CI	straightforward CI integration
uniform projects	consistent project structure

TAB 1. List of requirements

Design Driver	Description
easy to use	intuitive usability
dependency optimality	no/less dependency overhead
linting support	integrate a linting tool

TAB 2. List of design drivers

4. CONCEPT & IMPLEMENTATION

The use of meta-build systems provides the majority of the necessary functionality to meet the requirements for 2Simulate, as listed in Table 1. However, meta-build systems also introduce further complexity. Converting existing projects to use meta-build systems hence would require for at least one maintainer per project to have a deep understanding of the given meta-build system. Furthermore, within 2Simulate several interlinked subprojects exist. As these subprojects depend on each other, their respective maintainers would need to coordinate the conversion with one another. In order to minimize training and coordination requirements, the following concept introduces an abstraction layer to meta-build systems with a uniform project structure. For the given use case, CMake is selected as the meta-build system.

4.1. Concept

The following approach aligns with best practices, commonly adopted by web developers. For example, in frontend development, initial code served by the web-server typically consists of basic page structures. Additional page elements, styles, and logic are fetched asynchronously in the background. This approach is known as "lazy loading" [16, p.7]. While lazy loading primarily helps to achieve shorter loading times in the domain of web development, it is also applicable to build systems.
The foundation of the proposed concept is the division of CMake project definitions into two parts, one frontend and one backend part. The schematic of such a distinction is depicted in Figure 2. The frontend is "static" and only includes variable definitions as well as minimal code for fetching the backend (e.g. code for loading a second CMakeLists.txt over the network). The second part, which is more complex, is the backend of the project definition. It uses the variables defined in the frontend part to declare targets, add linking dependencies, and so on. Targets in this context are best described as desired configuration sets of the given code files. For instance, generating a static library requires at least a target to be configured as a static library and to include references to the relevant code files.
This approach allows one general logic to be defined within the backend part and used for multiple projects.

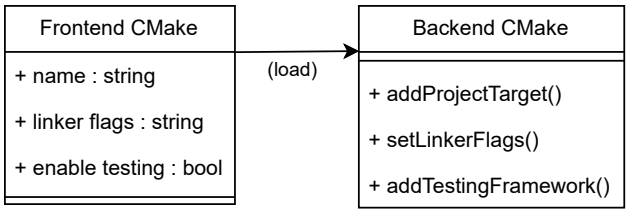
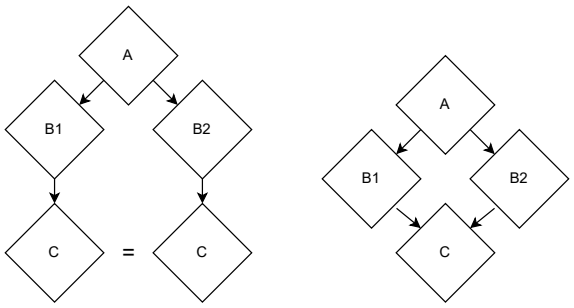


FIG 2. Concept schematic

Each project only needs to include the minimal redundant frontend part and define variables as needed. The given concept enables developers to maintain the build system of multiple projects by modifying a single point of code. Furthermore, it eliminates the need for manual changes to every dependent project when the underlying build logic changes. Because of this, the frontend part is called "static": apart from the variable values, it is never changed. Only the backend part is adjusted if necessary.

Additional objective is to reduce overhead in the dependency management. Within dependency management, the primary focus of the proposed approach lies on two key points. The first can be clarified by referring to the schematics in Figure 3a and Figure 3b. Assume a project A, which depends on B1 and B2. B1 and B2 are dependent on C. If B1 and B2 were to load their dependencies individually, this would result in C being loaded multiple times. To resolve this issue, it is essential to ensure that dependency resolution is unique. Even though the dependency C for example could potentially be requested with differing configuration, for the current use case 2Simulate only non-configurable projects exists.

The second point is more specific to the use case in question, as developers at AVES work on several subprojects of 2Simulate. In order to facilitate effective workflows with either a single project, or a combination of multiple dependent projects, it is necessary to distinguish between "solo" and "workspace" mode. When solo generating a project, it is essential that all its dependencies are loaded by the backend. However, when using a workspace generation, only dependencies outside the scope of the collection of dependent projects need to be loaded. The remaining dependencies need to get resolved locally, as the projects already exist on the drive.



(a) Unsolved dependency problem **(b) Solved dependency problem**

FIG 3. Dependency problem of shared resources

4.2. Implementation

In line with the specified concept, the overall project is divided into a template project and a core project. The template project corresponds to the frontend and the core project to the backend. CMake is currently the most popular meta-build system, with a significant lead over Meson and other similar systems. This is why CMake is used as the meta-build system for the following [7, p.17].

As outlined in Section 4.1, the template's primary logic, encapsulated in its CMakeLists.txt file, is structured in two sections. The first section contains all variable definitions and is used by the developer to configure the project. The second section uses CMake's FetchContent module to define and load the core project. At present, a developer can use the features listed in Table 3. As there may be several compiler flags needed for instance, the variables are implemented as lists wherever applicable.

Option	Description
project name	identifier of the project
project path	relative path in VS solution
project type	shared/static library; executable
enable testing	yes/no
linker language	C++/C
third-party workspace	internal dependencies
third-party extern	extern dependencies
Compiler Flags	-
Linker Flags	-

TAB 3. List of template configuration options

In addition to the CMakeLists.txt the template project also consists of a streamlined and unified project structure as depicted in Figure 4.

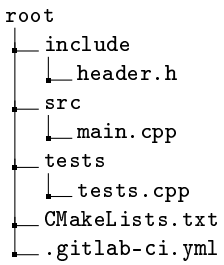
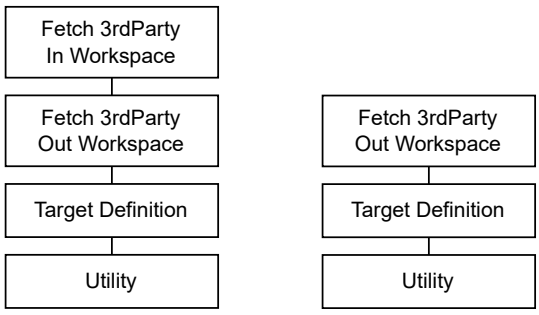


FIG 4. Template project structure

In contrast, the core project consists of one large CMakeLists.txt and a number of CMake scripts that are outsourced.

As outlined in Section 4.1, the logic behind the core project is divided into solo and workspace mode. When generating the workspace each third-party project outside the workspace is uniquely fetched. This is achieved by comparing each one against a list of already fetched target names and utilizing CMake's FetchContent methods. Following the resolution of third-party projects, the project's own target will be defined based on the variables, specified in the template project. The installation directories are set target and configuration (debug/release) dependent, ensuring the correct and consistent export of build results and header files when using the generated projects. In contrast, the solo generation also features fetching all internal third-party projects, prior to the declaration of the target. The comparison between both modes is illustrated in Figure 5a and Figure 5b.

Further features of the core project, listed in Table 4, are bundled in a utility section for both modes. Linting is carried out using the open-source tool C++ Lint (CppLint) [17]. By utilizing CMake's "add_custom_target" method, call-



(a) solo generation **(b) workspace generation**

FIG 5. Generation modes

ing the Python-based CppLint tool is encapsulated into the generated build-system projects. For every project, including dependencies, one linting target is created. Necessary input parameters, like the path to code-containing directories, get generated dynamically when creating the corresponding targets. For instance, when creating the project "2Simulate", a target called "run_cpplint_2Simulate" is created, which executes CppLint with the project specific source code paths. Every dependent project using the templates also creates their respective "run_cpplint_..." target. Configuration of the linting process happens per project through a CppLint-specific configuration file.
To generate streamlined documentation, Doxygen Documentation Generator (Doxygen) is used [18]. In contrast to linting, only one target is created for running Doxygen. This target only creates documentation for the current project, excluding dependencies.
Testing is integrated with Google's testing framework GoogleTest (GTest) [19]. Similar to the integration of Doxygen, testing is only possible for the current project. The execution of tests is enabled through a project-specific executable target, which is linked against the GTest-library.
To demonstrate the different targets created, an exemplary target-tree of "2Simulate" is depicted in Figure 6.

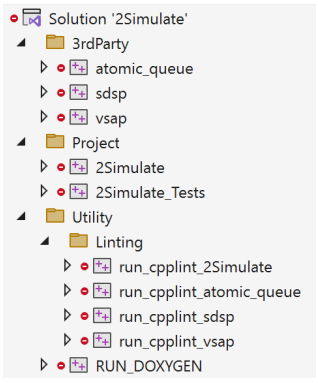


FIG 6. Overview of generated targets

While the mentioned features are common in the C++ domain, the licence check is unique to the proposed approach. The backend continuously checks for licence files when adding projects, comparing them against a list of authorized licence types. In the event of missing licences, a warning will be issued while any detected forbidden licences will result in an error. Currently, the configuration of permitted and prohibited licence types is managed centrally within the core project.
To enable features that fall outside the capabilities of the proposed approach, the generation process can be extended on a per-project basis through the use of additional CMake scripts.
In situations where a user may wish to create a project using an older version of the template, it is always possible to fetch older backend versions by specifying a Git tag or commit directly within the project's frontend.
In addition to new features, the proposed implementation also offers indirect improvements. The uniform project structure enables CI via a generic .yaml file. This file uses CMake's CMake Test (CTest) to facilitate testing. Furthermore, current forms of build result distribution can be replaced with CI-based artifacts distribution.

Feature	Description
linting	integration of CppLint as a target
documentation	integration of Doxygen as a target
testing	integration of GTest
licence check	check for allowed licences

TAB 4. List of features

5. CONCLUSION

The proposed approach makes the transition from existing build systems in AVES less challenging. For this purpose, a front-end template project is available for immediate use. Developers can now use the introduced abstraction layer to utilize a widely used meta-build system without special training. By default, CMake's sophisticated mechanisms are not exposed to the typical user, but can be accessed through custom code. By using a dedicated backend for the build logic, improvements, and changes can still be introduced later on and independent of the project's developer.
The primary requirements, including cross-platform, cross-version, and the integration of modern development concepts, are met by using the meta-build system CMake. Applying the proposed approach of splitting a CMake project into two projects, enabled a seamless integration of a complex meta-build system into the existing project structures of 2Simulate with only minimal effort. The decision to differentiate between solo and workspace generation enabled a space optimal solution to the dependency management problem of large projects. Furthermore, it allows flexibility in terms of solo project-based workflows to remain. The uniform implementation of all features through the proposed approach indirectly enabled the generalized integration of CI by using a single .yaml file. The usage of CMake also enhanced the testing integration into CI pipelines, enabling CI-workflows to rely on CTest for automated test execution and status reporting.

Contact address:

Noah.Wiederhold@dlr.de

References

[1] Jürgen Gotschlich, Torsten Gerlach, and Umut Durak. 2simulate: A distributed real-time simulation framework. In Jürgen Scheible, Ingrid Bausch-Gall, and Christina Deatcu, editors, *ASIM Mitteilung 149 / ARGESIM Report 42*. ARGESIM Verlag, 2014. ISBN: 9783901608438.

[2] Blackberry. Qnx momentics projects, 11.03.2025. https://www.qnx.com/developers/docs/8.0/com.qnx.doc.ide.userguide/topic/creating_qnx_project.html.

[3] Cmake, 23.04.2025. https://cmake.org/about/.

[4] Automake - gnu project - free software foundation, 23.04.2025. https://www.gnu.org/software/automake/.

[5] gn - git at google, 23.04.2025. https://gn.googlesource.com/gn/.

[6] Meson build system, 22.04.2025. https://mesonbuild.com/.

[7] Standard C++ Foundation. Cppdevsurvey 2024, 2024. https://isocpp.org/files/papers/CppDevSurvey-2024-summary.pdf.

[8] Standard C++ Foundation. Cppdevsurvey 2023, 2023. https://isocpp.org/files/papers/CppDevSurvey-2023-summary.pdf.

[9] Standard C++ Foundation. Cppdevsurvey 2022, 2022. https://isocpp.org/files/papers/CppDevSurvey-2022-summary.pdf.

[10] Kitware. Fetchcontent — cmake 4.1.0, 05.08.2025. https://cmake.org/cmake/help/latest/module/FetchContent.html.

[11] Meson wrap, 25.08.2025. https://mesonbuild.com/Wrap-dependency-system-manual.html.

[12] Cmake loc counter, 31.01.2025. https://codetabs.com/count-loc/count-loc-online.html.

[13] Phoronix Media. Gitstats - linux kernel, 23.06.2015. https://www.phoronix.com/misc/linuxstat-june-2015/lines.html.

[14] Lukas Gygi. *CppBuild: Large-scale, automatic build system for open source C++ repositories*. 2021.

[15] J. Elmsheuser, A. Krasznahorkay, E. Obreshkov, and A. Undrus. Large scale software building with cmake in atlas. *Journal of Physics: Conference Series*, 898(7):072010, 2017. ISSN: 1742-6596. DOI: 10.1088/1742-6596/898/7/072010.

[16] María Pilar Del Salas-Zárate, Giner Alor-Hernández, Rafael Valencia-García, Lisbeth Rodríguez-Mazahua, Alejandro Rodríguez-González, and José Luis López Cuadrado. Analyzing best practices on web development frameworks: The lift approach. *Science of Computer Programming*, 102:1–19, 2015. ISSN: 0167-6423. DOI: 10.1016/j.scico.2014.12.004.

[17] cpplint/cpplint: Static code checker for c++, 24.04.2025. https://github.com/cpplint/cpplint.

[18] Doxygen, 16.04.2025. https://doxygen.nl/.

[19] GitHub. google/googletest: Googletest - google testing and mocking framework, 24.04.2025. https://github.com/google/googletest.

[20] Jürgen Scheible, Ingrid Bausch-Gall, and Christina Deatcu, editors. *ASIM Mitteilung 149 / ARGESIM Report 42*. ARGESIM Verlag, 2014. ISBN: 9783901608438.

[21] The Linux Foundation, Victor Rodriguez. Cutting edge toolchain (latest features in gcc/glibc), 23.04.2025. https://www.youtube.com/watch?v=QXwxBM4sbYM.