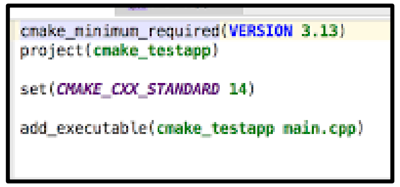
SAND2022-7018C

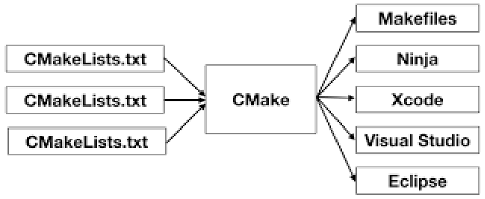
A new CMake Scripting Language?



```
cmake_minimum_required(VERSION 3.13)
project(cmake_testapp)

set(CMAKE_CXX_STANDARD 14)

add_executable(cmake_testapp main.cpp)
```

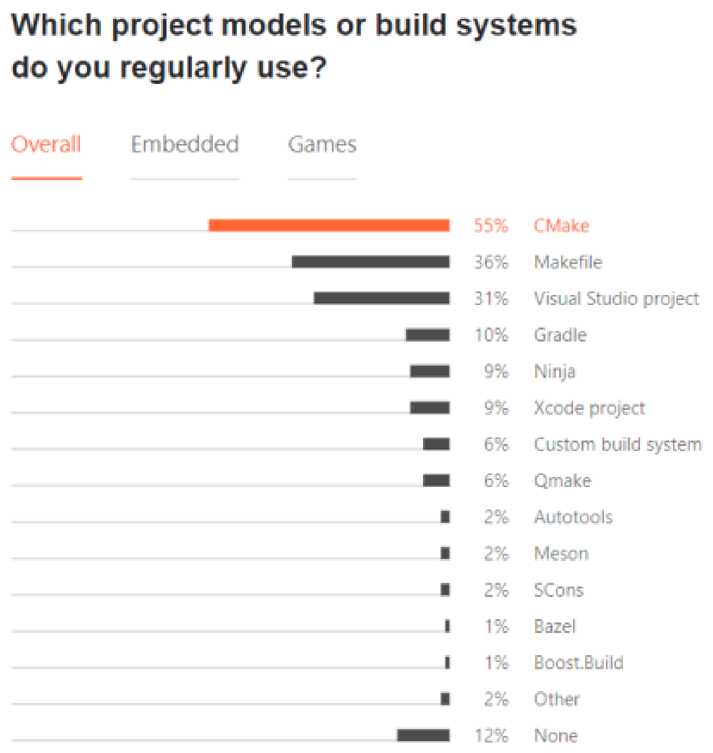


Addressing CMake weaknesses and avoid a possible expensive future migration to another build system

Roscoe A. Bartlett, Sandia National Laboratories, Department of Software Engineering Research
Bill Hoffman, Kitware VP
Brad King, Kitware CMake Lead

CMake Overview

- CMake is the de-facto standard C++ build system. It holds over 50% of the market for building C++ code (Jetbrains survey result).
- CMake is the standard build system in the Exascale Computing Project (ECP).
- CMake has broad and growing usage in the CSE and ASC communities.



Our Background with CMake

Roscoe A. Bartlett:

- Transitioned the Trilinos project build system from GNU Autotools to CMake starting in 2008.
- Developed and maintained a framework for a package-based architecture for CMake projects called the Tribal Build, Integration and Test System (TriBITS).
- Has run many contracts with Kitware to improve CMake, CTest, and CDash for many years going back to 2008 through today (2022).

Bill Hoffman

- The originator of the CMake project in 2000 as a build system for the Insight Toolkit
- Works to find CMake customers to fund the development of CMake

Brad King

- Lead maintainer of CMake

History and evolution of CMake language

- CMake began in 2000 as a build tool for created by Kitware for the ITK C++ project
- Core syntax established:
  - `command(arg1 arg2 “arg3 with space” ... argN)`
  - space-delimited arguments, no commas
  - quotes used for spaces within an argument
- **Initially declarative**
- Cascading makefile-per-directory model
- One library per directory/makefile
  - `library(mylib)`
  - `source_files(itkFoo itkBar)`
  - `include_directories(/path1 /path2)`
  - `link_libraries(dep1 dep2)`
  - `subdirs(dirA dirB)`
  - sources listed without suffixes (`itkFoo => itkFoo.{h,cxx}`)
  - no control flow, processed serially as implementation detail
  - `${VAR}` syntax only for predefined placeholders
- **Not designed to be a general-purpose scripting language**

History and evolution of CMake language

- Extended to support multiple libraries per directory (2001)
  - `source_files(mysrcs someClass1 someClass2 someClass3)`
  - `add_library(mylib ${mysrcs})`
  - “source lists” were the predecessor to variables
  - Stored as strings with semicolon-separated elements
  - `someClass1;someClass2;someClass3`
  - No syntax for nesting because they were just lists of source names
- **Converted to imperative to support flow control** (2001)

```
if(WIN32)
  source_files(mysrcs someWindowsOnlyClass)
endif()
```

- Source lists became variables (2001)
  - `source_files() => set()`
  - still used semicolon-separated syntax
  - directory scoped
- Flow control added over time
  - `foreach` (2001), `macro` (2002), `while` (2005)
  - still used `command(...)` syntax, even as block delimiters
  - `function` used dynamic scoping for variables in each directory (2007)
- **Evolved into a scripting language**

History and evolution of CMake language

- **Build system model converted to target-centric approach**
  - **a.k.a. Modern CMake (2013)**
  - `target_compile_definitions()`, `target_include_directories()`, **etc.**
  - **legacy directory-scoped commands still exist, still widely used**
- Evolution driven by use cases
  - Initially developed along with projects using it, changes deployed quickly
  - Had nightly testing, but no design or review process for almost 10 years
- Backward compatibility became a priority
  - Early: “get it working for now, we can always change it later”
  - [popularity explodes]
  - Later: “we can’t change it now because of backward compatibility”
- The “policy” mechanism was added (2008)
  - Allows some changes without breaking compatibility.
  - Many language fundamentals cannot be changed.

History and evolution of CMake language

- **Result:** A language that works well for many use cases, but with **surprising limitations and gotchas that make some use cases unnecessarily hard**, or even impossible, to achieve using documented interfaces, particularly when they involve processing arbitrary data.

- Example: Lists cannot easily store other lists as elements.

- Example: CTest uses the CMake scripting language to define test cases. Data involving special characters like quotes (`"`), semi-colons (`;`), and square brackets (`[]`) cannot easily be passed to tests as command-line arguments or environment variables.

CMake Alternatives and CMake Scripting Language Critiques

Meson (Started 2012):

- Simple python-like input syntax, small interface, preferred build system for Fedora packages, gaining popularity even in CSE and HPC communities (e.g. there is Spack support for Meson)

- https://www.admin-magazine.com/HPC/Articles/The-Meson-Build-System
  - "Finnish developer Jussi Pakkanen was frustrated by existing build systems with **foolish syntax in configuration files and unexpected behavior**."
- https://mesonbuild.com/Comparisons.html
  - CMake Cons: "**The scripting language is cumbersome to work with**. Some simple things are more complicated than necessary."
- https://gms.tf/the-rise-of-meson.html
  - "Since it was created after CMake (which was initially released in 2000), it had the chance to **learn from CMake's mistakes**. In comparison to CMake, Meson has better documentation, it doesn't support generating classic makefiles, it only supports out of source tree builds, and **its domain specific language is arguably much better designed**."
- https://www.rojtberg.net/1481/do-not-use-meson/
  - **"You might say that CMake is ugly"**

These are all criticisms of the CMake scripting language and legacy CMake commands but not of core CMake functionality!

Cost of moving away from CMake to an alternative?

- Transitioning **just the Trilinos project** to a new build and test system could cost **$1M+** over several years.
- **Transitioning just the SNL ASC-related codes** to a new build and test system could cost **$4M+** over several years.
- Transitioning all ASC codes to another build and test system could cost **$10M or more** and damage the productivity of developers during the transition process.
- **Example:** The transition of Trilinos from GNU Autotools and a custom testing system to CMake, CTest, and CDash from 2008 through 2022:
  - $500K+ in SNL developer time
  - $700K+ in for Kitware to extend and improve CMake, CTest, and CDash for Trilinos needs

In case you think a transition from CMake to another tools like Meson cannot happen, consider what CSE/HPC looked like 20 years ago with the dominance of GNU Autotools. Like CMake today, the Autotools people back then could have cited the large user community and large adoption rates for Autotools with confidence. But Autotools was largely replaced by CMake in many communities and **it can happen again unless we address the fundamental issues with CMake that would drive such a transition**!

What is CMake, really?

- A sophisticated "code model" representing a project's build system.
- Build rules are organized into high-level logical build "targets". Each target has:
  - A list of source files to be compiled and linked into a binary, or built via custom commands.
  - A list of dependencies on other targets (libraries).
  - Requirements for compilation and linking (include directories, compile definitions, link libraries, etc.).
  - Requirements for usage by other targets (include directories, compile definitions, link libraries, etc.).
- A "Configure" step processes CMakeLists.txt files to build the project's code model using imperative logic.
  - CMakeLists.txt files written in the CMake language.
  - Can defer some evaluation using "generator expressions".
- A "Generate" step processes the code model to generate a build system, e.g., Makefiles, Ninja.
  - Evaluates generator expressions using information not available until the generation step.

Modern target-based CMake commands and idioms are logical, robust, and proven over thousands of CMake projects.

The book "Professional CMake" largely solves the CMake Documentation problem.

The primary criticism of CMake and the draw to other build systems like Meson is the CMake scripting language and legacy CMake commands.

A new CMake language? This is not a new idea

- 2007
  - Mailing List Thread: CMake with Lua Experiment
  - CMake Community Wiki: Experiments With Lua
- 2014
  - Merge Request 3938: cmake_lua() command brainstorming branch
- 2016
  - Mailing List Thread: CMake alternative language
- 2019
  - CMake Issue 19863: Proposal: Lua as alternative imperative language
  - CMake Issue 19889: Proposal for a C++-like language: CMakeSL
  - CMake Issue 19891: Proposal for a declarative specification format
- None of these makes project specifications toolable

A new CMake language? Requirements

- Create two new interoperable CMake languages:
  - New small declarative CMake language (to allow auto-modification by tools and IDEs).
  - New small Turing-complete imperative CMake scripting language (to implement all logic).
- The new CMake languages:
  - Are formally-specified, robust, and can represent arbitrary data.
  - Exclude problematic legacy CMake commands and functionality that have better more modern replacements (e.g., no `add_defintions()` or `include_directories()`).
  - Are backward compatible with the underlying CMake C++ object model implementation.
  - Allow interaction between the new declarative and imperative CMake languages and files to construct CMake projects.
  - Allow projects and scripts with mixtures of files using the legacy CMake scripting language and the new CMake language(s).
  - Allow functions defined in legacy CMake language to be called in new language and visa vera.
- Incremental transition:
  - Allow for incremental transition of existing CMake projects and scripts file-by-file.
  - Allow new CMake projects to contain files using only the new CMake language(s).
- **Stretch Goal:** Research development of tool(s) to translate legacy CMake to new CMake language:
  - => May require some initial refactoring of legacy CMake code (e.g. to remove deprecated commands and functionality) before running the translation tool.

A new CMake language? The details

- Decouple internal representations from legacy CMake language
  - Project code model used by generators
  - Compiler/platform information used by generators
  - Typed C++ structures in place of list-of-strings representations
  - Several areas already converted, e.g., to generate file-api replies
- Design a declarative language for the majority of project specification
  - Toolable (editable by IDEs)
  - Maps to project code model (targets, sources, etc.)
  - Hooks for running imperative logic at various stages
  - Parallelizable
- Design an imperative language for scripting operations
  - Strongly typed
  - Real data structures
- Iterate designs through public review and discussion

The Funding and Staffing Challenge

- SNL ASC has successfully funded a lot of development in CMake over the years, however ...
- This effort could cost $1M or more over a several years.
- This is too much to expect any single laboratory or project to fund on their own (e.g. too much for just SNL ASC).
- Also, this effort should include direct input and feedback from many different customers, not just SNL ASC and Trilinos.
- This effort should be funded at a higher program level like ASC and/or ASCR.
- But, **there has never been a large effort like this with CMake**. Up until now, all additions and improvements to CMake have been very incremental and funded with smaller pots of money. It may require significant help outside of Kitware to complete a task like this. (But **Kitware has to be in charge of the design and implementation since they need to maintain this for years to come**.)

The Potential Payoff

- A $1M+ investment in a new CMake language(s) and translation tools could avoid a future massively expensive transition of ASC codes to another tool like Meson that could easily cost more than $20M across the ASC program and negatively impact developer productivity while the transition is taking place.

Summary

- CMake is powerful and has a very broad adoption.
- But, CMake has some critical issues with basic its scripting language that can't be fixed.
- Competitors to CMake have been created to address fundamental CMake issues (e.g. Meson)
- The transition from CMake to another build system for just ASC projects could cost $10M+.
- **Proposed solution, new CMake language(s)**: $1M+ investment over several years:
  - A new declarative language and a new Turing-complete imperative scripting language (backward compatible with the underlying CMake C++ object model).
  - Allow for incremental adoption and mixing legacy and new CMake files in same project.
- **Funding challenge:**
  - Too large for any one lab funding source (e.g. SNL ASC)
  - Funding and requirements should be spread across multiple keys users
- **The potential payoff:**
  - Reduce the time and cost for new developers to learn to use CMake.
  - Reduce the effort and cost to maintain CMake projects.
  - Avoid costly transition to a different build system (e.g. Meson) that could cost $10M+ for ASC.
  - Position CMake to be the most attractive and standardized build system for CSE software for the next 20+ years.

The End