Models de desenvolupament i Gestió de projectes
Source Code Management

Jordi Planes

Departament d'Informtica
Universitat de Lleida

Curs 2010/2011

Scheme

Introduction to Build automation

Definition

Build automation is the automation of usual tasks to build a software system, which consists of:

- Compiling
- Testing
- Packaging

Additionally, it could include also:

- Creation of documentation
- Deployment

Usually, it is a set of scripts.

Advantages

- Improve product quality
- Accelerate the compile and link processing
- Eliminate redundant tasks
- Minimize problems with builds
- Keeps history of builds and releases in order to investigate issues
- Save time and money

Requirements

- Frequent or overnight builds to catch problems early
- Support for Source Code Dependency Management
- Incremental build processing
- Reporting that traces source to binary matching
- Build acceleration
- Extraction and reporting on build compile and link usage

Additionally:

- Generate release notes and other documentation (e.g., help pages)
- Build status reporting
- Testing reporting (passed or failed)
- Summary of the features added/modified/deleted

Build Automation Systems

Selection of software:

- GNU autotools
- CMake
- iMake
- Apache Ant
- SCons

A more extensive list at [Build automation software].

Scheme

GNU Build system

Why we usually do:

```
$ ./configure
$ make
$ make test
$ make install
```

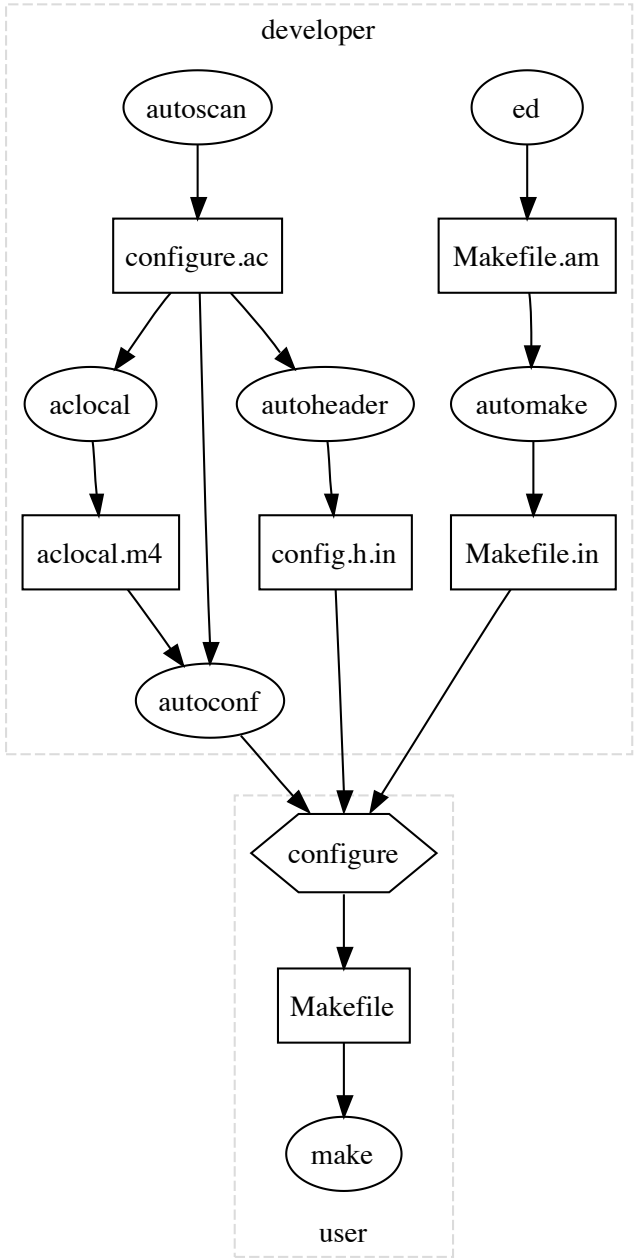
GNU Build system

- Autoconf (configure)
- Automake
- Make
- And other, like: Autoheader

Scheme

Autoconf

- GNU Autoconf is a tool for producing configure scripts
- A configure script configures a software package for installation on a particular target system
- After running a series of tests, it generates header files and a makefile from templates


developer
autoscan
ed
configure.ac
Makefile.am
aclocal
autoheader
automake
aclocal.m4
config.h.in
Makefile.in
autoconf
configure
Makefile
make
user

Autoconf
Advantages

- the configure script can get reasonable results on newer or unknown systems
- it allows administrators to customize their machines and have the configure script take advantage of the customizations
- there is no need to keep track of minute details of versions, patch numbers, etc., to figure out whether a particular feature is supported or not

Autoconf I
Criticisms

- General complexity of used architecture, most projects use multiple repetitions
- Generated 'configure' is written in Unix shell and thus Makefile generation is slow. Some projects, such as KDE, tried to work around this issue by introducing distinct Makefile generators written in other languages
- Generated 'configure' uses only manual-driven command-line interface without any standardization
- M4's behavior is cryptic and thus hard to debug
- Weak backward and forward compatibility
- Cannot be used when building under Mac OS X using Xcode, or under Windows using Visual C++
- Autoconf-generated scripts are large and cryptic, making debugging and hand tuning difficult

Autoconf II
Criticisms

- Autoconf makes it hard to port software to exotic architectures as it keeps a list of supported architecture in the config.sub file distributed with each software package, and will not run for an architecture which is not listed.

Autoconf
Usage overview

The developer specifies the desired behaviour of the configure script by writing a list of instructions in the GNU m4 language in a file called configure.ac

Autoconf transforms the instructions in configure.ac into a portable configure script.

Autoconf
configure.ac format

AC_PREREQ(version) macro can be used to ensure that a recent enough version of the autoconf program is available

AC_INIT(package, version, bug-report-address) This macro is required in every configure.ac file. It specifies

- name and version of the software package
- the email address of the developer

information on the package

checks for programs, for libraries, for header files, for types, for structures, for compiler characteristics, for library functions, for system services

AC_CONFIG_FILES(file...)

AC_OUTPUT

Scheme

Make

- Make tries to create a target [Make]
- It follows the intrusctions from a Makefile
- For every target, there are a set of sources
- It checks whether each source has been modified after the target. In such a case, the target is re-made

Advantages and disvantages

- Dependencies created and maintained by hand → Several tools created to automatize the process
- Platform dependant options
- When using a version control system, file dates can be in the past
- Syntax: tab and whitespaces look the same → use of dedicated editors
- Syntax: declarative programming oriented (vs. imperative programming)

Make
Structure

Comments are started with the hash(#) symbol.
target [target ...] : [component ...]
$command_1$
.
.
.
$command_n$

- Consists of dependency lines which define a target followed by a colon and, optionally, a set of sources
- After, optionally a set of lines tab-indented which define how to transform the set of files into the target
- Multiple targets are allowed

Make
Structure

Comments are started with the hash(#) symbol.
target [target ...] : [component ...]
command$_1$
.
.
.
command$_n$

- the set of commands could be empty mrproper : clean doc−clean
- To create a target $ make target
  By default, the first target in the Makefile
- Any line can be split with \
- To skip command printing: @

Make
Macros

- File inclusion include file
- Conditional ifeq (first , second)
  Command
  endif

Make
Special targets

- Not real targets: .PHONY : list of targets
  e.g. clean, run, dirs, ...
- Suffixes used: .SUFFIXES : list of suffixes
- Default action to make: .DEFAULT : action

Wildcards

- For a command rm −f ∗.o
- For a source print : ∗.c
- For a set obj_files = ∗.o
- As a function obj_files = $(wildcard ∗.o)

Make
Variables

- We can deal with variables
- Assignment by value Variable:=Value
- Assignment by command Variable :=‘command‘
- Assignment by function Variable:=$(function)
- Getting the value $(Variable)
- Concatenation +=

Make
Predefined variables

- Inside a rule
  - target: $@
  - first source: $¡
  - all sources: $∼
  - modified sources: $?
- Global
  - Looking for sources: VPATH
  - Command remove: RM
  - Command make: MAKE
    Make could be recursively called with make −C makefile
  - C language: CC, CXX, CFLAGS, LFALGS, CCFLAGS

Functions

Common functions

String: substr patsubst strip findstring filter sort word wordlist firstword, lastword

Files: dir, notdir, suffix, basename, addsuffix, wildcard, realpath, abspath

Conditionals: if, or, and

Other: foreach, call, value, eval, shell

Implicit rules

- There are rules not needed to be defined
- In this case, there is no need to state the rules for .o files:

```
target : source1.o source2.o
    $(CC) -o target source1.o source2.o $(CFLAGS)
```

- Languages with implicit rules: C, C++, Pascal, Fortran, Modula-2, Assembler, Linking a single object file, Yacc for C, Lex for C, TEXand Web, Texinfo, ...
- Implicit rules work with variables for programmes: AR, AS, CC, CXX, CPP, FC, M2C, PC, CO, YACC, TEX, MAKEINFO, ...
- and variables for arguments: ARFLAGS, ASFLAGS, CFLAGS, CXXFLAGS, ...
- There can be chains of implicit rules. e.g. Yacc $\rightarrow$ C $\rightarrow$ Object $\rightarrow$ Executable

Pattern

- A rule defined for a set of targets
  [TARGETS ...:] TARGET−PATTERN: SOURCE−PATTERN ...
  Commands
- The patterns could contain the stem character %
  $(objects) : %.o : %.c
  $(CC) −c $(CFLAGS) $< $@

Scheme

Automake

GNU Automake is a programming tool that produces portable makefiles for use by the make program
The makefiles produced follow the GNU Coding Standards.

Automake

Automake aims to allow the programmer to write a makefile in a higher-level language, rather than having to write the whole makefile manually.
In simple cases, it suffices to give:

- a line that declares the name of the program to build;
- a list of source files;
- a list of command-line options to be passed to the compiler (for example, in which directories header files will be found);
- a list of command-line options to be passed to the linker (which libraries the program needs and in what directories they are to be found).

Automake

From this information, Automake generates a makefile that allows the user to:

- compile the program;
- clean (i.e., remove the files resulting from the compilation);
- install the program in standard directories;
- uninstall the program from where it was installed;
- create a source distribution archive (commonly called a tarball);
- test that this archive is self-sufficient, and in particular that the program can be compiled in a directory other than the one where the sources are deployed.

Automake

Automake also takes care of automatically generating the dependency information, so that when a source file is modified, the next invocation of the make command will know which source files need to be recompiled. If the compiler allows it, Automake tries to make the dependency system dynamic: whenever a source file is compiled, that file's dependencies are updated by asking the compiler to regenerate the file's dependency list. In other words, dependency tracking is a side effect of the compilation process.

Scheme

References

- http://en.wikipedia.org/wiki/Build_automation
- http://en.wikipedia.org/wiki/List_of_build_automation_software
- http://en.wikipedia.org/wiki/Make_(software)
- http://en.wikipedia.org/wiki/GNU_Automake