BACHELOR INFORMATICA
UNIVERSITEIT VAN AMSTERDAM

INFORMATICA — UNIVERSITEIT VAN AMSTERDAM

Autoconfiscation of Imake Configured Projects

Ivo Tamboer
0306088

August 29, 2007

Supervisor(s): Bob Diertens, Inge Bethke

Signed: ____________

Contents

CHAPTER 1

Introduction

This document is a report on my findings in creating a conversion guide for Imake configured programs to an Autotools equivalent. The process of using Autotools to configure software is called “autoconfiscation”.

The relevance of this is that one of popularity. In the past, many ./configure programs existed. Eventually, many of these had their functionality absorbed into the Autotools project, making Autotools the de facto standard.

Imake was created for the configuration of the X Window System. The X Window System itself is being converted to Autotools. Because Imake will not be used for the X Window System any more, it is likely to become deprecated soon. Other packages will have to migrate as well. The goal of this project is to create a guide to convert Imake configured projects into Automake configured projects.

The versions used here are GNU Autoconf 2.61 and GNU Automake 1.10. These are the latest stable versions of the tools. There are incompatibilities with older versions of the tools, but it is always advised to use the latest stable version of a program if possible. Besides, the Autotools are not required for compilation itself because Autotools made distributions come with a complete configure script leaving the Autotools version requirements to the package maintainers only.

1.1 PSF Toolkit

A secondary goal to this project is to autoconfiscate the PSF Toolkit [13]. PSF is a formalism for the specification of all kinds of processes based on Process Algebra and Algebraic Specification of data. The toolkit contains tools to compile, rewrite, simulate and visualise these specifications. These various tools have varying library dependencies and also contain source code written in languages other then C. Also, it has a custom Imake template and various custom functions which make it difficult to autoconfiscate.

1.2 Document structure

This document will first discuss the difference between Imake and Autotools and how both tools work. Then, it will discuss how Imake compilation targets are to be translated to Automake. After that, the process of autoconfiscation for several examples will be discussed, followed by a list of problems encountered during the process. Finally, some concluding remarks.

Though this document contains all information to autoconfiscate Imake configured projects; a separate document will act as the actual guide. This document will explain thoroughly about several aspects of this process and how it has been applied to several programs. A more compact, less verbose guide under the name “Imake autoconfiscation guide” has been included as appendix C on page 29.

1.3 History

The problem of incompatibility between different systems is very old. There have been several programs made to try to overcome this problem. The most notable ones being Imake and Cygnus and GCC configure scrips; and Autotools.

Today the functionality of Cygnus and GCC scripts have been implemented into Autotools. Also, other configuration projects have been started, for example CONS, SCONS and CMake.

1.4 Portability

The goal of both Autotools and Imake is to help the developers in the process of porting their software to several operating systems, using a single code base. Different systems have different capabilities, functions and programs installed. The idea is to make use of a standardised set as much as possible and write replacement functions for those not portable. For example, `bcopy` is a function which is usually available on BSD systems, but not always on Linux. There is an equivalent function, `memmove`. If the function `bcopy` is used in the source code, the configuration program should have a function or macro to optionally replace `bcopy` calls with `memmove` ones.

1.5 Make

Both Imake and Autotools require Make. The purpose of the make utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them. In order to use make, a Makefile is required.

The makefile specifies which files are to be created and how they should be created; and which files are needed to do that. Though this can be very general in some cases; most of the time system specific information is required making it virtually impossible to create a single Makefile compatible with each system available. Both Imake and Autotools are designed to create system specific Makefiles for the systems they are running on.

1.6 Imake

Imake is the name of the configuration tool controlled by X.org for the X Window System. It clearly defines the variables used and has a "database" of variables which can be used in the configuration phase.

Imake also includes a number of `CPP` macros which are directives to simplify building `C` and `C++` programs. Under normal circumstances, there is an Imake template, which will be put together with the Imakefile. This then expands into a big Makefile. Alternatively, it is possible to create a custom Imake template in case the standard template does not suffice.

1.7 Autotools

Autotools is the name for a set of programs created by the GNU project. Its major components being autoconf, automake and libtool. The Autotools make extensive use of GNU M4 macro processing language.

In a typical GNU fashion, it is easy to set-up a project for a specific system using the Autotools and expand its portability later. However, the downside of this approach is that it is unclear how to make a project portable using Autotools in the first place.

1.7.1 Autoconf

Autoconf handles the configuration part for the Autotools toolkit. It creates the ./configure script out of the configure.ac file. configure.ac contains M4 macros which will be expanded to standard shell code. The ./configure script will do the actual configuration.

Autoconf is not responsible for compiling. It is responsible for locating the programs; libraries and functions available and check for compatibility.

1.7.2 Automake

Automake is responsible for the build step (compiling).

Automake takes a Makefile.am file and processes it to create the file Makefile.in. Automake accepts special make like variables which make it easier to maintain and understand as regular makefiles. Makefile.am does not need to specify how a program is built in standard (C/C++) cases but accepts Makefile directives to define how something should be built.

CHAPTER 2

Principles of Usage

Software configuration problems are of various types and do not have one single way of which it could be fixed. Hence the programs used for software configuration are usually set up in a very generic way.

2.1 Imake

Imake makes use of a database of software configuration profiles and library locations. This information should be used in combination with the program targets defined in Imake.rules.

Imake is called (usually) by xmkmf. Imake itself then calls the CPP macro processor to generate the makefiles. Input to the CPP Macro Processor is the Imake template (Imake.tmpl). The standard Imake template includes other standard Imake files and the Imakefile. Figure 2.1 on page 10 shows a graphical representation of this process.

When, for example, a library isn't defined in Imake yet including it's requirement is usually done either by defining a recognising script in the Imakefile or creating a custom Imake template which does the same.

2.2 Autotools

Autotools has a very decoupled way of handing software configuration problems. Autoconf will takes care of program recognition, identification and checks if the dependencies are met. Automake will take care of the building step. A graphical representation of the way the Autotools work can be found in figure 2.2 on page 10.

Autoscan creates the file configure.scan. This is converted by the user to configure.ac. aclocal takes configure.ac and optionally acinclude.m4 to create aclocal.m4. configure.ac and aclocal.m4 are then used by autoconf to create the ./configure script. Makefile.am and configure.ac are used by automake to create Makefile.in. This file is used by the configure script to generate the final Makefile.

2.2.1 Autoscan

Autoscan scans the source code to find out as much as possible about the libraries it uses; programs it would require; etc. It generates configure.scan which ideally is a drop-in replacement for configure.ac.

2.2.2 aclocal

aclocal is a program which takes the configure.ac script and examines it for macros. It scans acinclude.m4 and the standard macros (usually installed in /usr/share/aclocal-[version] and /usr/share/aclocal).

There is a project called the Autoconf Macro Archive. This is a collection of autoconf macros which can be used free of charge. This archive can be downloaded from http://autoconf-archive.cryp.to/.

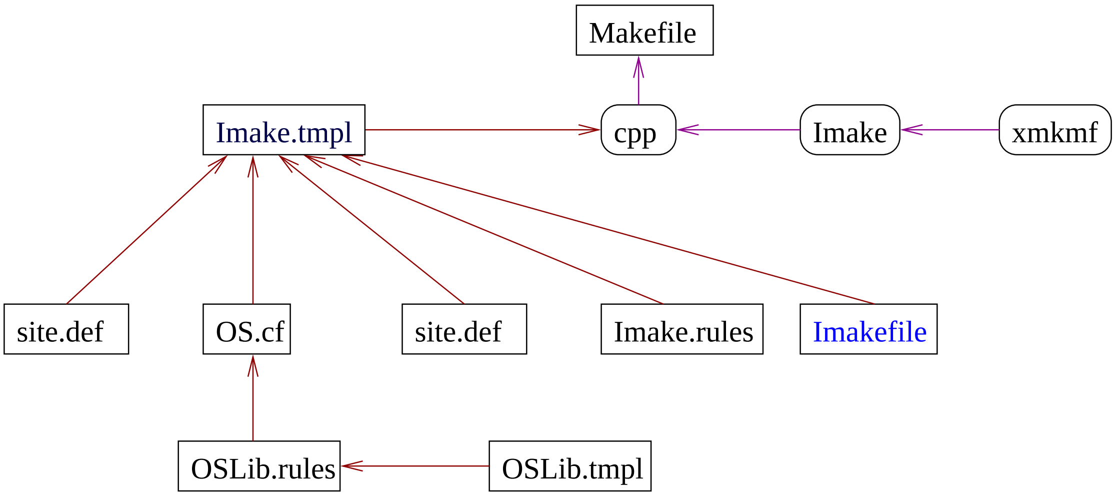

Figure 2.1: Imake data flow.

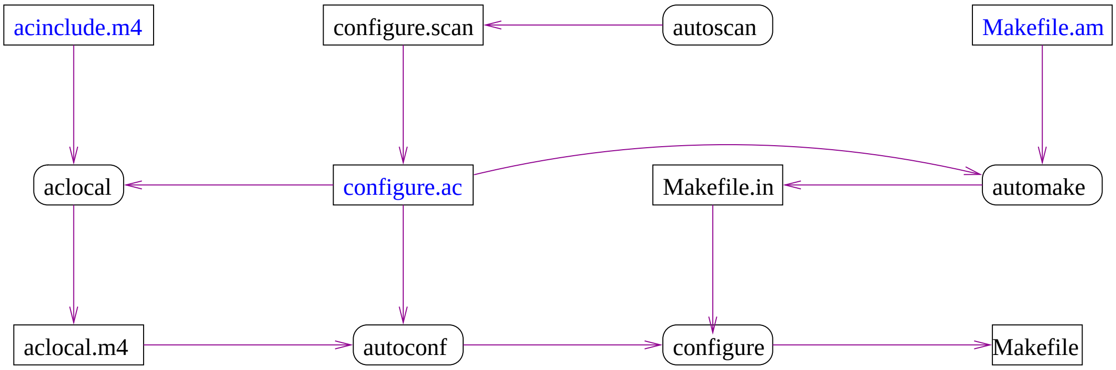

Figure 2.2: Autotools data flow.

The macros from the Autoconf Macro Archive could be installed by package managing software, but it is still advised to include the macros you used from the Autoconf Macro Archive in the acinclude.m4 file because some of these package managers don't include the archive macros in the aclocal directory.

2.2.3 Autoconf

Autoconf takes the file configure.ac and process it's macros using the file aclocal.m4. These expanded macros result into a portable shell script which actually configures the software and creates the Makefiles and other (user defined) configuration files.

configure.ac has a preamble which contains autoconf and automake initialisation macros. Also, it defines the source directories. Then several checks are made for different aspects of the program.

The advised order in which to check for problems is programs, libraries, header files, typedefs, structures and compiler characteristics, library functions and files. This information is also included in the configure.scan file created by autoscan. On the end of the configure.ac file `AC_CONFIG_FILES` and `AC_OUTPUT` are defined identifying the files that should be created by autoconf.

2.2.4 Automake

Automake relies on the Makefile.am and is used with the building process. In the most general case one can define a few standard automake macros and it compiles. In more complicated situations one could use a make like syntax to define how these files should be built.

The programs used by Makefile.am should be defined using autoconf. For example, the autoconf macro `AC_PROG_LATEX` will look for the proper LaTeXprocessor and define it as `$(latex)`. Makefile.am will only contain the `$(latex)` calls but not it's definition.

CHAPTER 3

Mapping Imake onto Autotools

The file Imake.rules contains the macros which the package maintainer should use in order to compile the software. It also contains a few “helper macro’s” such as for example concat, which concatenates two strings.

However; most of the Imake.rules file is irrelevant for our purpose. The goal of Imake is to expand the Imake template to a makefile; and most of the rules are constructed to accommodate this. The macros which are important to us are the ProgramTarget macros. These define what is to be compiled.

3.1 Relevant Imake Macros

3.1.1 NormalProgramTarget (program,objects,deplibs,locallibs,syslibs)

One of the main differences between Imake and AutoTools is the way programs are built. Imake defines CPP macros which expand to makefiles while AutoTools has AutoMake. Because of the semi-automatic library support of AutoConf, the following will suffice to correctly compile the program:

Makefile.am:

```
bin_PROGRAMS=program
program_SOURCES=srclist
```

srclist begin a list of source files which compile into objects.

Sometimes not all libraries can be correctly recognised by autoconf. In that case, the library flags can be derived the same way they are derived in Imake and the following should suffice (provided the contents of deblibs, locallibs and syslibs is copied):

Makefile.am:

```
bin_PROGRAMS=program
program_SOURCES=srclist
program_LDADD = deplibs locallibs syslibs
```

3.1.2 SimpleProgramTarget (program)

SimpleProgramTarget creates program out of program.c. In order to achieve the same using the Autotools, the following construction can be used:

Makefile.am:

```
bin_PROGRAMS=program
program_SOURCES=program.c
```

There are also a number of numbered SimpleProgramTarget macros. These work exactly the same as the normal SimpleProgramTarget.

3.1.3 ComplexProgramTarget (program)

`ComplexProgramTarget` uses the Imake standard variables to compile using `$(SRCS)` and link using `$(OBJS)`. This works the same as the NormalProgramTarget (section 3.1.1), except it also installs the manual if available. In order to install the manual one must use the `MANS` primary. Usually manuals are defined in `man_MANS`. This would mean that other then what's given in the NormalProgramTarget translation, the following should be added to the `Makefile.am` file.

```
man_MANS=program.man
```

Keep in mind though man files are usually not source code. Generally, they are compiled out of texi files. If the man pages should be included in the distribution, this must be explicitly defined. This is done by adding the `dist_` prefix.

```
dist_man_MANS=progam.man
```

3.1.4 ComplexProgramTarget_*number* (program,locallib,syslib)

Here the same counts as with the regular ComplexProgramTargets. However; this time, there are two other arguments to the list, `locallib` and `syslib`. These are additional arguments because the macros `locallib` and `syslib` are already in use for the normal `ComplexProgramTarget`.

number can be any number between 1 and 10 here. This construction is to allow multiple ComplexProgramTargets to be compiled by one Imakefile.

3.2 Other Macros

There are quite a few more macros available to Imake then discussed above. However; quite a few of these macros are to accommodate deprecated debug engines, work around shortcomings of CPP, or define the compilation structure within Imake.

The use of these macros is rare and usually clear enough to be understood on it's own. Therefore, no extra attention is given to these macros in this documents. In case a macro will be found in an Imakefile which is unfamiliar, usually one can examine Imake.rules [16] to find out it's purpose.

CHAPTER 4

Converting Applications

In order to get a clear understanding of Imake and the Autotools work I have tried to implement it with several projects. The first program to have been converted is a game called 'Yiff', which is a small (400 kilobyte) C program made to test the supposedly simplified C compilation structures. The other project I have converted is the base for this Bachelor thesis. This has been converted to investigate the working of automake on unknown source code, and the PSF Toolkit; to examine it on a larger, more complex project.

4.1 Yiff

Yiff is a game originally created for DOS and later ported to X11 under the GNU GPL. It depends on ncurses and allegro library and came with a static Makefile. The makefile itself was written in a specific way so it suffered from false dependencies; depending among things on a 486 CPU, the allegro library being present in /usr/local/allegro, and the standard header files in the exact same (non standard) location as the original coder had.

In order to autoconfiscate Yiff, it seemed like a good idea to get rid of the (heavily dependent) Makefile, and start from scratch. The first step was to run autoscan. Autoscan creates configure.scan, which is processed by autoconf to create the ./configure script. Ideally configure.scan can be renamed to configure.ac and it 'just works'. In practice, several changes had to be made to configure.scan in order to be correct.

4.1.1 configure.ac

configure.scan is a fairly generic C oriented file. It checks for a C compiler and standard C headers. Quite some tinkering needs to be done to make it working. There are two AC_CHECK_LIB macros.

The AC_CHECK_LIB(library,symbol[,success,failed]) macros are autoconf macros which check for availability of a certain symbol (such as, for example, a function) inside a library. It will prepend -l[library] to the comiler flags and define the preprocessor macro HAVE_LIB[library. Optionally, one can define shell codes for when the check is succeeded or failed.

The ncurses library is compatible with the AC_CHECK_LIB macro, but allegro proves a different case. After searching the internet it seems that AM_PATH_ALLEGRO must be used for the right results.

The AM_PATH_{LIB_NAME}([minimum_version,success,failed]) macro works differently from the AC_CHECK_LIB macro. All arguments are optional. If a minimum version is supplied; it is checked wether or not an up to date version of the library is installed. The success and failed arguments are again shell codes which are executed upon success or failure.

The macro AM_PATH_{lib_name} is usually needed if the library is shipped with a {lib_name}-config program. This script is used to find the information for the macro. It is not necessary to execute this program.

AM_PATH_{lib_name} doesn't add values to the compile flags. Instead it defines libname_DLADD and libname_FLAGS. These must be added to the compiler flags manually.

It is advised to know all dependencies and have the development packages installed. Some macros, such as the allegro macro discussed above, are installed by the package itself (allegro) instead of the Autotools.

4.1.2 Makefile.am

Also, a variable `YIFFDIR` has been defined as `$DATADIR` in Makefile.am to allow for the data files to be installed into the file system dependent data directory rather then the standard string provided in the source code.

The compilation of Yiff was pretty simple. The first step was to define `bin_PROGRAMS` and `yiff_SOURCES`. The only program to be compiled is yiff and it's sources are all .c and .h files.

However; allegro proved to be a bit more cumbersome yet again. The dependency on the allegro library had to be explicitly defined. The `yiff_LDADD` macro has to be included and contain `$(allegro_LIBS)`. Also; the macro `yiff_CFLAGS` needs to include `$(allegro_CFLAGS)` and `-DYIFFDIR=\¨$(yiffdir)\¨`. The `-DYIFFDIR` macro is included to make sure the data install path and the path the program expects the data to be at will be the same.

The `yiff_LDADD` and `yiff_CFLAGS` macros are there to ensure the compiler knows where to find the allegro library. Other ways to achieve this are to use the `LDADD` and `AM_CFLAGS`.

The macro `LDADD` exists to add commands to the linker command line. For example; libraries which aren't found by configure (such as, in this case, allegro). The `AM_CFLAGS` macro exists to define global compilation flags arguments for the compile step. The `yiff_CFLAGS` macro is specifically for the program “yiff”. If a specific program macro is defined, the global macro is ignored. However; one can include the global macro in the program specific macro.

```
yiff_CFLAGS=$(AM_CFLAGS) yiff_specific_cflags
```

4.2 Bachelor Thesis

This very document has also been autoconfiscated. The source code to this document consists out of LaTeXsource code. This can be compiled to DVI and PDF using latex and pdflatex, or texi2dvi and texi2pdf.

Unlike with C; LaTeXcompilation pretty unsupported. The Autoconf Macro Archive can provide us with the autoconf macros `AC_PROG_LATEX` and `AC_PROG_PDFLATEX` to check for availability of latex and pdflatex.

The various documents have been saved in the doc/ directory. The Makefile.am file is used to generate a (portable) Makefile. For this example we have defined the `.tex.dvi:` and `.tex.pdf:` targets using the variables `$latex` and `$pdflatex`. We generated a list of PDF and DVI files which could be created and added them to the pdf and dvi targets. `EXTRA_DIST` includes the tex source files so they will also be included in a tarball.

4.3 PSF Toolkit

In order to compile the PSF toolkit, it was first compiled with Imake and the output logged in a log file. This information can be valuable for the autoconfiscation process. For example, it exposes the compile and link flags used on this particular system for each source file without needing to examine the Imakefiles.

4.3.1 configure.ac

In order to create a starting point for configure.ac, autoscan is used. The configure.scan file is copied to configure.ac and it's values for `AC_INIT` are filled in. Since it's already sure we will be using Automake for this process, the `AM_INIT_AUTOMAKE()` macro is added right after the `AC_INIT` macro.

We already know there are lex, yacc and perl files, so a check is added for these programs. For lex and yacc these are `AC_PROG_LEX` and `AC_PROG_YACC`. Note that the configure script will not fail if these programs are not around. This is because the make dist command will include the files created by lex and yacc, so these programs are optional for end users. For more information about lex and yacc support please check chapter 9.7 on the Automake info page [2]

perl

Perl is another matter. There is no particular program check for perl; so we have to make use of the generic one, `AC_CHECK_PROG`. `AC_CHECK_PROG` takes 3 required and 3 optional arguments. The first argument is the variable to save the result in. The second one is the program to look for in `PATH` and the third argument is the The optional fourth argument is the value the variable should have if the program is not found. The fifth argument is the path to search and the sixth argument is a full path to a file which will always be rejected as a valid result.

The macro used in this case is `AC_CHECK_PROG([PERL],[perl],[$(which perl)], [None])`. We're looking for "perl", and if it's found the variable "PERL" is set to "$(which perl)". If not found "PERL" is set to "None".

`$(which perl)` is used instead of just `perl` because we need to know the absolute path to the perl program. the which program and `$()` construction are part of standard shell code and therefore don't need to be checked for.

Because perl is a required program it needs to be tested if it works. If the perl check failed, `PERL` would be initiated to `None`. So we write a test for it using standard shell code and an autoconf macro.

if test "$PERL" = "None"; then AC_MSG_ERROR(Cannot find perl); fi

`AC_MSG_ERROR` is an autoconf macro which will print it's contents and then exits the configure script. It is advised to test required programs and be verbose about what condition is not met. This information is valuable to end users who's system does not yet meet the requirements.

The final two lines in configure.ac should be `AC_CONFIG_FILES` and `AC_OUTPUT`. For the purpose of autoconfiscating the toolkit, I've created a shell script which can keep this up to date automatically. It can be found in appendix A on page 25.

4.3.2 Makefile.am

It is best to autoconfiscate the different programs of a package one at the time. In order to do so one also needs to address one directory at a time. The source directory structure of the PSF toolkit contains 82 directories and each time one needs to traverse into a subdirectory; this must be listed in the Makefile.am.

The top Makefile.am contains only `SUBDIRS=src`. The Makefile.am file in the "src" directory would contain only a list of it's subdirectories. However; during the autoconfiscation process; it would be best to start off with just one, and gradually add more as the process continues. This would lead into a cumbersome administration of which directories are already autoconfiscated and which still need to be processed.

SUBDIRS

In order to make this easier, I've created two bash scripts and a python script which work together to administer the Makefile.am files which only contain a SUBDIRS directive.

The python script calls the shell scripts which will delete all the Makefile.am files with only a `SUBDIRS` macro. Then, it generates a list of the remaining Makefile.am files and recreates the Makefile.am SUBDIRS files structure.

The shell scripts can be found in appendix B on page 27.

Though this scheme works while autoconfiscating, before the code can be distributed, the SUBDIRS structure may need to change. Sometimes files depend on other targets in other directories and Make does not know how to build these. The `SUBDIRS` macro is followed as written. So after all targets have been defined, the `SUBDIRS` macros must be rearranged so each target is only build when it's dependencies are built as well.

Makefile.common

As can be seen in the in the log file discussed in 4.3 on page 16 each file is compiled with a couple of standard flags. In order not having to include this information in each and every Makefile.am, a common makefile is created in the top directory of the project. This file has been called Makefile.common. In order to include the content of this file in every Makefile each makefile must have an include directive. A special variable is available to point to the top directory of the project; top_srcdir. The include directive would look like this:

```
include $(top_srcdir)/Makefile.common
```

Though this would still require a common string to be included in each and every Makefile.am; it would simplify things if the common compile flags need to be changed.

Since Makefile.common is defined to keep project level variables, it contains the compile flags and an LDADD directive concerning a library which will be discussed in section 4.3.3 on page 18

The actual Makefile.am

In order to get the relevant contents of the Makefile.am file, the corresponding Imakefile must be examined. Each ProgramTarget macro represents a program to be build. So the name of the ComplexProgramTarget is added to a PROGRAMS macro in the Makefile.am. Since every program will be added to the regular binary directory, for the PSF Toolkit this will be bin_PROGRAMS.

Each ComplexProgramTarget has it's own set of OBJS and SRCS macros. The OBJS macro is irrelevant to Automake, but the SRC macro content has to be copied to the {program}_SOURCES macro. Though the Imakefile only defines the C files, it is advised to also include the header (.h) files. Though this does not affect the compilation, all files listed in a SOURCES macro are included in the make dist distribution.

Man files

Normally, man files are generated out of texi documents, and thus the man files are normally not included in the distribution. However; in the case of the PSF toolkit, the man files are not compiled. The regular way to include define the man files is with the man_MANS directive. In order for the man pages to be included in the distribution there are two options. Either to create a macro EXTRA_DIST and add the content of the man_MANS; or to use the dist_ prefix.

For example, the code for manual inclusion of writetil is as follows:

```
dist_man_MANS=writetil.man
```

4.3.3 Convenience libraries

The PSF toolkit had a lot of source code which is shared by multiple programs. There is a directory src/include which contains these files. In the Imake solution; the required files were soft linked and compiled separately with every different program.

Inspired by [7], a different choice has been made for the PSF toolkit. Instead, a library will compiled the same way as the program would've been compiled, only instead of a PROGRAMS directive, a LIBRARIES directive is used. Since libraries are usually installed, and in this case we don't want that to happen, the noinst_ prefix is used.

So instead of linking and compiling each file, the library directory is added to the list of directories to be searched for header files in compile time. In link time, the corresponding library is added as a link flag, so it would be linked against.

This way, each "library" only has to be compiled once rather then once per program which requires it.

CHAPTER 5

Problems

In the process of writing the guideline and testing it on the PSF Toolkit I've stumbled across a few problems which are worth mentioning. These include not only Imake autoconfiscation problems but also general problems encountered with the Autotools. I hope this chapter gives insight in these problems and how to solve them. The versions used here are GNU Autoconf 2.61, GNU Automake 1.10.

5.1 Autotools project name

When calling `AC_INIT(project_name, version, email)`, the project_name must not contain any white space. If it does, `make dist` will fail. Somehow, the `project_name` will then not be part of the file name, and thus there will be worked with a file named -version. Some operations done on a file starting with a minus (-) sign will fail, as they will within `make dist`. Probably a `project_name` starting with a minus sign will fail as well.

5.2 Dodgy including

Within the PSF Toolkit, several files are "included" by creating symbolic links to the files in the include directory. These files are then compiled into .o object files. This results in several files being compiled several times for different tools.

An more elegant solution is offered in section 4.3.3 on page 18.

5.3 Manuals

Man pages are not automatically included by `make dist` because they usually are compiled out of .texi files. With the PSF Toolkit, these manuals were not compiled but written as is. To include these man pages the `man_MANS` macro must be prepend by `dist_` so automake knows these files have to be rolled into the tarball.

5.4 Lex and Yacc

With Imake, when lex and yacc files are compiled, their output files are the same as they would be when invoking the programs from the command line. With the Autotools, however; compiled lex and yacc files are renamed to fit the original file name. So lexer.l will be compiled to lexer.c.

The PSF Toolkit had many lex and yacc files with the same base name. Also, the other files depended on lex and yacc files to be compiled using their standard output file name. To overcome this problem, the lex files had to be renamed to lex.yy.l to it would compile to lex.yy.c and the yacc files had to be renamed to y.tab.y, to it would compile to y.tab.c and y.tab.h.

5.5 y.tab.h

Though Autotools compiles yacc files to C files with no problem once they are included in a SOURCES macro, sometimes, files wouldn't want to compile because they couldn't find y.tab.h, or a way to compile it. In order to fix this; a y.tab.h makefile directive had to be made. Because y.tab.h will be compiled together with y.tab.c is compiled, defining a dependency to y.tab.c is enough.

CHAPTER 6

Conclusion

Software configurations is a field which characteristics depend on the software built and the platform it is built on. The problem has been around since the second computer was built; and will probably stay around forever, since there will always be different computers; different operating system; and different libraries; doing things in a different way.

6.1 Software Configuration

The key to minimise the effort is to define the different problems within the subject and deal with them one at the time. First, a working platform on which the software should run and compile properly needs to be defined. This platform will be the reference platform. The software must be able to run and compile on this computer.

For the software part of the configuration finding the right inclusion directives and define statements for the compile step is first priority. After that one should find the right link directives for the link step. Both of these steps should be done using standard aclocal macros as much as possible. Also, the Autoconf Macro Archive is a good source for macros. The pre built macros are preferred over custom macros because they are generally well tested on many different platforms.

In order to make the software work on other platforms then the working platform, one needs to find and eliminate the differences between both platforms. Where possible it is advised to write software in a standard compatible, portable way; and use software configuration to do function checks and rewrite incompatible functions and define the non existing ones.

6.2 Imake and Automake

The process of translating Imake projects to Automake ones is hard because both programs work in a different way. Though both leave enough freedom to create configurations which are easily interchangeable; this same freedom enables users to create their own solutions which cannot be translated properly. The best way in most cases is to copy the project and start to configure it with the Autotools, using the Imakefiles as reference for each step on the way.

6.3 PSF Toolkit

Autoconfiscating the PSF Toolkit proves to be a challenge. Not only because of the differences between Imake and Autotools, but also because the different constructions used. Some files are constructed using strange schemes. These have to be understood before they can be translated properly. Another big problem is the dependency to certain compile flags. The log of compilation using Imake has been of great help in solving these matters.

The Imake autoconfiscation guide is sufficient to complete the autoconfiscation of the PSF toolkit. The PSF toolkit is not yet completely autoconfiscated the PSF toolkit is, at time of writing not yet complete. However, each Imakefile in the PSF toolkit has been examined and assessed for autoconfiscatability.

Bibliography

[1] The Autoconf info page

[2] The Automake Info page

[3] The imake man page

[4] The C PreProcessor (http://gcc.gnu.org/onlinedocs/cpp/)

[5] The GNU M4 Macro Processor (http://www.gnu.org/software/m4/manual/m4.html)

[6] Imake related software and documentation (http://www.snake.net/software/imake-stuff/)

[7] GNU Autoconf, Automake and Libtool book (http://sourceware.org/autobook/)

[8] OReilly “Software Portability with imake” (http://examples.oreilly.com/imake/)

[9] Tools used in the build process (http://www.cse.iitb.ac.in/~sameera/seminar/finalSeminar/seminar.ps)

[10] The Autoconf Macro Archive (http://ac-archive.sourceforge.net/)

[11] Imake Frequently Asked Questions (http://www.snake.net/software/imake-stuff/imake-faq.html)

[12] Imake contra I make (http://www.informatik.uni-osnabrueck.de/um/92/92.1/imake/imake.html)

[13] The PSF Toolkit (http://staff.science.uva.nl/~psf/toolkit/)

[14] Yiff for X11 (http://www.beastwithin.org/users/wwwwolf/games/xyiff.html)

[15] Carbon (for the Makefiles) (http://home.gna.org/mlmm/)

[16] Imake.rules (usually /usr/lib/X11/config/Imake.rules)

APPENDIX A

ac_config_files_fixer.sh

APPENDIX B

Makefile.am structure

B.1 nido ac_config_files fixer.py

B.2 nido Makefile.am deleter.sh

B.3 nido test SUBDIRS del file.sh

APPENDIX C

Imake autoconfiscation guide

C.1 The tools used

The goal of this guide is to give insight in how to autoconfiscate Imake configured projects. The method used by this guide is to do autoconfiscation the regular way, completed with information gathered from the Imake files. This section will discuss the tools of the trade. For more information about specific programs and how they work; it is advised to check out the corresponding info page.

C.1.1 automake

The function of automake is to define the build step in software compilation. It is based on regular make; and together with autoconf it should be capable of building any type of program. There are special primaries for certain types of code such as C, JAVA and python, but it has also been used successfully for languages unknown to automake itself.

Automake depends on autoconf to define machine specific variables; such as the location of the compiler, header files, libraries; etcetera. Automake itself defines how the program is built.

recursion Unlike Imake, Automake does not automatically recurse over the Makefile.am files. So, if the top Makefile.am contains a directive, Makefile.am files in subdirectories will not know these directives.

Automake does have an include directive which can be used to emulate this behaviour. It is also possible to create a “common” makefile in the root directory of the source tree. There is a special variable called `$(top_srcdir)`, which refers to the source root. Keep in mind this “common” makefile must be explicitly included by every Makefile.am

primaries Automake works with primaries which define what should be made. There are several standard primaries which can be used. With automake 1.5 the primary names are `PROGRAMS`, `LIBRARIES`, `LISP`, `PYTHON`, `JAVA`, `SCRIPTS`, `DATA`, `HEADERS`, `MANS` and `TEXINFOS`.

These primaries are prefixed with a macro pointing to the directory in which it should be installed. A few of them have already been defined. For example, `bindir`, `sbindir` and `datadir`. One of the most used automake macros is probably `bin_PROGRAMS`.

C compilation The compilation of a C program consists out of two steps. First, the C files are compiled to object files; then the object files are linked into an executable or library.

In order to compile a C program with automake, one must first define a `PROGRAMS` macro. This macro must be prefixed with a directory macro. For example; if you wish to build the program “hallo”, and install in in the standard binary directory, the correct macro would be `bin_PROGRAMS=hallo`

For each program in a `PROGRAMS` macro; there must exist a `SOURCES` variable. This will define the source code to be compiled and linked together. Say the hallo program requires the files “hallo.c” and “hallo.h”. the `SOURCES` macro would look like this: `hallo_SOURCES=hallo.c hallo.h`

Generally this should suffice. Hoever; sometimes the process requires flags to be set at compile time. This can be done in two ways.

Compile step Sometimes, a piece of C code required specific flags to be set for compilation. For example, macro definitions or include statements for unsupported libraries.

AM_CFLAGS is a global compile flags macro. Here all needed compile flags can be set. Also compile time defines and such should be defined here. There is also a CFLAGS macro, but this one is for the user to define, so it shouldn't be touched. AM_CFLAGS is for global compile flags.

Programs have their own special CFLAGS macro. This is usually empty but can be defined. If, for example, one program requires a specific define which no other program needs. If the program specific CFLAGS macro is defined, it will ignore the AM_CFLAGS macro for the compilation of that program. However; it is possible to include the contents if the AM_CFLAGS in the program CFLAGS macro. For example, if the hallo program requires the AM_CFLAGS and -DWORLD, it would look like this: hallo_CFLAGS=$(AM_CFLAGS) -DWORLD

Link step Also during the link step, specific flags may need to be set. If you need to link against libraries that are not found by 'configure', you can use LDADD to do so. This variable actually can be used to add any options to the linker command line. Next to the global LDADD macro, there is a program specific version program_LDADD, in case not all programs in the same directory share the same link-time requirements.

The LDADD macros are inappropriate for passing linker flags other then library flags (such as -l and -L. Instead, use the LDFLAGS macros. Just as with CFLAGS, there is a user macro LDFLAGS which is not to be used. Instead, use it's automake shadow macro AM_LDFLAGS, or the program specific version {program}_LDFLAGS.

C.1.2 autoscan

Autoscan is a program designed to create a configure.scan file. This file can be used as a preliminary configure.ac. In that case, the configure.scan file needs to be examined manually before it can be renamed to configure.ac. Autoscan can also be used to check the configure.ac completeness. In that case, you need to examine the difference between configure.ac and configure.scan, and consider the suggestions given by configure.scan.

C.1.3 autoconf

Autoconf is responsible for making sure the program given can compile. It relies on configure.ac and aclocal.m4. aclocal.m4 can be created automatically using aclocal. These macros will result in variables which can be used in automake. For example, $(CC) is defined by autoconf, as are most library link flags.

A typical configure.ac consists of initialisation code, followed by compatibility checks, followed by a list of files to create.

C.1.4 aclocal

aclocal creates aclocal.m4 by grabbing macros from acinclude.m4 and system macro's, usually installed in /usr/share/aclocal/.

C.2 Guide through the process

This guide will aim give a better insight in how the Autotools work and how Imakefiles can be used as a reference to for information. It will discuss how to create a configure.ac file and Makefile.am files

C.2.1 Imake

One thing that may be useful to know is what Imake actually does. The easiest way to get to know this is to compile the project using Imake and log the output. This will give all arguments used in each compile and link step. It will be useful to guess libraries used and other flags set in case the Imakefiles do not provide good information.

C.2.2 project layout

GNU has a detailed description about how a software tree should look. However, in this case we already have a way the software is set up and we can use the old layout rather then to adhere to GNU standards and find more problems because of moved source code.

There are, however; a few files which automake requires. Most of these files could be created using the command automake –add-missing –copy. A Few files cannot be added this way, for example NEWS and README. Their contents, however, is irrelevant, so they can just be created by touch.

Be careful with the automake –add-missing –copy command. This will include, for example the GNU General Public Licence. Make sure you are comfortable with the files copied. Automake does not check the files contents.

C.2.3 configure.ac

If you are happy with your project layout the first step is to run autoscan. This results in a file called configure.scan. Intentionally this file can be copied to configure.ac and used without modification. However, om reality some things still need to be done in advance of this being a proper configure.ac file.

Firstly, the `AC_INIT` variable should be properly set. `AC_INIT` initialises autoconf. It requires three variables. The project name, version and an email address. For example, this is the `AC_INIT` macro used for this document.

```
AC_INIT(project_name, 0.2, email@address.edu)
```

Another macro we'll add to the configure.ac file is AM_INIT_AUTOMAKE. This ensures autoconf will play nice with automake. If suffices to simple add `AM_INIT_AUTOMAKE`.

C.2.4 Makefile.am

For each directory containing source code, an Makefile.am file must be created. This file will contain the compilation directives. Most information of this can be derived from the Imakefile.

First, the `SUBDIRS` variables need to be defined. For each direct subdirectory of the given directory which needs to be processed by Automake, it's name must be an entry of the given directories `SUBDIRS` variable. For example, most of the time, the top Makefile.am contains only one macro, being `SUBDIRS=src`.

The Imakefile corresponding to the Makefile.am that is to be created will contain some sort of `ProgramTarget`. This contains the name of the program to be built. This program depends on a number of object files in the `OBJS` macro. These object files are created out of source code, These sources are in a macro called `SRCS`. One exception to this is the case of the `SimpleProgramTarget` the source code consists out of it's argument`.c`. So for the macro `SimpleProgramTarget(timer)`, the program to be compiled will be `timer`, and it's source code will be `timer.c`

One must create a `_PROGRAMS` macro containing the program to be built. This macro must be prefixed with an install directory. Examples of predefined install directories are `bindir sbindir libexecdir` and `pkglibdir`. You can also define your own directory. The prefix to use is equal to the directory macro minus the `dir` part. For example, `bin_PROGRAMS=foo` will install program foo in `bindir`. One can also use a non-installing prefix called `noinst`. `noinst_PROGRAMS=bar`, will build, but not install the program bar.

For each program defined in a `_PROGRAMS` macro, a `_SOURCES` macro can be defined. If this macro is not defined, a default value of `progname.c` will be used. The `_SOURCES` macro consists of a canonical version of the program name, appended by `_SOURCES`. The canonical version of a program name is the program name with all characters except for letters, numbers, the at (@) and underscore symbol turned into underscore symbols. For example, canonical version of `program-name.sh` is `program_name_sh`.

The value which should be used for the _SOURCES variable can be derived from the Imakefile or the Imake build log. Regular ProgramTargets depend on a OBJS macro. This macro defines the object files used by the ProgramTarget to link the program. These object files are usually created from the from source files with the same name, except for the .o is replaced by .c. When this isn't the case, special build directives can be found and destilated from the Imakefile.

C.2.5 Completing autoconfiscation

If that is done, configure.ac must be modified again to include the AC_CONFIG_FILES macro, which specifies which Makefiles have to be made. A script called ac-config-files-makefile-finder.sh has been created which, if executed in the project's root, echoes the macro correctly so it can be included in configure.ac. The prefered location in the file for this macro is before AC_OUTPUT and after anything else.

Then you run aclocal. This will install the aclocal.m4 file needed by autoconf and automake. After that it is advised to run `automake --add-missing` to include files needed for Autotools to function normally. There are some other files required but not installed by `automake --add-missing`, these can be just touched but it is adviced to fill them with relevant content. The files in question are NEWS, AUTHORS and ChangeLog.

The program may also complain about config.h.in. This file can be created using autoheader.

When all this is done one can run autoreconf which should run all required programs again in the right order. If any files are edited it is adviced to run autoreconf and ./configure again.

C.2.6 Lex and Yacc

Some sources in these files are lex or yacc files. In order to compile these, autoconf must first be made aware of these programs. The usual way to define the variables of these programs is to add the macros AC_PROG_YACC and AM_PROG_LEX. The input files for these programs can also added to the progname_SOURCES variable. With the given macros automake is now able to compile it. The YFLAG and LFLAG variables can be copied from the Imake file into AM_YFLAGS and AM_LFLAGS.

C.2.7 other source code

When the program you wish to compile is written in a language which is not a C like language, it is still possible to use Autotools. One can still define the _PROGRAMS variable. There are, however, a few things to take care of for it to work.

For starters, the _SOURCES variable requires C like source code, so one must define the _SOURCES variable empty.

However; automake still needs to know how to compile the program. One can include normal make code which defines the compile step. Keep in mind though not to use automake variables for programs. For example: $(PERL). These variables must be defined using autoconf.