The Rules build system based on GMake

Paolo Walter Cattaneo

INFN Pavia, Via Bassi 6 - 27100, Pavia, Italy
Email: Paolo.Cattaneo@pv.infn.it

Abstract

The necessity of using a maintenance system based on the UNIX make utility in any non trivial software project is widely recognized. Nevertheless, the writing of Makefiles for each part of an evolving software project can be difficult, tedious and error prone. Therefore a higher level build system based on the GNU Make program (GMake) is proposed that relieves the user from most the task of Makefile writing. The resulting system is easy to use, still remaining flexibile and powerful.

1 Why Make?

Make is a program developed under Unix to help the programmer to manage his/her project. That is done providing in a file, usually called makefile, a set of rules to relate a **target** (most often, but non necessarily, a file) with **dependencies** (again most often, but non necessarily, files), providing rules to build the target from the dependencies. Make checks the latest times at which the dependencies were modified and the rules are run only when the target is older that some of the dependencies. A careful explanation of the Make functionality and goals is provided in [1].
All Unix flavour and some non-Unix systems have their own Make programs, but portability is limited. Instead the GNU Makefile (GMake) has been designed to run on all major platforms and implements several useful extensions [2].
Assuming that the software project is a traditional programming projects, in which some source code files (that, for sake of example, are assumed C++ code in the following) are trasformed into object files and then linked together in executables and/or libraries, it is possible to write explicitly rules for each target file listing all the dependencies in each step of the compile-linking chain and providing the necessary rules.
The disadvantages of this approach are obvious: for any non trivial project, large Makefiles must be written, that need to be continuosly updated for any modification of the file structure of the projects, like adding or removing source code files or include files, or of the compiling-linking conditions, like flags and libraries.
A partial solution to this problem is provided within the Make program itself, with the implicit rules that provide built in rules for several standard target-dependencies relations, so that, for example, the rule for compiling a C++ source file .cc in a object .o file is provided. In [2] the list of the implicit rules and of predefined names is detailed. The names relevant for C++ are the compiler name, CXX, the compilation flag, CXXFLAGS, the link flags, LDFLAGS, the archive program, AR.

2 The Rules package

Basing on the experience of a previous experiment (NOMAD), I have developed a package (Rules) that allows the exploitation of Make potential with good flexibility and ease of use. It is centered around three files: Compiler.defaults, Make.deftargets and Make.rules located in the directory $RULESSRC. In Compiler.defaults, platform and compiler dependent variables are defined depending on the architecture determined using *uname*: the compiler name, the compiler flags, the link program and some library.
The languages supported are C, C++, F77, F90.
The platforms supported are Compaq Alpha, SunOs, Solaris, HP-UX, IBM-AIX, Linux.
The file Make.rules includes the list of dependencies that can be associated to targets. The file Make.deftargets defines all the rules that, together with the built in rules, govern the relations between target and dependencies.
The user must define the name and the type of the target. Starting from source files, the choice is between an executable, an archive or a shared library; starting from Tex files the only type is a postscript file.

2.1 The compilation targets

The package is better explained through examples. We start with an example of directory tree for building an executable and an archive library. A characteristic directory organization of the top directory of the package (using CVS) is:

```
CVS/
Linux/
Makefile
doc/
include/
lib/
src/
```

The Makefile of this package consists only of

```
-include $(RULESSRC)/Make.rules
-include $(RULESSRC)/Make.deftargets
```

This Makefile has no target and its only task is looping on all the subdirectories not belonging to the list NOUSE, defined in Make.rules, by

```
NOUSE=CVS doc jobs main $(BIN_DIR) $(LIB_DIR) Rules ps input output
```

where BIN_DIR and LIB_DIR are the directories where executable and libraries are stored (by default the name of the architecture as returned by uname). These directories are excluded either because they are directories created by Make itself where no Makefile is present or because no building is required in these directories. An exception is doc, where often documentation files are kept, that might need building, but that does not need to be run every time. When documentation is to be

remade, "make doc" will do it.
Therefore Make do not descend in CVS, Linux and doc. It runs the Makefile in include, where the relevant .h are kept and where nothing is to be rebuilt, so that another do-nothing Makefile is sufficient.
In lib the source code files (.cc in this case) are kept, they must be compiled and linked together to give an archive library to be put in LIB_DIR. The Makefile reads

```
-include $(RULESSRC)/Make.rules
PRODLIB=MyLib
-include $(RULESSRC)/Make.deftargets
```

The line with PRODLIB gives a target of type archive library and name MyLib, so that the .cc file are compiled by CXX and then archived by AR.
A list of dependencies of each file *.cc in lib is generated through the script gendepend (also in $RULESSRC) that calls the standard program makedepend. The resulting list is put into a file with the same stem and the extension .d and all these files are automatically included in the Makefile. Appropriate rules of dependency between *.d and *.cc regenerate automatically these files when the source files are modified.
Alternatively a shared object library is obtained setting PRODSO=MyLib.
In the directory src the Makefile reads

```
-include $(RULESSRC)/Make.rules
PROD=elefdsim
-include $(RULESSRC)/Make.deftargets
```

The line with PROD gives a target of type executable and name elefdsim in $BIN_DIR. The dependencies files *.d are produced as well. At variance with the PRODLIB target, the object files *.o, that are intermediate targets, are not removed.

2.2 The auxiliary packages

There are a set of packages that may be used by some projects but not necessarily by all, that might be included in the Rules compile and linking sequence.
They are :

- CERNLIB, GEANTLIB
- QT
- CLHEP
- ROOT
- XERCESC

In order to include any of them, an appropriate environmnet variable needs to be set.
They are in order

- CERN: top directory of the CERN area (e.g. /cern)

- QTLIB: top directory of the QT area (e.g. /usr/lib/qt-2.3.1)
- CLHEPHOME: top directory of the CLHEP area (e.g. /usr/local/lhcxx/specific/redhat73/gcc-3.2/CLHEP/new)
- ROOTSYS: top directory of the ROOT area (e.g. /usr/local/root/v3.05.07/rh73_gcc32/root)
- XERCESCROOT: top directory of the XML reader XERCES (e.g. /usr/local/lib/xerces-c-src2_0_0)

Only the packages with the environmnet variable set are used by Rules. The others are ignored.

2.3 The include and library directories

The system identifies the default include directories and libraries required for compiling and linking. The standard include list is defined in Compiler.defaults by the local include directories; the directories are

```
. ./include/ ../include/ ../../include ../../../include ../*/include
../../*/include -I/usr/X11R6/include
```

This list is added the include directories of the auxilary packages required.

2.4 Modifying default variables

The default variables can be substituted or modified by the user for either using the command line or adding lines to the local Makefile, between the includes. The modification applies for the directory where the Makefile belongs to and all its subdirectories.
The most common case is the addition of an include directory to the compile command or a library to the link phase. That can be done either through the command line or modifying the Makefile file. In the first case, gmake is run as

```
gmake userincludes=my_inc userlibs=my_libs
```

in the second, a line must be added in the Makefile as

```
userincludes=my_inc
userlibs=my_libs
```

where my_libs may be a single library or a directory preceded by -L and followed by a list of libraries. Multiple include directories or multiple libraries need to be white spaced.
Other variables can be modified in the same way, e.g. any of the package environemnt variable previously defined.
Another commom case is excluding the directory mydir from the compilation. That is obtained with the statement "NOUSE+=mydir".
Excluding the file my_file from compilation is obtained with "FILENOUSE=my_file.o".
Limiting the compilation to a subset of directories from the command line, is obtained writing

```
gmake my_dir1 my_dir2
```

A subset of files is compiled with

```
gmake my_fil1.o my_fil2.o
```

2.5 The documentation targets

2.5.1 The doxigen targets

PRODDOX is a specific target related to the use of doxigen as documentation tool. The Makefile should look like

```
-include $(RULESSRC)/Make.rules
PRODDOX=doxigen
NOUSE+=html latex
-include $(RULESSRC)/Make.deftargets
```

and usually resides in a doc directory with a doxygen_config file.
PRODDOX is a dummy target with no dependencies and runs the following command

```
doxygen doxygen_config
```

every time make is executed. The line with NOUSE prevents the attempt of runnning non existing Makefile in the doxigen automatically generated directories.
This configuration allows to run the doxigen documentation from the directory above with gmake doc.

2.5.2 The TEX targets

The target PRODTEX builds a postscript file from a TEX with the same name. Some scripts located in $RULESSRC creates for each TEX file in the directory a .dtex file with the list of the postscript, encapsulated postscript, tex and bibliography files required for compilation. These files are searched in the usual include pattern given by INCLUDES.
The final postscipt file is created via an intermediate .dvi file geneareted by running latex and bibtex in the appropriate order.
For reason of space, the postscript and encapsulated postscript files are usually stored compressed in cvs as .Z (compress) or .gz (gzip). There are intermediate rules that decompress these files for inclusion and then remove them.

2.5.3 The PDF targets

The target PRODPDF builds a .pdf file from a TEX with the same name.postscript. It uses as intemediate target the postscipt file produced with the previous rule.

3 ROOT rules

The use of ROOT or, more specifically, rootcint, the C++ interpreter, poses some very specific requirements on the building procedure. Rootcint requires the creation of a C++ file and a header file, the Dictionary files, from some header files: the ones containing classes to be streamed and the LinkDef file. Furthermore the source code files containing the implementations of these classes need to be put into a shared objects library [3].
The solution to these problems consists in the target PRODCINT that triggers the building of a shared object. This target looks for files of name xxxLinkDef.h and xxxDict.cc and tries to build the shared object from xxxDict.cc and those xxx.cc containing the ROOT classes implementation (ClassImp). If the Dict file is missing or out of date, it is rebuilt by an intermediate rule running rootcint on xxxLinkDef.h and those xxx.h containing the ROOT classes declaration (ClassDef).
These rules, even if they are a litte abstruse, guarantee the correct building of up to date shared object and executable at the same time.
This Makefile generates a shared obeject and an executable with the name package from the include and C++ source file available.

```
-include $(RULESSRC)/Make.rules
PRODCINT=package
PROD=package
-include $(RULESSRC)/Make.deftargets
```

If the Makefile is in a directory library, substituting PROD=package with PRODLIB=package, an archive library is produced together with the shared object for ROOT.

4 Keeping order

The compilation steps and the mechanism for keeping tracks of dependencies generate many intermediate files, both for executable and library creation and for postscript creation, e.g. the object files, the .d and .dtex files. Some of them are removed automatically during generation of the target, if they are not required by the rules governing the generation of the target. Other files, line the .d and .dtex, are such that if they are not present, the target is always regenerated every time make is run. Hence normally they are not removed.
Nevertheless it might be necessary to remove them. That may happen because a fresh start of the compilation is needed, because a directory developed outside CVS must be imported or because the directory needs to be tarred for distribution.
Rules supports several instructions for cleaning these files:

- texclean Cleaning only intermediate files related to the generation of .ps from .tex
- locclean Run texclean plus cleaning intermediate files related to compilation
- dirclean Run locclean recursively on subdirectories
- clean Run locclean plus removing target files and directories recursively

5 Project makefiles

In a software project it is often the case that some addition is needed to the files Make.deftargets and Make.rules. The variables can be redefined or integrated with extra values. Furthermore a not-yet supported platform may need to be included.
Rather than modifying the files in $RULESSRC or including in the project makefiles extra files, the solution consists in defining a new environment variables $PROJRULES, pointing to a directory with two files Project.deftargets and Project.rules. If $PROJRULES is defined and if these files are present, they are included at the end of Make.deftargets and Make.rules respectively.
If the definitions in Project.rules are relevant only for a specific operating system OpSys (name obtained by uname) or if they have to be defined for a new one, they have to be surrounded by the if clause

```
ifeq ($(ARCH), OpSys)
.........
endif
```

Some of the makefile variables are not initialized in Make.rules and is a user task to do so. They are

```
FLAGSFC      Extra flags for the F77 compiler
FLAGSF90C    Extra flags for the F90 compiler
FLAGSFCC     Extra flags for the C compiler
FLAGSFCXXC   Extra flags for the C++ compiler
FLAGSLD      Extra flags for the linker
userincludes User include patterns
userlibs     User libraries
```

Other variables can be redefined by the user in Project.rules if the default needs to be changed. The list with the default with Linux is

```
   FC = g77             F77 compiler
   F90 = f90            F90 compiler
   CC =  gcc            C compiler
   CXX = g++            C++ compiler
   LEX = flex           Lex compiler
   FFLAGS =             F77 flags
   OPT_FFLAGS = -O      Optimization F77 flags
   DBG_FFLAGS = -g      Debug F77 flags
   CFLAGS =             C flags
   OPT_CFLAGS = -O      Optimization C flags
   DBG_CFLAGS = -g      Debug C flags
   CXXFLAGS =           C++ flags
   OPT_CXXFLAGS = -O    Optimization C++ flags
   DBG_CXXFLAGS = -g    Debug C++ flags
```

OS	F77	F90	C	C++	Lex
OSF1	f77	f90	cc	cxx	lex
SunOS	f77	f90	cc	CC	lex, flex
HP-UX	f77	f90	cc	aCC	lex
AIX	xlf, g77	xlf	xlc, gcc	xlc, g++	lex, flex
Linux	g77		gcc	g++	flex

Table 1: Operating system and compilers supported by Rules

```
F90FLAGS =           F90 flags
OPT_F90FLAGS = -O    Optimization F90 flags
DBG_F90FLAGS = -g    Debug F90 flags
F90DEFFLAGS  = -m vo Flags for F90 dependencies
LFLAGS =             Lex flags
link_F = $(FC)       Linker for FC
link_F90 = $(F90)    Linker for F90
link_C = $(CC)       Linker for CC
link_CXX = $(CXX)    Linker for CXX
LDFLAGS =            Linker flags
F77LIBS =  -lg2c     Libraries for linking Fortran files with link_C
SOFLAGS =  -shared   Shared object flags
otherlibs =          Extra libraries
```

6 Distribution

Even if the most popular and powerful distribution method is through CVS, it is occasionally useful to produce tar files for distribution.
In a given directory running

```
gmake tar
```

will produce a compressed tar file. It can be later decompressed running tar xzvf xxx.tar.

7 Conclusions

A brief explanation of the Rules package for code building is presented. The explanation is oriented for potential package users not for potential developers who, of course, can be developers of the source code.

Appendix

The operating systems and compilers supported by Rules are summarized in the table Tab.1.

References

[1] A. Oram and S. Talbott. *Managing projects with Make.* O'Reilly & Associates, Inc, Sebastopol, CA, 1993.

[2] R.M. Stallman and R. McGrath. Gnu make. Published by Free Software Foundation.

[3] R. Brun et al. Root, users guide, 3.1c. Available from root.cern.ch.