Study of tools used in build process

Sameera Deshpande

November 16, 2005

Contents

1 Introduction

1.1 Introduction to configure and build process

Generally, the software programs written must be system independent, i.e they must be adaptable so that they can work in different computing environments. The following features of computing environments could differ:

- Processor : The architectural differences can be taken care of if the software is written in a High level language, and a compiler is available for that language which generates code for desired processor.
- Operating system : Even for the same processor, computing environments may have different operating systems. Hence, for each operating system, the system calls may differ. Depending upon which operating system is under consideration, the function used to perform specific operation may differ. These differences are taken care of by compilers and associated language implementation tools.
- Versions and installations of operating system : For the same operating system there can be different versions, which have different file formats, tools, or even locations at which system executables are kept. The programmer has to take care of these differences explicitely.

The process of adapting a software to the above changes is called *porting*. Porting can be labourious and error-prone. Configure and Build Systems [4] allow a programmer to specify these differences to simplify the process of porting. *Configuration* refers to instanciating software to incorporate above variations whereas *building* refers to recompilation of the adapted software. *Configurator* is a set of scripts which carries out task of configuration, whereas *builder* is a set of scripts which performs building.

To facilitate configuration, software is written generically in terms of macro variables which capture system variations. Configurator attempts to guess correct values for system dependent variables used during compilation. It individually checks for presence of each feature that the software for which it is written might need. It also accepts some external inputs from user giving him freedom to override few default attribute values. It then uses all this collected information to generate the scripts or headers which are used for building. Builder makes use of the information collected by configurator, and then generates appropriate port.

In this report, I am focusing on GNU configure and build system. I will be discussing roles played by GNU automated tools like `autoscan, autoconf, automake, autogen` in configure and build process.

In GNU projects, `configure` is the configurator which takes `makefile.in` as an input. It collects information about system features, that the software might need to generate another shell script `config.status`, which in turn generates `makefile` in each

subdirectory. The `Makefile` is then used by builder `make` to generate binaries. These binaries are desired executables.

Developers can write all input files like `configure`, `Makefile.in`, <header>.h.in etc. on their own to suite their requirement. But as size of software package increases, the complexity of `configure` script also increases drastically. Here Configure and Build system tools come into picture. The use of any tool is not necessary, it just makes task of developer easy.

1.2 Configure and Build Process

The set of GNU configure and build tools can categorised as following depending upon their applications :

- Tools to generate Configurators : The tools which generate files that are needed by configurator or generate configurator itself, come under this category.
- Tools to generate Builders : The tools which generate files that are used by builder come under this category.
- Generic tools : There are some tools in GNU project which can be used to generate configurators, builders or even source files. Such tools are categorised as generic tools.

Here we will see formal process flow of configure and build tools. Please refer figure 1.

- **autoscan**[1] is configurator generator tool used to generate `configure.scan` file for the source package. This program examines source files in directory for common portability problems and generates checks for programs, header files, libraries, types, functions etc. The file generated by this program is then renamed as `configure.`[`in/ac`] or any other file name which is given as input to `autoconf` on command line after manually examining the file and making appropriate changes if needed (explained in later section).
- If header file is to be generated by configure script, then **autoheader** program is invoked. `autoheader` scans the `configure.`[`in/ac`] file to check if macro `AC_CONFIG_HEADER` is defined or not. If it is defined, then template file of C `#define` statements is created.
- **automake**[2] program is builder generator tool which is used if `makefile.in` file is not manually written and is to be generated. To generate all `makefile.in`s for the package, `automake` program is run in top level directory, with no arguments. 'automake' automatically finds appropriate `makefile.am` and generate corresponding

[1] http://www.gnu.org/software/autoconf/manual/autoconf-2.57/html_mono/autoconf.html#SEC11
[2] http://www.gnu.org/software/automake/manual/html_mono/automake.html

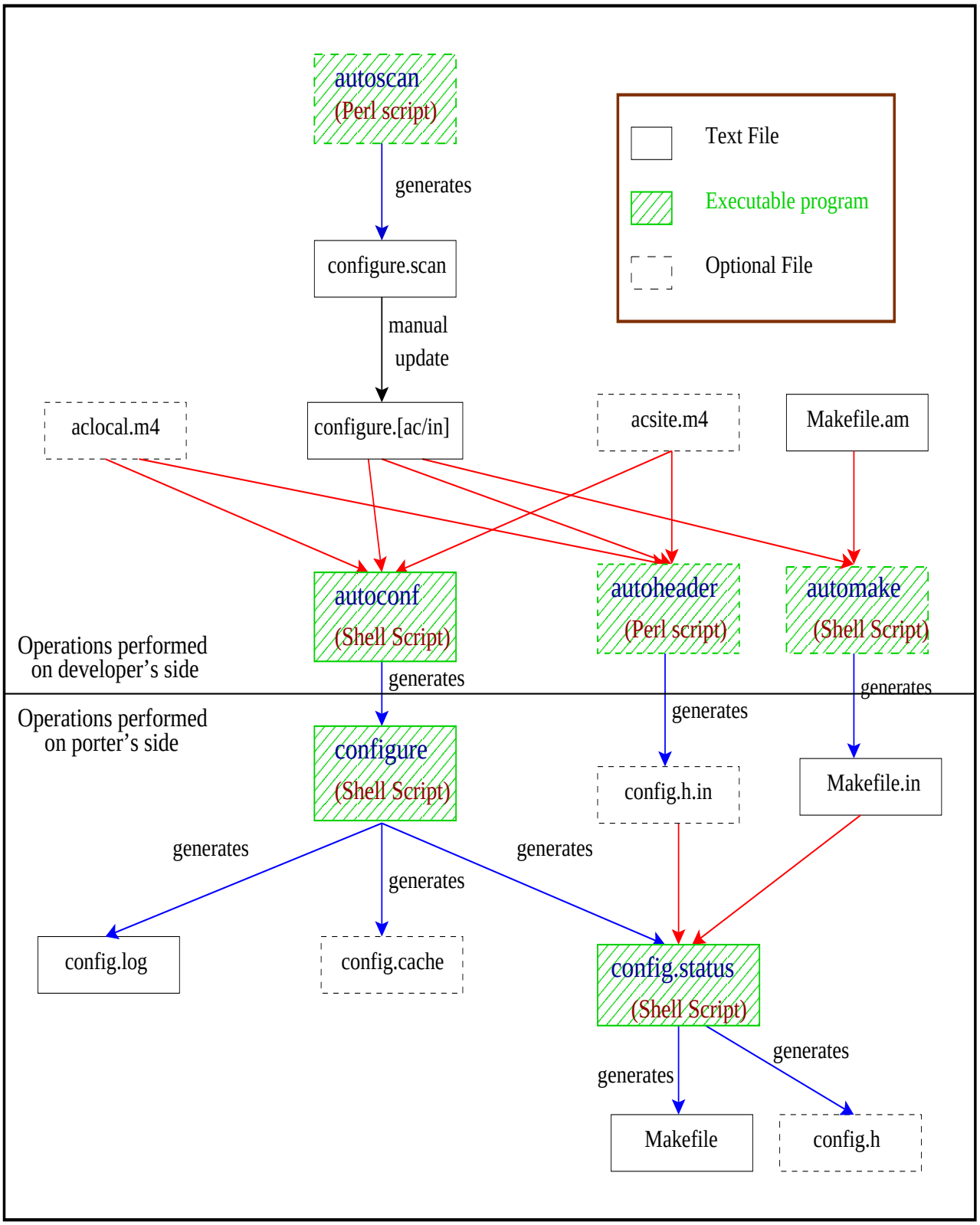

Figure 1: Relationship of different tools in configure and build process

`makefile.in`. `automake` runs `autoconf` to scan `configure.in` and its dependencies. It just scans `configure.in`, and doesn't generate configure files. So, `autoconf` is needed to run seperatetly for generating `configure` script.

- `automake` includes a number of `autoconf` macros which can be used in package; some of them are actually required by `automake` in certain situations. These macros must be defined in your `aclocal.m4`; otherwise they will not be seen by `autoconf`. **aclocal**[3] is generic tool that scans all the '.m4' files it can find, looking for macro definitions. Then it scans `configure.in`. Any mention of one of the macros found in the first step causes that macro, and any macros it in turn requires, to be put into `aclocal.m4`. `aclocal` tries to be smart about looking for new `AC_DEFUN`s in the files it scans. It also tries to copy the full text of the scanned file into `aclocal.m4`, including both '#' and 'dnl' comments.

- **autoconf**[4] is the configurator generator tool written as shell script which takes `configure.`[`in/ac`] file as an input and generate `configure` script as an output. The configuration scripts produced by `autoconf` require no manual user intervention when run; they do not normally even need an argument specifying the system type. Instead, they individually test for the presence of each feature that the software package they are for might need. The script produced by `autoconf` is independent of `autoconf`, when it is executed.

- **configure** script is the configurator itself when executed, generates `config.status`, `makefile` in each subdirectory, header files containing system dependent macro definitions, and other files like `config.cache, config.log`, etc.

- **make** utility is the builder used to generate desired binaries. It determines which piece of code has modified, and hence needs to be recompiled, and issues the commands accordingly.

- **Cygnus configure** [5] is another configurator which takes `configure.in, config.sub, config.guess, makefile.in` and makefile fragments reciding in `config/*` files as an input, and then generates `config.status` and `makefile` as an output in each subdirectory. The `configure.in` file taken as input by `Cygnus configure` is very different from one that is used by `autoconf`.

- **autogen**[6] is the generic tool, which is used when one wants to generate program files having repeatative text with varied substitutions. It can be used to generate source files or even configurators or builder generators. It takes template and definition files as input and using predefined macros, perform intelligent substitution to generate desired files.

[3] http://www.gnu.org/software/automake/manual/html_mono/automake.html#Invoking%20aclocal
[4] http://www.gnu.org/software/autoconf/manual/autoconf-2.57/html_mono/autoconf.html
[5] http://www.chemie.fu-berlin.de/chemnet/use/info/configure/configure.html
[6] http://www.gnu.org/software/autogen/manual/html_mono/autogen.html

1.3 Case Study for GCC package.

GCC is very large software package, having large number of source files written in 'C'. Lets have a look at configure and build tools used by well-known software package GCC. Consider the figure 2

- GCC uses **autogen** as builder generator. There are many host and target modules, for which action taken is just same. So, to avoid repetition of text, those modules are named in definition file, and then template file is used to finally define `all, check and install` targets using these definitions. The sizes of input `Makefile.*` and `Makefile.in` does not differ much in this case, as Autogen cannot intelligently detect dependencies unlike automake.

- GCC uses **Cygnus configure** as configurator. It detects various system dependent variables that are needed by GCC to get successfully built, either as native compiler or cross compiler. It takes `config.sub, config.guess, configure.in, Makefile.in` generated by `autogen` as an input, along with makefile fragments in `config` directory which contain specifications of hosts or targets like for processors arm, hp, sparc; or versions sysv, sysv4, sysv5 etc. The `configure` generates `Makefile` along with `config.status` in build directory. If we try to generate `configure` script using `autoscan` and `autoconf` tools, the size of `configure` file generated if `autoconf` were used turns out to be very large as compared to `Cygnus configure` file.(`Autoconf` generates `configure` of 5432 lines whereas size of `Cygnus configure` is 1609 lines.)

2 Detailed study of tools

2.1 Configurator generator tools

2.1.1 autoscan

1. **Advantages using autoscan**
   The `autoscan` program helps to create and maintain a `configure.`[`in/ac`] file for a software package. `autoscan` examines source files in the directory tree rooted at a directory given as a command line argument, or the current directory if none is given. It searches the source files for common portability problems and creates a file `configure.scan` which is a preliminary `configure.`[`in/ac`] file.

2. **Output of autoscan**
   The `configure.scan` file so generated contains checks for

   - programs
   - library archive files
   - header files

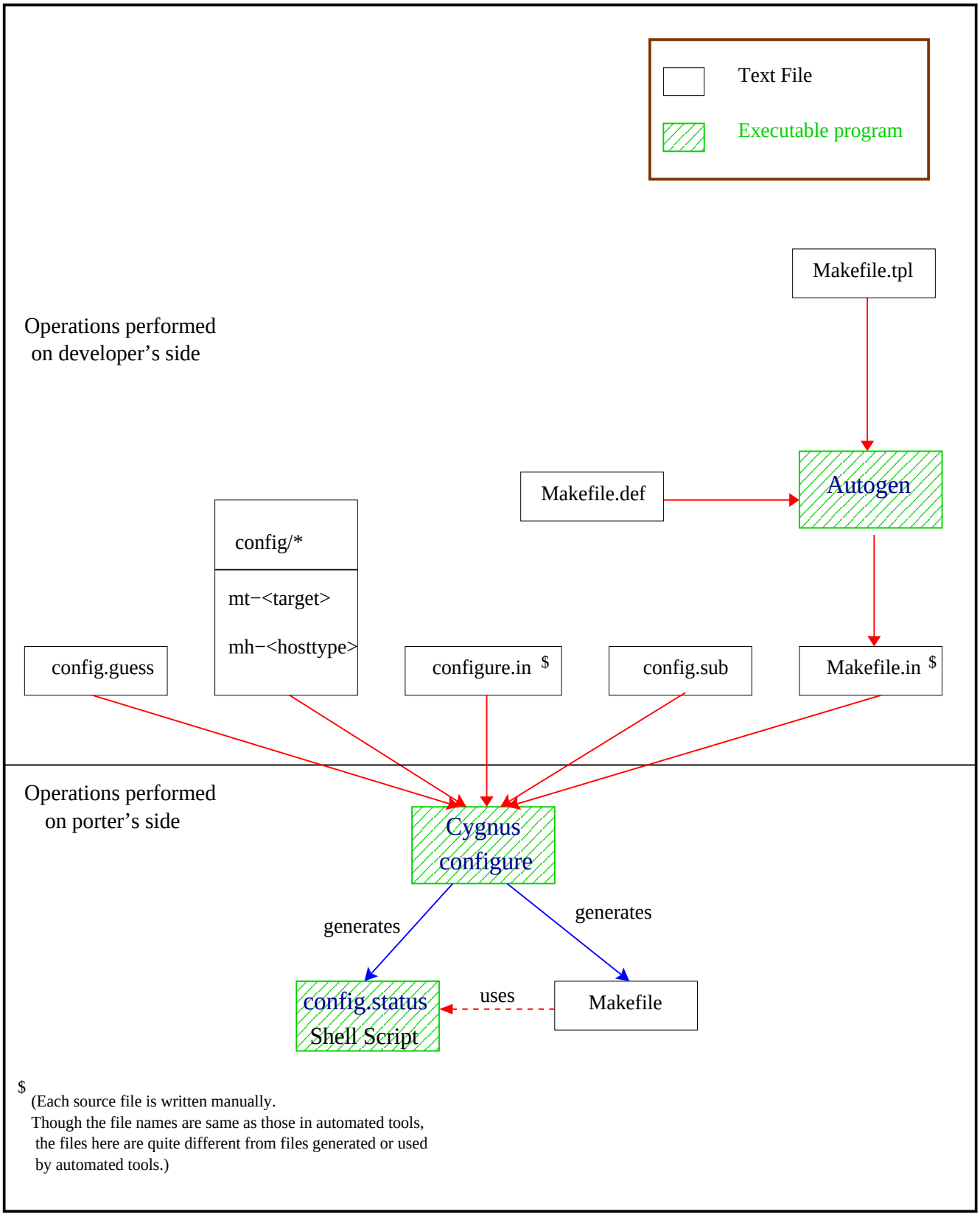

Figure 2: Configure and Build tools used for configuring and building GCC.

- particular C library functions
- typedefs, structures, and compiler characteristics

`autoconf` has a set of macros for the system features which have already demonstrated their usefulness. Hence, `autoscan` checks for these default features which are needed by package for which finally `configure` script is to be generated. Moreover, `autoscan` has its own list of features which are used frequently. Those features are also defined using generic `autoconf` macros. So, checks for those features are generated by `autoscan.`

3. **Short comings of autoscan**
   The `autoscan` script is written in perl. After looking at the perlscript for `autoscan`, we can infer that it has predetermined list of programs, library archives, header files, C library functions and types. It scans the files for known directives and extract the information. Hence, `autoscan` can not generate checks for features other than these default features. Such features are needed to be added in final `configure.`[`ac/in`] file manually.

   Consider simple example, in which we want to generate .dvi file using given .tex file. `latex` and `texi2dvi` commands are not in list of default programs. Hence, `autoscan` cannot generate check for them. But, our `configure` will need to check whether the command is present on system or not, and accordingly either will generate the makefile, which uses `latex` if the command `latex` is present on system, or `texi2dvi` if `texi2dvi` is present,or give error that makefile cannot be generated, if both are absent. So, the macro will have to be written to check if 'programs' `latex` and `texi2dvi` are present on system in `configure.`[`in/ac`].

   Secondly, there may be some macros which depend upon some other macros as they check previously set value of some variable to decide what action is to be taken. `autoconf` does not consider such dependencies and occasionally outputs a macro in wrong order relative to another macro, so `autoconf` produces warning. Such macros must be moved manually.

   For example, even if the macros for checking for 'programs' `latex` and `texi2dvi` were present in above example, the relation that the `makefile` should use `latex` first, and if `latex` command is absent then only check for `texi2dvi` will not be infered by `autoscan`, which will result in warning at the time of running `autoconf`.

   Hence, when using `autoscan` to create a `configure.`[`ac/in`], one should manually examine and refine `configure.scan` before renaming it to `configure.ac`, it will probably need some adjustments.

2.1.2 autoconf

1. **Advantages of autoconf**
   Configure is the process which tries to guess correct values for various system

dependent variables used during compilation. The configure script individually test for the presence of each feature that the package to be built will require. As size of package goes on increasing the task of writing configure script manually becomes more and more difficult.

`autoconf` is a tool that generates `configure` script to configure source code package to multiple platforms. The `configure` script so generated is independent of `autoconf`. For each software package that `autoconf` is used with, it creates a configuration script from a template file `configure`[`ac/in`] which lists the system features that the package needs or can use. Thus `autoconf` reliabally detects the system-specific build and run-time information.

2. **Input of autoconf**
   `configure.`[`ac/in`] is the template file which is used by `autoconf` to generate configure script. `autoconf` has list of system-specific information which is used extencively. But this may not cover all features that are needed to be checked for specific package. Hence, we may need to write our own tests to suppliment those that come with `autoconf`. These site's or the package's own feature tests can be written into `aclocal.m4` and `acsite.m4` files, which are given as input to `autoconf`.

3. **How does autoconf work?**
   `autoconf` is implemented as a macro expander: a program that repeatedly performs macro expansions on text input, replacing macro calls with macro bodies and producing a pure sh script in the end. Instead of implementing a dedicated `autoconf` macro expander, it is natural to use an existing general-purpose macro language, such as M4, and implement the extensions as a set of M4 macros.
   So, `autoconf` requires GNU M4 in order to generate the scripts. It uses features that some UNIX versions of M4, including GNU M4 1.3, do not have. One must use version 1.4 or later of GNU M4.

4. **Output of autoconf**
   `autoconf` creates shell file `configure` which when run tests for presence of features that the software package to be built will require.

   `autoconf`-generated `configure` scripts need some information about how to initialize, how to find the package's source files and about the output files to produce. This information is taken from script writer through `autoconf` macros.

   There are only two macros that must be defined: `AC_INIT` and `AC_OUTPUT`.

5. **Implementation**
   I have taken a small software package which has 3 different programs to be built in 3 different sub-directories. One program generates .pdf document from .tex file. Other program generates shell script to perform operations similar to `mail` command on unix. Third program generates Dot file parser from grammer provided

using flex and bison I have generated small configure.scan file using autoscan and modified it to have feel of how autoconf configure.in file can be written. Here I will explain macros I have used. I have tried to make use of most of the core autoconf macros, but still, this list of macros is not exhaustive. For complete documentation please refer autoconf manual [3].

- AC_INIT processes any command line options and perform various initializations and verifications. It sets name of package and its version. It also initializes some M4 macros, output variables and preprocessor symbols.

  ```
  AC_INIT(cs699, 1.1.1, sameera@cse.iitb.ac.in)
  ```

  This gives name of package, its version and email address to which users should send bug reports.

- AC_CONFIG_SRCDIR is the macro, which specifies any file in the source, which is used by configure, to make sure that the directory that is told to hold source code actually does hold it.

  ```
  AC_CONFIG_SRCDIR([parul_sameera/myTry.y])
  ```

  Here, the file myTry.y in sameera_parul source directory is given for verification. If the file is not found, the configuration process terminates giving an error message `cannot find sources (parul_sameera/myTry.y) in . or ...`

- AC_OUTPUT generates config.status and launches it. Each configure.[ac/in] file must call this macro once at the end.

  ```
  AC_OUTPUT
  ```

  After configure script is done with system examination, it creates shell script config.status. It then calls config.status. config.status is the script which takes care of actions to be taken depending upon the results of configure. Typical task of config.status is to instanciate files. config.status creates the Makefiles and other files resulting from configuration. There are four instanciating macros : AC_CONFIG_FILES, AC_CONFIG_HEADERS, AC_CONFIG_COMMANDS and AC_CONFIG_LINKS, which are used by macro AC_OUTPUT to create files, commands or links.

- AC_CONFIG_HEADER generates header file given as an parameter from default file <filename>.in. If name of input file is different it can be specified in parameter to the macro after name of file to be created in colon seperated form.

  ```
  AC_CONFIG_HEADER([config.h]
  ```

  This macro generates header file config.h from config.h.in file. seperated form.

- AC_CONFIG_FILES make AC_OUTPUT create each '<file>' by copying an input file (by default '<file>.in'), substituting the output variable values.

```
[if [ "$LATEXVAR" = "yes" ]]
[then]
AC_CONFIG_FILES([bhavana_sameera/Makefile])
[else]
[echo "The latex was not found, hence makefile is not generated."]
[fi]
AC_CONFIG_FILES([parul_sameera/makefile])
```

  Here the code snippet conveys that we can have shell script written in `configure.in` file which is passed to configure script for execution. But it should be noted that the script is needed to be enclosed within the M4 quote characters '[' and ']'. Here the file to be created or not is decided by looking at variable LATEXVAR. If the variable holds zero, then only makefile for directory bhavana_sameera is created else, not. AC_CONFIG_FILES lists the files to be created by `config.status`.

- AC_CHECK_PROG checks whether the program named is present or not. If path is given, it searches the program in that path, otherwise it searches it in default path. If the program is found, it defines the variable whose name is passed to the macro as first argument.

  Running configure in varying environments can be extremely dangerous. The environment variables affecting such situations, are precious variables. It is strongly recomended to declare the variable passed to AC_CHECK_PROG as precious. The way to record the results of tests is to set output variables, which are shell variables whose values are substituted into files that configure outputs. AC_SUBSTis used to create an output variable from a shell variable. Which may be used in creating makefile also.

  The macro AC_ARG_VAR declares variable as a precious variable, and include its description in the variable section of `./configure --help`.

```
AC_SUBST(LATEXVAR)
AC_ARG_VAR(LATEXVAR,[To check whether latex is present])
AC_CHECK_PROG(LATEXVAR,latex,[yes],[no])
```

  Here, the variable LATEXVAR is created as an output variable, its description is also created, and then it is initialised as 'yes' or 'no' depending upon whether the program 'LATEX' present or not. If the program LATEX is not present, then there is no use of creating `Makefile` in sub-directory bhavana_sameera as it requires the program LATEX.

```
[if [ "$LATEXVAR" = "yes" ]]
```

```
[then]
AC_CONFIG_FILES([bhavana_sameera/Makefile])
[else]
[echo "The latex was not found, hence makefile is not generated."]
[fi]
```

It can be seen in this snippet that the variable `LATEXVAR` is used to decide if file `bhavana_sameera/Makefile` is to be created or not.

- `AC_DEFUN` is used to define new macro. When a feature test that could be applicable to more than one software package is written, it is best to encapsulate it as new macro.

```
#AC_DECL_MYHEADERS()
AC_DEFUN([AC_DECL_MYHEADERS],[if [[ -f "$2/$1" ];] then
echo $1 " File found!!!!!";
else
echo $1 " File not found!!!!!"; fi])
AC_DECL_MYHEADERS(myGraph.h, parul_sameera)
```

For example, here the macro `AC_DECL_MYHEADERS` is written to check for header files in given path. This functionality may be needed by many packages, because if any of the user written header file is not present in the package source, the program cannot be built completely from that source package. Hence, that functionality is defined as seperate macro, so that it can now be used by `configure.`[`ac/in`] writer for other packages also. Such autoconf macro archive is available at http://autoconf-archive.cryp.to/

Along with these macros, there are few macros which test for specific files, programs, libraries or types and structures. Those an be found in any autoconf manual [3].

2.1.3 Cygnus configure

Unlike others, `Cygnus configure` is not configurator generator, but configurator itself. It is the shell script which guesses and sets up the environment in which the software will get built correctly, for the underlying system.

1. **Input to Cygnus configure**
   The files used by `Cygnus configure` are

   - configure.in : The shell script fragments recide here. `Configure.in` file is partitioned into 4 sections, which itself is a shell script fragment. `Configure` shell executes each fragment at appropriate time. The parameters to fragments are passed through set of shell variables. The sections in `configure.in` are

per-invocation : `configure.in` file begins with this section. This fragment is executed just after parsing command-line arguments. The variables 'srctrigger' and 'srcname' must be set here.

per-host : This section begins from line beginning with #per-host. This section is invoked when configure is determining host-specific information.

per-target : This section begins with line beginnign with #per-target. Configure invokes the commands in the per-target section when determining target-specific information, and before building any files, directories, or links.

post-target : This section is optional. And if it exists it begins with #post-target. configure invokes this section once for each target after building all files, directories, or links.

- makefile.in : This is the template used by `configure` to generate output `makefile`.
- config.sub : This shell script is used to convert machine name into triplate in the form *cpu-vendor-os*.
- config.guess : If the system type is not specified on command line while invoking configure, then configure script uses this shell script to determine underlying system of host machine.
- config.status : This is the script generated by configure in final stage. This script is then used to generate and maintain makefile.
- config/* : `Configure` uses three types of makefile fragments, which are stored in directory `srcdir/config`. In a generated `Makefile` they appear in the order: target fragment, host fragment, and site fragment, which are used for target, host and site dependent compile time options respectively.

2. **Output of Cygnus configure**
   The `Cygnus configure` generates `makefile`s in each sub-directory, from `makefile.in` template. It also generates `config.status` which can be used by `makefile` to reconfigure itself. It generates `.gdbinit` in build directory, to find program's source code.

3. **How does Cygnus configure work**
   The main purpose of `Cygnus configure` is to guess correct values of system dependent variables. `Cygnus configure` achieves this by doing the following :

   - It generates `makefile` from template `makefile.in` in each relevent subdirectory. Each `makefile` contains variables which have been assigned values according to specifcations.
   - The values of certain shell veriables are given to `configure` either on command line or through `configure.in` file. Depending upon values taken by these

veriables, it customizes the build process as per specifications. If the values taken up by these variables are not specified, then it tries to guess system information using the script `config.guess`.

- It creates build directories in which compiled code will recide before installing.
- It generates `config.status` script which is used by `makefile`, to reconfigure itself whenever needed.

The main disadvantage of `Cygnus configure` is that even though many configuration variables are available, most of the script is needed to be written manually, to test all the features that the software might need. This can be error-prone. The `Cygnus configure` script predates `autoconf`. All of its interesting features have been incorporated into `autoconf`. Hence, new programs should not be written to use the `Cygnus configure` script. However, the `Cygnus configure` script is still used in a few places: at the top of the Cygnus tree like in GCC and in a few target libraries in the Cygnus tree.

2.2 Builder generator tools

2.2.1 automake

1. **Advantages of automake**
   GNU makefile standards document is long, complicated and subject to change. The task of maintaining `makefile`s to these standards is difficult to handle. Hence `automake` is used, which automatically generates `makefile.in` from `makefile.am`.

2. **Input and output of automake**
   `automake` takes `makefile.am` as an input. The input file is collection of different variable definitions along with some rules. It generates `makefile.in` file in each directory in source tree having `makefile.am` file.

   `automake` requires perl in order to generate `makefile.in`. But once the `makefile.in` is generated, it is independent of perl. Secondly, it asumes that the configure file will be generated using the tool `autoconf`. This is because it needs some `autoconf` macro-values to generate `Makefile.in`s.

   The main advantage of `automake` is that, the distribution created by `automake` is fully GNU standards compliant. `automake` provides 3 levels of strictness, i.e. levels to which standards are maintained. This facilitates users who do not want to use all GNU conventions.

   - Foreign : `automake` checks for the constraints which are absolutely necessary for proper functioning.
   - gnu : `automake` tries to be compliant with gnu standards.
   - gnits : `automake` perform checking for gnit standards, which are based on GNU standards, but even more detailed.

3. **How does automake work?**

   To create all Makefile.ins in subdirectories of source, automake is run without arguments. It detects automatically all Makefile.am in source tree, by running autoconf to scan configure.in and its dependencies. It needs some autoconf macros and some variables must be defined in configure.in, using which automake tailors its output.

   The veriable definitions and targets in makefile.am are copied verbatim into makefile.in. automake recognizes some special veriables and add make rules to the output file. These special veriables are used by automake to determine which sources are part of built program or which libraries it is to be linked with.

4. **Implementation**
   automake scans configure.in file for getting some information about software package. The macros which are necessary to be defined in configure.in for this purpose are :

   - AM_INIT_AUTOMAKE : It runs many macros required for proper functioning of generated makefiles. By default it defines package and version using autoconf macro AC_DEFUN.

     ```
     AM_INIT_AUTOMAKE
     ```

   - AC_CONFIG_FILES :
   - AC_OUTPUT : automake uses these two macros to determine which files are to be created. The listed file is considered to be automake generated makefile, if there is <filename>.am file present in directory. i.e. AC_CONFIG_FILES(dir/makefile) will cause automake to generate dir/makefile.in if dir/makefile.am is present.

     ```
     [if [ "$LATEXVAR" = "yes" ]]
     [then]
     AC_CONFIG_FILES([bhavana_sameera/Makefile])
     [else]
     [echo "The latex was not found, hence makefile is not generated."]
     [fi]
     AC_CONFIG_FILES([parul_sameera/makefile])
     AC_OUTPUT
     ```

     In above example, if there are bhavana_sameera/Makefile.am and parul_sameera/makefile.am present then automake generates bhavana_sameera/Makefile.in and parul_sameera/makefile.

   These are the only macros which are neccessary to be defined in configure.in file. The macros which are not issencial but can be used by automake can be found in automake manual [1].

In order to build the program, one needs to tell `automake` which sources are part of it and which libraries are to be linked with.

The name of the program which is to be built in specific directory is specified in `makefile.am` through veriable `bin_PROGRAMS` as `bin_PROGRAMS=<program-name>` Sources to the program are specified through veriable `<program-name>_SOURCES`.

```
bin_PROGRAMS=dotParser
dotParser\_SOURCES=lex.yy.c y.tab.c y.tab.h myGraph.h
```

Here, the executable to be generated is `dotParser`. The sources needed by `dotParser` to get built completely are specified in variable `dotparser_SOURCES`.

2.2.2 imake

`imake` generates `makefile` from template, cpp macro functions (for generating rules) and per directory input file `Imakefile`. This allows machine dependencies to be kept seperate from different items to be built.

`imake` invokes gcc with name of file containing definitions of macros `IMAKE_TEMPLATE, INCLUDE_IMAKEFILE` and including of `IMAKE_TEMPLATE`.

```
#define IMAKE_TEMPLATE "Imake.tmpl"
#define INCLUDE_IMAKEFILE <Imakefile>
#include IMAKE_TEMPLATE
```

The `IMAKE_TEMPLATE` is master template file, by default, which typically reads in a file containing machine-dependent parameters (specified as cpp symbols), a site-specific parameters file, a file defining variables, a file containing cpp macro functions for generating make rules, and finally the Imakefile (specified by `INCLUDE_IMAKEFILE`) in the current directory. Imake.tmpl file written by me looks as simple as

```
#include "Imake.rules"
#include "Imakefile"
```

The Imakefile uses the macro functions to indicate what targets should be built; imake takes care of generating the appropriate rules. For example, in the example I wrote, it uses macros `runGcc, runLex, runYacc, clean` defined by me in `Imake.rules`.

```
runGcc(dotParser, myTry.y.c lex.yy.c)
runLex(lex.yy.c, myTry.l)
runYacc(myTry.y.c, myTry.y)
clean(dotParser lex.yy.c myTry.y.c myTry.y.h myTry.y.output)
```

Imake configuration files contain 2 type of variables, make variables and imake variables. The imake variables are interpreted by cpp when imake is run. The make variables are

written into the Makefile for later interpretation by make.

The rules file (usually named `Imake.rules` in the configuration directory) contains a variety of cpp macro functions that are configured according to the current platform. `Imake` replaces any occurrences of the string @@ with a new-line to allow macros that generate more than one line of make rules. Imake also replaces any occurrences of the word XCOMM with the character # to permit placing comments in the Makefile without causing invalid directive errors from the preprocessor. My `Imake.rules` file looks like

```
XCOMM This is to define rules

#define runLex(dest, src)       @@\
dest::src       @@\
        flex src

#define runYacc(dest, src)      @@\
dest::src       @@\
        bison -o dest src -tvd

#define runGcc(dest, src)       @@\
dest::src       @@\
        gcc -o dest src -ll

#define clean(src)      @@\
clean::         @@\
        rm src -rf
```

Here, `dest, src` are `Imake` variables. The macro gets expanded when it is invoked from `Imakefile`. Some complex imake macros require generated make variables local to each invocation of the macro, often because their value depends on parameters passed to the macro. Such variables can be created by using an imake variable of the form `XVARdefn`, where n is a single digit. A unique make variable will be substituted. Later occurrences of the variable `XVARusen` will be replaced by the variable created by the corresponding `XVARdefn`.

2.3 Generic tools

2.3.1 autogen

1. **Advantages of autogen?**
   `autogen` is a tool designed for generating program files that contain repetitive text with varied substitutions.Its goal is to simplify the maintenance of programs that contain large amounts of repetitious text. This is especially valuable if there are several blocks of such text that must be kept synchronized in parallel tables.

The main features of `autogen` are:

(a) The definitions are completely separate from the template. By completely isolating the definitions from the template it greatly increases the flexibility of the template implementation. A secondary goal is that a template user only needs to specify those data that are necessary to describe his application of a template.

(b) Each datum in the definitions is named. Thus, the definitions can be rearranged, augmented and become obsolete without it being necessary to go back and clean up older definition files, thereby reducing incompatibilities.

(c) Every definition name defines an array of values, even when there is only one entry. These arrays of values are used to control the replication of sections of the template.

(d) There are named collections of definitions. They form a nested hierarchy. Associated values are collected and associated with a group name. These associated data are used collectively in sets of substitutions.

(e) The template has special markers to indicate where substitutions are required, much like the $VAR construct in a shell here doc. These markers are not fixed strings. They are specified at the start of each template. Template designers know best what fits into their syntax and can avoid marker conflicts.

(f) These same markers are used, in conjunction with enclosed keywords, to indicate sections of text that are to be skipped and for sections of text that are to be repeated.

(g) Finally, we supply methods for carefully controlling the output. Sometimes, it is just simply easier and clearer to compute some text or a value in one context when its application needs to be later. So, functions are available for saving text or values for later use.

2. **input to autogen**

Input to `autogen` is divided into two files:

Definition file : In order to instantiate a template, one normally must provide a definitions file that identifies itself and contains some value definitions.The definitions file is used to associate values with names. Every value is implicitly an array of values, even if there is only one value. Values may be either simple strings or compound collections of name-value pairs.

- Identification definition : The first definition in this file is used to identify it as a `autogen` file. It consists of the two keywords, 'autogen' and 'definitions' followed by the default template name and a terminating semi-colon.

  ```
  AutoGen Definitions template-name;
  ```

- Named Definitions : There are two kinds of definitions, 'simple' and 'compound'.

  ```
  compound\_name '=' '\{' definition-list '\}' ';'
  simple\_name '=' string ';'
  no\_text\_name ';'
  ```

  Here, definition-list is a list of definitions that may or may not contain nested compound definitions.

Template file : The `autogen` template file defines the content of the output text. It is composed of two parts. The first part consists of a pseudo macro invocation and commentary. It is followed by the proper template. This pseudo macro is special. It is used to identify the file as a `autogen` template file, fixing the starting and ending marks for the macro invocations in the rest of the file, specifying the list of suffixes to be generated by the template and, optionally, the shell to use for processing shell commands embedded in the template.

3. **How does autogen work?** `autogen` copies text from the template to the output file until a start macro marker is found. The text from the start marker to the end marker constitutes the macro text. `autogen` macros may cause sections of the template to be skipped or processed several times. The process continues until the end of the template is reached. The process is repeated once for each suffix specified in the pseudo macro.

4. **Implementation** `autogen` can be used to generate various files from template and definition files. It is not restricted to generate `makefile`s only. But, GCC uses `autogen` to generate `Makefile.in`. It should be noted that `Makefile.tpl` written holds all code that is required in `Makefile.in`. Only the places where same operation is to be performed on verious data, the autoGen macros come into picture.

   To show how `makefile` can be written using `autogen` I have tried to generate small `makefile` for a project in java, where repeatatively same commands are needed to be executed on different input files. Here, are all the macros I have used.

   Template file: Each template file must start with pseudo macro which tells Autogen how to process a template. In pseudo macro we specify

   - The set of characters to demarcate start of macro. In this case, @: is macro demarker.
   - The start marker must be followed by keywords `Autogen5` and `template`.
   - Zero or more suffix specifications to tell `autogen` how many times template is to be processed. In this case, only one file `makefile.in` is to be generated, hence suffix `in` is specified.

- comment lines.
- End-macro marker. In our case, end macro marker is :@.

```
@: Autogen5 template
in
:@
```

The part written outside the macro-markers is copied into the output file as it is. The part into macro-markers is processed as per definitions in definition file(explained later) and code is generated.

eg:

```
# This is the comment in makefile.

SRCS=@:FOR mfile:@@:filename:@.class @:ENDFOR mfile:@
```

This part gets expanded as

```
# This is the comment in makefile.

SRCS=Database.class Databasej.class Databaseq.class EServlet.class
FriendReq.class Home.class MYLdap.class Logout.class Pending.class
Search.class SearchPage.class TServlet.class ViewAll.class
Request.class TestHttp.class TestRequest.class
```

Here, the statements between start and end markers "@:" and ":@" are considered as macro text, and processed to generate code as per definitions in definition file.

There are many `autogen` macros like `FOR, CASE, IF, DEFINE, EXPR, WHILE`, etc. which can be used in templates to expand various definitions in definition file.I have used the macro `FOR` which uses value-name in definition list and generates list of all source files.

Definition file: Each definition file must start with identification definition, which consists of two keywords `autogen` and `definitions` followed by default template name and terminating semi colon. This template name is used to find corresponding template file.

```
Autogen Definitions makefile.tpl;
```

Any name can have multiple values associated with it in definition file. For, eg, in our example the name `mfile` is compound name which is having multiple instances.

```
mfile={filename="Database";};
mfile={filename="Databasej";};
mfile={filename="Databaseq";};
mfile={filename="EServlet";
dep[0]="Databaseq";
dep[1]="Home";};
mfile={filename="FriendReq";
dep[0]="Databaseq";
dep[1]="Home";
dep[2]="Database";};
mfile={filename="Home";
dep[0]="Databaseq";};
:
:
:
```

Hence, all these copies are expanded using macro `FOR` in template file. in `mfile={filename= "FriendReq";dep[0]="Databaseq"; dep[1]="Home";dep[2]="Database";};`, there are 2 definitions, one to name `filename` and other to array `dep`.

In case of arrays, all values in the range need not be filled, in which case the array becomes sparse array. The macro `FOR` takes care of sparse array, while emiting the code. The difinition

```
mfile={filename="SearchPage";dep[0]="Databaseq";dep[2]="Database";};
```

generates the code

```
SearchPage.class:chirkut/SearchPage.java Databaseq.class
Database.class chirkut.conf
```

when emited by template

```
@:FOR mfile:@@:filename:@.class:chirkut/@:filename:@.java
@:FOR dep:@@:dep:@.class
@:ENDFOR:@chirkut.conf
```

This part is just an introduction to how to write definition and template files for `autogen`. For more details please refer `autogen` manual [2].

3 Conclusion

This report discusses how to use important GNU configure and build tools by actually demonstrating them on simple software packages. Depending upon type of software being

configured and built, specific configuration and build tool is selected. The tool proving very good for one package need not work that efficiently with other software. Hence while choosing build and configure tools for specific software, the structure and requirements of the software are needed to be considered. The documentation available for these tools is not very good, and does not convey for which type of software, which tool is to be used. Hence, a lot of experimentation may be required to determine appropriate configure and build tool for specific software. The examples I have generated for this report are available at www.cse.iitb.ac.in/ sameera/seminar.

References

[1] http://www.gnu.org/software/automake/manual/html_mono/automake.html

[2] http://www.gnu.org/software/autogen/manual/html_mono/autogen.html

[3] http://www.gnu.org/software/autoconf/manual/autoconf-2.57/html_mono/autoconf.html

[4] http://www.airs.com/ian/configure/configure_toc.html

[5] http://www.gnu.org/software/autoconf/manual/autoconf-2.57/html_mono/autoconf.html#SEC11

[6] http://www.gnu.org/software/automake/manual/html_mono/automake.html #Invoking%20aclocal