Managing VHDL Models with Makefiles

QUALIS™

Janick Bergeron
Qualis Design Corporation
janick@qualis.com

Abstract

This methodology brief explains how makefiles and the Unix utility *make*(1) are used to efficiently manage VHDL models and deal with the complexities of compilation dependencies. It also demonstrates how makefiles can easily be generated and maintained using the productivity tools *vmk* and *lmk*.

A Makefile Primer

A makefile is a configuration and control file for the UNIX utility *make*. This utility has proven to be so powerful and popular that it is available on other operating systems. Software engineers have long been confronted with the problem of keeping software systems up-to-date, making sure that all modified files were properly recompiled and that all required components were available.

Before we tackle makefiles for VHDL, we must become familiar with their original intended use: maintaining C programs. Figure 1 shows three inter-related C source files: a header file (`utils.h`) declaring externally available functions in a corresponding utility source file (`utils.c`) and a top-level source file (`main.c`) that uses some utility functions.

Figure 1: Example of C source code

utils.h:

```
char* get_userid(int uid);
```

utils.c:

```
#include "utils.h"
char* get_userid(int uid)
{
   ...
}
```

main.c:

```
#include "utils.h"
int main(
{
   ...
   name = get_userid();
   ...
}
```

If a change is made to the file `utils.h`, both files `utils.c` and `main.c` should also be recompiled to ensure that they all agree on the usage of the functions declared in the header file. Similarly, if the file `utils.c` is changed, only it needs to be recompiled; however, all software systems that use it must also be rebuilt. In this example, an ANSI compiler would report the fact that the call *get_userid* in the file `main.c` lacks an argument. Since neither the operating system nor the C language enforce compilation requirements, not recompiling the file `main.c` would not prevent building the software system but a run-time failure would occur. Finding the cause of the problem would require the use of a source-

© 1997 Qualis Design Corporation

Permission to copy or distribute available upon request.

level debugger and significantly more effort than would have been required had the compiler found the error.

The brute-force approach to solving this problem would be to compile each and every file whenever a change is made to any file. This may be acceptable for small software systems, but it would be prohibitively inefficient in large systems composed of dozens of large files. A more efficient approach would be to define dependency rules indicating what needs to be done if a specific file changes. A file specifying such dependency rules is called a makefile and the *make* program performs any necessary steps required to update a specific software system based on the rules in the makefile.

Makefile rules use the file modification times to determine if a file is out-of-date with respect to another. If a file is found to be out-of-date, it is updated using one or more user-specified commands. Figure 2 shows the makefile for the example in Figure 1 and defines the following rules:

- The object file `utils.o` must be updated if either of the files `utils.c` or `utils.h` are newer (i.e. have been modified since the last compilation).
- The object file `main.o` must be updated if either of the files `main.c` or `utils.h` are newer.
- The binary file `doit` must be rebuilt if either of the object files `main.o` or `utils.o` are newer

Figure 2: Makefile for the Example C Source Code

makefile:

```
utils.o: utils.c utils.h
	cc -c utils.c

main.o: main.c utils.h
	cc -c main.c

doit: main.o utils.o
	cc -o doit main.o utils.o
```

The entire software system *doit* can now be brought up-to-date using a minimum number of commands by simply using the following command:

```
% make doit
```

COMPILATION DEPENDENCIES

Hardware engineers have traditionally not had to deal with compilation order problems before: simulators simply took a list of files and compiled them before each and every simulation run. Syntax errors were quickly identified and fixed since the simulation could not proceed when they were present. Potential design

© 1997 Qualis Design Corporation

Permission to copy or distribute available upon request.

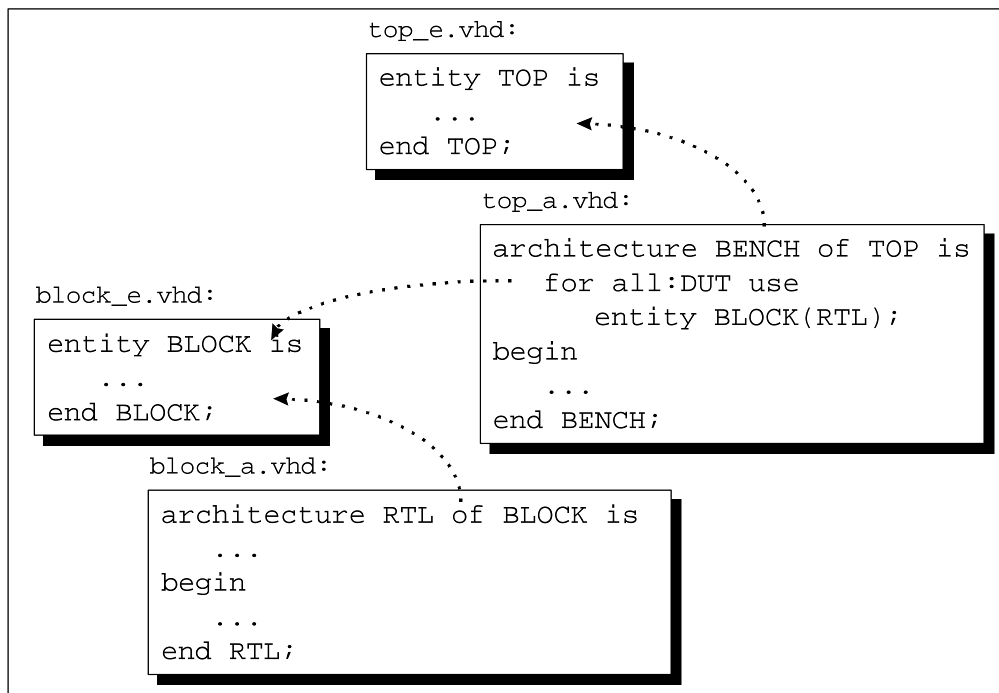
mistakes, such as two inputs connected together, wrong number of pins used, or truncated busses, often went unreported until a problem during simulation was eventually traced to the mistake. Only then were these mistakes fixed.

Early programming languages, such as BASIC, C, and FORTRAN operated in a similar fashion. Software engineers realized several years ago that a lot of those mistakes could be automatically caught at compile time, with their time better spent looking for and fixing the more serious problems at run time. Thus came the concept of strong typing used in more modern languages such as ANSI C. Strong typing had only one problem: to be checked, the source file had to be compiled, but nothing actually enforced compilation when a file change was made. Software systems had to rely on manually maintained makefiles.

ADA and VHDL thus took strong typing and conformance checking one step further: by taking over the management of the object file from the operating system, the system could ensure that source files were subsequently recompiled whenever another file they depended on had been recompiled.

Figure 3 shows a small VHDL model example with arrows showing the compile-time dependencies. If the file `block_e.vhd` is recompiled, whether or not it has actually been modified, the files `top_a.vhd` and `block_a.vhd` will need to be recompiled before the simulator will allow a simulation to proceed.

Figure 3: Example VHDL Source Code



If a VHDL system produced a visible object file like C compilers do, the makefile shown in Figure 4 could be used to maintain the model of Figure 3.

© 1997 Qualis Design Corporation

Permission to copy or distribute available upon request.

QUALIS

Figure 4: Hypothetical Makefile for VHDL Model Example

```
makefile:
block_e.o: block_e.vhd
	vhdl block_e.vhd

block_a.o: block_a.vhd block_e.o
	vhdl block_a.vhd

top_e.o: top_e.vhd
	vhdl top_e.vhd

top_a.o: top_a.vhd top_e.o block_e.o
	vhdl top_a.vhd

simulation: top_e.o top_a.o block_e.o block_a.o
	sim TOP
```

VHDL AND MAKEFILES

Although VHDL systems enforce compilation dependencies before a simulation can proceed, they provide little help in resolving compilation dependency problems and fail to automate the maintenance of up-to-date VHDL models. There are two problems:

- They do not detect that a source file has been modified and therefore needs to be recompiled. In the example shown in Figure 3, the user must manually compile the file `block_e.vhd` after any modifications.
- Some tools can perform required compilations by recompiling an internal copy of the last source code that was compiled. In the example, upon request by the user, it would automatically recompile the internal copy of the source code that was previously compiled from files `block_a.vhd` and `top_a.vhd`. Any modifications made to the actual source files would not be included.

Makefiles are the obvious solution to the compilation dependency problem. Any modifications to any source files will trigger the compilations necessary to bring the model up-to-date.

Some VHDL tools (e.g. Synopsys's *VSS* or Model Technology's *V-System*) include a makefile generation utility. Unfortunately, the makefile generation process requires a compiled model to determine the dependencies. The user must manually compile any new file then manually update, in the correct order, the entire model before a new makefile can be generated. This makes using such makefiles useful only when a model has reached the maintenance stage. It is of little use while the model is actively being developed. These makefiles are also tool-specific and cannot be used to port a model to another VHDL system.

Makefiles can be manually maintained by adding source files as they are created. This process has the advantage that the makefile can be used from the start, even

© 1997 Qualis Design Corporation

Permission to copy or distribute available upon request.

before all syntax errors have been fixed. It does, however, present several serious problems:

- VHDL compilers do not generate clearly visible object files like C compilers do. Instead, compiled code is stored in a library. Since the implementation, form, and structure of the library is tool-specific, it will be necessary to reverse-engineer the library structure to determine the existence, location and name of each object file. This work may have to be repeated as vendors are free to (and often do) modify their implementation of the library from release to release.
- Dependencies are between design units, not source files. A source file may contain more than one unit, each independently compiled as if they were in different files. There is a one-to-many correspondence between a source file and the resulting object files. The makefile must thus deal with the units the file contains, not just the file itself.
- Manual maintenance is tedious, error prone, and inneficient.

Small tool-independent utilities have emerged to help maintain makefiles for VHDL models: they peruse source files, identify units, deduce dependencies then generate an appropriate makefile. Most work even in the presence of syntax errors. Some are targeted to specific VHDL tools, using the information stored in the library. Others, including Qualis' *vmk*, are tool-independent and maintain their own pseudo object files.

The following sections will illustrate how to maintain a VHDL model using *vmk*. The guidelines presented here facilitate the task of managing and maintaining large VHDL models, and are not *vmk*-specific. Information on obtaining *vmk* can be found at the end of this methodology brief.

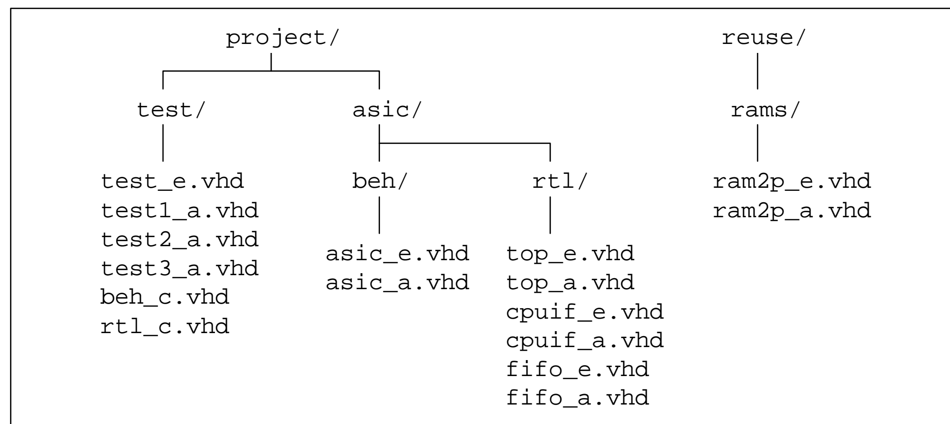
USING MAKEFILES

Libraries and VHDL units are analogous to directories and files in an operating system. Libraries keep units organized in separate name spaces but do not provide any hierarchy. Since libraries group VHDL units together, it is a good practice to group the source files for these units in the same directory. Similarly, as libraries keep their units separate from one another, you should keep source files for units of different libraries in different directories. Figure 5 shows a typical directory structure for a large model where the behavioral model, RTL model, testbench, and reusable components are kept in separate libraries. All the source files associated with a given library can be found in a single directory, and that directory only contains source files for that library.

© 1997 Qualis Design Corporation

Permission to copy or distribute available upon request.

Figure 5: Example VHDL Source Code Structure



vmk generates a makefile for a single library. Since there are 4 libraries, 4 makefiles must be generated. To generate each makefile, one must first change the working directory to the directory where the compilation will be performed (typically the same as the source files) then invoke *vmk* in no particular order:

```
% cd …/project/test
% vmk -o Makefile *.vhd
% cd ../asic/beh
% vmk -o Makefile *.vhd
% cd ../rtl
% vmk -o Makefile *.vhd
% cd …/reuse/rams
% vmk -o Makefile *.vhd
```

Any library can be brought up-to-date at any time by invoking *make* in the source directory. By default, *vmk* can only resolve dependencies within the same library since it only deals with source files from a single library and does not know the location of the source files for other libraries. Very often, a unit in one library will depend on a unit in another. If that other library is also maintained using *vmk*, it is possible to instruct *vmk* to infer a cross-library dependency to a unit that resides in that other library. This can be done through *library* directives in a *vmk* configuration (*vmkrc*) file mapping library logical names to the directory where their makefile resides:

```
library TESTLIB …/project/test
library RTLLIB  …/project/asic/rtl
library RAMLIB  …/reuse/rams
library BEHLIB  …/project/asic/beh
```

To recompile the entire model, *make* must be invoked for each library in the correct order:

```
% cd …/reuse/rams; make
% cd …/project/asic/beh; make
% cd ../rtl; make
% cd ../../test; make
```

© 1997 Qualis Design Corporation

Permission to copy or distribute available upon request.

This process of generating makefiles for individual libraries then invoking *make* in each library in the correct order can be cumbersome. That's why *vmk* comes with a companion tool called *lmk* designed to handle this task. *Library* directives in *lmk*'s configuration files (*lmkrc*) define the available libraries, their logical name, the directory where compilation must take place, and how to get all the source files for that library from that directory. For the above example, the *lmkrc* file might look like this:

```
library TESTLIB …/project/test     *.vhd
library RTLLIB  …/project/asic/rtl *.vhd
library RAMLIB  …/reuse/rams       *.vhd
library BEHLIB  …/project/asic/beh *.vhd
```

Refreshing the makefile in all known libraries, then generating a shell script to invoke *make* in all the same libraries in the correct order is achieved using the simple command:

```
% lmk -l all -u -o recompile
```

Bringing up-to-date a model made up of an arbitrary number of files in an arbitrary number of libraries is now reduced to invoking the shell script:

```
% ./recompile
```

MINIMIZING COMPILATION REQUIREMENTS

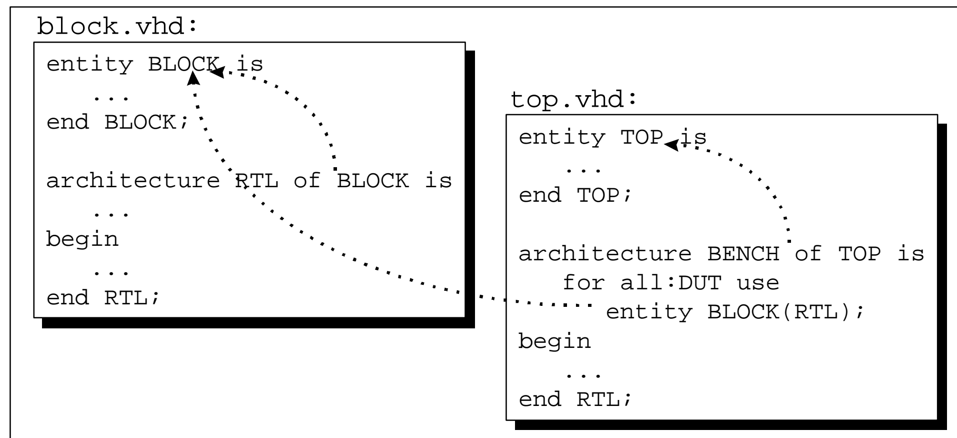
The compilation dependencies of VHDL can be frustrating. However, users can minimize the pain by following a simple principle: keep entities, architectures, configurations, packages, and packages bodies in separate files. Previous requirements from the Synopsys synthesis tool and Verilog experience often entices people to put entities and architectures in the same file, as well as packages and their bodies. Doing so is no longer good design practice.

Figure 6 shows the same source code as the example shown in Figure 3 but structured in 2 files instead of 4. Notice that there are still 3 compilation dependencies: dependencies are between units, not source files, hence some dependencies can be contained in the same file.

© 1997 Qualis Design Corporation

Permission to copy or distribute available upon request.

Figure 6: VHDL Source Maximizing Compilation Requirements

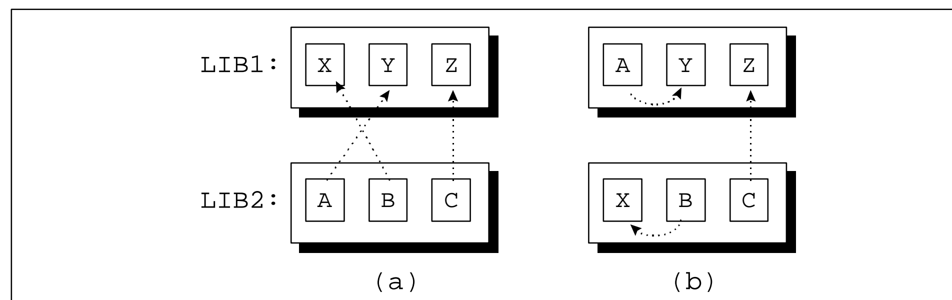


Let's examine the compilation requirements when the architecture of *BLOCK* is modified. If the Figure 6 structure is used, the modification will require the recompilation of file `block.vhd` which will cause the entity *BLOCK* to be recompiled as well, generating a new object file for it. This will make the architecture of *TOP* obsolete and require the file `top.vhd` to be recompiled, which will recompile the entity *TOP* as well. This cascade effect ripples all the way through the hierarchy of the model.

On the other hand, if the structure shown in Figure 3 is used, the modification to the architecture of *BLOCK* requires that the file `block_a.vhd` be recompiled. Since no other units depend on that architecture, no other recompilations are required.

When structuring a model into different libraries, care must be taken to minimize the number of cross-library dependencies to minimize the effect of a library update on others. Figure 7(a) shows a model divided between two libraries with cross-library dependencies. When library LIB1 is refreshed, there is a high probability that LIB2 will need to be refreshed as well. Figure 7(b) shows the same model with units rearranged to minimize cross-library dependencies, reducing the probability that LIB2 will be affected by a refreshing of LIB1.

Figure 7: Alternative Library Structures

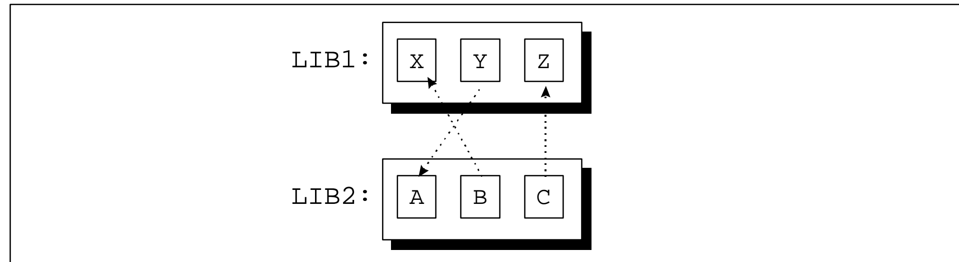


Furthermore, cycles in cross-library dependencies, as shown in Figure 8, should be avoided. To compile such models, it is necessary to bounce back and forth between libraries to achieve a correct order. For example, to compile the model

© 1997 Qualis Design Corporation
97-09-16
Permission to copy or distribute available upon request.

shown below, a compilation order with the minimum library switch would be LIB1.Z, LIB1.X, LIB2.C, LIB2.B, LIB2.A, then LIB1.Y. Without cycles, libraries can be completely compiled one at a time. *lmk* does not handle cycles in cross-library dependencies.

Figure 8: Cycles in Cross-Library Dependencies



OBTAINING A COPY OF *VMK*

You can access *vmk* from the Qualis technology library at:

`http://www.qualis.com`

Alternatively, you can download *vmk* via anonymous FTP from:

`ftp://ftp.qualis.com/pub/vhdl/vmk`

VMK is currently available for Sun/Sparc, HP-700.and Windows-NT.

To obtain a free evaluation license, send email to *vmk@qualis.com*. Please provide the hostid of a machine to be used as a network license server.

SUMMARY

Makefiles specify compilation dependency rules and help automate the maintenance of large software systems. They can facilitate the maintenance of VHDL models and remove the user from compilation order and dependency details. v*mk*, and its companion tool *lmk*, can be used to automatically maintain makefiles for VHDL models made up from an arbitrary number of files containing any number of library units, scattered across several libraries. Keeping library units in separate files also greatly reduces the compilation requirements when a library unit is modified.

ABOUT THE AUTHOR

Janick Bergeron is the Director of Technology at Qualis Design where he helps shape the industry's most advanced verification, modeling and implementation methodologies. He actively consults with leading-edge engineering teams on design methodologies, and has created and taught several introductory and advanced classes on HDL-based design. Janick frequently contributes his experience to the hardware design community through conference papers, tutorials, panel discussions, and newsgroup articles. He can be reached via email at janick@qualis.com or by phone at (503) 644-9700.

© 1997 Qualis Design Corporation
Permission to copy or distribute available upon request.

Qualis applies its knowledge, expertise and fresh perspective to the world's toughest system design challenges. With extensive million+ gate ASIC and system design experience, the minds at Qualis routinely help technology leaders get to the future, *faster*. Qualis also shares its knowledge through unique courses in HDL-based design.

Learn more about Qualis at `http://www.qualis.com`.

© 1997 Qualis Design Corporation

Permission to copy or distribute available upon request.