Make-system evaluation for Hall-D software

D. Lawrence
Jefferson Lab

May 13, 2004

1 introduction

In this document I present an evaluation of "Make-systems" which could be used for the Hall-D software. A make-system is the software used to orchestrate the compiling and linking of executable programs and libraries from the source code. A good make system is essential for efficient software development of any large software project.

Currently, the Hall-D software imploys the GNU Makefile system(gmake) implemented through a series of independant Makefiles. The gmake system is by far the most commonly used make system for the platforms commonly used in experimental particle physics. Because the Hall-D software was basically built one program at a time, it does not employ a common system throughout, leading to situations which will grow into hard to manage problems over time. Some of these are:

> Large overlap in the contents of the makefiles. This comes about from each directory having its own makefile which implements platform-dependant tweaks. If a minor change is needed (due to say, a new verison of RedHat), then the fix would need to be added to many makefiles.
>
> No convienient mechanism for passing variables. Basically, if you have a makefile in one directory invoke a makefile in a subdirectory, the variables are not passed down the chain which could lead to inconsistancies.
>
> Non-standard naming. Currently, some packages use "GNUMakefile" (the recomended name by the makers of gmake) while other use "Makefile" (also accepted by gmake and much more commonly used).

Before simply implementing a system based on gmake, a quick evaluation was made to explore what other options exist. The gmake system has a few shortcomings which might be overcome by newer systems. A simple program was used to evaluate three make systems and compare them. A review of documentation for a few others was also done. This note presents the results of this study.

2 Evaluation Program

Before begining the evaluation, a simple source tree was constructed with which to test the systems. An outline of the directory/file structure is shown in figure 1. Note that this is not intended to be an exact represenatation of the directory sturcture the Hall-D software will have. In particular, the bin and lib directories will likely not reside in the "src" directory. However, it served for the purposes of this test.

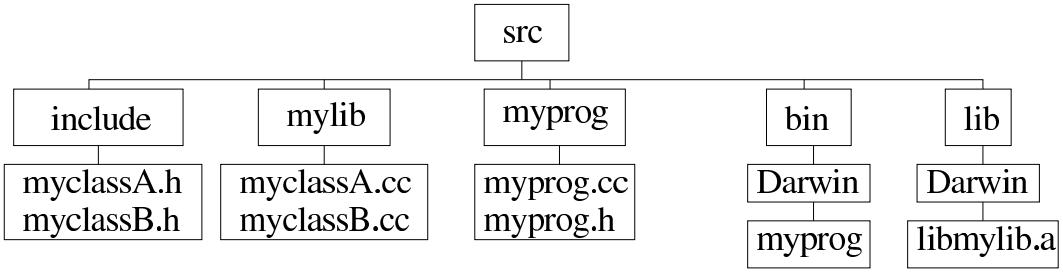

Figure 1: Directory tree of source code used in evaluations. ("Darwin" is the uname for Mac OSX. The environment variable OSNAME was used to identify the platform directories for the binaries.)

The source code itself is quite simple and is included in Appendix A for completeness.

3 gmake

GNU Make or "gmake" is the clearly accepted standard for software development on Linux. It takes a "Makefile" as input which defines a set of rules that gmake uses to build the binaries.

Here is the top-level Makefile used:

```
LIBS = -Llib/$(OSNAME) -lmyprog

bin/$(OSNAME)/myprog:
make -C mylib -f ../Makefile.common
make -C myprog -f ../Makefile.common
ranlib lib/$(OSNAME)/libmyprog.a
$(CXX) $(CXXFLAGS) $(LIBS) -o $@
```

I did this in a slightly unconventional way by not actually putting a makefile in the mylib or myprog directories. They both used the exact same makefile,

"Makefile.common", which was explicitly referenced in the top level Makefile. Here is the Makefile.common file:

```
CXXFLAGS = -I../include
CXXSRC = $(wildcard *.cc)
OBJS = $(CXXSRC:.cc=.o)

all:../lib/$(OSNAME)/libmyprog.a($(OBJS))
../lib/$(OSNAME)/libmyprog.a($(OBJS)):
```

Here are some observations about gmake:

gmake has a lot of implicit (built-in) rules for compiling and linking which can be very powerful when used properly.

gmake does a good job of building long chains of rule dependancies which can create any number of intermediate files. The intermediate files are removed automatically making for a cleaner build environment.

There is no underlying programming language to gmake. If you really want to do something fancy, you need to put it in some other executable and invoke it as a shell command from gmake.

gmake has no built in dependancy scanning (include files). The documentation recommends writing a rule based on gcc's -M or -MM flags that turns into a much more complicated mess than should be neccessary.

There is really no easy way to pass the build environment from one makefile down to another. Specifically, if the common method of using "make -C subdir" is employed, the build environment is not passed to the Makefile in "subdir".

Summary: The gmake system is ubiquitous. It is installed on every Linux system which has a C/C++ compiler installed. It is the most commonly used make system in particle physics. This makes it the yardstick by which other make systems are measured. The single biggest drawback to gmake is the lack of an implicit rule for dependancy scanning. Every reasonably large software project which uses gmake must overcome this in basically the same way.

4 CONS

CONS is based on perl and has the capability to include perl code in the configuration files. The top level "Construct" file looked like:

```
$BINDIR = "$ENV{'PWD'}/bin/$ENV{'OSNAME'}";
$LIBDIR = "$ENV{'PWD'}/lib/$ENV{'OSNAME'}";

$env = new cons(
CC => 'gcc',
CXX => 'g++',
CPPPATH => 'include',
LIBPATH => $LIBDIR,
LIBS => '-lmylib',
);

Export qw(env BINDIR LIBDIR);

Build qw(

myprog/Construct
mylib/Construct
);
```

And here is the Construct file in mylib looked like:

```
$core_name = mylib;
$libname = "lib$core_name.a";

Import qw(env BINDIR LIBDIR);

$csrc = `cd $core_name; ls *.cxx`;
Library $env  $libname, 'myclassA.cxx', 'myclassB.cxx';

Install $env $LIBDIR, $libname;
```

Here are a few observations about the CONS make system:

The tool was fairly easy to configure and understand. A top-level "Construct" file was only about 13 lines long. The subdirectory construct files only needed to be about 3-4 lines long. However, they were not generic in that file names were given explicitly, but presumably, one could use perl code to produce a list of files.

The system was clearly set up to do C/C++ programming with no obvious way to extend to other languages (like fortran). In fact, the system required a .cxx extension for c++ files(it invoked the C compiler for .cc). There is apparently no way to modify the suffix list either.

The built-in cleaning mechanism (cons -r) did not clean up all of the intermediary files. For example, invoking "cons myprog" would correctly compile the source in the mylib and myprog directories. However, "cons -r myprog" would not delete the lingering object and library files left in the mylib directory. Nor did it delete the library in the lib/Darwin directory.

Lots of lingering objects were left. The "Install" method only copied the final library file to its install directory. This resulted in three copies of the object files on the disk (the .o file, the one stored in mylib/libmylib.a and the one in lib/Darwin/libmylib.a). There is apparently no built-in mechanism for keeping objects only in library archives, or deleting intermediary files at the end of the build.

Documentation was basically one long webpage. It was sufficient to get going, but not great.

Summary : Using perl as a platform for this gives you the powerful flexibility of a programming language. This means there is certainly a way to overcome the shortcomings of cons by programming around it. But then, what's the point? CONS has one main advantage over gmake which is built-in dependancy scanning (looking for include files and rebuilding if they change). This may be a debatable point though. See the conclusion for a discussion. Another advantage CONS has over gmake is the ability to pass the build environment (i.e. CC, CXX, LD ,...) easily down through the heirachy of Construct files. The lack of housekeeping is a big minus IMHO for CONS.

5 SCONS

SCONS is a make utility very similar to CONS. SCONS, however, is built on python as the underlying language. The top level "SConstruct" file looked like:

```
import os
BINDIR = os.environ["PWD"]+'/bin/'+os.environ["OSNAME"]
LIBDIR = os.environ["PWD"]+'/lib/'+os.environ["OSNAME"]

env = Environment(
CC = 'gcc',
CXX = 'g++',
CPPPATH = '../include',
LIBPATH = LIBDIR,
LIBS = '-lmylib'
)

Export('env', 'BINDIR', 'LIBDIR');

SConscript(['myprog/SConstruct',
```

```
            'mylib/SConstruct'])
```

And here is the SConstruct file in mylib looked like:

```
Import('env', 'BINDIR', 'LIBDIR')

list = Split('myclassA.cc myclassB.cc')
env.Library('mylib', list);

env.Install(LIBDIR, 'libmylib.a')
```

Here are a few observations about the SCONS make system:

The SCONS philosphy seems to map almost exactly with that of CONS. It seems the basic differences are just in syntax. With relatively minor editing, I was able to convert the "Construct" scripts for CONS into "SConstruct" scripts for SCONS.

SCONS suffers from the same limitations in regard to lingering objects that CONS does. See the CONS review for details.

Unlike CONS, SCONS used the default target if none was specified on the command line. This may be a minor point, but with CONS, you had to type "cons myprog" while with SCONS, I only needed to type "scons".

The documentation for SCONS seemed a bit better than for CONS. There were sections though that were clearly taken straight from the CONS manual.

Summary : SCONS might have a few slight advantages over CONS, mainly in the documentation. The choice for an individual programmer though, would probably be driven by whether they liked perl or python better since they seem so philosophically ans structurally similar. The same arguments given for/against CONS would therefore apply here as well.

6 Other Make Systems

Several other make systems exist. A nice list can be found on the web at *http://directory.google.com/Top/Computers/Software/Build_Management/Make_Tools*. I found at least one other web page which listed make tools, but it seemed to be a subset of this one.

Here's a summary of observations gathered from the documentation of several of these packages (note that only freeware packages were considered):

Ant This is an incredibly popular make tool apparently used by most Java programs. The documentation does not mention any restrictions such as "it can only be used to build java apps". It also does not mention using it for anything other than Java. This is clearly a Java tool which has not yet had any real effort put into making it a general use make system which could be used for C/C++.

NAnt This is an Ant port to compile other types of code, but is built on C# and .NET technologies which are widely used on Windows, but have not yet gained much footing in Unix.

A-A-P This could be a promising system. It is based on python. It may eventually suffer from trying to be too much. It tries to implement version control and web publishing. It does however, seem to be well developed with a respectable user base. More study may be required here.

Cook This is package implements its own file format which they call "cookbooks". The documentation is a single PDF file which was clearly derived from a set of man pages. It does not appear to have a tutorial. It was apparently written and is maintained by a single person without much of a user base.

Jam This is an open source package created by Perforce Software. On the jam homepage, it gives an example of how a jam file compares to a makefile. It is misleading because the makefile example does not use any implicit rules, making it look much more complicated than the one line jamfile. Aside from that, it appears to have all of the features of gmake with the addition of some flow control statements which allow for some limited programming.

MakeXS A modified GNU make which uses the M4 language. The documentation for makexs admits that it's documentation is a bit terse and that one needs a pretty good understanding of gmake and M4 to really get use out of it. This seems more of a competitor for autoconf and imake than for gmake.

jmk This is another java make tool. It apparently is built on make.

PVM GNU Make This is another modified version of GNU Make. Its main goal appears to be "parallel, distributed build". This is not expectded to be the driving feature for Hall-D software.

Boost Build Looking through the tutorial, this looks as though it is a variant of the Jam system (the command is "bjam" and the default configuration file is named "Jamfile"). The logo touts it as "C++ Boost" which leads one to believe it is really geared toward C++.

7 Conclusion

A few make systems were evaluated (GMAKE, CONS, and SCONS) and several others were briefly reviewed. As was pointed out earlier, the gmake system is the most commonly used, by far in the Unix environment. It is also the only one that comes standard on any Linux system with a C/C++ compiler installed. As such it is the default choice unless significant benefit can be gained from another package. The benefit must be measured by a single parameter: How many manhours the collaboration, as a whole, spends on configuring and using the make system. Anything other than gmake already carries a deficit just in the time it takes to download and install a non-standard system on each computer that the Hall-D software will be compiled on. Any problems encountered by a collaborator will either trigger them spending a day to learn system-X in order to fix it, or an exchange of e-mails to have an expert do it. This is, of course, a little hard to predict.

The biggest shortfall of gmake is its implimentation of dependancy scanning. This is also where other systems seem to surpass it. Gmake does have a recommended method, however which relies on using gcc or g++ with the **-M** or **-MM** parameter.

Most other packages (Jam perhaps being the exception) don't do as good of a job in housekeeping as gmake. Lingering objects not only uneccessarily clutter directories, but have the potential of causing link errors.

My reccomendation is to use gmake. This is capable of doing what is required with sufficient compententcy. The advantages of the other systems do not appear suffcient to warrant deviating from the standard. The one reservation is that the makefiles must be thoughtfully written and centralized. See the discussion below for some ideas along this line.

8 Considerations for the Hall-D make system

Here are some things to consider for the Hall-D make system:

Make program builds independant. This may seem obvious, but if you are working on a program and just want to rebuild it, you should not have to issue a make which spans the entire directory heirachy.

Wherever possible, use includes rather than separate invocations. In particular, try to use "include (subdir/Makefile)" rather than "make -C subdir". This allows gmake to build a more complete dependancy tree. This may be difficult to resolve with the above point.

Use a standard naming convention. I'd suggest going against the recommendation by the makers of gmake to "GNUMakefile" and use the more common "Makefile" instead. This still allows one to type simply "make" without having to explicitly specify the configuration file name.

The system should be centralized. Virtually all Makefiles should consist of a single include statement. This makes it much, much easier to propagate changes to the entire source tree.

Avoid autoconf. Autoconf is widely used, but with a relatively small pool of experts. Its main job is to handle platform-specific dependancies. Most common packages now come with utilities which handle this for you and can be easily incorporated in the makefile. See, for example, *root-config*, *mysql_config*, or *cernlib*.

A Appendix: Source code

myprog.h

```
#include "myclassA.h"
#include "myclassB.h"
```

myprog.cc

```
#include "myprog.h"

int main(int argc, char argv[])
{

    myclassA mcA;
    mcA.DoA("classA message");

    myclassB mcB;
    mcB.DoB("classB message");

    return 0;
}
```

myclassA.h

```
class myclassA
{
    public:
    myclassA();
    ~myclassA();

    int DoA(char *mess);
};
```

mclassA.cc

```
#include <iostream>
#include <iomanip>
using namespace std;

#include "myclassA.h"
```

```
myclassA::myclassA()
{
}

myclassA::~myclassA()
{
}

int myclassA::DoA(char *mess)
{
    cout<<mess<<endl;

return 0;
}
```

---

myclassB.h

```
class myclassB
{
    public:
    myclassB();
    ~myclassB();

    int DoB(char *mess);
};
```

---

myclassB.cc

```
#include <iostream>
#include <iomanip>
using namespace std;

#include "myclassB.h"

myclassB::myclassB()
{
}

myclassB::~myclassB()
{
}
```

```
int myclassB::DoB(char *mess)
{
    cout<<mess<<endl;

return 0;
}
```