Investigation of a new build system: SCons

Marco Corvo - XI SuperB workshop

December 3, 2009

Why investigate a new build system

- The current build system is based on Makefiles which have become almost unreadable and difficult to improve or even debug
- The same inner organization of the source code is flat, that is all source and header files are in the same directory
- This requires Makefiles to filter them depending on their final usage (e.g. lib files or bin files or root macro files) which is not very efficient.

Why investigate a new build system

- The current build system is based on Makefiles which have become almost unreadable and difficult to improve or even debug
- The same inner organization of the source code is flat, that is all source and header files are in the same directory
- This requires Makefiles to filter them depending on their final usage (e.g. lib files or bin files or root macro files) which is not very efficient.

Why investigate a new build system

- The current build system is based on Makefiles which have become almost unreadable and difficult to improve or even debug
- The same inner organization of the source code is flat, that is all source and header files are in the same directory
- This requires Makefiles to filter them depending on their final usage (e.g. lib files or bin files or root macro files) which is not very efficient.

Why SCons

- SCons is a pure Python tool designed to allow building of software projects without Makefiles
- It can be interfaced to many different "build" tools like gcc or fortran
- The main advantage is that it's written in a fully debuggable language which allows to dig into code to understand build failures
- The other advantages are modularity and flexibility

Why SCons

- SCons is a pure Python tool designed to allow building of software projects without Makefiles
- It can be interfaced to many different "build" tools like gcc or fortran
- The main advantage is that it's written in a fully debuggable language which allows to dig into code to understand build failures
- The other advantages are modularity and flexibility

Why SCons

- SCons is a pure Python tool designed to allow building of software projects without Makefiles
- It can be interfaced to many different "build" tools like gcc or fortran
- The main advantage is that it's written in a fully debuggable language which allows to dig into code to understand build failures
- The other advantages are modularity and flexibility

Why SCons

- SCons is a pure Python tool designed to allow building of software projects without Makefiles
- It can be interfaced to many different "build" tools like gcc or fortran
- The main advantage is that it's written in a fully debuggable language which allows to dig into code to understand build failures
- The other advantages are modularity and flexibility

Packages structure

- The power os SCons comes out when the software project is well organized in distinct packages, each with a clear directory structure, like e.g.

```
/package/src
/package/include
/package/scripts
/package/tests
```

- but this practice is a good one despite of the build system we want to use because it keeps package structure clean and more readable . . .

Packages structure

- The power os SCons comes out when the software project is well organized in distinct packages, each with a clear directory structure, like e.g.

```
/package/src
/package/include
/package/scripts
/package/tests
```

- but this practice is a good one despite of the build system we want to use because it keeps package structure clean and more readable . . .

The heart of SCons

- The heart of SCons is the *Build Engine*, a Python module that manages dependencies between objects
- The *Build Engine* uses a Python API for specifying source (input) and target (output) objects, rules for building/updating objects, rules for scanning objects for dependencies
- to use the *Build Engine* for dependency management we need to interact with it through *Construction Environments*

The heart of SCons

- The heart of SCons is the *Build Engine*, a Python module that manages dependencies between objects
- The *Build Engine* uses a Python API for specifying source (input) and target (output) objects, rules for building/updating objects, rules for scanning objects for dependencies
- to use the *Build Engine* for dependency management we need to interact with it through *Construction Environments*

The heart of SCons

- The heart of SCons is the *Build Engine*, a Python module that manages dependencies between objects
- The *Build Engine* uses a Python API for specifying source (input) and target (output) objects, rules for building/updating objects, rules for scanning objects for dependencies
- to use the *Build Engine* for dependency management we need to interact with it through *Construction Environments*

SCons *Environments*

- A *Construction Environment* is made of one or more associated `Scanner` objects and `Builder` objects
    - A Scanner object specifies how to examine a type of source object (C source file, scripts file) for dependency information
    - A Builder object specifies how to update a type of target object: executable program, object file etc ...

SCons *Environments*

- A *Construction Environment* is made of one or more associated `Scanner` objects and `Builder` objects
  - A Scanner object specifies how to examine a type of source object (C source file, scripts file) for dependency information
  - A Builder object specifies how to update a type of target object: executable program, object file etc . . .

SCons *Environments*

- A *Construction Environment* is made of one or more associated `Scanner` objects and `Builder` objects
  - A Scanner object specifies how to examine a type of source object (C source file, scripts file) for dependency information
  - A Builder object specifies how to update a type of target object: executable program, object file etc ...

Overall SCons features

- Automatic dependency analysis built-in for C, C++ and Fortran, extensible through user-defined dependency Scanners for other languages or file types
- Built-in support for C, C++, D, Java, Fortran, Yacc, Lex, Qt, SWIG, TeX and LaTeX, extensible through user-defined Builders for other languages or file types
- Reliable detection of build changes using MD5 signatures plus optional, configurable support for traditional timestamps
- Support for parallel builds
- Global view of all dependencies (no more multiple build passes or reordering targets to build everything)
- Ability to share built files in a cache to speed up multiple builds similar to ccache but for any type of target file

Overall SCons features

- Automatic dependency analysis built-in for C, C++ and Fortran, extensible through user-defined dependency `Scanners` for other languages or file types
- Built-in support for C, C++, D, Java, Fortran, Yacc, Lex, Qt, SWIG, TeX and LaTeX, extensible through user-defined `Builders` for other languages or file types
- Reliable detection of build changes using MD5 signatures plus optional, configurable support for traditional timestamps
- Support for parallel builds
- Global view of all dependencies (no more multiple build passes or reordering targets to build everything)
- Ability to share built files in a cache to speed up multiple builds similar to ccache but for any type of target file

Overall SCons features

- Automatic dependency analysis built-in for C, C++ and Fortran, extensible through user-defined dependency Scanners for other languages or file types
- Built-in support for C, C++, D, Java, Fortran, Yacc, Lex, Qt, SWIG, TeX and LaTeX, extensible through user-defined Builders for other languages or file types
- Reliable detection of build changes using MD5 signatures plus optional, configurable support for traditional timestamps
- Support for parallel builds
- Global view of all dependencies (no more multiple build passes or reordering targets to build everything)
- Ability to share built files in a cache to speed up multiple builds similar to ccache but for any type of target file

Overall SCons features

- Automatic dependency analysis built-in for C, C++ and Fortran, extensible through user-defined dependency `Scanners` for other languages or file types
- Built-in support for C, C++, D, Java, Fortran, Yacc, Lex, Qt, SWIG, TeX and LaTeX, extensible through user-defined `Builders` for other languages or file types
- Reliable detection of build changes using MD5 signatures plus optional, configurable support for traditional timestamps
- Support for parallel builds
- Global view of all dependencies (no more multiple build passes or reordering targets to build everything)
- Ability to share built files in a cache to speed up multiple builds similar to ccache but for any type of target file

Overall SCons features

- Automatic dependency analysis built-in for C, C++ and Fortran, extensible through user-defined dependency `Scanners` for other languages or file types
- Built-in support for C, C++, D, Java, Fortran, Yacc, Lex, Qt, SWIG, TeX and LaTeX, extensible through user-defined `Builders` for other languages or file types
- Reliable detection of build changes using MD5 signatures plus optional, configurable support for traditional timestamps
- Support for parallel builds
- Global view of all dependencies (no more multiple build passes or reordering targets to build everything)
- Ability to share built files in a cache to speed up multiple builds similar to ccache but for any type of target file

Overall SCons features

- Automatic dependency analysis built-in for C, C++ and Fortran, extensible through user-defined dependency `Scanners` for other languages or file types
- Built-in support for C, C++, D, Java, Fortran, Yacc, Lex, Qt, SWIG, TeX and LaTeX, extensible through user-defined `Builders` for other languages or file types
- Reliable detection of build changes using MD5 signatures plus optional, configurable support for traditional timestamps
- Support for parallel builds
- Global view of all dependencies (no more multiple build passes or reordering targets to build everything)
- Ability to share built files in a cache to speed up multiple builds similar to ccache but for any type of target file

Nice and useful features

- SCons has an internal dependencies tree (a graph), build by Scanners, that can be dumped and analyzed
- A complex software project rarely requires the same level of compiler optimization, debug, etc. options. Hence we need a mechanism allowing to build different targets in different ways: SCons uses Environments
  - Since they are plain Python objects they can be cloned, derived from other Environments and extended
- In principle you could rely on a single SConstruct file (that's a sort of main file for SCons) but for complex projects it clearly useless: SCons' Hierarchical builds
  - Along with a SConstruct file you can define many SConscript ones, namely one for every package in your project

Nice and useful features

- SCons has an internal dependencies tree (a graph), build by `Scanners`, that can be dumped and analyzed
- A complex software project rarely requires the same level of compiler optimization, debug, etc. options. Hence we need a mechanism allowing to build different targets in different ways: SCons uses `Environments`
  - Since they are plain Python objects they can be cloned, derived from other `Environments` and extended
- In principle you could rely on a single `SConstruct` file (that's a sort of `main` file for SCons) but for complex projects it clearly useless: SCons' Hierarchical builds
  - Along with a `SConstruct` file you can define many `SConscript` ones, namely one for every package in your project

Nice and useful features

- SCons has an internal dependencies tree (a graph), build by Scanners, that can be dumped and analyzed
- A complex software project rarely requires the same level of compiler optimization, debug, etc. options. Hence we need a mechanism allowing to build different targets in different ways: SCons uses `Environments`
  - Since they are plain Python objects they can be cloned, derived from other `Environments` and extended
- In principle you could rely on a single `SConstruct` file (that's a sort of `main` file for SCons) but for complex projects it clearly useless: SCons' Hierarchical builds
  - Along with a `SConstruct` file you can define many `SConscript` ones, namely one for every package in your project

Where to go from here for SuperB use cases

- The main doubt about SCons is its scalability
- Most of what I learned about SCons comes from Igor Gaponenko and Andrei Salnikov (Slac people) who use this system to build LCLS software
  - but their project has just order of 20/30 packages to be managed while SuperB scales up to hundreds
- The idea is to:
  - select a few self consistent SuperB packages (to create a sort of independent subproject)
  - clean and reorganize their inner structure (that is create the `/src`, `/include`, `/test`, `/scripts` structure)
  - clean up sources in order to stick to one, maybe two executables and consequently reduce the numbers of libraries they depend on to better understand how SCons manages dependencies

Where to go from here for SuperB use cases

- The main doubt about SCons is its scalability
- Most of what I learned about SCons comes from Igor Gaponenko and Andrei Salnikov (Slac people) who use this system to build LCLS software
    - but their project has just order of 20/30 packages to be managed while SuperB scales up to hundreds
- The idea is to:
    - select a few self consistent SuperB packages (to create a sort of independent subproject)
    - clean and reorganize their inner structure (that is create the `/src`, `/include`, `/test`, `/scripts` structure)
    - clean up sources in order to stick to one, maybe two executables and consequently reduce the numbers of libraries they depend on to better understand how SCons manages dependencies

Where to go from here for SuperB use cases

- The main doubt about SCons is its scalability
- Most of what I learned about SCons comes from Igor Gaponenko and Andrei Salnikov (Slac people) who use this system to build LCLS software
  - but their project has just order of 20/30 packages to be managed while SuperB scales up to hundreds
- The idea is to:
  - select a few self consistent SuperB packages (to create a sort of independent subproject)
  - clean and reorganize their inner structure (that is create the `/src, /include, /test, /scripts` structure)
  - clean up sources in order to stick to one, maybe two executables and consequently reduce the numbers of libraries they depend on to better understand how SCons manages dependencies

Where to go from here for SuperB use cases

- The main doubt about SCons is its scalability
- Most of what I learned about SCons comes from Igor Gaponenko and Andrei Salnikov (Slac people) who use this system to build LCLS software
  - but their project has just order of 20/30 packages to be managed while SuperB scales up to hundreds
- The idea is to:
  - select a few self consistent SuperB packages (to create a sort of independent subproject)
  - clean and reorganize their inner structure (that is create the `/src`, `/include`, `/test`, `/scripts` structure)
  - clean up sources in order to stick to one, maybe two executables and consequently reduce the numbers of libraries they depend on to better understand how SCons manages dependencies

Where to go from here for SuperB use cases

- The main doubt about SCons is its scalability
- Most of what I learned about SCons comes from Igor Gaponenko and Andrei Salnikov (Slac people) who use this system to build LCLS software
    - but their project has just order of 20/30 packages to be managed while SuperB scales up to hundreds
- The idea is to:
    - select a few self consistent SuperB packages (to create a sort of independent subproject)
    - clean and reorganize their inner structure (that is create the `/src`, `/include`, `/test`, `/scripts` structure)
    - clean up sources in order to stick to one, maybe two executables and consequently reduce the numbers of libraries they depend on to better understand how SCons manages dependencies

Where to go from here for SuperB use cases

- The main doubt about SCons is its scalability
- Most of what I learned about SCons comes from Igor Gaponenko and Andrei Salnikov (Slac people) who use this system to build LCLS software
    - but their project has just order of 20/30 packages to be managed while SuperB scales up to hundreds
- The idea is to:
    - select a few self consistent SuperB packages (to create a sort of independent subproject)
    - clean and reorganize their inner structure (that is create the `/src, /include, /test, /scripts` structure)
    - clean up sources in order to stick to one, maybe two executables and consequently reduce the numbers of libraries they depend on to better understand how SCons manages dependencies

Where to go from here for SuperB use cases

- The main doubt about SCons is its scalability
- Most of what I learned about SCons comes from Igor Gaponenko and Andrei Salnikov (Slac people) who use this system to build LCLS software
  - but their project has just order of 20/30 packages to be managed while SuperB scales up to hundreds
- The idea is to:
  - select a few self consistent SuperB packages (to create a sort of independent subproject)
  - clean and reorganize their inner structure (that is create the `/src, /include, /test, /scripts` structure)
  - clean up sources in order to stick to one, maybe two executables and consequently reduce the numbers of libraries they depend on to better understand how SCons manages dependencies