Automated WAVES Testset Compilation*

Michael Doll and Ranga Vemuri
Laboratory for Digital Design Environments
Department of Electrical and Computer Engineering and Computer Science
University of Cincinnati, M.L. #30
Cincinnati, OH, 45221-0030
mdoll@ece.uc.edu, ranga.vemuri@uc.edu
http://www.ece.uc.edu/~ddel/waves.html
Ph: (513) 556-4784

Abstract

When a VHDL design is synthesized from a specifica-
tion, it becomes necessary to test the synthesized
design description and eventually the actual device. While test vectors and a testbench may already exist
for the specification, there may be many differences in
the two designs that make the original testbench and
test vectors unusable for the synthesized design/actual
device. The new design may have extra pins not in-
cluded in the original design, and most likely the timing
information will have changed from the original. This paper discusses a methodology for transforming
a VHDL/WAVES testbench for the specification model
into a VHDL/WAVES testset to test either a synthesized
design or the actual device.

1 Introduction

WAVES, the Waveform And Vector Exchange Speci-
fication was released as IEEE standard 1029.1 in 1991
to standardize the representation of test data. Its pur-
pose is to provide an unambiguous method of exchang-
ing stimulus and response data among a design and
test environments.

Figure 1 shows a typical vertically integrated design
process. The left column shows the different level of
abstractions for the synthesized design. With each
new level of abstraction, we need a new set of stim-
ulus/response data. This can be done either through
a VHDL testbench or a WAVES testset. It is important
to mention that the test/response data shown for the
testing of the device can be a WAVES testset, but not

Figure 1: Typical Vertically Integrated Design Process

a VHDL testbench. When a design is synthesized its
stimulus/response data is not synthesized with it. We
have developed a methodology for creating a WAVES
testset for a synthesized VHDL/WAVES design. The
method can be used to synthesize between any two
levels of abstraction. For example, we could synthe-
size the gate testset from the behavioral testset.

1.1 Terminology and Conventions

Below is a brief description of some of the terms used
in this paper.

- **UUT** - Unit Under Test. This is used to refer ei-
ther to the instantiation of the entity to be tested
in the testbench or to a actual device being tested

---

*This work was done at the University of Cincinnati and was
sponsored in part by the Air Force Material Command under
an SBIR award to MTL Systems Inc. (# 18338).
by an ATE.

- **WAVES Data Set** - Set of files which defines the WAVES structures and data needed to create and apply a waveform to a UUT.

- **Testbench** - A VHDL model which instantiates an entity and applies tests to it.

- **WAVES testbench** - A VHDL file which instantiates the UUT and interfaces with the WAVES data set. It also provides all the functionality of a tester environment that is necessary to communicate with the WAVES data set. (This includes the maintenance of match flags used for comparing expected and actual results.)

- **WAVES Test Set** - The combination of the WAVES Data Set and the WAVES Testbench.

- **Testbench Compiler** - A program that automatically creates a testbench.

- **WAVES Testset Compiler** - Any program that automatically creates a WAVES testset - the subject of this paper.

Although the methodology proposed in this paper can be used to compile a WAVES testset from a design specification at any level of abstraction, we shall refer to the design specification as the Behavioral level. Similarly, although the synthesized design can also be at any level of abstraction we shall refer to it as the structural level. When referring to the waveforms of the two designs, we shall call the waveform for the behavioral level the source waveform and the waveform for the structural level the target waveform.

1.2 **Brief overview of WAVES**

WAVES is a subset of VHDL. It contains two levels. Level 1 is simple and very restrictive and can be used for the majority of waveforms encountered in practice. Level 1 is a subset of Level 2 which is less restrictive and is similar to a programming language. The remainder of this paper shall discuss only Level 1, since ATE vendors are only expected to support Level 1 initially.

WAVES data sets interact with a tester environment and/or a VHDL testbench through the use of one or two signals. The first, which is required, is called the WAVES port list (wpl). The waves port list contains on array of N elements where N is the number of primary inputs and outputs of the UUT. Each element contains the value to apply to or sense from the UUT. Each element also contains a match flag which controls the matching of the primary outputs. The wpl also indicates what type of delay to apply, either a timed delay or a handshake delay. In the case of a handshake delay the pin and value to handshake on is also passed through the wpl. The second signal is a WAVES match list. This contains the results of all the match events. It is the responsibility of the testing environment (whether it is a Automatic Test System or a VHDL testbench) to set match flags which indicate the results of these match events.

The waveform is generated through a Waveform Generator Procedure (WGP). The only parameters allowed are the WAVES port list and the WAVES match list. The waveform is constructed by applying a series of slices. A slice is a specification of all the events occurring during a fixed period of time for all the signals in the waveform. Each slice can be of different duration. The time of each event is specified in relation to the start of the slice. The slice can be further divided into frames, which are simply lists of all the events on a single signal during a slice. Figure 2 shows how a waveform is constructed from slices and frames.
The frames are created before simulation and are used as building blocks to form slices during simulation using a call to a procedure called 'apply'. The frame is represented by a single character called a pin code. In Figure 2, frame 1 could be represented by the character 'c'. Thus, if we applied the pin code 'c' during a slice, the WGP would schedule a 'drive.1' event 50 NS from the start of the slice and a 'drive.0' event 150 NS from the start of the slice. The D signal would need three different pin codes to represent the different possible frames shown. The same pin code can be used for each pin, but could have different meanings. For example frame 1 and frame 2 in Figure 2 could both be represented by '1'. Its meaning would then depend on which pin it is being applied to. Pins can be grouped together in pin sets such that pin codes have the same meaning for each pin in the pin set.

In the example shown each slice is of the same duration. Another way to define a slice is on a single change basis. Any time a signal changes a new frame is started. Thus all events are at 0 NS in relation to the start of a slice. This is the format used in our WAVES testset compiler system. In this format only one pin code is needed for each possible logical value in the simulation.

The pin codes for each slice can be provided in an external data file that must meet a certain format. The time period of the slice is also specified in the external pattern file.

In addition to signal value events, other events can be created by the Waveform Generator Procedure. These include start and stop matching events and handshaking events. The matching events can be used to determine when to check the results on the test probes. Also a match function can be used to check the state of the UUT and determine the next vector to apply. The handshaking event can be used to suspend the application of vectors until a given value is detected on a specified signal.

In order to have a working WAVES data set, the following components must be present.

1. **Header File** - The header file simply contains information on the order to compile files, and other documentation information for the testset.

2. **Test pin type declaration** - The test pin type declaration is an enumerated type that lists all the signals in the UUT. This information is usually placed in its own file since it is not reusable across different designs.

3. **Logic values type declaration** - The logic values type declarations lists the possible values that a signal can have. In our implementation we are using logical values which correspond to the IEEE 1164 standard logic.

4. **Value dictionary function declaration** - The value dictionary defines the meaning of the logical values by stating their state, strength, direction, and relevancy.

5. **Pin codes constant declaration** - The pin codes constant declaration is a list of all the legal pin codes allowed.

6. **Frame set declaration** - The Frame set declaration defines what each pin code means for each pin, as described earlier.

7. **Waveform Generator Procedure** - see above.

8. **External pattern file** - The external pattern file is optional, but it greatly simplifies the WGP. It contains the pin codes needed to form the slices.

This was just a very brief and informal description of WAVES, and in some portions only described the aspects relevant to our system. For more information on WAVES the reader is encouraged to refer to [1, 3, 6].

# 2 Motivation For a Testset Compiler

To illustrate the need for a testset compiler let us look at an example. Figure 3 shows the source waveform of a test for a behavioral model of a 3 bit counter and the corresponding structural model. Notice that the structural model has a reset pin which will also need to be tested.

In the source waveform the counter starts off at 0 and increments at every rising clock edge with no delay. Thus all events occur on a clock edge. The clock has a period of 10 NS.

In the target waveform we need to initialize the counter to 0 by using the reset. There is a delay of 6 NS before the counter is reset. The period of the counter has now changed to 30 NS. There is also a worst case delay of 9 NS between the rising edge of the clock to when the count changes. At the end of the simulation we need to add an additional test of the reset to see that it resets the signals asynchronously. Note that the signals in the target waveform tend to
Figure 3: Source and Target Waveforms for a 3 Bit Counter

have the same sequence of values as the corresponding signals in the waveform but with different timing.

Figure 3 shows two CNT buses for the target waveform. The first is the actual value of the CNT bus which is observed on the UUT in the testbench (or the actual device in the tester). The second is the value that the WAVES port list will tell the testbench that it is expecting to see. Since there is a 9 NS delay between the rising edge of the clock and when the CNT bus becomes valid, we don’t care what value is on the bus during this period. Thus we program the WPL to send the testbench a don’t care value during this period. Thus, any intermediate values (not shown) on the actual CNT bus will not be interpreted as errors.

Now we must create a program which will read in the source waveform and perform some operations on it that will transform it into the target waveform. In Figure 3 it is the expected version of CNT that we want to produce for the target waveform. With the information given above, we know enough to do this.

3 A Methodology for Transforming WAVES Testsets

Figure 4 shows that our methodology has three steps. The first step is to insert a monitor process into the testbench of the behavioral design. The monitor process will print out the vectors to a file in the WAVES external pattern file format. If the specification design’s testbench is a WAVES testset that is using IEEE 1164 standard logic then the monitor process can be generated using a small program we have written. The values written must be legal IEEE 1164 standard logic. If the testbench uses another logic system then a monitor process will need to be written by hand. It should not be too difficult to develop a program which generates this procedure based on your own synthesis environment.

In the second step the testbench/WAVES testset is simulated and an intermediate WAVES external data file is created. Since the test vectors are an integral part of the testset a new testset will need to be compiled.
for each separate test. Note, it is not necessary to repeat step 1. If the vectors for the specification WAVES testset are already in a "print on change", or the definition of the frames are fairly simple, then steps 1 and 2 can be skipped, and the WAVES external file for the behavioral testset can be used as the intermediate WAVES external data file needed for step 3.

Step 3, however, is where the majority of the work lies. In this step we must specify how to transform the vectors from the source waveform to the target waveform. This is done by a program called a transformer, which is described in the next section.

4 Transformer Programs

To perform the transformation the user must write a program called a waveform transformer or simply transformer which includes the transformer library and uses it to perform the necessary operations. When the transformer program is executed the entire WAVES data set will be generated. A separate program is used to create the WAVES testbench. We are using a modified version of a program called wstb, written by Robert Hillman of US-AN Rome Labs, to perform this task (available at http://vhdl.org/vi/waves/wwwpages/wavestools.html).

Figure 5 shows the basic concept behind a transformer. The transformer program reads in the source waveform and writes out the target waveform. Two internal clocks keep track of the time for each waveform. The clocks are advanced such that the time of the target clock corresponds to the time of the source clock, taking into account the timing differences of the two designs. The target waveform is then formed by copying the source waveform.

Some of the more important data types provided in the transformer package to implement this functionality are:

- Clock
- Signal Type enumeration
- Signal
  - Source Signal
  - Target Signal
- Bus
  - Source Bus
  - Target Bus
- Transformer (Class)

Figure 5: Example of Waveform transformation requiring handshaking.

4.1 Clocks

Clock is a class used to keep track of the simulation time. Two global clocks are provided inside the transformer package, Source_clock and Target_clock. The Source_clock is used to keep track of the simulation time in the source waveform, thus the value of the Source_clock represents the time associated with the last vector that was read in from the source waveform. The Target_clock maintains the time of the target waveform and is used to determine the length of each slice that is written to the WAVES external data file. Its value represents the time of the last vector written to the external data file.

The two global clocks are "read only". They cannot be changed directly. They can only be advanced through the transformer class, discussed later. If needed the user can create additional clocks to help calculate times and other small tasks. User declared clocks can be advanced directly. The global clocks cannot be set back, but the user declared clocks can, although a run
time warning is given.

4.2 Signal Type enumeration

The signal type enumeration corresponds to the logic value enumeration in WAVES. The enumeration is used to determine the value set of a signal.

4.3 Signals

There are two types of signals provided, source signals and target signals. Source signals are used to store the current value of the signals in the source waveform. The only way these signal values can be changed is through reading a new vector from the source waveform. The user can create additional signals to be used as temporary variables. They can be either source or target signals.

Target signals represent the current value of the signals in the target waveform. They also store the direction of the signal as well as a string containing the name of the pin they represent. These two pieces of information are specified when the target signal is declared and cannot be changed afterwards. The direction and name are used to help create the WGP. The signal values can be changed by copying the values of other signals into them (either source or target signals) or by assigning it a specific value. This action is done through the transformer class, described later.

The signals can also indicate if they have had an event. An event on a source signal indicates that the value of the signal changed during the last read of the source waveform. For a target signal it means the value has changed since the last write to the WAVES external file.

4.4 Busses

Busses are simply arrays of signals. There are two types of Busses provided, source and target. Each bus is a collection of either source signals or target signals. Mixing is not allowed. The bus class simply allows shorthand notation. If signals tend to have the same timing they can be grouped together in a bus. Thus, a group of source signals can be copied to a group of target signals simultaneously. Signals that have been grouped together in a bus can still be changed individually.

4.5 Transformers

The transformer is the most important class. All the other classes support this class. The transformer is responsible for reading in new vectors from the source waveform and writing out the target waveforms. It is also responsible for generating the entire WAVES data set.

There are several basic types of functions provided by this class,

- Copy functions,
- Advance functions,
- Handshake functions.

4.5.1 Copy Functions

Copy functions are used to copy values to target signals. The value being copied can be a source signal, a target signal or a constant signal value. In addition, a source or a target bus can be copied to a target bus. While copying a bus, the first signal in the bus being copied is copied to the first signal in the target bus, etc. The copy function cannot be used to change the value of source signals as they should only be changed by reading in a vector from the source waveform.

For example, for the 3 bit counter design of figure 3 assume we have declared a transformer called counter, a source signal called src_clk, a source bus called src_cnt, a target bus called trg_cnt and target signals called trg_clk and trg_reset. We can now set the the values of the target signals as follows:

counter.copy(src clk, trg clk);
counter.copy(src cnt, trg cnt);
counter.copy(forcing_1, trg reset);

4.5.2 Advance functions

Advance functions are used to advance the clocks. There are two basic types of advances, those that advance the source clock and those that advance the target clock. When the target clock is advanced, a time is specified. This indicates the amount of time elapsed since the last advancement of the target clock. The function then writes this time as the length of time for the previous slice. It then writes the current value of all the target signals to the external data file. The length of this slice is determined by the time of the next advance.

Note, the advance functions can be redefined so that the specified time represents the duration of this slice, rather then the previous slice. The difference is that in the former, we need to know the time the previous slice started, and in the latter, we need to know the
time that the next slice begins. The examples shall use the former definition.

Advances on the source clock can be broken down into three basic types. The first specifies an amount of time. The transformer increments the source clock by that amount and reads all the vectors in the source waveform until the time of the vector is greater than or equal to the time of the source clock. If more than one vector occurs at the time of the source clock, then it reads in all of them.

The second form does not require any parameters and implies that the transformer should read in the next vector, regardless of its time and advance the source clock by the appropriate amount.

The final form advances the source clock until a given condition is met. This is done by passing a boolean function to the advance function. The advance will read in a vector and check the function, if it returns false it will loop and continue to read vectors until the function returns true. The source clock gets updated to the time the condition became true.

The following statements demonstrate the use of some of the above advances to create the majority of the target waveform.

```java
while (more slices in source waveform) {
    counter.advance(rising_edge_of_clock);
    counter.copy(src_clk, trg_clk);
    counter.copy(dont_care, trg_cnt);
    counter.advance(target_clock, 15 NS);
    counter.copy(src_cnt, trg_cnt);
    counter.advance(target_clock, 9 NS);
    counter.copy(forcing_0, trg_clk);
    counter.advance(target_clock, 6 NS);
}
```

In the first call to advance we send a function called rising_edge_of_clock, which returns true when there is a rising edge of the clock in the source waveform. In this case it is implied that the source clock is the clock to be advanced by the fact that the only parameter sent is a function. In the second advance call we advance the target_clock by 15 NS. It is advanced by 15 NS because we know at this point that the last event in the target waveform was a falling edge of the clock. The next event, the rising edge of the clock and the don’t cares on the cnt bus, occurs 15 NS later. Before this call, the WAVES external data file might look something like:

```
%CLK CNT2 CNT1 CNT0 RST
...
0 0 1 0 0 : 15 NS;
1 - - - 0 : 9 NS;
1 0 1 1 0 : 6 NS;
0 0 1 1 0
```

After the call to advance the file will look like:

```
%CLK CNT2 CNT1 CNT0 RST
...
0 0 1 0 0 : 15 NS;
1 - - - 0 : 9 NS;
1 0 1 1 0 : 6 NS;
0 0 1 1 0 : 15 NS;
1 - - - 0
```

Thus the time sent to the advance functions indicates the amount of time between the start of the last slice and the start of this slice.

This is just one method that could be used. We could have advanced the source clock by 5 NS knowing that all events in the source waveform occur on the clock edges, which are every 5 NS.

4.5.3 Handshaking functions

The handshaking functions allow the application of the target waveform to be suspended until a certain condition is met on the UUT. Thus, if the exact delay of an event is not known or cannot be determined before testing, the WPL can suspend the application of new stimulus until a given condition is met. Two types of handshaking are supported, fine and coarse. Fine handshaking is done through WAVES handshaking events, while coarse is done through WAVES match events.

Fine handshaking can only be done on one signal at a time, due to the WAVES implementation of handshaking. Fine handshaking allows the exact time of an event to be extracted from the testing environment. The WGP waits until the condition is met and then applies the next slice.

With coarse handshaking the exact time of the event is not determined. The WGP applies the current slice for a small finite amount of time. It then checks to see if the condition has been met. If not, it loops and applies the same slice for the same amount of time specified by the user and continues until the condition
Example of a Handshake Transformation:

<table>
<thead>
<tr>
<th>Source Waveform</th>
</tr>
</thead>
<tbody>
<tr>
<td>RD</td>
</tr>
<tr>
<td>ADDR</td>
</tr>
<tr>
<td>DATA</td>
</tr>
<tr>
<td>REQ_ACK</td>
</tr>
</tbody>
</table>

70 NS

<table>
<thead>
<tr>
<th>Target Waveform</th>
</tr>
</thead>
<tbody>
<tr>
<td>RD</td>
</tr>
<tr>
<td>ADDR</td>
</tr>
<tr>
<td>DATA (Actual)</td>
</tr>
<tr>
<td>DATA (Expected)</td>
</tr>
<tr>
<td>REQ_ACK</td>
</tr>
</tbody>
</table>

??? NS

Figure 6: Example of Waveform transformation requiring handshaking.

is met. Thus the time of the event is only as accurate as the amount of time the slice was applied for.

Figure 6 shows a transformation that will require handshaking. The figure shows a few signals from a typical microprocessor. For brevity the figure only shows the signals needed to demonstrate the handshaking; the memory read line, RD, the address bus, ADDR, the data bus, DATA and request acknowledge signal from the memory, REQ_ACK. In the source waveform we know the amount of time that transpires between the assertion of the RD signal and the assertion of the REQ_ACK signal since this is the output of a simulation. However, we do not know the corresponding amount of time for the target waveform. Thus after the RD signal is asserted we must handshake on the REQ_ACK signal before applying other vectors, since we do not know the length of this slice. To create a handshake delay we need to call the handshake function. Below is an example of how to do this:

```c
micro.advance(target_clock, 10 NS);
// amount of time since the last vector.
if (trg_rd == forcing_1) {
    micro_handshakeFine(trg_req_ack, forcing_1);
}
```

In the call to the handshake function we must pass the name of the signal, and the value to wait for. This example is for a fine handshake. If we had used the coarse handshake function we would have needed to send the amount of time between samples of the signal.

5 Generating and Compiling the WAVES Testset

In order to run a WAVES testset with our methodology, the following files need to be analyzed, or present:

- Header File - partially produced by Testset Compiler.
- Design Files - provided by the user.
- UUT.test_pins - produced by Testset Compiler.
- Waves_objects - file is provided with Testset Compiler.
- Waveform Generator Procedure (WGP) - produced by Testset Compiler.
- WAVES Testbench - produced by Testset Compiler.
- WAVES External file - produced by Testset Compiler.

Before any of these files are analyzed the WAVES library files must be analyzed into WAVES_STD and a WAVES/VHDL interface file for IEEE 1164 must be compiled into WAVES_1164. These files can be obtained from the WAVES web site at http://vhdl.org/vi/waves/. These files only need to be compiled once.

The header file does not need to be analyzed for the WAVES testset to be run by a VHDL simulator, in fact it can not be analyzed by a VHDL analyzer, but the WAVES standard requires that it be generated. It will be needed to run the WAVES testset on an actual tester.

The UUT.test_pins.vhd file, the WGP and the WAVES external file are all created by the transformer. The
UUT_test_pins.vhd file only contains the enumerated type called test_pins. If no handshaking is needed then the WGP’s format is quite simple. The procedure itself is just a loop which reads a vector and applies it. If there is handshaking then it is more complex. A list of pins is generated to indicate the pins that handshaking will be done on. The xth element of the list represents the pin used for the xth handshake event. The handshake event is detected when an integer is read in from the external pattern file rather than a time value. The integer can be decoded to determine the type of handshaking, fine or coarse, and the value to handshake on.

Waves_object.vhd is a file containing several packages and definitions. It does not need to be generated or edited. It must be analyzed after the UUT_test_pins.vhd file because the packages depend on this enumerated type. It must be analyzed before the WGP.

The WAVES testbench is the last file to be analyzed. As mentioned earlier, it is generated by a separate program which is a modified version of wstb by US-AF Rome Labs.

6 Implementation

The transformer library package is currently implemented in C++. We choose C++ for the transformer because of its strong object orientation. Also using an already developed programming language relieved us of the burden of developing a syntax along with a compiler for it.

Although the Testset Compiler currently uses IEEE 1164 standard logic, redefining this enumeration, and rewriting a few procedures in the library should allow other logic value systems to be used.

7 Discussions/Conclusions

The purpose of this paper was to introduce the reader to a methodology of compiling WAVES testsets, which has been developed at the University of Cincinnati. The concept of a transformer was introduced and some of its important capabilities were described. The methodology and tools introduced in this paper are being enhanced to support complete VHDL/WAVES testset compilation for multichip boards or MCM-level designs, with optional boundary-scan capability.

The transformer library can be used to develop testset compilers compatible with a variety of design and synthesis methodologies, tools and conventions. For example, if a synthesis tool generates designs with a particular entity interface protocol convention (say, two-phase clocking, reset, and handshaking with the environment through privileged signals) then a “canned” testset compiler can be written to generate complete WAVES testsets for any design synthesized using that synthesis tool. Similarly, if the interfacing timing of an ASIC is specified in a certain format (such as pin-to-pin models in VITAL [2] and SDF [4]) then a testset compiler can be written to generate complete testsets for any ASIC model conforming to that format. We expect that designers, test engineers and other end-users would use these testset compilers developed to support specific design flows and methodologies. The transformer library described in this paper facilitates the development of such “canned” testset compilers. Where such canned compilers don’t exist a custom testset compiler can be easily written for a specific design or a new design methodology using the transformer library.

The transformer library and the testset compiler described in this paper will be freely available for non-commercial use through MTL Systems Inc., Dayton. For further information on this and related projects the readers can refer to the project home pages on the world wide web at http://www.ece.uc.edu/~ddel/waves and http://www.mtl.com/waves.

Acknowledgments

We thank Jim Hanna, Bob Hillman, Praveen Chawla, Tim Noh, Herb Hirsch and John Hines for their constant support and encouragement during the course of this project. We also thank Raghu Vutukuru for his earlier work on test bench compilers in conjunction with the the DSS synthesis system and Bob Hillman for the development of the wstb program and his permission to allow us to use it.

References


