Proposal for a new build system with CMake

The CMake system

Marco Corvo

University of Padova

XIII SuperB General Meeting

September 30, 2010

What is *CMake*?

1. Generates native build environments
   - UNIX/Linux: **Makefiles**
   - Windows: **VS Projects/Workspaces**
   - Mac OS: **Xcode**
2. Opensource
3. Cross-platform
4. Integrates testing and packaging systems

CMake features

1. Manage complex, large build environments (KDE4)
2. Very Flexible and Extensible
   - Support for Macros
   - Modules for finding/configuring software (bunch of modules already available)
   - Extend CMake for new platforms and languages
   - Create custom targets/commands
   - Run external programs
3. Very simple, intuitive syntax
4. Support for regular expressions (*nix style)
5. Support for In-Source and Out-of-Source builds
6. Cross Compiling
7. Integrated Testing and Packaging (Ctest, CPack)

Why Use *CMake*?

PROS

1. CMake depends only on C++ compiler
2. CMake supports great variety of platforms (basically every ***ix, Mac OS, Windows**)
3. CMake generates only Makefiles for all supported platforms
4. CMake additionally can produce project files for IDE's (KDevelop, XCode, VStudio)

Why Use *CMake*?

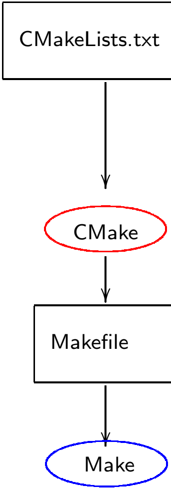
Pros (cont'd)

1. More usefull error messages when making a mistake in editing input files
2. Easy to use configure-like framework
3. CMake has simple syntax
4. CMake has a testing framework
5. CMake is faster than autotools (does not use libtools)

CMake basics

- CMake works with CMakeLists.txt files, written in the CMake syntax, which have the function of configuring the project and the single packages.
    - There are two kinds of CMakeLists.txt: the main one in the root directory of the project, then one in each package directory
- Every package has its own CmakeLists.txt containing the package parameters, like e.g. source files, special c++ flags, libraries, platform conditions . . .
- Every file then specifies which libraries and executables, if any, should be built.

Workflow



How to use *CMake*

1. Create a build directory (out-of-source build concept)
   - I choose to create my build directory into the top dir of the release
   - cd FastSim/V0.2.X
   - mkdir Build ; cd Build
2. Configure FastSim for your system
   - cmake [options] <path to main CMakeLists.txt>
3. Build the package
   - make

Basic structure for *CMake* and FastSim

1. The main CmakeLists.txt file is in the top level of the release

```
# Top-Level CmakeLists.txt
project( FastSim )
. . .
# Load some basic macros which are needed later on and find some usefull package
include(MyMacros)
find_package(CLHEP)
. . .
set(EXECUTABLE_OUTPUT_PATH ¨path to binary dir¨)
set(LIBRARY_OUTPUT_PATH ¨path to library dir¨)
. . .
ADD_SUBDIRECTORY( KalmanTrack )
. . .
```

- when *CMake* finds an **ADD_SUBDIRECTORY** it stops execution, enters the directory and looks for a new CMakeLists.txt to execute

2. The CMakeLists.txt in the package subdir declares which libraries and executables to build

```
# Subdir (package) level CmakeLists.txt
. . .
ADD_LIBRARY(KalmanTrack $sources)
ADD_EXECUTABLE( TestKalmanTrack )
TARGET_LINK_LIBRARIES( TestKalmanTrack lib1 lib2 . . . )
. . .
```

Basic structure for *CMake* and FastSim (cont’d)

- I choose to keep in-package CMakeLists.txt file as simple as possible
- All relevant things are inside CMake macros and the main CMakeListsFile.txt
- This means that to add a new package to the cmake system it’s just a matter of putting a template CMakeLists.txt file and eventually add all package specific stuff

```
#
# Set specific compiler flags for the package
#
set(${pkgname}_CXX_FLAGS “-Wall -Wno-sign-compare -Wno-parentheses -fpermissive -DCLHEP_CONFIG_FILE=
set(CMAKE_CXX_FLAGS “${CMAKE_CXX_FLAGS} ${${pkgname}_CXX_FLAGS}”)
configPkg(${pkgname}) # main macro
#
# Generate library
#
add_library(${pkgname} ${Sources})
target_link_libraries(${pkgname} ${LIB_LINK_LIST})

add_executable(exe ${source file})
target_link_libraries(exe ${LIB_LINK_LIST})
add_test(testExe exe)
```

Very quick summary on *CMake* syntax

- # This is a comment
- Commands syntax: COMMAND(arg1 arg2 . . .)
- Lists A;B;C # semi-colon separated values
- Variables
- Conditional constructs
  - IF() . . . ELSE()/ELSEIF() . . . ENDIF()
  - Very useful: IF(APPLE); IF(UNIX); IF(WIN32)
  - WHILE() . . . ENDWHILE()
  - FOREACH() . . . ENDFOREACH()
- Regular expressions

Very quick summary on *CMake* syntax (cont'd)

- ADD_EXECUTABLE
- ADD_LIBRARY
- ADD_DEPENDENCIES(target1 t2 t3) target1 depends on t2 and t3
- ADD_DEFINITIONS(-Wall -ansi -pedantic)
- TARGET_LINK_LIBRARIES(target-name lib1 lib2 . . .) Individual settings for each target
- LINK_LIBRARIES(lib1 lib2 . . .) All targets link with the same set of libs
- MESSAGE(STATUS—FATAL_ERROR message)
- INSTALL(FILES f1 f2 f3 DESTINATION .)
    - DESTINATION relative to ${CMAKE_INSTALL_PREFIX}

Very quick summary on *CMake* syntax (cont'd)

- SET(VAR value [CACHE TYPE DOCSTRING [FORCE])
- LIST(APPEND | INSERT | LENGTH | GET | REMOVE_ITEM | REMOVE_AT | SORT . . .)
- FILE(WRITE | READ | APPEND | GLOB | GLOB_RECURSE | REMOVE | MAKE_DIRECTORY . . .)
- FIND_FILE
- FIND_LIBRARY
- FIND_PROGRAM
- FIND_PACKAGE
- EXEC_PROGRAM(bin [work_dir] ARGS . . . [OUTPUT_VARIABLE var] [RETURN_VALUE var])
- MESSAGE(STATUS | FATAL_ERROR message)