Using Clearmake with Gnu Makefiles

Overview

1. Why use Gnu-compatible mode ?

2. Implementation strategy.

3. Recent improvements in clearmake Gnu-compatibility.

4. Remaining incompatibilities.

5. User experiences using Gnu-compat.

Why use ‘clearmake -C gnu’ ?

- Makefile portability
  - ***into*** Clearcase
  - ***out of*** Clearcase

- Intrinsic reasons...

Gnu Make features

BSD Make + System V Make + Gnu additions/generalizations

- ‘VPATH’
- **included Makefiles**
- **pattern substitution in variable refs**
- **pattern rules**
- **...**

- **string manip. functions**
- **conditional Makefiles**
- **static pattern rules**
- **pattern rules/multiple targets**
- **selective** ‘vpath’ **search**
- **simply-expanded variables**
- **include-file search-path**
- **‘.PHONY’ target declaration**
- **...**

<http://www.ns.utk.edu/gnu/make/make_toc.html>

Implementing Gnu-compatibility

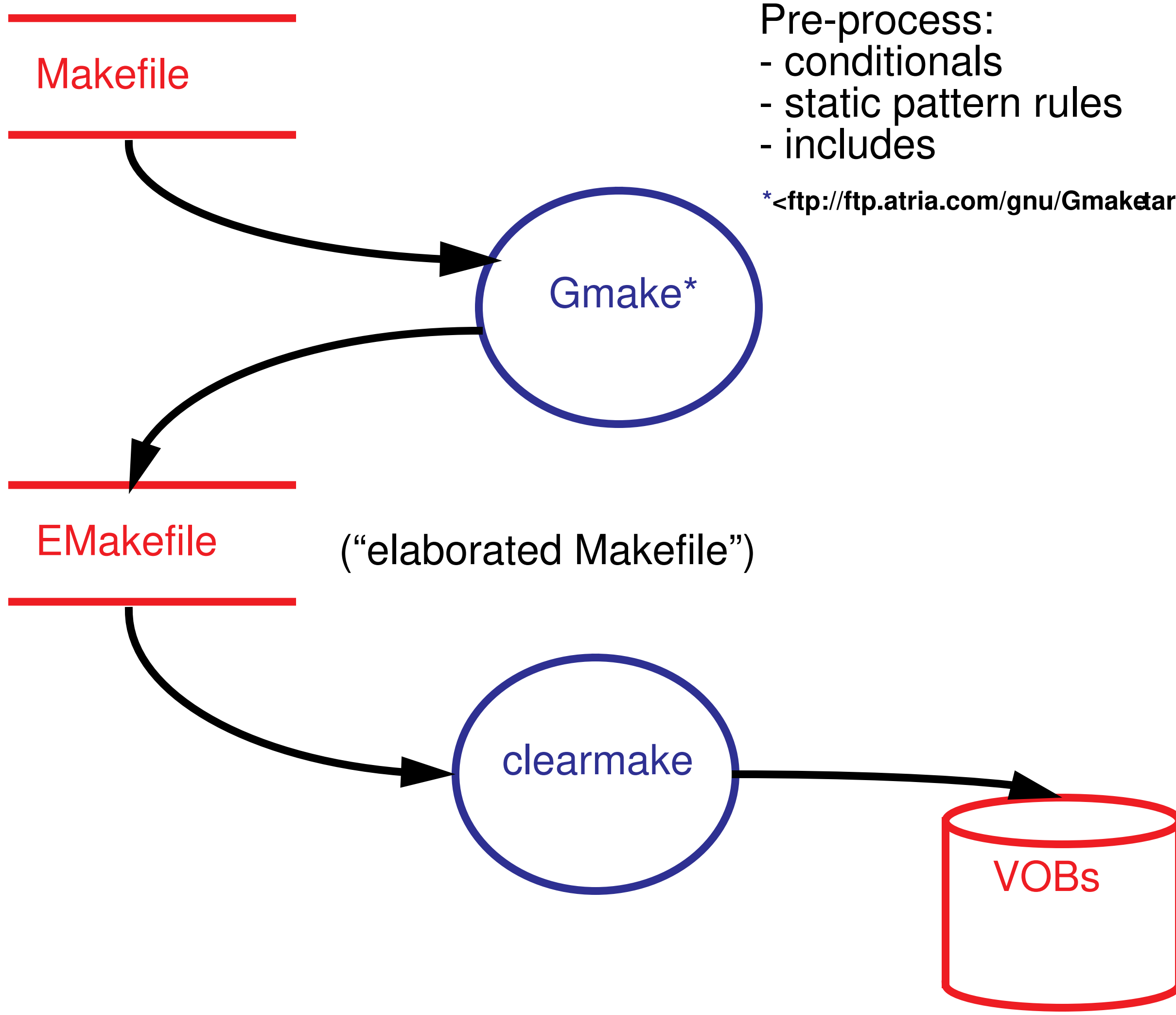
Choices:

1. add Clearcase features to Gnu Make, eg:
   - parallel/distributed building
   - Clearcase Makefile semantic declarations (.NO_CMP_SCRIPT, .NO_CONFIG_REC, .NOTPARALLEL, .NO_WINK_IN, ...)
   - BOS-file (Build Option Spec) files
   - ...
2. add Gnu Make features to Clearmake ✓
   - ***pro***: no legal issues with FSF.
   - ***pro***: 'gmake -p' ==> (somewhat) vanilla Makefile, allowing incremental implementation of compatibility.
   - ***con***: not 100% compatibie.

Implementing Gnu-compatibility

Pre-process:
- conditionals
- static pattern rules
- includes

*<ftp://ftp.atria.com/gnu/Gmake.tar.gz>



Recent improvements in Gnu-compatibility

Clearcase V2.0.2 + latest patches:

- ‘**VPATH**’ improvements (‘**$<**‘ correct).
- corrrect semantics for ‘**vpath**’.
- correct semantics for simply-expanded macros.
- correct implicit rule search.
- ‘.PHONY’ target declarations.
- permit multiple `%’ on right hand side of pattern rule.
- recognise ‘**+**’ build rule prefix (“always execute, overrides ‘-n’ “).
- escaped comment chars in build scripts: ‘**\#**’.

Recent improvements in Gnu-compatibility

Clearcase V2.1 (available soon):

- *full* support for all text manipulation functions:

  dir, notdir, basename, suffix, wildcard, addprefix, addsuffix, join,

  filter, filter-out, sort,

  word, words, firstword,

  strip, findstring, subst, patsubst,

  shell, foreach, origin.

- Gmake based on latest Gnu Make (version 3.74).

- ability to specify '**CLEARCASE_MAKE_COMPAT**' in a local BOS file.

Remaining incompatibilities

More important

1. **VPATH/vpath** and automatic variables '$^', '$?'.
2. Passing options to sub-makes:

   '-**I** include_path',

   '-r'.

   Workaround: explicitly set environment variables for sub-makes.
3. Static pattern rules and '$*'.

   Workaround: use the string builtin functions to edit '$<' or '$@'
4. New automatic variable: '?+': list of (non-unique) dependencies.
5. Command line option : `--directory=DIR'.
6. Multi-line macro 'define/endef'.

   Workaround: use ordinary variable assignment.
7. The `unexport' directive.

Remaining incompatibilities

Less important

1. Automatic Makefile regeneration.
2. 100% command-line compatibility.
3. Library dependencies written like ‘**-lname**’.

Nits

1. Multiple makefiles named in one ‘include’ directive.
2. ‘**MAKEFILES**’ environment variable.
3. The “what if” command-line option (`-W’).
4. Reading the makefile from stdin.
5. Deletion of intermediate files created by implicit rule-chaining.

Gnu Make features: conditionals

Eg 1:

```
libs_for_gcc = -lgnu
normal_libs =


foo: $(objects)
ifeq ($(CC),gcc)
        $(CC) -o foo $(objects) $(libs_for_gcc)
else
        $(CC) -o foo $(objects) $(normal_libs)
endif
```

Eg 2:

```
ifndef ARCH
ARCH := $(shell machine)
endif
```

Gnu Make features: static pattern rules

Syntax:

```
TARGETS ...: TARGET-PATTERN: DEP-PATTERNS ...
        COMMANDS
```

Eg 1:

```
srcs := $(wildcard *.c)
objects := $(patsubst %.c,%.o,$(srcs))
mf_objects := $(filter mf_%.o, $(objects))
bld_objects := $(filter bld_%.o, $(objects))
$(mf_objects): %.o: %.c
        $(CC) -c $(MF_CFLAGS) $< -o $@
$(bld_objects): %.o: %.c
        $(CC) -c $(BLD_CFLAGS) $< -o $@
```

Eg 2:

```
bigoutput littleoutput : %output : text.g
        generate text.g -$* > $@
```