Bell-Northern Research
VHDL Group

Automatic Generation of Makefiles for VHDL
Application Note

Himanshu M. Thaker
Dept. 5S71

May 31, 1991

1. Introduction

This document describes how to automatically generate makefiles for VHDL source code, containing dependency rules as per document "Writing Makefiles for VHDL models" by Janick Bergeron. There is a tar'd and compressed file in (SITE NOT YET FOUND) on the machine (SITE NOT YET FOUND) containing the source code, instructions on creating the executable, and a troff man page.

It is assumed that the reader is familiar with VHDL, and somewhat familiar with make files (at least knows what they do, and has seen them before... for those who are not, read the man pages on make, to obtain sufficient knowledge).

2. Usage

vmkr [options] file1 file2 ...

To use the VHDL makefile maker, list the files you wish vmkr to examine and determine interdependencies. For example,

vmkr -ts *.vhd > makefile

will parse all of the files in the current directory ending in .vhd, determine the interdependencies, and generate a makefile, called "makefile", based on those dependencies, for the Synopsys toolset.

The files need not be restricted to the current directory only. For example,

vmkr *.vhd ../otherdir/*.vhd > makefile

will parse files from the current directory as well as the files from otherdir.

If vmkr exits with -1, then there was a warning or error message generated. Examine the makefile generated to determine what the message was.

NOTE THAT VMKR EXPECTS THERE TO BE ONLY ONE DESIGN UNIT PER FILE!!! If this rule is violated, vmkr may not work properly. You can use the program vsplit to comply with this rule.

Refer to the man pages for more information on how to use vmkr.

3. Options

Vmkr has four option flags. They are :

- -a This is used to *suppress* the code to make all of the "loose" units. A "loose" unit (e.g. package, entity, architecture) is one that is not used by a top level architecture or configuration, and thus not entered into the dependency list.
- -c This option is there for historical reasons, and new users probably don't want to use it. It is used to keep the vmkr "state" files in the current directory.
- -l This option is used to indicate to vmkr to include dependencies on packages, architectures, etc. that reside in other libraries, and are not in the list of files passed to vmkr. The name of the libraries will be identified as variables near the top of the output generated, and it is up to the user to assign the path to the libraries to these variables.
- -t This option is used to generate makefiles for other toolsets. Currently, it supports In-

termetrics, Vantage, Synopsys, Modeltech, Dasix, and Racal-Redac. It also has an option for VHDL-XL. However, it cannot be used with the XL toolset, as you must be within their own environment to execute their compile commands. However, vmkr can be used to create the makefile, and then you can extract the order of execution required by "make -n".

4. Output

Vmkr creates output that is divided into five regions. These are : user modifiable variables, package and entity-architecture dependencies on other packages, top level architecture and configuration dependencies on other entity-architectures and configurations, "loose" list, and finally, the update rules, to analyse, model generate, and possibly build, basic units.

5. Dependency Rules

Primary units are : entity declarations, package declarations, and configuration declarations. Secondary units are : package body, and architecture body. Order of analysis is governed by three rules. These rules are :

1) A secondary unit can be analyzed only after its corresponding primary unit has been analyzed.

2) Any library unit that references a primary unit can only be analyzed only after that primary unit has been analyzed.

3) An architecture body referenced in a configuration declaration must be analyzed before the configuration declaration; likewise, one configuration declaration referenced in a second configuration declaration must be analyzed before that second configuration declaration.

- above quoted from VHDL : Hardware Description and Design

- by Lipset et al, pg 40

Intermetrics VHDL design Environment (VDE) User’s Manual gives the following dependency rules regarding the order of analysis:

1) A package body depends on its package declaration. A package body may not be analyzed into a library unless the package declaration is present in the same library, although the package declaration and the package body do not have to be analyzed at the same time.

2) An architecture body depends on its entity declaration. An architecture body may not be analyzed into a library unless the entity declaration is present in the same library, but the entity declaration and the architecture body do not have to be analyzed at the same time.

3) Any unit that contains a reference to a primary unit (package declaration, entity declaration or configuration declaration) depends on that primary unit. The referenced primary unit will have been made visible by a VHDL use clause or a VHDL library clause and a VHDL use clause.

Intermetrics also gives the following rules for the order of model generation

1) A package declaration must be model generated before its corresponding package body can be model generated.

2) A package declaration must be model generated before any unit that uses the package declaration (including an architecture body of an entity declaration that uses the package declaration) can be model generated.

3) All the architecture bodies and configuration declarations implied by a configuration declaration must be model generated before the configuration declaration itself can be model generated.

5.1. Some Definitions

There are four phases required in executing VHDL code. They are :

- analysis -this is where syntactic and static semantic analysis occurs
- model generation - high level code is generated from VHDL, then compiled and placed into the design database
- build phase - a simulation kernel is created by linking the object modules (created by the model generation phase)
- simulation phase - execution of the kernel created by the build phase

The VHDL toolset does not affect the timestamps of the VHDL source codes. Thus, it is not possible to determine in what state of execution the toolset is, simply from examining the timestamps of the source files. One solution around this, to keep the state information, is to create 'dummy' files, whose existence (along with their timestamps) store this information (even though these 'dummy' files are empty).

Vmkr creates and depends on these files for such information. By definition, these files will begin with a '.', so that when the 'ls' UNIX command is executed (with no options, of course), the 'dummy' files are not listed. Three extensions are defined. A file ending in :

- ".ana" indicates that the corresponding unit has been analysed
- ".mg" indicates that the corresponding unit has been model generated, and
- ".bd" indicates that the corresponding unit has been built.

5.2. A Quick Review of the Unix make command

Vmkr examines the VHDL files presented to it, and after building an internal representation of the dependencies, generates statements that are compatible with the UNIX make command. The make command requires statements to be of the following format :

```
TARGET : dependency1 dependency2 ... dependencyn
        command1
        ...
        commandn
```

When asked to make the TARGET (either directly from the user, or from another statement within the makefile), make examines the timestamp of TARGET, and compares it to the timestamps of its dependencies. If the TARGET is older than any of the dependencies, then the commands are executed. There is a special command called "touch $@" which touches, or changes the timestamp of its TARGET to the current time. Thus, if the touch is the last command in the list, after executing all the commands, TARGET will be made up to date relative to its dependencies (because its timestamp now will be newer than the rest).

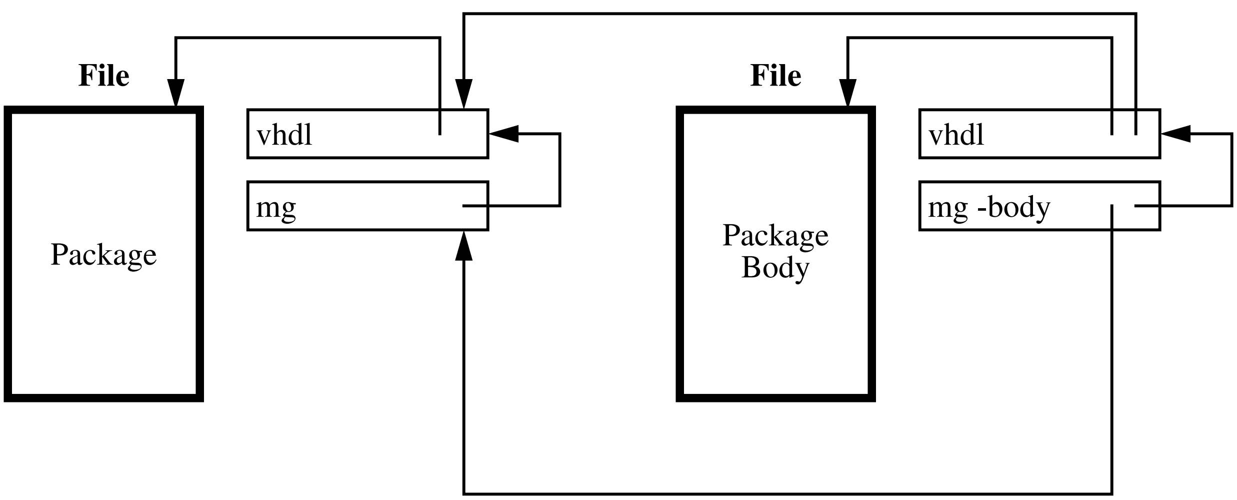
Any command that is preceded by an @ will not be echoed to the terminal. Thus "@touch $@" executes silently.

5.3. Graphical Representation of the Dependency Rules

The dependencies are created based on the rules presented in section 5. on page 3. The following illustrates graphically what those rules indicate.

5.3.1 Package and Package Bodies

A package and package body have the following dependencies.



Package and Package Body Dependencies

Figure 1

This diagram indicates that :

1) the analysis (vhdl) of the package depends on the file that the package resides in.

2) the model generation (mg) of the package depends on the analysis.

3) the analysis of the package body depends on the file that the package body resides in, as well as the analysis of the package.

4) the model generation of the package body depends on the analysis of the package body, as well as the model generation of the package.

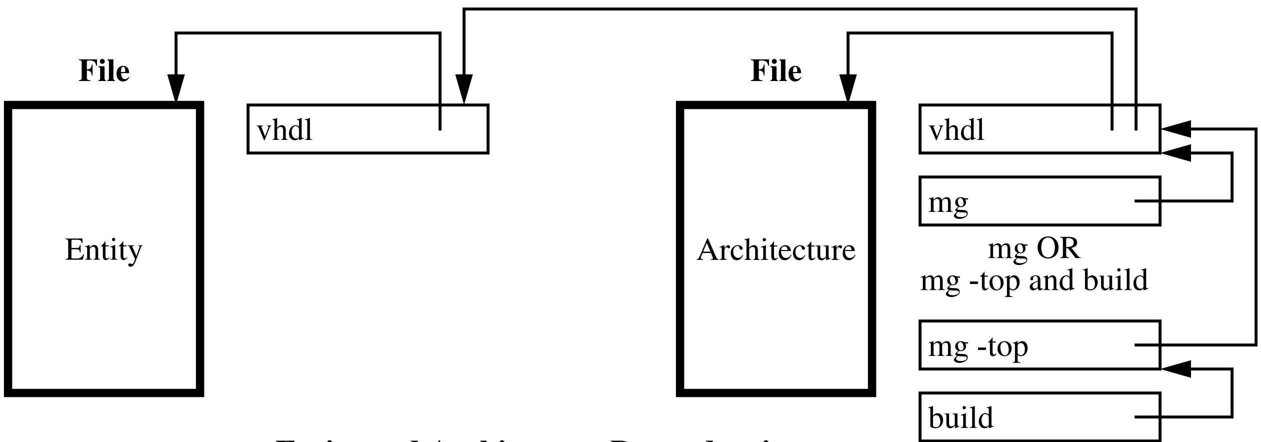
As an example of dependency changes, the above diagram indicates that if the package should change, then the dependency rules would indicate :

- the package should be reanalysed
- then, the package body should be analyzed, and the package should be model generated (it does not matter, without additional dependency rules, which occurs first)
- finally, the package body should be model generated.

There are a few variations as to how package and package bodies can exist. For example, the package and package body may reside in the same file. Additionally, a package body may not exist. Modification of the rules are straightforward to accommodate these variations.

5.3.2 Entities and Architectures

Entity and architecture dependencies are shown in Figure 2. Note that an entity could reside in the



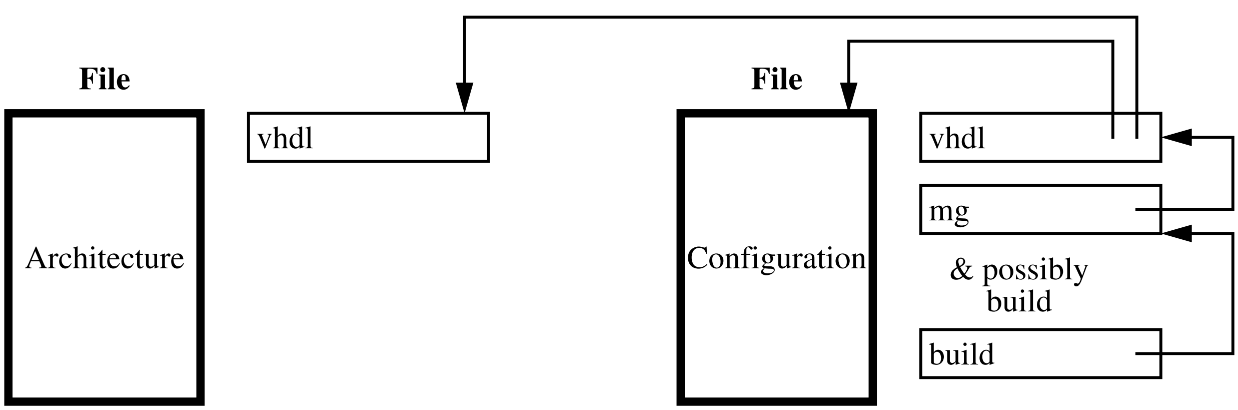
Entity and Architecture Dependencies

Figure 2

same file as the architecture. Additionally, there could be multiple architectures for every entity (but not vice versa).

5.3.3 Configurations

Configuration dependencies are shown in Figure 3. A configuration could reside in the same file as the entity or architecture. These variations have simple changes to the dependency diagrams, and are not illustrated here.

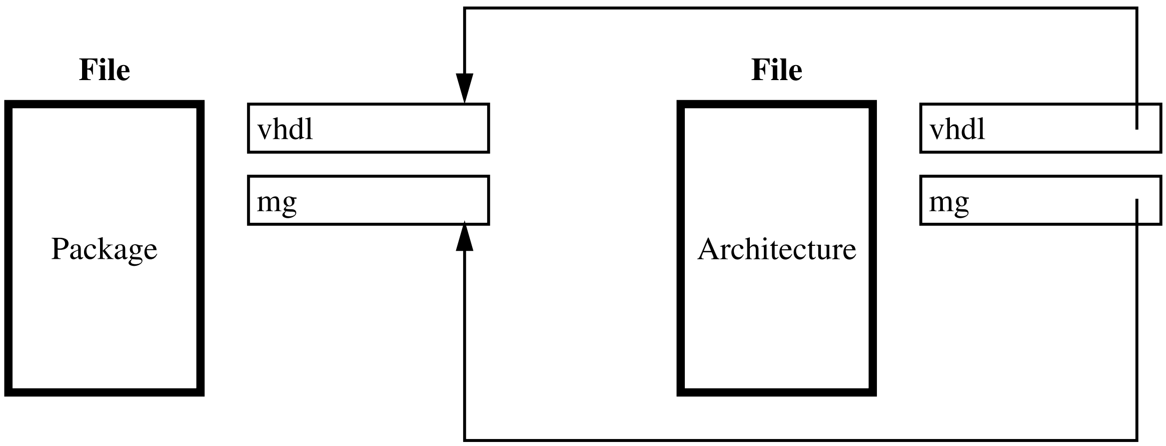


Configuration Dependencies

Figure 3

5.3.4 Additional Dependencies - Architectures, Entities and Packages

If an architecture uses a package, then the analysis of the architecture is dependent on the analysis of the package. Similarly for the model generation, the architecture is dependent on the model generation of the package. If an entity uses a package, then the analysis of the entity is dependent on the analysis of the package, and all of the architectures of the entity (recall that there can be more than one architecture per entity) depend on the package being model generated before they can be model generated. Note that the package body does not come into play, whether it exists or not. The graphical depiction of these dependencies are shown in Figure 4 and Figure 5.



Architecture Depends on Package

Figure 4

5.3.5 Additional Dependencies - Architectures and other Entity-Architecture Pairs

If an architecture depends on another entity architecture pair (i.e. the USE ENTITY clause, when binding a component to a specific entity-architecture), then the model generation of the architecture is dependent on the analysis of the entity being used, as shown in Figure 6.

5.3.6 Additional Dependencies - top level kernels on Architectures and Configurations

Before a top level kernel can be built for an architecture (or configuration, though not depicted in the graphical representation), all of the architectures and configurations used (recursively, that is, not only the ones used, but the ones used by the ones used, etc.) must be model generated.

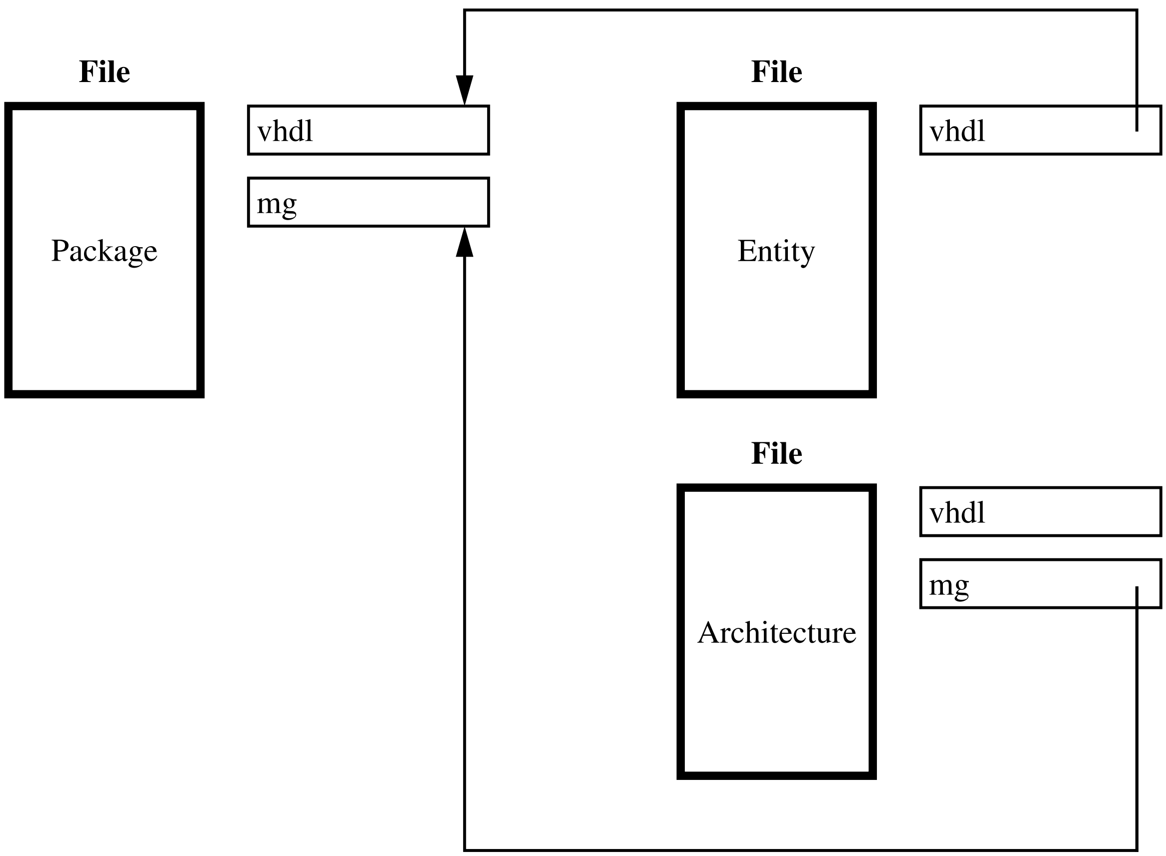
5.3.7 Additional Dependencies - Configurations on its components

Before a configuration can be analyzed, all its components must be analyzed, recursively. Additionally, before a configuration can be model generated, all its components must also be model generated, recursively.

6. Concluding Remarks

The knowledge base for make (the . files) can occasionally get corrupted (e.g. by accidentally touching a .file). When this happens, make may not be able to determine the order of execution required. One solution could be to remove all of the . files, and start from scratch.

You will probably wish to tailor the makefile for your personal taste. Examples of changes include deleting unnecessary top level dependency information, adding a vls setlib command to ensure that

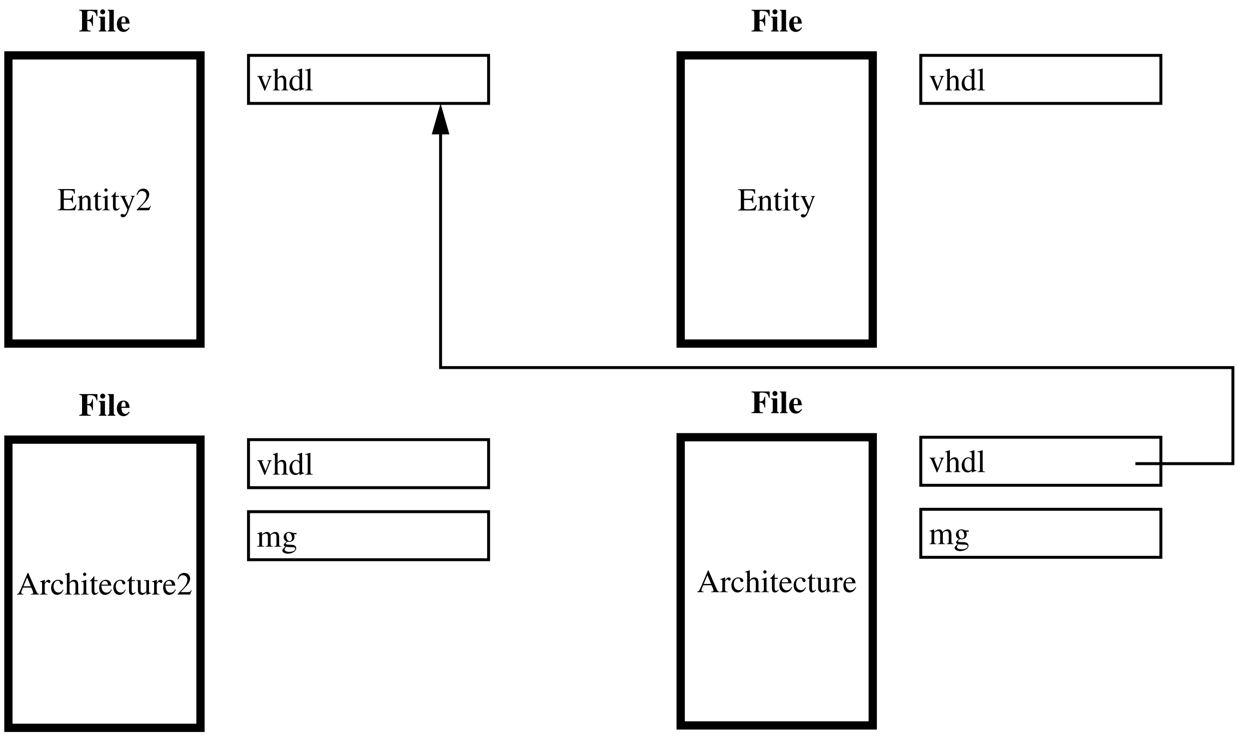


Entity Depends on Package

Figure 5

the compiler compiles into the intended vls directory, adding a sim command, adding other dependencies, etc.

Any information on bugs, suggestions for improvements, or general comments would be welcome. Please email these to hemi@bnr.ca. A description of how vmkr works can be found in the document titled "Internal workings of vmkr", by Himanshu M. Thaker.



Architecture Depends on an Entity-Architecture Pair

Figure 6