Encounter® Test: Guide 6: Test Vectors

Product Version 12.1.101
February 2013

Contents

List of Figures

Preface

About Encounter Test and Diagnostics

Encounter® Test uses breakthrough timing-aware and power -aware technologies to enable customers to manufacture higher-quality power-efficient silicon, faster and at lower cost. Encounter Diagnostics identifies critical yield-limiting issues and locates their root causes to speed yield ramp.

Encounter Test is integrated with Encounter RTL Compiler global synthesis and inserts a complete test infrastructure to assure high testability while reducing the cost-of-test with on-chip test data compression.

Encounter Test also supports manufacturing test of low-power devices by using power intent information to automatically create distinct test modes for power domains and shut-off requirements. It also inserts design-for-test (DFT) structures to enable control of power shut-off during test. The power-aware ATPG engine targets low-power structures, such as level shifters and isolation cells, and generates low-power scan vectors that significantly reduce power consumption during test. Cumulatively, these capabilities minimize power consumption during test while still delivering the high quality of test for low-power devices.

Encounter Test uses XOR-based compression architecture to allow a mixed-vendor flow, giving flexibility and options to control test costs. It works with all popular design libraries and automatic test equipment (ATE).

Typographic and Syntax Conventions

The Encounter Test library set uses the following typographic and syntax conventions.

- Text that you type, such as commands, filenames, and dialog values, appears in Courier type.

  Example: Type `build_model -h` to display help for the command.

- Variables appear in Courier italic type.

  Example: Use `TB_SPACE_SCRIPT=`*`input_filename`* to specify the name of the script that determines where Encounter Test binary files are stored.

- Optional arguments are enclosed in brackets.

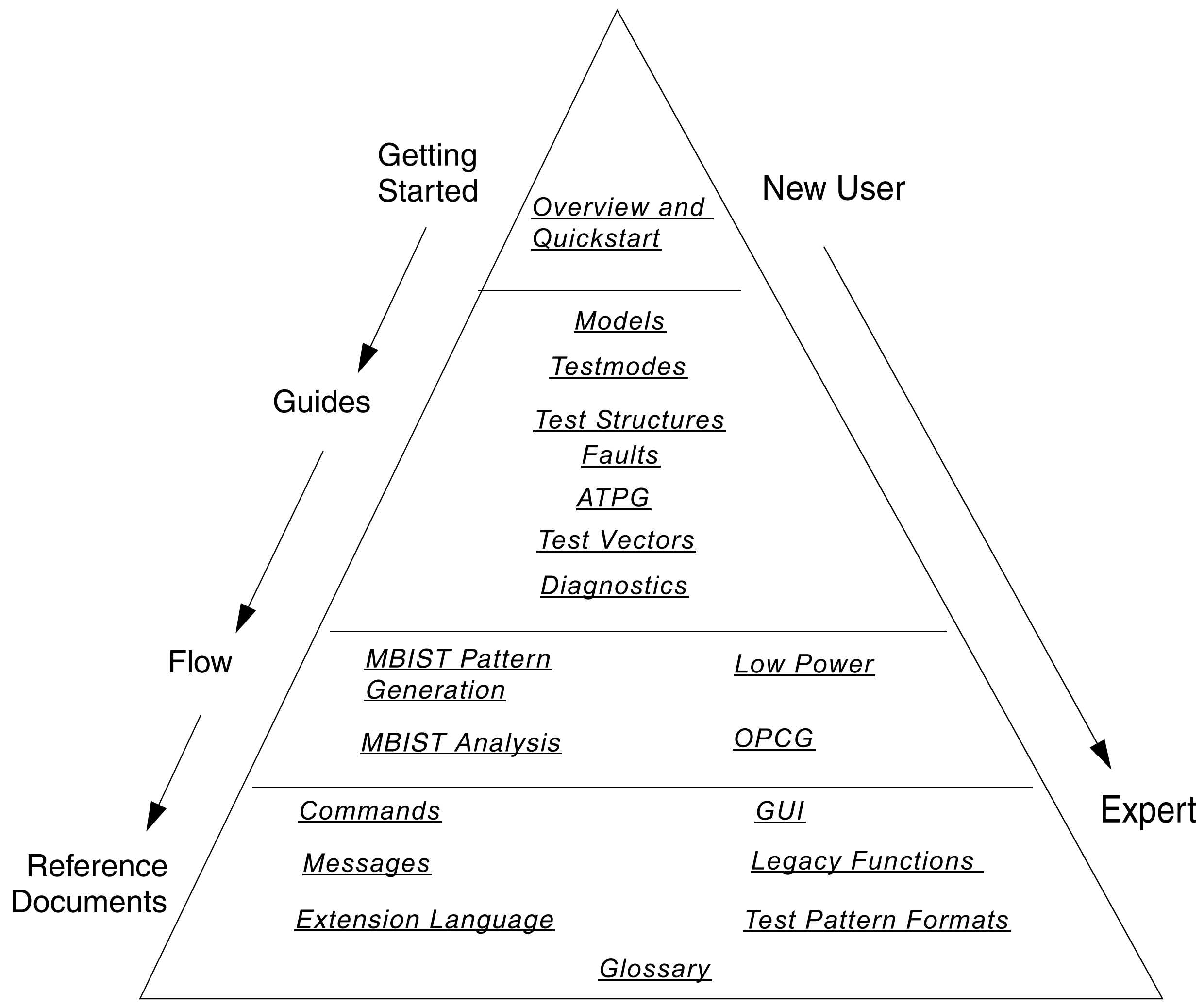
Example: `[simulation=gp|hsscan]`

- User interface elements, such as field names, button names, menus, menu commands, and items in clickable list boxes, appear in Helvetica italic type.

  Example: Select *File - Delete - Model* and fill in the information about the model.

Encounter Test Documentation Roadmap

The following figure depicts a recommended flow for traversing the documentation structure.



Getting Help for Encounter Test and Diagnostics

Use the following methods to obtain help information:

1. From the *<installation_dir>*`/tools/bin` directory, type `cdnshelp` at the command prompt.
2. To view a book, double-click the desired product book collection and double-click the desired book title in the lower pane to open the book.

Click the *Help* or *?* buttons on Encounter Test forms to navigate to help for the form and its related topics.

Refer to the following in the *Encounter Test: Reference: GUI* for additional details:

- "Help Pull-down" describes the *Help* selections for the Encounter Test main window.
- "View Schematic Help Pull-down" describes the Help selections for the Encounter Test View Schematic window.

Contacting Customer Service

Use the following methods to get help for your Cadence product.

- Cadence Online Customer Support

  Cadence online customer support offers answers to your most common technical questions. It lets you search more than 40,000 FAQs, notifications, software updates, and technical solutions documents that give step-by-step instructions on how to solve known problems. It also gives you product-specific e-mail notifications, software updates, service request tracking, up-to-date release information, full site search capabilities, software update ordering, and much more.

  Go to http://www.cadence.com/support/pages/default.aspx for more information on Cadence Online Customer Support.

- Cadence Customer Response Center (CRC)

  A qualified Applications Engineer is ready to answer all of your technical questions on the use of this product through the Cadence Customer Response Center (CRC). Contact the CRC through Cadence Online Support. Go to http://support.cadence.com and click the *Contact Customer Support* link to view contact information for your region.

- IBM Field Design Center Customers

Contact IBM EDA Customer Services at 1-802-769-6753, FAX 1-802-769-7226. From outside the United States call 001-1-802-769-6753, FAX 001-1-802-769-7226. The e-mail address is edahelp@us.ibm.com.

Encounter Test And Diagnostics Licenses

Refer to "Encounter Test and Diagnostics Product License Configuration" in *Encounter Test: Release: What's New* for details on product license structure and requirements.

Using Encounter Test Contrib Scripts

The files and Perl scripts shipped in the `<ET installation path>/ etc/tb/contrib` directory of the Encounter Test product installation are not considered as "licensed materials". These files are provided AS IS and there is no express, implied, or statutory obligation of support or maintenance of such files by Cadence. These scripts should be considered as samples that you can customize to create functions to meet your specific requirements.

What We Changed for this Edition

There are no significant modifications specific to this version of the manual.

1 Writing and Reporting Test Data

This chapter provides information on exporting test data and sequences from Encounter Test.

The chapter covers the following topics:

Writing Test Vectors

Encounter Test writes the following vector formats to meet the manufacturing interface needs of IC manufacturers:

- Standard Test Interface Language (STIL): an ASCII format standardized by the IEEE.
- Waveform Generation Language (WGL): an ASCII format from Fluence Technology, Inc.
- Verilog: an ASCII format from Cadence Design Systems., Inc.
- Tester Description Language (TDL)

Refer to Encounter Test: Reference: Test Pattern Formats for detailed descriptions of these formats.

To write vectors using the graphical interface, refer to "Write Vectors" in the *Encounter Test: Reference: GUI*.

To write vectors using commands, refer to "write_vectors" in the *Encounter Test: Reference: Commands*.

The syntax for the `write_vectors` command is given below:

```
write_vectors workdir=<directory> testmode=<modename> inexperiment=<name>
language=<type>
```

where:

- `workdir` = name of the working directory
- `testmode=` name of the testmode of experiment to export pattern
- `inexperiment=` name of the experiment from which to export data
- `language=stil|verilog|wgl|tdl` = Type of patterns to export

The following table lists the commonly used options for `write_vectors` for all languages:

Language	Parameter	Description
`STIL`	`signalsfile=yes|no`	Create a file containing common elements (signals). Default is `yes`.
	`singletimeplate=no|yes`	Create a single timeplate per file. Default is conditional.
`WGL`	`signalsfile=yes|no`	Create a file containing common elements (signals). Default is `yes`.
	`singletimeplate=no|yes`	Create a single timeplate per file. Default is conditional.
`Verilog`	`scanformat=serial|parallel`	Select the format of the scan in the output vectors.
	`singletimeplate=no|yes`	Create a single timeplate per file. Default is conditional.
	`includenetnames=no|yes`	Include net name in scan out miscompare messages. Default is `no`.

Language	Parameter	Description
`TDL`	`scanformat=serial|parallel`	Select the format of the scan in the output vectors.
	`singletimeplate=no|yes`	Create a single timeplate per file. Default is conditional.
	`configfile=<filename>`	Specify the file that contains the TDL design configuration information
`All languages`	`testperiod=<decimal 0.000000 or greater>`	Global test timing - Set the test period
	`scanperiod=<decimal 0.000000 or greater>`	Global scan timing - Set the scan period
	`scanoverlap=yes|no`	Overlap the scan out with the scan in. Default is `yes`.
	`EXPORTDIR=<outdir>`	Directory to include export data (default is part directory)
	`outputfilename=<string>`	Change the default output file name to be based on this name.

Write Vectors Restrictions

Restrictions differ depending on the selected language format. Refer to subsequent STIL, WGL, Verilog, and TDL sections for details.

Write Vectors Input Files

Write Vectors uses an Encounter Test Experiment or Committed Test Data as input. The command also uses an optional keyword `removepinsfile`, which allows removal of the specified pins from the output test vector files (by default, all pins are written to the output test vectors). Refer to removepinsfile in the *Encounter Test: Reference: Commands* for more information.

When writing TDL output, you need to specify the `configfile` keyword to define the TDL design configuration information. Refer to subsequent TDL section for more information.

Write Vectors Output Files

The output differs depending on the selected language format. Refer to subsequent STIL, WGL, Verilog, and TDL sections for details.

To limit the number of test vector output files, specify `combinesections=yes`, which combines multiple test sections based on test types and creates a maximum of four files, one each for storing static scan tests, static logic tests, dynamic scan tests, and dynamic logic tests.

Refer to write_vectors in the *Encounter Test: Reference: Commands* for more information.

In this case, by default, the files are named as *`language.testmode.testtype`*, which can be overridden using the `outputfilename` keyword. If the `outputfilename` keyword is specified, the output file is named as *`outfilename.testtype`*, where *`testtype`* is static_scan, dynamic_scan, static_logic, or dynamic_logic to indicate the type of test vectors contained within the output test vector file.

Note: The `write_vectors` command does not write the PPIs used for test generation in the output vector.

Default Timings for Clocks

For WGL, STIL, and Verilog vectors, default placement of test clocks within the test cycle (specified by the value for `testperiod`) is based on the following algorithm:

- `clockoffset` is initialized to: (the higher value of either `testpioffset` or `testbidioffset`) + `testpulsewidth`
- If, and only if, there are A_SHIFT_CLOCKs and/or pure C-clocks (not ES or BS):
  - They are placed at clockoffset for a duration of the value specified for `testpulsewidth`
  - `clockoffset` is incremented by 2 times the `testpulsewidth` value
- If, and only if, there are E-clocks:

- They are placed at clockoffset for a duration of the `testpulsewidth` value
- `clockoffset` is incremented by 2 times the `testpulsewidth` value

- If, and only if, there are B-SHIFT_CLOCKs and/or P-clocks:
  - They are placed at `clockoffset` for a duration of the `testpulsewidth` value
  - `clockoffset` is incremented by 2 times the `testpulsewidth` value
- Note that `clockoffset` is incremented only if a particular clock type exists. This means that clock placement is dependent on the type of clocks used in the design.
- Final `clockoffset` time must be <= `testperiod` to enable all clocks to fit within the specified test cycle.
- Default placement of scan clocks within the scan cycle is identical except that `scanperiod`, `scanpioffset`, `scanbidioffset` and `scanpulsewidth` values are used and only A, E, and B clocks are placed (no C or P clocks).
- Any clock may be explicity placed by using `testpioffsetlist` or `scanpioffsetlist`.

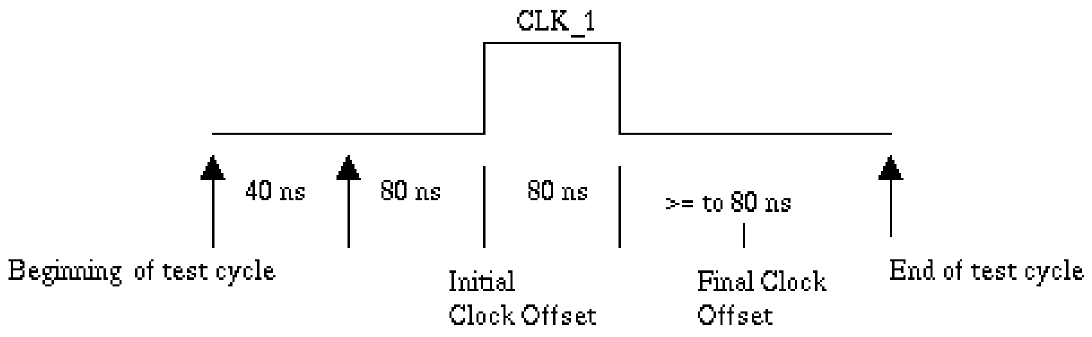
The default timing for `scanpioffset`, `scanbidioffset`, and `scanstrobeoffset` varies depending on the order of the `Set_Scan_Data` and `Measure_Scan_Data` events in the scan sequences. The following defaults apply for designs without compression:

- If the `Measure_Scan_Data` event precedes the `Set_Scan_Data` event, the defaults are the following:
  - `scanstrobeoffset=0`
  - `scanpioffset=scanbidioffset =0+2 * scanpulsewidth`
- If the `Set_Scan_Data` event precedes the `Measure_Scan_Data` event, the default are the following:
  - `scanpioffset=scanbidioffset =0`
  - `scanstrobeoffset=0+scanpulsewidth`

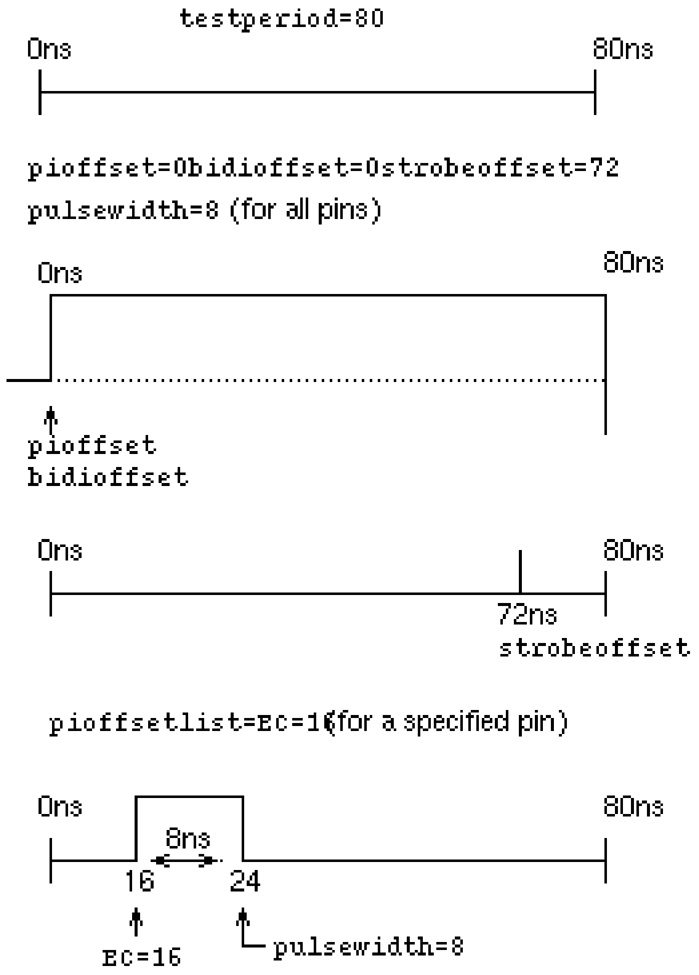
If `Set_Scan_Data` and `Measure_Scan_Data` events do not exist in all test modes, the default timing is the following:

- `scanpioffset=scanbidioffset=scanstrobeoffset=0`

Specify `testpioffset=CLK_1=120 testpulsewidth=80` to generate the timing shown in the following figure:



Specify `testperiod=80 pioffset=0 bidioffset=0 strobeoffset=72 pulsewidth=8 pioffsetlist=EC=16` to generate the timings shown in the following figure:


testperiod=80
0ns
80ns
pioffset=0bidioffset=0strobeoffset=72
pulsewidth=8 (for all pins)
0ns
80ns
pioffset
bidioffset
0ns
80ns
72ns
strobeoffset
pioffsetlist=EC=16(for a specified pin)
0ns
80ns
8ns
16
24
EC=16
pulsewidth=8

Adding Extra Timing Set for Initialization Sequences

By default, the timing of the modeinit sequence matches the timing of the test sequences. However, you can change the modeinit timing by using any of the following keywords and specifying a value different than the current timing value for the test sequences:

- `initbidioffset`
- `initperiod`
- `initpioffset`
- `initpioffsetlist`

- `initpulsewidth`
- `initstrobeoffset`
- `initstrobetype`
- `inittimeunits`

Refer to write_vectors in the *Encounter Test: Reference: Commands* for more information.

`write_vectors` uses the modeinit timings only if they are different from the test sequence timings. In case of different modeinit timings:

- The modeinit timings replace the test timings for all the processed modeinit sequences.
- The modeinit timings are used for all events encountered within the init Test_Sequence except for Scan_Load events. `write_vectors` uses scan timings to process any Scan_Load event it encounters.
- Using the modeinit timings, `write_vectors` writes out accumulated values immediately after processing the modeinit sequence, prior to processing the first test. In other words, `write_vectors` does not compress patterns when transitioning between modeinit timings and test timings.
- `write_vectors` prints the modeinit timings in the header area for each output vector file.

Processing Dynamic Sequences

While processing dynamic sequences, `write_vectors` only creates dynamic timeplates that are unique. In other words, if a cycle of a dynamic sequence matches an already existing dynamic sequence then the first dynamic sequence is used. For example, if the release cycle of a dynamic sequence matches the capture cycle of any dynamic sequence, then the first encounter cycle of the dynamic sequence is used and referenced throughout the output vector files. In addition, if the keyword `limittimeplates` is specified to `yes` and the release timeplate matches the capture timeplate then `write_vectors` defines only the capture timeplate.

Specify `compressdynamictimeplates=no` to disable the automatic compression of dynamic timeplates.

Refer to write_vectors in the *Encounter Test: Reference: Commands* for more information.

Changing Default Pin Order

As an optional input, you can provide a file that includes the order in which to sequence pin data while creating test data. Use the `write_vectors` keyword `pinorder=`*<pinorder file name>* to specify a pin order file.

Refer to "write_vectors" in the *Encounter Test: Reference: Commands* for more information.

Note: If you do not specify a pin order file, the command writes the pin data in the order they were configured in the Encounter Test model (`PIentry` and `POentry` order).

Define the order of pins in the file by including the pin names separated by one or more blank spaces. Block and line comments are supported in the file.

Tip

If the pin order specified for an invocation of `write_vectors` differs from the previously used order, you must recreate the test vectors for the previously exported Test_Sections.

Limiting Dynamic Timeplates for Vectors

Specify `limittimeplates=yes` to limit the number of timeplates for the test vectors. When you set `limittimeplates=yes`, Encounter Test defaults the `singletimeplate` keyword to `yes`, which limits the test and scan timeplates to 1. If dynamic timings exist, then two additional dynamic timeplates are created, the Release timeplate and the Capture timeplate.

Refer to write_vectors in the *Encounter Test: Reference: Commands* for more information.

Specifying the keywords as mentioned above results in the following dynamic timeplate configuration for `write_vectors`:

- The dynamic clock events use the timeplates in the order of Release and Capture. For example, the first dynamic clock event goes in the Release timeplate, the second dynamic clock event goes in the Capture timeplate, the third event in the Release timeplate, and so on.
- The dynamic Stim_PI events are included in the timeplate associated with the next dynamic clock event.
- All Measure PO events are in the Capture timeplate.

Creating Cycle Map for Output Vectors

Use the `cyclemap` keyword to generate cycle information for the output vectors.

The `cyclemap` keyword can have any of the following values:

- `yes` (default) - creates the cycle map file for all scan and PO measures. All output vectors will be created.
- `only` - creates the cycle map only for all scan measures. No output vectors will be created.
- `measures` - creates the cycle map file for all scan and PO measures. No output vectors will be created.
- `no` - does not create a cycle map file

The generated cycle map file is named as `cycleMap.<testmode>|.<inexperiment>` and saved in the `workdir/testresults/EXPORTDIR` directory.

Refer to write_vectors in the *Encounter Test: Reference: Commands* for more information.

```
--------------------------------------------------------------------------------
Total  Relative Relative   Test     Pattern    Event      Cycle  Scan   Relative
Cycle  Cycle    Simulation Sequence Odometer   Type       Offset Length Scan
Count   Count      Time(ns)   Number                                    Number
// The vector file output name:  STIL.FULLSCAN.scan.ex1.ts1.stil

2      2        ET10.1     1        1.1.1.2.1.1 Scan_Load   0      85     ET10.1
87     87       ET10.1     1        1.1.1.2.1.1 end_of_scan 84     0      ET10.1
97     97       ET10.1     1        1.1.1.2.1.9 Scan_Unload 0      85     ET10.1
182    182      ET10.1     1        1.1.1.2.1.9 end_of_scan 84     0      ET10.1
...................
...................
...................
```

The cycle map file has the following columns:

- `Total Cycle Count` - `write_vectors` starts from the top of TBDbin with value zero and continues to count the cycles across all output files.
- `Relative Cycle Count` - Beginning from the top of each output vector file, `write_vectors` starts the cycle count at zero on every new output file.
- `Test Sequence Number` - The test sequence number in the TBDbin.
- `Pattern Odometer` - The odometer value of the reported test sequence in the TBDbin.
- `Event Type` - The event type associated with the reported test sequence in the TBDbin.

- `Cycle Offset` - This is zero at the start of the scan event and the end of the scan event, the value is *`scan length`* `- 1`. For all `measure_PO` events, this value is `-1`.
- `Scan Length` - The number of clock shifts during a scan operation.

Writing Standard Test Interface Language (STIL)

Write Vectors accepts either a committed vectors or an uncommitted vectors file for a specified test mode and translates its contents into files that represent the `TBDbin` test data in STIL format.

- To write STIL vectors via the GUI, select *STIL* as the *Language* option on the *Write Vectors* form
- To write STIL vectors using commands, specify `language=stil` for write_vectors.

STIL Restrictions

The following restrictions apply:

- Test data created for an assumed scan chain test mode cannot be written.
- Overlapping is not allowed when using diagnostic measure events.

STIL Output Files

Write Vectors creates the following output files in STIL format:

- An uncommitted input `TBDbin.`*`testmode.inexperiment`* file results in one or more files named `STIL.`*`testmode.inexperiment.testsectiontype`*`.ex#.ts#.stil` in the specified directory.
- A committed input `TBDbin.`*`testmode`* vectors file results in one or more files named `STIL.`*`testmode.inexperiment.testsectiontype`*`.ex#.ts#.stil` in the specified directory.
- Input of either uncommitted vectors or committed vectors results in a file named `STIL.`*`testmode.inexperiment`*`.signals.stil` in the specified directory if the option to generate this file is selected.

  The *`testmode`* and *`inexperiment`* fields contain the testmode and uncommitted test names from the source `TBDbin` file.

The *`testsectiontype`* field contains the test section type value from the `TBDbin` and is used to identify the type of test data contained within the STIL file, for example, logic.

The *`ex#`*, *`ts#`*, and optional *`tl#`* fields differentiate multiple files generated from a single `TBDbin`. The `TBDbin` hierarchical element id is substituted for `#`, i.e., `ex#` receives the uncommitted test number within the `TBDbin`, `ts#` receives the test section number within the uncommitted test, and `tl#` receives the tester loop number within the test section.

Committed and uncommitted TBDbins contain test coverage and tester cycle count information for each test sequence.

The optional signals file contains declarations common to the STIL vector files, for example, I/O signal names, in order to eliminate redundant definition of these elements.

The preceding files represent default names if keyword `outputfilename` (or *Write Vectors* form field *Set the output file names*) is not specified. If a value for `outputfilename` is specified, multiple output files are differentiated by the presence of a numeric suffix appended to the file name. For example, multiple committed vectors files are named *`outputfilename_value`*`.1.stil`, *`outputfilename_value`*`.2.stil`, and so on. The signals file is named *`outputfilename_value`*`.signal.stil`.

Use the `latchnametype` keyword to specify whether to report the output vectors at the primitive level or at the cell level.

Note: If a Verilog model does not have the cells and macros correctly marked and you specify `latchnametype=cell`, then the instance names might match and be invalid. If the model does not have a cell defined, then `write_vectors` uses primitive level for the instance name.

Writing Waveform Generation Language (WGL)

Write Vectors accepts either a committed vectors or an uncommitted vectors file for a specified test mode and translates its contents into files that represent the `TBDbin` test data in WGL format.

- To write WGL vectors via the GUI, select *WGL* as the *Language* option on the *Write Vectors* form
- To write WGL vectors using commands, specify `language=wgl` for write_vectors.

WGL Restrictions

The following restrictions apply:

- Test data created for an assumed scan chain test mode cannot be written.
- Overlapping is not allowed when using diagnostic measure events.

WGL Output Files

Write Vectors creates the following output files in WGL format:

- An uncommitted input `TBDbin.`*`testmode.inexperiment`* file results in one or more files named `WGL.`*`testmode.inexperiment.testsectiontype`*`.ex#.ts#.wgl` in the specified directory.
- A committed input `TBDbin.`*`testmode`* vectors file results in one or more files named `WGL.`*`testmode.inexperiment.testsectiontype.`*`ex#.ts#.wgl` in the specified directory.
- Input of either uncommitted vectors or committed vectors results in a file named `WGL.`*`testmode.inexperiment.`*`signals.wgl` in the specified directory if the option to generate this file is selected.

  The *`testmode`* and *`inexperiment`* fields contain the testmode and uncommitted test names from the source `TBDbin` file.

The *`testsectiontype`* field contains the test section type value from the `TBDbin` and is used to identify the type of test data contained within the WGL file, for example, logic.

The *`ex#`*, *`ts#`*, and optional *`tl#`* fields differentiate multiple files generated from a single `TBDbin`. The `TBDbin` hierarchical element id is substituted for `#`, i.e., `ex#` receives the uncommitted test number within the `TBDbin`, `ts#` receives the test section number within the uncommitted test, and `tl#` receives the tester loop number within the test section.

Committed and uncommitted TBDbins contain test coverage and tester cycle count information for each test sequence.

The optional signals file contains declarations common to the WGL vector files, for example, I/O signal names, in order to eliminate redundant definition of these elements.

The preceding files represent default names if keyword `outputfilename` (or *Write Vectors* form field *Set the output file names*) is not specified. If a value for `outputfilename` is specified, multiple output files are differentiated by the presence of a numeric suffix appended to the file name. For example, multiple committed vectors files are named *outputfilename_value*.1.wgl, *outputfilename_value*.2.wgl, and so on. The signals file is named *outputfilename_value*.signal.wgl.

Writing Tester Description Language (TDL)

Write Vectors accepts either a committed vectors or an uncommitted vectors file for a specified test mode and translates its contents into files that represent the `TBDbin` test data in TDL format.

To write TDL vectors using commands, specify `language=tdl` for write_vectors.

TDL Restrictions

The following restrictions apply:

- Test data created for an assumed scan chain test mode cannot be written.
- Overlapping is not allowed when using diagnostic measure events.

TDL Output Files

Write Vectors creates the following output files in TDL format:

- Input of either uncommitted vectors or committed vectors results in a file named *<pattern_set_name>_#*.tdl in the specified directory.
- Input of either uncommitted vectors or committed vectors results in a signals file named <*pattern_set_name*>.signals.tdl in the specified directory if the option to generate this file is selected.

  The optional signals file contains declarations common to the TDL vector files, for example, I/O signal names, in order to eliminate redundant definition of these elements.

  pattern_set_name is the pattern set name specified in the input config file and `#` is the test section number within the test.

The preceding files represent default names if keyword `outputfilename` (or *Write Vectors* form field *Set the output file names*) is not specified. If a value for `outputfilename` is specified, multiple output files are differentiated by the presence of a numeric suffix appended to the file name. For example, multiple committed vectors files are named *outputfilename_value*.1.tdl, *outputfilename_value*.2.tdl, and so on. The signals file is named *outputfilename_value*.signal.tdl.

Writing Verilog

Write Vectors accepts either a committed vectors or an uncommitted vectors file for a specified test mode and translates its contents into files that represent the `TBDbin` test data in Verilog format.

- To write Verilog vectors via the GUI, select *Verilog* as the *Language* option on the *Write Vectors* form
- To write Verilog vectors using commands, specify `language=verilog` for write_vectors.

Parallel vs Serial Timing

Parallel Verilog patterns always measure the nets at the time frame 0 as opposed to serial patterns that measure the nets at other poissible times. Overriding the Test or Scan Strobe Offsets must have the measures before any stimulus to get the correct parallel simulation results.

Miscompares in parallel mode result are reported on internal nets at the beginning of the scan cycle and are not flagged on a specific scan out pin.

Parallel Pattern Simulaton with OPCG Logic

If the internal PLL runs at the same speed as the reference OSC clock then `write_vectors scanformat=parallel` may not allow enough time for the OPCG logic to reset between TRIGGER (GO signal) events. To avoid this problem (again only parallel pattern simulation), specify the following to add a few more clock cycles delay:

```
write_vectors scanformat=parallel   waitcyclescanprecond=32
```

If you are running SERIAL simulation or if your internal PLL clock is 2x or more faster than the reference OSC clock, there would be no issues with parallel pattern simulation.

Verilog Restrictions

The following restrictions apply:

- Test data created for an assumed scan chain test mode cannot be written.
- Overlapping is not allowed when using diagnostic measure events.

Verilog Output Files

Write Vectors creates the following output files in Verilog format:

- An uncommitted input `TBDbin.`*`testmode.inexperiment`* file results in one or more files named `VER.`*`testmode.inexperiment.testsectiontype.`*`ex#.ts#.verilog` in the specified directory.
- A committed input `TBDbin.`*`testmode`* vectors file results in one or more files named `VER.`*`testmode.inexperiment.testsectiontype.`*`ex#.ts#.verilog` in the specified directory.
- Input of either uncommitted vectors or committed vectors results in a file named `VER.`*`testmode.inexperiment.`*`signals.verilog` in the specified directory if the option to generate this file is selected.

  The *`testmode`* and *`inexperiment`* fields contain the testmode and uncommitted test names from the source `TBDbin` file.

The *`testsectiontype`* field contains the test section type value from the `TBDbin` and is used to identify the type of test data contained within the Verilog file, for example, logic.

The *`ex#`*, *`ts#`*, and optional *`tl#`* fields differentiate multiple files generated from a single `TBDbin`. The `TBDbin` hierarchical element id is substituted for `#`, i.e., `ex#` receives the uncommitted test number within the `TBDbin`, `ts#` receives the test section number within the uncommitted test, and `tl#` receives the tester loop number within the test section.

Committed and uncommitted TBDbins contain test coverage and tester cycle count information for each test sequence.

The optional signals file contains declarations common to the Verilog vector files, for example, I/O signal names, in order to eliminate redundant definition of these elements.

The preceding files represent default names if keyword `outputfilename` (or *Write Vectors* form field *Set the output file names*) is not specified. If a value for `outputfilename` is specified, multiple output files are differentiated by the presence of a numeric suffix appended to the file name. For example, multiple committed vectors files are named *`outputfilename_value.`*`1.verilog`, *`outputfilename_value.`*`2.verilog`, and so on. The mainsim file is named *`outputfilename_value.mainsim.`*`v`.

NC-Sim Considerations

When using Write Vectors for subsequent simulation by NC-Sim, the following keywords may be specified for NC-Sim:

`+DEBUG`	Specify this keyword to trace the process flow.
`+HEARTBEAT`	Specify this keyword to produce progress status messages.
`+FAILSET`	Specify this keyword to produce a Chip-Pad-Pattern record for each miscomparing vector. Refer to "Reading Failures" in the *Encounter Test: Guide 7: Diagnostics*.
`+START_RANGE`	Specify an odometer value to indicate the beginning of a pattern range to be simulated.
`+END_RANGE`	Specify an odometer value to indicate the end of a pattern range to be simulated.
`+TESTFILE`*`num=filename`*	Specify an input data file to NC-Sim. Specify multiple files by incrementing the *`num`* string. For example, specify `+TESTFILE1=data1 +TESTFILE2=data2`.
`+DEFINE+simvision`	Specify this optional compiler directive keyword to compile code proprietary to Cadence Design Systems., Inc. Do not include this for non-Cadence simulators. This command enables the automatic dump of waveform data but does not invoke a waveform dump.
`+simvision`	Specify this optional keyword to invoke the SimVision dump of waveform data during simulation.
`+vcd`	Specify this optional keyword to invoke the SimVision dump of VCD data during simulation.

The following is an example syntax:

```
ncxlmode \
+DEFINE+simvision  \
+simvision    \
+vcd \
+HEARTBEAT \
+FAILSET \
+START_RANGE=1.1.1.1.1.1. \
+END_RANGE=1.1.14.1.8 \
VER.${TESTMODE}*mainsim.v \
+TESTFILE1=VER.testmode.data.testsection.ex.ts1 \
+TESTFILE2=VER.testmode.data.testsection.ex.ts2
```

Another NC-sim option is to set `SEQ_UDP_DELAY+<delay in ps>` during elaboration state to allow Encounter Test to add some delay to the sequential elements. This option can help fix many miscompares when running in a zero delay simulation mode.

An example to add 50ps delay is given below:

```
If using NC Verilog command line:
ncverilog \
   +ncseq_udp_delay+50ps \
```

Specify the following if using the NC Elaboration command line:

```
ncelab \
   -seq_udp_delay 50ps \
```

Using the TB_VERILOG_SCAN_PIN Attribute

The `TB_VERILOG_SCAN_PIN` model attribute is used to control the selection of scan, stim, and measures points in exported Verilog test vectors when specifying the `write_vectors scanformat=parallel` option, or via the graphical user interface, selecting a `Scan Format` of parallel.

The `TB_VERILOG_SCAN_PIN` attribute may be placed on any hierarchical pin on any cell. However the pin must be on the scan path (or intended to be on the scan path if used within a cell definition).

When encountered on input pins, the parent net associated with the attributed pin is selected as a stimuli net. When encountered on output pins, the associated parent net is selected as a measure net. This attribute may be selectively used, i.e., default net selection takes place if no attribute is encountered for a specific bit position.

The following example depicts the placement of the `TB_VERILOG_SCAN_PIN` attribute on an instance within a cell:

```
DESFQ
    \$blk_DESFQ$14
        (.Q ( \$net__G001 ) //! TB_VERILOG_SCAN_PIN="YES"
        ,.C ( \$net__H05 )
        ,.D ( \$net__H01 )  //! TB_VERILOG_SCAN_PIN="YES"
        ,.R ( \$net_NET$1 )
        ,.S ( \$net_NET$2 )
        );
```

Understanding the Test Sequence Coverage Summary

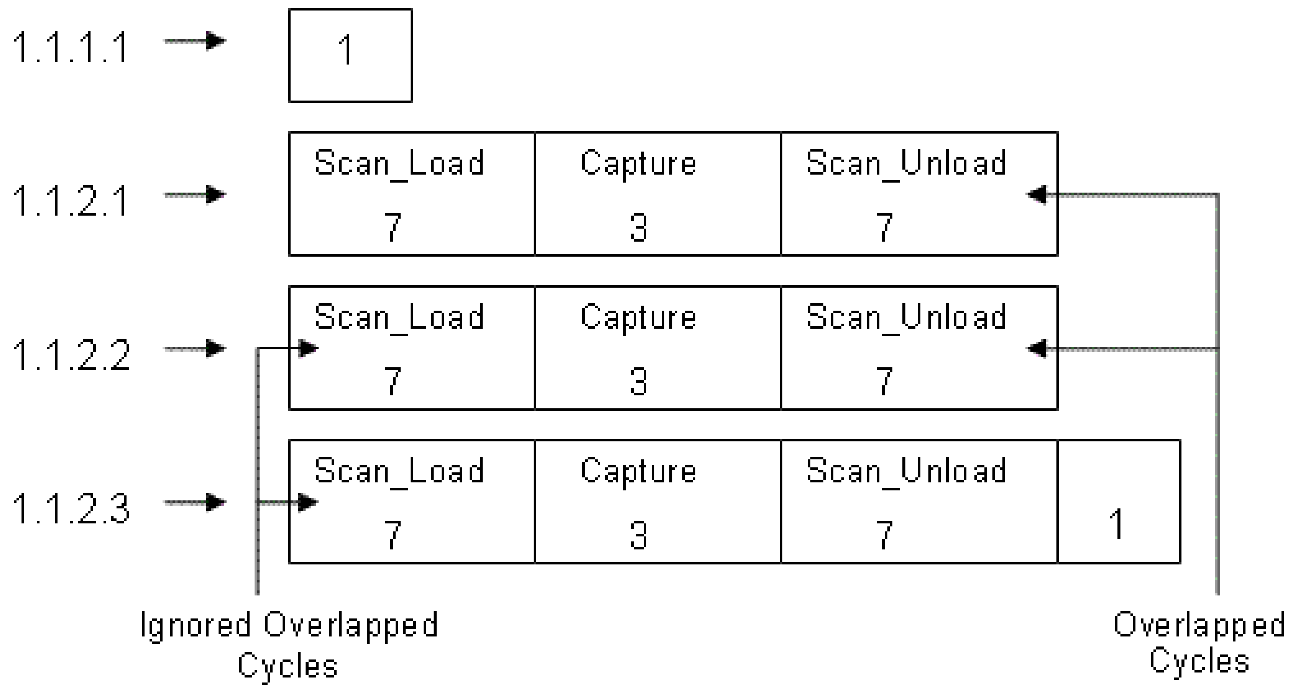
The following is a sample Test Sequence Coverage Summary Report produced by `write_vectors`:

```
--------- -------- -------- --------  -------- --------  ----------  ------
Test      Static   Static   Dynamic   Dynamic  Sequence  Overlapped  Total
Sequence  Total    Delta    Total     Delta    Cycle     Cycle       Cycle
          Coverage Coverage Coverage  Coverage Count     Count       Count
--------- -------- -------- --------  -------- --------  ----------  ------
1.1.1.1.1   0.00    0.00     0.00      0.00      1          0          1
1.1.1.2.1   38.31   38.31    24.14     24.14     17         7          18
1.1.1.2.2   54.55   16.24    32.18     8.04      10         7          28
1.1.1.2.3   64.94   10.39    35.63     3.45      11         0          39
```

The preceding report is produced for STIL, Verilog, and WGL vectors.

Figure 1-1 on page 33 shows a graphical representation of the scan cycles for the reported vectors:

Figure 1-1 Scan Cycle Overlap in Test Sequence Coverage Summary Report



The output log includes the following columns:

- `Test Sequence` - shows the sequence for which the test coverage and tester cycle information is reported
- `Static Total Coverage` - shows the static fault coverage for the test sequence
- `Static Delta Coverage` - shows the difference in static fault coverage between the current and the preceding vector. For example, the static fault coverage for vector

1.1.1.2.2 is 54.55 while the coverage of the preceding vector 1.1.1.2.1 is 38.31. Therefore, the difference in the coverage by the two vectors is 16.24, which is reported in the `Static Delta Coverage` column.

- `Dynamic Total Coverage` - shows the dynamic fault coverage for the test sequence
- `Dynamic Delta Coverage` - shows the difference in dynamic fault coverage between the current and the preceding vector. For example, the dynamic fault coverage of vector 1.1.1.2.2 is 32.18 and the coverage of the preceding vector 1.1.1.2.1 is 24.14. Therefore, the difference in the coverage by the two vectors is 8.04, which is reported in the `Dynamic Delta Coverage` column.
- `Sequence Cycle Count` - shows the clock cycles taken by the test sequence
- `Overlapped Cycle Count` - shows the cycles that are overlapping with the next reported vector. For example, for vector 1.1.1.2.1, out of 17 cycles, 7 cycles overlap with the next vector 1.1.1.2.2.

  While calculating the `Sequence Cycle Count` for a vector, the cycles overlapping with the preceding vector are ignored.

  For example, as shown in the figure, out of the 17 cycles taken by vector 1.1.1.2.2, 7 cycles overlap with the preceding vector 1.1.1.2.1. Therefore, though vector 1.1.1.2.2 takes total of 17 cycles, the 7 cycles that overlap with the preceding vector 1.1.1.2.1 are ignored and not reported in the `Sequence Cycle Count` column for the vector.
- `Total Cycle Count` - shows the cumulative sequence cycle count of the current and all preceding vectors. For example, the total cycle count reported for vector 1.1.1.2.2 is the combination of the cycle count for this vector (10) and all the vectors reported before it, i.e, 1.1.1.2.1 (17) and 1.1.1.1.1 (1). Therefore, the reported cycle count for the vector is 28.

Create Vector Correspondence Data

When the input TBDpatt file contains vector format data, there must exist a specification of the correspondence between vector bit positions and design pins and blocks. This correspondence data is contained in a file (TBDvect) that is separate from the TBDpatt data. The data is maintained separately since it is concerned only with formatting, and not data content. The same vector format file is used both for importing TBDpatt data and for printing a TBDpatt file.

There is a single vector correspondence file per test mode. The intent is that there be only one set of vector correspondence data ever used for a given mode. Making changes to the vector correspondence file should be undertaken only with utmost caution. If you want to import a TBDpatt file that was produced by Encounter Test, the vector correspondence must not be altered during the interim.

You can change the ordering within vectors by changing the positions of pins or latches in the vector correspondence file. This may be useful if you are going to use TBDpatt as an intermediate interface to some other format which requires that the vectors be defined in a specific way. In such a case, you might be able to modify TBDvect in such a way as to avoid re-mapping the vectors in another conversion program.

Test data written in vector format contains a Vector_Correspondence section consisting of commented lines. Vector correspondence data is stored in the TBDvect file. The vector correspondence lists define each position of the vectors that are used in stim events, measure events, and weight events. For example, the fifth position in a Stim_PI vector is the value to be placed on the fifth input in the primary input correspondence list.

The vector correspondence data includes the following lists:

- Primary Input Correspondence List
- Primary Output Correspondence List
- Stim Latch Correspondence List
- Measure Latch Correspondence List
- MISR Latch Correspondence List
- PRPG Latch Correspondence List

A scan design in Encounter Test, even if composed of flip-flops is always modeled using level-sensitive latches. Each latch that is controlled by the last clock pulse in the scan operation must necessarily end up in the same state as the latch that precedes it in the register. In rare circumstances, the preceding latch may also be clocked by this same clock, and then both latches have the same state as the next preceding latch (and so on). Some latches may be connected in parallel, being controlled by the same scan clock and preceded by a common latch. These latches also must end up at a common state. Encounter Test identifies all the latches that have common values as the result of the scan operation, and each such set of latches is said to be “correlated”. TBDpatt does not explicitly specify the value for every latch, but only for one latch of each group of correlated latches. This latch in each group is called a “representative stim latch” or “representative measure latch”, depending upon whether the context is a load operation or an unload operation. The Stim Latch Correspondence List is a list of all the latches that can be independently loaded by a scan or load operation. This consists mainly of the representative stim latches, but contains also “skewed stim latches” which require special treatment for the Skewed_Scan_Load event described in the *Encounter Test: Reference: Test Pattern Formats*. The Measure Latch Correspondence List is a list of the representative measure latches.

In the case of PRPG and MISR latches, used for built-in self test, similar correlations may exist; the representative latches for these are called “representative cell latches”. The PRPG and MISR correspondence lists include the representative cell latches for the PRPGs and MISRs respectively.

See TBDpatt and TBDseqPatt Format in the *Encounter Test: Reference: Test Pattern Formats* for information on vector correspondence language definition.

To perform Create Vector Correspondence using the graphical interface, refer to “Write Vector Correspondence” in the *Encounter Test: Reference: GUI*.

To perform Write Vector Correspondence using command line, refer to “write_vector_correspondence” in the *Encounter Test: Reference: Commands*.

The syntax for the `write_vector_correspondence` command is given below:

```
write_vector_correspondence workdir=<directory> testmode=<modename>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode

Create Vector Correspondence Output

If creating the vector correspondence file, the output will be written into `tbdata/TBDvect.<TESTMODE NAME>`.

Specify `vectorcor=yes` to include the output in `TBDpatt` files.

Reporting Encounter Test Vector Data

You can report the binary `TBDbin` test vectors generated by Encounter Test into an ASCII (`TBDpatt`) file. This is useful when you need to understand how the generated test vectors exercise your design.

To verify cell models created to run with Encounter Test, you can simulate vectors created by our system with other transistor level simulators. To do this, it is necessary to expand the scan chain load and unload data into individual PI stims, clock pulses, and PO measures. Select the *Expand scan operations* option on the *Report Vectors Advanced* form or specify the keyword pair `expandscan=yes` on the command line to expand any scan operations found in the test data.

To report Encounter Test vectors using the graphical interface, refer to "Report Vectors" in the *Encounter Test: Reference: GUI*.

To report Encounter Test vectors using command line, refer to "report_vectors" in the *Encounter Test: Reference: Commands*.

The syntax for the `report_vectors` command is given below:

```
report_vectors workdir=<directory> testmode=<modename> experiment=<name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode of experiment to export pattern
- `experiment`= name of the experiment from which to export data

The commonly used keywords for the `report_vectors` command are:

- `format=vector|node` – Specify the pattern format. Vector produces a smaller file but requires vector correspondence to find individual bit information. Default is `vector`.
- `vectorcor=yes|no` – Whether to include vector correspondence data. Default is `yes`.
- `outputfile=STDOUT|<outfilename>` – Output file name or STDOUT. Default is output file name which is written to the `testresults` directory.
- `testrange=<odometerRange>` - Select a range of tests. Default is all patterns.

Refer to "report_vectors" in the *Encounter Test: Reference: Commands* for more information on these parameters.

Input Files for Reporting Encounter Test Vector Data

`report_vectors` requires a valid model and experiment or committed test data as input.

Output Files for Reporting Encounter Test Vector Data

The `report_vectors` command produces the following default output files:

- Reporting committed vectors results in a file named `TBDpatt.`*`testmode`* in the *`<workdir>`*`/testresults` directory.

- Reporting expanded committed vectors results in a file named `TBDpatt.`*`testmode.experiment`*`.EXPANDSCAN` in the in the *`<workdir>`*`/testresults` directory.
- Reporting uncommitted vectors results in a file named `TBDpatt.`*`testmode.experiment`* in the *`<workdir>`*`/testresults` directory.
- Reporting expanded uncommitted vectors results in a file named `TBDpatt.`*`testmode.experiment`*`.EXPANDSCAN` in the *`<workdir>`*`/testresults` directory.
- The output file name can be changed by specifying the `outputfile=`*`<filename>`* keyword for the `report_vectors` command or by entering the file name in the GUI *Output file name* entry field.

Reporting Sequence Definition Data

Encounter Test supports the reporting of sequence definitions. The `report_sequences` command reports sequence definitions in `TBDpatt` format. Sequence definitions identify the patterns used to establish the testmode initialization state and to perform the scan operation. These are not used by themselves to drive a tester, but are useful in the ASCII `TBDpatt` format for analysis or to be used as a basis for writing custom mode initialization or scan sequences. For more information about test mode initialization or custom sequences, refer to "Mode Initialization Sequences (Advanced)" and Defining a Custom Scan Sequence in the *Encounter Test: Guide 2: Testmodes*.

To report sequence definition data using the graphical interface, refer to "Report Sequences" in the *Encounter Test: Reference: GUI*.

To report sequence definition data using command line, refer to "report_sequences" in the *Encounter Test: Reference: Commands*.

The syntax for the `report_sequences` command is given below:

```
report_sequences workdir=<directory> testmode=<modename>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode to report the sequences on

The commonly used keywords for the `report_sequences` command are:

- `format=vector|node` - Specify the pattern format. Vector produces a smaller file but requires vector correspondence to find individual bit information. Default is `vector`. `Node` format is required if the file is used for future `build_testmodes` processes.
- `outputfile=STDOUT|<outfilename>` - Output file name or STDOUT. Default is output file name which is written to the `testresults` directory.

Refer to “report_sequences” in the *Encounter Test: Reference: Commands* for more information on these pa-rameters.

Input Files for Reporting Sequence Definition Data

`report_sequences` requires a valid model and testmode as input.

Output Files for Reporting Sequence Definition Data

Reporting Encounter Test sequence definitions results in a file named `TBDseqPatt.`*`testmode`* in the *`<workdir>`*`/testresults` directory.

The output file name can be changed by specifying the `outputfile=`*`<filename>`* keyword for the `report_vectors` command or by entering the file name in the GUI *Output file name* entry field.

Converting Test Data for Core Tests (Test Data Migration)

The `convert_vectors_for_core_tests` command converts a standard `TBDbin` file for a core to a structure-neutral form of `TBDbin` that references none of the internal structure of the core. This is necessary so the migration process for the core test data does not require the presence of a core logic model. The structure-neutral `TBDbin`, called `TBDtdm`, closely resembles the standard `TBDbin` except for those events whose interpretation relies on knowledge of the internal structure of the core are replaced by events that require knowledge of only the core I/O pins. The structure-neutral file can be printed using the utility `report_vectors_for_tdm`. See “Reporting a Structure-Neutral TBDbin” on page 40 for details.

To convert vectors for core tests using the graphical interface, refer to “Convert Vectors for Core Tests” in the *Encounter Test: Reference: GUI*.

To convert vectors for core tests using commands, refer to “convert_vectors_for_core_tests” in the *Encounter Test: Reference: Commands*.

The syntax for the `convert_vectors_for_core_tests` command is given below:

```
convert_vectors_for_core_tests workdir=<directory> testmode=<modename>
inexperiment=<name> exportfile=<file>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode of experiment to export pattern
- `inexperiment`= name of the experiment from which to export data
- `exportfile`= file name of exported data

Input Files for Converting Test Data for Core Tests (TDM)

Converting vectors for core tests requires a valid model and experiment data.

Output Files for Converting Test Data for Core Tests (TDM)

The `convert_vectors_for_core_tests` command creates the output file `TBDtdm.`*`cellname`*.

Reporting a Structure-Neutral TBDbin

The structure-neutral file can be printed using the command `report_vectors_for_tdm`.

The output of `convert_vectors_for_core_test` can be reported and viewed using the `report_vectors_for_core_tests` command. Performing `convert_vectors_for_core_tests` is a prerequisite to executing `report_vectors_for_core_tests`.

Note: `report_vectors_for_core_tests` reads only the data produced by `convert_vectors_for_core_tests`.

Refer to "report_vectors_for_core_tests" in the *Encounter Test: Reference: Commands* for the syntax.

The syntax for the report_vectors_for_core_tests command is given below:

```
report_vectors_for_core_tests workdir=<directory> inputfile=<file>
```

where:

- `workdir` = name of the working directory
- `inputfile`= file name of exported data

Input for Reporting Structure-Neutral TBDbin

The output from `convert_vectors_for_core_test`, `TBDtdm.`*`cellname`*, is the input to `report_vectors_for_tdm`.

Output for Reporting Structure-Neutral TBDbin

Specify any meaningful path/filename to write the output.

Note: This output is disallowed as input to `read_vectors` (tbdpatt) or `read_sequence_definition_data`.

2

Utilities and Test Vector Data

This chapter discusses the commands and concepts related to test patterns and the ATPG process.

The chapter covers the following topics:

Committing Tests

All test generation runs are made as uncommitted tests in a test mode. Commit Tests moves the uncommitted test results into the committed vectors test data for a test mode. Refer to "Performing on Uncommitted Tests and Committing Test Data" on page 44 for more detailed information about the overall test generation processing methodology.

To perform Commit Tests using the graphical interface, refer to "Commit Tests" in the *Encounter Test: Reference: GUI.*

To perform Commit Tests using command line, refer to "commit_tests" in the *Encounter Test: Reference: Commands*.

The syntax for the `commit_tests` command is given below:

```
commit_tests workdir=<directory> testmode=<modename> inexperiment=<name>
```

where:

- `workdir` = name of the working directory
- `testmode=` name of the testmode for dynamic ATPG
- `inexperiment=` name of the experiment to save

The most commonly used keyword for the commit_tests command is:

- `force=no/yes` - To force uncommitted tests to be saved even if potential errors have been detected. Some potential errors are:
    - TSV was not run or detected severe errors
    - Date/time indicates that the patterns were created before a previous save

Refer to "commit_tests" in the *Encounter Test: Reference: Commands* for more information on the keyword.

Prerequisite Tasks

Complete the following tasks before running Test Simulation:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in the *Encounter Test: Guide 1: Models*.
2. Create a Test Mode. Refer to "Performing Build Test Mode" in the *Encounter Test: Guide 1: Models*.
3. Build a fault model for the design. See "Building a Fault Model" in the *Encounter Test: Guide 4: Faults* for more information.
4. Run ATPG and create a pattern to be saved. Refer to "Invoking ATPG" on page 35 for more information.

Performing on Uncommitted Tests and Committing Test Data

Encounter Test creates test data in the form of uncommitted tests. You can inspect the uncommitted test results to decide whether it is worth keeping or that it should be discarded. If the test results from an uncommitted test are deemed worth saving, there are two means available for continuing additional test generation from where an uncommitted test left off.

- You can append to this uncommitted test when you run test generation again.
- You can commit (save) the test data.

Note: A committed test or experiment may not be reprocessed.

If you use the uncommitted test as input to another uncommitted test, append to it and the results of the second uncommitted test are not satisfactory, both uncommitted tests will have to be thrown away since they will have been combined into a single uncommitted test.

By committing an uncommitted test, you append the results of the uncommitted test to a “committed vectors” set of test data for the test mode. The committed vectors for a test mode is the data deemed worth sending to manufacturing. After an uncommitted test has been committed, it is removed from the system and can no longer be manipulated independent from the other committed vectors data. For future reference, the names of the committed tests are saved with the committed test patterns.

After test data from one or more uncommitted test has been committed to the committed vectors set of test data, any subsequent test generation uncommitted tests will target only those faults left untested by the committed test patterns. To get the most benefit from this, it is advisable to achieve acceptable uncommitted test results before beginning a new uncommitted test. While it is possible to run many different test generation uncommitted tests in parallel, if they are committed there will be many unnecessary test patterns included in the committed vectors test set since the uncommitted tests may have generated overlapping test patterns that test the same faults.

When uncommitted test results are committed to the committed vectors for a test mode, the commit process checks whether it is allowed to give test credit for faults in any other test modes that have been defined. This processing is referred to as *cross mode fault mark-off*. This is a mechanism that allows a fault that was detected in one test mode to be considered already tested when a different test mode is processed in which that fault is also active. Sometimes cross mode fault mark-off is explicitly disabled, such as when two test modes are targeting the same faults, but for different testers (and different chip manufacturers).

Deleting Committed Tests

Use the Delete Committed Tests command to

- Remove all committed and uncommitted tests for a testmode
- Remove individual experiments from a testmode if the experiment does not have any saved fault statistics.
- Reset the fault status for the specified testmode.

To perform Delete Commit Tests using the graphical interface, refer to "Delete Committed Tests" in the *Encounter Test: Reference: GUI*.

To perform Delete Commit Tests using command line, refer to “delete_committed_tests” in the *Encounter Test: Reference: Commands*.

The syntax for the `delete_commit_tests` command is given below:

```
delete_commit_tests workdir=<directory> testmode=<modename>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode

The most commonly used keywords for the `delete_committed_tests` are:

- `keepuncommitted=no -` Remove uncommitted experiments
- `gmexperiment=<name>` - Name of the good machine experiment to remove from committed test data.

Refer to "delete_committed_tests" in the *Encounter Test: Reference: Commands* for more information on these keywords.

Prerequisite Tasks

Complete the following tasks before running Test Simulation:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in the *Encounter Test: Guide 1: Models*.
2. Create a Test Mode. Refer to "Performing Build Test Mode" in the *Encounter Test: Guide 2: Testmodes*.
3. Build a fault model for the design. See "Building a Fault Model" in the *Encounter Test: Guide 4: Faults* for more information.
4. Commit test data. Refer to "Committing Tests" on page 43 for more information.

Deleting a Range of Tests

Use the `delete_testrange` command to remove one or more test vectors or a range of test vectors from an experiment before committing the test data. The command removes the tests or the range of tests specified for the `testrange` keyword. Refer to delete_testrange in the *Encounter Test: Reference: Commands* for more information.

After removing the tests from the experiment, resimulate the new set of tests in the following conditions:

- If the tests in the experiment are order dependent

- If fault status exists for the experiment. Resimulation is required to recalculate and reset the fault status for the remaining tests.

  Refer to "Simulating Vectors" on page 70 for more information.

The syntax for the `delete_testrange` command is given below:

```
delete_testrange workdir=<directory> testmode=<modename> inexperiment=<name>
testrange=<odometer>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode
- `inexperiment`= name of experiment from which to delete patterns
- `testrange`= odometer value of the test patterns to delete. The values can be:
  - `all` - process all test sequences
  - `odometer` - process a single experiment, test section, tester loop, procedure, or test sequence in the TBDbin file (for example 1.2.1.3)
  - `testNumber` - process a single test in the set of vectors by specifying the relative test number (for example 12)
  - `begin:end` - process range of test vectors. The couplet specifies begin and end odometers (such as 1.2.1.3.1:1.2.1.3.15) or relative test numbers (such as 1:10)
  - `begin:` - process the vectors starting at the beginning odometer or relative test number to the end of the set of test vectors.
  - `:end` - process the test vectors starting at the beginning of the set of test vectors and ending at the specified odometer or relative test number
  - `:` - process the entire range of the set of test vectors (the same as testrange = all)
  - A zero (`0`) value in any odometer field specifies all valid entities at that level of the TBD hierarchy. For example, 1.1.3.0.1 indicates the first test sequence in each test procedure.

Refer to delete_testrange in the *Encounter Test: Reference: Commands* for more information on these keywords.

Deleting an Experiment

Use the delete_tests command to remove all data associated with the specified uncommitted experiment(s) from the working directory.

To perform Delete Tests using the graphical interface, refer to "Delete Tests" in the *Encounter Test: Reference: GUI*.

To perform Delete Tests using command line, refer to "delete_tests" in the *Encounter Test: Reference: Commands*.

The syntax for the `delete_tests` command is given below:

```
delete_tests workdir=<directory> testmode=<modename> experiment=<name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode
- `experiment`= name of experiment to delete

Encounter Test Vector Data

Encounter Test creates, stores, and processes test data in a compact binary format known as `TBDbin`. The test data in `TBDbin` is in a hierarchical form which consists of the following entities:

Figure 2-1 Hierarchy of TBDbin Output

```
Experiment
  Define_Sequence
    Timing_Data
      Pattern
        Event
  Test_Sequence
    Tester_Loop
      Test_Procedure
        Test_Sequence
          Pattern
            Event
```

If the `TBDbin` contains signature based TBD data, the `TBDbin` hierarchy consists of these following entities:

```
Experiment
   Test_Section
      Tester_Loop
         Test_Procedure
            Signature interval
               Iteration 1
               Iteration 2
               Iteration 3
               Iteration 4
               Iteration 5
            Signature interval 2
               Iteration 1
               Iteration 2
                     .
                     .
                     .
```

Although these elements are not numbered in the `TBDbin`, individual elements can be referenced in Encounter Test tools by using a hierarchical numbering scheme (commonly referred to as odometer represent levels in the hierarchy shown in Figure 2-1 on page 48. So, as an example, 1.2.3.4.5.6.7 would be uncommitted test 1, test_section 2, tester_loop 3, test_procedure 4, test_sequence 5, pattern 6, event 7.

The following gives a brief description of each of these levels of hierarchy. Additional information about this and other data contained in the `TBDbin` is included in "Test Pattern Data Overview" in the *Encounter Test: Reference: Test Pattern Formats*.

Refer to "Viewing Test Data" in the *Encounter Test: Reference: GUI* for details on viewing and analyzing test data using the graphical interface.

Experiment

The experiment groups a set of `test_sections` generated from "one" test generation run. The experiment name is specified at the invocation of the test generation application. The "one" run may actually be multiple invocations of test generation appended to the same experiment name. Refer to "Experiment" in the *Encounter Test: Reference: Test Pattern Formats* for details.

Define_Sequence

This is basically a template of a test sequence that is used within this experiment. There will be one of these for each unique test sequence template used in the experiment. The sequence definition includes timing information for delay testing and a description (template) of the patterns and events. The `Define_Sequence` currently is found only within experiments that contain timed dynamic tests. Refer to "TBDpatt and TBDseqPatt Format" in the *Encounter Test: Reference: Test Pattern Formats*.

Timing_Data

This defines a tester cycle and the specific points within the tester cycle when certain events (defined in terms of the event type and signal or pin id) are to occur. Refer to "AC Test Application Objects" in the *Encounter Test: Reference: Test Pattern Formats* for a more detailed description of timing data. The sequence definition may contain several `Timing_Data` objects, each one defining a different timing specification for testing the product. The choice of which `Timing_Data` object is controlled by the `Test_Sequence` which uses this sequence definition.

Test_Section

The `test_section` groups a set of `tester_loops` within an experiment that have the same tester setup attributes (such as tester termination), and the same types of test (see "General Types of Tests" on page 30). Refer to "Test_Section" in the *Encounter Test: Reference: Test Pattern Formats* for details.

Tester_Loop

The `tester_loop` groups a set of `test_procedures` that is guaranteed to start with the design in an unknown state. Thus, the `tester_loops` can be applied independently at the tester. Refer to "Tester_Loop" in the *Encounter Test: Reference: Test Pattern Formats* for details.

Test_Procedure

The `test_procedure` is a set of `test_sequences`. `Test_Procedures` are sometimes able to be applied independently at the tester. If the `test_procedures` within a `tester_loop` cannot be applied independently, an attribute to that effect is included in the containing `tester_loop`. Refer to "Test_Procedure" in the *Encounter Test: Reference: Test Pattern Formats* for details.

There are attributes in the `test_procedure` to inform manufacturing about the effectiveness of the `test_sequences` it contains. This is a statement of the number of static faults, dynamic faults, and driver/receiver objectives that were detected with these `test_sequences` and the test coverage calculated up to this point in the `TBDbin`.

Test_Sequence

A `test_sequence` is a set of test patterns that are geared toward detecting a specific set of faults. The patterns are intended to be applied in the given order. Refer to "Test_Sequence" in the *Encounter Test: Reference: Test Pattern Formats* for details.

Pattern

A pattern is a set of events that are applied at the tester in the specific order specified. Refer to "Pattern" in the *Encounter Test: Reference: Test Pattern Formats* for details.

Event

An event is the actual stimulus or response data (or any other data for which ordering is important). The data within the event is represented as a string of logic values called a vector. The pin to which the value applies is determined by the relative position of the logic value in the vector. Encounter Test maintains a vector correspondence list, that indicates the order of the primary inputs, primary outputs and latches as they appear in vectors.

There is a file created in Encounter Test called `TBDvect`. This file contains the vector correspondence information. It can be edited if you need to change the order of the PIs, POs, and/or latches in the vectors. Refer to "Event" in the *Encounter Test: Reference: Test Pattern Formats*.

The following sections give information on providing test data to manufacturing and analyzing test data.

Test Data for Manufacturing

There is no one vector format accepted by all component manufacturers. Therefore, Encounter Test provides these vector formats that can be used for interfacing with various manufacturers.

1. `TBDbin` - the compact binary format used by Encounter Test applications can also be used as a manufacturing interface. The `TBDbin`, together with various other Encounter Test files are packaged into a Encounter Test Manufacturing Data (TMD) file which is accepted by manufacturing sites.
2. Waveform Generation Language (WGL)** - this format can be translated to many tester formats using Fluence Technologies, Inc. WGL In-Convert applications. Also, some manufacturers have their own translators for WGL.

See “WGL Pattern Data Format” in the *Encounter Test: Reference: Test Pattern Formats* for more detail.

3. Verilog** - this format is used by many manufacturers to drive a “sign off” simulation of the vectors against the component model prior to fabrication. It can also be translated into many tester formats by the manufacturers.

   See “Verilog Pattern Data Format” in the *Encounter Test: Reference: Test Pattern Formats* for more detail on Verilog.

4. Encounter Test Pattern Data (`TBDpatt`) - The `TBDpatt` that is output from Encounter Test contains all the same “manufacturing data” as the `TBDbin`, WGL, and Verilog forms. The `TBDpatt` is an ascii form of the `TBDbin`.

   See “TBDpatt and TBDseqPatt Format” in the *Encounter Test: Reference: Test Pattern Formats* for additional information.

5. STIL data - Encounter Test exports test vectors in STIL format, conforming to the IEEE Standard 1450-1999 Standard Test Interface Language (STIL), Version 1.0 standard.

   See “STIL Pattern Data Format” in the *Encounter Test: Reference: Test Pattern Formats* for additional details.

Note: See “Writing and Reporting Test Data” on page 15 for information about creating (exporting) these test data forms.

Test Vector Forms

Test vectors may be created in any of the following forms:

Static

Static tests are structured to detect static (DC) defects. The detection of DC defects does not require transitions, so the design is expected to settle completely before the next event is applied.

Dynamic

Dynamic tests are structured to create a rapid sequence of events. These events are found inside the dynamic pattern and are identified as release events (launch the transition), propagate events (propagate the transition to the capture latch) and capture events (capture the transition). Within the dynamic pattern the events are expected to be applied in rapid succession. The speed is at the discretion of the manufacturing site, since there are no timings to describe how fast they can be applied.

Dynamic Timed

Dynamic timed tests are dynamic tests that have associated timings. In this case the speed with which the events are expected to be applied at the tester is explicitly stated in the timing data.

Tests for Sorting Product

It is possible to create tests that can be used to sort the product by applying the test at several rates of speed. Not all manufacturers support this, so you must contact your manufacturer before sending them test data with several different sets of timing data.

The manufacturing process variation curve is a normal distribution. Sorting the tests is done by selecting a point on the process variation curve through the setting of the coefficients (best, nominal, worst) of a linear combination.

3

Reading Test Data and Sequence Definitions

This chapter discusses the commands and concepts to read in test patterns and test sequences into Encounter Test.

The chapter covers the following topics:

Reading Test Data

Test vectors from other formats can be read into Encounter Test. Test vectors are read and stored into the Encounter Test format called `TBDbin`. You can then use these test vectors in `TBDbin` format as input to any Encounter Test command that takes existing experiments.

You can read from the following types of formats:

- Encounter Test Pattern Data (`TBDpatt`)
- Standard Test Interface Language (STIL)
- Extended Value Change Dump (EVCD) file

To read Encounter Test Pattern Data using the graphical interface, refer to "Read Vectors" in the *Encounter Test: Reference: GUI*.

To read Encounter Test Pattern Data using command lines, refer to "read_vectors" in the *Encounter Test: Reference: Commands*.

The syntax for the `read_vectors` command is given below:

```
read_vectors workdir=<directory> testmode=<modename> experiment=<name>
language=<type> importfile=<filename>
```

where:

- `workdir` = name of the working directory
- `testmode=` name of the testmode for dynamic ATPG
- `experiment=` name of the output experiment resulting from simulation
- `language=` type of patterns to be imported
- `importfile=` location of patterns to be imported

The most commonly used keywords for the `read_vectors` command are:

- `language= stil|tbdpatt|evcd` - Type of language in which the patterns are being read
- `importfile=STDIN|<infilename>` - Allows a filename or piping of data
- `uniformsequences=no|yes` - STIL import options to create test procedures with uniform clocking sequences. Default is `no`.
- `identifyscantest=no|yes` - STIL import options to identify a scan integrity test, if exists.

Refer to “read_vectors” in the *Encounter Test: Reference: Commands* for more information on these keywords.

Prerequisite Tasks

Complete the following tasks before running Test Simulation:

1. Import a design into the Encounter Test model format. Refer to “Performing Build Model” in the *Encounter Test: Guide 1: Models*.
2. Create a Test Mode. Refer to “Performing Build Test Mode” in the *Encounter Test: Guide 2: Testmodes*.
3. Have the test patterns in the supported format.

Read Vectors Restrictions

The following restrictions apply to conversion of all test data formats:

- `TBDpatt` that contains Load or Unload events, most likely created by Convert Vectors for Core Tests, cannot be read.

- Conversion of vectors produced from designs with grandparent test modes is not supported. Refer to "Building Multiple Test Modes" in the *Encounter Test: Guide 2: Testmodes* for related information.

Reading Encounter Test Pattern Data (TBDpatt)

You can read `TBDpatt` files into Encounter Test. `TBDpatt` data files produced by writing test data can be edited and resimulated by Encounter Test. This gives you the ability to perform uncommitted test with test vectors as you work to achieve additional fault coverage on a design. For information on how to create `TBDpatt` format data, refer to "Writing and Reporting Test Data" on page 15.

`TBDpatt` is the recommended format for entering manually generated test vectors into Encounter Test because the following reasons:

- `TBDpatt` format translates directly into `TBDbin` format

  The probability of information loss is less when translating between `TBDpatt` and `TBDbin` formats.

- `TBDpatt` format can be easily created

  You can create a `TBDpatt` file for input to Encounter Test by first generating test vectors (even just random ones) and then converting the resulting `TBDbin` file into a `TBDpatt` file. You can then edit the `TBDpatt` file to reflect the desired primary input and latch stimulus values.

To read Encounter Test Pattern Data using the graphical interface, refer to "Read Vectors" in the *Encounter Test: Reference: GUI*.

To read Encounter Test Pattern Data using command lines, refer to "read_vectors" in the *Encounter Test: Reference: Commands*.

Encounter Test Pattern Data Output Files

Encounter Test Pattern Data stores test vectors in an uncommitted vectors file with the format `TBDpatt.`*`testmode.experiment`*. You must specify an uncommitted test name, which will identify the read test vectors.

Reading Standard Test Interface Language (STIL)

Encounter Test reads data that conforms to the IEEE 1450-1999 STIL, Version 1.0 standard. The `TBDbin` produced by STIL can be used by any Encounter Test simulation or conversion tool that accepts `TBDbin` format.

Initial support for STIL that conforms to the IEEE 1450.1 standard is available. Refer to "Support for STIL Standard 1450.1."

Encounter Test reads, parses, and performs semantic checks on the constructs found in the input STIL file in addition to validating structural references and test vector data.

For information on Encounter Test STIL pattern data, refer to the following sections of the *Encounter Test: Reference: Test Pattern Formats*.

- "STIL Pattern Data Format"
- Appendix E, "STIL Pattern Data Examples"

Support for STIL Standard 1450.1

Encounter Test currently supports reading the IEEE P1450.1 standard constructs as follows:

- The `WFCmap` construct within the Signals Block
- `X` (cross-reference) statements
- `UserKeywords` statement extension
- `resource_id` tags (1450.3)
- `InheritWaveForTable`
- `Specify` block with multiple `Category` definitions
- `WaveFormTable` with signal blocks having definitions for waveform characters
- Parse-only support for the following constructs:

ActiveScanChains, AllNames, CellIn, CellOut, Constant, Cycle, Data, Design, E, Else, Enumeration, Enumerations, Environment, Equivalent, Extend, Fixed, FileFormat, FileReference, FileType, FileVersion, Format, If, Independent, InitialValue, Instance, Iteration, InheritEnvironment, InheritNameMap, Integer, IntegerConstant, IntegerEnum, Lockstep, NameMaps, Offset, ParallelPatList, PatSet, PatternOffset, Real, ScanChainGroups, ScanEnable, SignalVariable, SignalVariableEnum, SyncStart, TagOffset, Type, Usage, Values, Variables, Version, Wait,While

Support for STIL Standard 1450.2

Encounter Test currently has parse-only support for the following IEEE 1450.2 standard constructs:

- `DCLevels`
- `Environment`

STIL Restrictions

It is strongly recommended that the STIL data that you read should be in an Encounter Test-generated file, produced by "Writing Standard Test Interface Language (STIL)" on page 25. The results for STIL data generated outside of Encounter Test are not guaranteed.

STIL scan protocol must match the scan procedures automatically generated by Encounter Test. Custom Scan Protocol procedures are not supported. If you do use custom scan protocol, we recommend that the read tests be simulated prior to sending data to the tester. Refer to "Custom Scan Sequences" in *Encounter Test: Guide 2: Testmodes* for more information.

Encounter Test does not support reading STIL data for designs with scan pipelines in compression testmodes.

Identifying Scan Tests in STIL Vectors

A sequence is a scan test if the following conditions are present:

1. The first event in the sequence is the `Scan_Load` event.

2. The `Scan_Load` event loads all scan strings with alternating pairs of 1's and 0's. Note that this may start at any point (for example, 00110011..., 011001100..., 11001100..., or 1011001100...).
3. There may be optional `Stim_PI` and `Measure_PO` events after the first event and before the last event.
4. If there are any *pulse* events after the first event and before the last event, these *pulse* events may only pulse clocks flagged as scan clocks.
5. The last event in the sequence is the `Scan_Unload` event. The `Scan_Unload` event measures alternating pairs on each scan chain.

Identifying Mode Initialization Sequences in STIL Vectors

The `read_vectors` command, when working on STIL input data, compares the initial patterns in the input STIL data against the mode initialization sequence as identified while creating the test mode.

This comparison helps identify the mode initialization sequence even when the end of the sequence is merged with the first real test pattern in the STIL data.

Note: The `read_vectors` command compares only the events directly controlled by the tester. Internal events in the mode initialization sequence, such as the stimulation of pseudo-primary inputs or the pulsing of pseudo-primary input, clocks will be ignored in this comparision.

If the first few STIL vectors match the mode initialization sequence, then read_vectors replaces the matching vectors with the mode initialization sequence identified for this test mode.

This replacement enables the resulting TBDbin (binary file of TBDPatt) to correctly identify any internal events associated with the mode initialization. This enables `read_vectors` to support the automatic creation of internal events for the mode initialization sequence, but not for normal test patterns.

If the first few patterns in the STIL data do not match the mode initialization sequence identified in the test mode, then the `read_vectors` command explicitly adds a mode initialization sequence from the TBDseq file to the output patterns before converting any of the STIL vectors. This makes sure that the mode initialization sequence occurs before the patterns in the STIL data in the resulting test pattern file.

As the mode initialization sequence is taken from the TBDseq file, this comparision will also correctly identify it as a mode initialization sequence in the resulting TBDbin file. When you resimulate the patterns, the simulator will not insert another copy of the mode initialization sequence.

Note: The `read_vectors` command does not identify a mode initialization sequence that is functionally equivalent to the mode initialization sequence of the test mode, but does not exactly match the test mode initialization sequence.

Reading Extended Value Change Dump (EVCD) File

NC-Verilog creates extended value change dump (EVCD) files that conform to the IEEE 1364-2001 Verilog standard. You can then read the EVCD file into Encounter Test for creating Verilog test vectors, simulating and fault grading functional vectors, and detecting and analyzing miscompares.

An EVCD file is an ASCII file that contains information about value changes on selected variables in the design. The file contains header information, variable definitions, and the value changes for all specified variables. Encounter Test accepts EVCD files through the Read Vectors application.

To produce EVCD output from the Cadence NC Simulator, add the following code in the Verilog testbench.

```
initial  $dumpports (< instance > ,"My_Functional_Patterns.EVCD",,2) ;
```

where:

`<instance>` = The instance name of the logic block whose ports are to be dumped. This is usually the chip under test.

`My_Functional_Patterns.EVCD` = ASCII file that NC Simulator will produce. Specifying quotes is mandatory. This file will be imported into Encounter Test

`,,2` = Formatting place holders for *`$dumpports`*. `2` produces the specific variation of the EVCD output that Encounter Test accepts. This format conforms to the IEEE standard

For more information on creation and content of the files, refer to the following sections in *Cadence® NC-Verilog® Simulator Help*:

- Generating a Value Change Dump (VCD) File
- Generating an Extended Value Change Dump (EVCD) File for a Mixed-Language Design
- Generating an Extended Value Change Dump (EVCD) File

Tip

It is recommended to follow some rules while reading EVCD:

- ❑ Ensure that the input data only changes when the clock(s) are OFF.
- ❑ Do not have the data and clock signals changing at the same time.
- ❑ Do not have overlapping clocks (unless they are correlated).

To read the generated EVCD file into Encounter Test using the graphical interface, refer to "Read Vectors" in the *Encounter Test: Reference: GUI*. Select *EVCD* for *Input Format*.

To read the generated EVCD file into Encounter Test using command lines, refer to "read_vectors" in the *Encounter Test: Reference: Commands*. Specify `language=evcd`.

Tip

The TBDbin consists of a single Experiment, Test_Section, and Tester_Loop. The type of Test_Section created in the TBDbin depends on the specified value for `testtype` or *Test section type* if using the GUI. The default Test_Section type is `logic`. Refer to "Encounter Test Vector Data" on page 48 for more information.

The `termination` keyword sets the tester termination value to be assumed for the Test_Section. The default termination setting is `none`.

The `inputthreshold` keyword defines the input threshold. Signals less than or equal to the specified threshold are interpreted as `Z`.

The `outputthreshold` keyword defines the output threshold. Signals less than or equal to the specified threshold are interpreted as `X`.

The `measurecyclesboundary` keyword is used to insert a `Measure_PO` event on a cycle boundary when no measure exists in the EVCD input.

EVCD Restriction

When a user requests that Encounter Test import EVCD patterns, it usually is with the expectation that functional test patterns will be fault simulated. However, the following limitations are to be considered:

- The functional patterns are imported as STATIC patterns. You cannot fault simulate and have Encounter Test detect DELAY or PATH Delay faults. Fault simulation will only detect STATIC faults.

- Static fault detection may be disappointing. Note that a fault is detected only when it reaches a measure point. For functional patterns only the chip output pins are measure point. Faults that propagate to internal Register, Parity and Error checking logic are not declared detected. Those faults must further propagate to an output pin.
- The concept of an internal strobe point similar to `$fs_strobe` for Cadence Verifault is not available.
- The `write_vectors` output of these captured patterns, by default, has static fault timing. The tester cycle is 80ns and the Stim, Pulse and Measure events are widely spaced. Users who want to run these patterns at functional clock speeds need to adjust the timings via `write_vectors` parameters and verify with SDF annotate timing in NCsim.
- Encounter Test supports only the single variable type of port. A message is issued if a port of any other variable type is detected.

EVCD Output

EVCD values that refer to Primary Inputs are translated to `Stim_PI` or `Pulse` events. Primary Output values are translated to `Measure_PO` events. Refer to "Event" in the *Encounter Test: Reference: Test Pattern Formats* for descriptions of these events.

Reading Sequence Definition Data (TBDseq)

Besides reading tests in the form for direct application, Encounter Test also supports reading test sequence definitions. Sequence definitions are used in cases where the same input sequence is used repetitively.

There are two broad categories of sequence definitions:

- Test

  This type of sequence defines the clocking template for a set of tests in terms of which clocks are to be pulsed, and in what order. The test sequence also specifies any other events that are to be applied, such as the scan operation and stims, on any pins that do not vary across tests.

- Others

  A sequence definition of any type other than test is used for operations that prepare for a test or read out the results of a test. There are many different such types, for example scan preconditioning (`scanprecond`) and scansequence. Refer to Define_Sequence in the *Encounter Test: Reference: Test Pattern Formats* to know more about the sequence definition types recognized by Encounter Test.

A sequence definition is similar to a subroutine in programming. It is not necessary for every sequence definition to have a recognized type. User-written sequence definitions can, in general, invoke other sequence definitions might not have an identified 'type'.

After reading the test vector, you can use the Test Simulation application to simulate the vectors. You can choose to perform good machine or fault machine simulation and forward or reverse vector simulation.

To read Sequence Definition Data using the graphical interface, refer to "Read Sequence Definition" in the *Encounter Test: Reference: GUI*.

To read Sequence Definition Data using command lines, refer to "read_sequence_definition" in the *Encounter Test: Reference: Commands*.

The syntax for the `read_sequence_definition` command is given below:

```
read_sequence_definition workdir=<directory> testmode=<modename>
importfile=<filename>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode for dynamic ATPG
- `importfile`= location of sequences to import

The most commonly used keyword for the `read_sequence_definition` is:

`importfile=STDIN|`*`<infilename>`* - Allows a file name or piping of data

Refer to "read_sequence_definition" in the *Encounter Test: Reference: Commands* for more infrormation on the keywords.

Prerequisite Tasks

Complete the following tasks before running Test Simulation:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in the *Encounter Test: Guide 1: Models*.
2. Create a Test Mode. Refer to "Performing Build Test Mode" in *Encounter Test: Guide 2: Testmodes*.

Sequence Definition Data Output Files

Sequence definition data produces the file `TBDseq.`*`testmode`* that contains the initialization sequences.

4 Simulating and Compacting Vectors

Encounter Test has capabilities to simulate and manipulate experiments to reduce the number of patterns, fault simulate patterns, calculate new values, and verify existing patterns.

This chapter discusses the following:

Compacting Vectors

With limited tester resources, when you want to limit the number of ATPG patterns to be loaded onto a tester, it is strongly recommended that you compact the vectors to produce a more ideal set of patterns before they limit the patterns being applied to the tester. Typically, the test pattern set created during ATPG does not represent the most ideal order of patterns to apply at a tester. For example, because of different clocking and random fault detection, pattern #100 might test more faults than pattern #60. If you can only apply 70 patterns, they would want pattern #100 to be applied because of the number of faults it detects.

Compact Vectors is designed to help order the patterns to allow the steepest test coverage curve.

By default the compact vectors sorts an experiment from ATPG based on the number of faults tested per experiment. The tools round the test coverage down to the nearest 0.05% and use this as the cut-off coverage number. If the experiment is based on dynamic patterns, both the static and dynamic coverages are reduced by 0.05%, and both numbers need to be reached before stopping. The coverage number is based on AT-Cov or adjusted fault coverage.

When the input consists of multiple experiments, test sections, or tester loops (refer to "Encounter Test Vector Data" on page 48 for more information on these terms), Compact Vectors tries to combine all the vectors into a single test section and tester loop. However, for that to happen, all the following conditions must be met:

- Test section type must be logic or WRP
- Test types (static, dynamic) must be the same
- Tester termination must be the same
- Pin timings must be the same
- Any linehold fixed-value latch (FLH) values must be consistent (that is, for each FLH, its value must be the same for all experiments or test sections being combined)
- There must be consistent usage (or non-usage) across the test sections of tester PRPGs, tester signatures, product PRPGs, and product signatures

To perform Compact Vectors using the graphical interface, refer to "Compact Vectors" in the *Encounter Test: Reference: GUI*.

To perform Compact Vectors using command line, refer to "compact_vectors" in the *Encounter Test: Reference: Commands*.

The syntax for the `compact_vectors` command is given below:

```
compact_vectors workdir=<directory> testmode=<modename> inexperiment=<name>
experiment=<name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode for dynamic ATPG
- `inexperiment`= name of the experiment to run re-order

- `experiment=` output experiment name

The most commonly used keywords for the `compact_vectors` command are:

- `resimulate=yes|no` - Set to `no` to not resimulate the result patterns. Default is `yes` to resimulate the results.
- `reordercoverage=both|static|dynamic` - Specify the fault types to drive sorting. The default is based on the type of ATPG patterns that are being simulated.
- `maxcoveragestatic=#` - Stop patterns at a specific static coverage number (for example `99.00`)
- `maxcoveragedynamic=#` - Stop patterns at a specific dynamic coverage number (for example `85.00`)
- `numtests=#` - Stop at a specific pattern count. This is an estimate as multiple patterns are simulated in parallel, so the total final pattern count might be slightly higher (less than 64 away).

Refer to "compact_vectors" in the *Encounter Test: Reference: Commands* for more information on these keywords.

Prerequisite Tasks

Complete the following tasks before running Test Simulation:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in the *Encounter Test: Guide 1: Models*.
2. Create a Test Mode. Refer to "Performing Build Test Mode" in *Encounter Test: Guide 2: Testmodes*.
3. Run ATPG or have experiments to compact. Refer to "Invoking ATPG" on page 35 for more information.

Output

The output includes the re-ordered experiment. Use this output experiment for exporting or committing test data.

Output Message

The following are some messages that can be seen:

- INFO (TWC-016)

A sample output log is given below:

```
          -------------Total Fault Simulation Statistics----------

Tests Simulated..........................41   #Number of patterns simulate
Effective Tests..........................32   #Number of patterns kept
Detects..................................1455 #Fault detection information
3-STATE Contentions
       #Identify any contention found
       Expect/Measure
       Good Compares ..........................2035 #Good/Bad compares found
       Misc
       Compares .............................0
```

Simulating Vectors

Simulate Vectors reads an ASCII pattern set and simulates it with one command. Another way to do this is to perform Read Vectors (refer to "Reading Encounter Test Pattern Data (TBDpatt)" on page 57) and then Analyze Vectors (refer to "Analyzing Vectors" on page 72).

To perform Test Simulation using the graphical interface, refer to "Simulate Vectors" in the *Encounter Test: Reference: GUI*.

To perform Test Simulation using command lines, refer to "simulate_vectors" in the *Encounter Test: Reference: Commands*.

The syntax for the `simulate_vectors` command is given below:

```
simulate_vectors workdir=<directory> testmode=<modename> experiment=<name>
language=<value> importfile=<filename>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode for dynamic ATPG
- `experiment`= name of the output experiment from simulation
- `language`= Type of patterns to import
- `importfile`= Location of patterns to import

The most commonly used keywords for the `simulate_vectors` command are:

- `language= stil|tbdpatt|evcd` - Type of language in which the patterns are being read.

- `gmonly=no|yes` - Perform good machine simulation (no fault mark off). Default is `no`.
- `simulation=gp|hsscan` - Select the simulator to use. Refer to "Test Simulation Concepts" on page 75 for more information.
- `delaymode=zero|unit` - Select the simulation type. Only applicable while using `simulation=gp`. Refer to "Zero vs Unit Delay Simulation with General Purpose Simulator" on page 84 for more information.

Refer to "simulate_vectors" in the *Encounter Test: Reference: Commands* for more information on these keywords.

Prerequisite Tasks

Complete the following tasks before running Simulate Vectors:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in the *Encounter Test: Guide 1: Models*.
2. Create a Test Mode. Refer to "Performing Build Test Mode" in *Encounter Test: Guide 2: Testmodes*.
3. Run ATPG or have experiments to compact. Refer to "Invoking ATPG" on page 35 for more information.
4. Have STIL or TBDpatt pattern files. For limitations on patterns, refer to "Reading Encounter Test Pattern Data (TBDpatt)" on page 57 for more information.

Refer to "Overall Simulation Restrictions" on page 89 for general simulation restrictions.

Output

The output is a new experiment containing the simulated results of the input patterns. The output also contains the new calculated values in case of miscompares.

Output Messages

Some of the messages generated by `simulate_vectors` are given below:

- INFO (TIM-800)
- INFO (TIM-805)
- INFO (TWC-015)

A sample output is given below:

```
--------------Total Fault Simulation Statistics----------

Tests Simulated..........................41 #Number of patterns simulate
Effective Tests..........................32 #Number of patterns kept
Detects..................................1455#Fault detection information
3-STATE Contentions # Identify any contention found
Expect/Measure
Good Compares ...........................2035 #Good/Bad compares found
Miscompares ..............................0
```

Analyzing Vectors

Use Analyze Vectors to simulate an existing Encounter Test experiment. This can be useful when cross checking simulation results, manipulating patterns, or creating scope sessions for further debug.

To perform Test Simulation using the graphical interface, refer to "Analyze Vectors" in the *Encounter Test: Reference: GUI*.

To perform Test Simulation using command lines, refer to "analyze_vectors" in the *Encounter Test: Reference: Commands*.

The syntax for the `analyze_vectors` command is given below:

```
analyze_vectors workdir=<directory> testmode=<modename> inexperiment=<name>
experiment=<name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode for dynamic ATPG
- `inexperiment`= name of the experiment to be simulated
- `experiment`= name of the output experiment resulting from simulation

The commonly used keywords for the `analyze_vectors` command are:

- `gmonly=no|yes` - Perform good machine simulation (no fault mark off). Default is `no`.
- `simulation=gp|hsscan` - Select the simulator to use. Refer to "Test Simulation Concepts" on page 75 for more information.
- `delaymode=zero|unit` - Select the simulation type. Only applicable if using `simulation=gp`. Refer to "Zero vs Unit Delay Simulation with General Purpose Simulator" on page 84 for more information.

- `contentionreport= soft|hard|all|none` - The type of contention to report on. Default is `soft`.
- `watchnets=all|file |none|scan|trace` - Specify to create simvisions waveforms files. Default is `none`. Requires `simulation=gp`.

Refer to "analyze_vectors" in the *Encounter Test: Reference: Commands* for more information on these keywords.

Prerequisite Tasks

Complete the following tasks before running Analyze Vectors:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in the *Encounter Test: Guide 1: Models*.
2. Create a Test Mode. Refer to "Performing Build Test Mode" in *Encounter Test: Guide 2: Testmodes*.
3. Run ATPG or have experiments to analyze. Refer to "Invoking ATPG" on page 35 for more information.

Refer to "Overall Simulation Restrictions" on page 89 for general simulation restrictions.

Output

The output is a new experiment containing the simulated results of the input patterns. The output also contains the new calculated values in case of miscompares.

A sample output is given below:

```
--------------Total Fault Simulation Statistics----------
Tests Simulated..........................41 #Number of patterns simulate
Effective Tests..........................32 #Number of patterns kept
Detects..................................1455#Fault detection information
3-STATE Contentions # Identify any contention found
Expect/Measure
Good Compares ...........................2035 #Good/Bad compares found
Miscompares .............................0
```

Timing Vectors

Use Time Vectors to time/re-time an existing Encounter Test experiment. This will time only dynamic events found in the test patterns. This simulation is good machine only.

To perform Time Vectors using the graphical interface, refer to "Time Vectors" in the *Encounter Test: Reference: GUI*.

To perform Time_Vectors using command lines, refer to "time_vectors" in the *Encounter Test: Reference: Commands*.

The syntax for the `time_vectors` command is given below:

```
time_vectors workdir=<directory> testmode=<modename> inexperiment=<name>
experiment=<name> delaymodel=<name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode for dynamic ATPG
- `inexperiment`= name of the experiment to simulate
- `experiment`= name of the output experiment resulting from simulation
- `delaymodel`= name of the delay model to use

The most commonly used keywords for the time_vectors command are:

- `constraintcheck=yes|no` - Ability to turn off constraint checks. Default is `yes` to check constraints.
- `earlymodel/latemode` - Ability to customize delay timings. Default is `0.0,1.0,0.0` for each keyword. Refer to "Process Variation" on page 142 for more information.
- `clockconstraints=`*`<file name>`* - List of clocks and frequencies to perform ATPG. Refer to "Clock Constraints File" on page 138 for more information.
- `printtimings=no|yes` - Print timing summary for each clock

Refer to "time_vectors" in the *Encounter Test: Reference: Commands* for more information on these keywords.

Prerequisite Tasks

Complete the following tasks before running Time Vectors:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in the *Encounter Test: Guide 1: Models*.
2. Create a Test Mode. Refer to "Performing Build Test Mode" in *Encounter Test: Guide 2: Testmodes*.

3. Have an existing dynamic experiment.

Refer to "Overall Simulation Restrictions" on page 89 for general simulation restrictions.

Output

The output a new experiment that is timed to the new timings found in the clock constraint files or the specified options.

Test Simulation Concepts

Encounter Test can be used to simulate test patterns that already exist. In order to simulate the test patterns, they must be available in a `TBDbin` file format. If you have the `TBDbin` file from a previous test generation uncommitted test, that file can be simulated by providing the uncommitted test name to the Test Simulation application. If the test patterns to be simulated are not currently available in a `TBDbin` file format, you can import the patterns into a `TBDbin` format using the import test data tool.

For more information, refer to "Reading Test Data and Sequence Definitions" on page 55.

Once the test patterns have been imported, they can be simulated by Encounter Test. The types of simulation available in Encounter Test are:

- High Speed Scan-Based Simulation - high speed simulation for designs and patterns that adhere to the scan design guidelines.
- General Purpose Simulation - simulation for designs and patterns independent of whether they adhere to the scan design guidelines.
- Good Machine Delay Simulation - simulation for designs and patterns that have a delay model to accurately reflect the timing in the design.

High Speed Scan-Based Simulation provides high-speed fault simulation for efficient test pattern generation. It requires designs and input patterns to adhere to the LSSD and GSD guidelines for Encounter Test. Input patterns must adhere to the following constraint:

- Patterns cannot place a test inhibit (TI) function pin or a test constraint (TC) function pin away from its stability value.

General Purpose Simulation provides a more flexible simulation capability than High Speed Scan-Based Simulation. A design is not required to adhere to design guidelines. Since this is a fully event-driven simulator, general sequential logic is handled. Also, this simulation does not impose particular clock or control input sequencing constraints on the input patterns. This type of simulation is used:

- to supplement High Speed Scan-Based Simulation for parts of a design that are not compatible with automatic test generation.
- to provide a means to insert functional tests into the tester data stream and to perform fault grading of functional tests.
- to verify test patterns before they are sent to a design manufacturer.

General Purpose Simulation compares its measurement results to those previously achieved and alerts you to any differences. For example, when a miscompare between an expected result and a simulator predicted result is found, analysis is needed to determine why the miscompare occurred. You can use analyze test patterns using *View Vectors*. Select the View Vectors icon on the Encounter Test schematic display to view simulation results and provide visual insight into the test data.

Note: Although General Purpose Simulation provides a less constrained simulation capability compared to High Speed Scan-Based Simulation, it does so at the price of simulation speed. General Purpose Simulation will typically be at least an order of magnitude slower than High Speed Scan-Based Simulation on the same design with the same patterns. The key here is that General Purpose Simulation should only be used in those cases where High Speed Scan-Based Simulation cannot be applied.

General Purpose Simulation provides some features for optimizing its performance under varying conditions of design size, fault count, and computer resource availability. The primary means for controlling the run time and memory usage during simulation is a technique known as multi-pass fault simulation. This means that General Purpose Simulation can simulate a given pattern set multiple times. For each pass, the unsimulated subset of the fault list is chosen for simulation. This reduces the memory requirements of fault simulation.

There is no way to determine exactly how much memory is required and hence calculate the ideal number of faults that should be simulated for a given pass. General Purpose Simulation makes decisions for this number, but you can specify the Number of Faults per Pass and the Maximum Number of Overflow Passes to control the multi-pass process.

Another feature of General Purpose Simulation that controls run time is fault dropping. As simulation progresses, the simulator detects that certain faults are consuming an inordinate amount of run time. When detected, these faults are dropped from the simulation for the current run. Note that any faults so dropped are attempted for simulation on successive runs, thus detection of the fault is still a possibility. The Machine Size sets the threshold at which fault dropping occurs.

Good Machine Delay Simulation is similar to General Purpose Simulation in that it is an event driven simulation. As the name implies, however, it does not perform fault simulation. Good

Machine Delay Simulation should be used when simulation of a design with delays and/or time-based stimulus and measurements is required. Good Machine Delay Simulation can not be used for weighted-random (WRPT) or LBIST tests unless these tests are first converted to stored pattern format using the Manipulate Tests tool.

Notes:

1. To perform Test Simulation using the graphical interface, refer to "Simulate Vectors" in the *Encounter Test: Reference: GUI*.
2. To perform Test Simulation using command lines, refer to "simulate_vectors" in the *Encounter Test: Reference: Commands*.
3. Also refer to "Simulating Vectors" on page 70.

Specifying a Relative Range of Test Sequences

Use `testrange =<relative test sequence number>` to specify a relative range of test sequences. A relative test sequence number is an integer value that identifies the position of the test sequence relative to the start of the pattern file. The first test sequence in the file has a relative test sequence number of 1. The next test sequence has a relative test sequence number of 2, and so on.

For example, specify `write_vectors testrange=1:5` to report the first five test sequences. The following is a sample output:

```
                TEST SEQUENCE COVERAGE SUMMARY REPORT

Test       |Static  |Static  |Dynamic |Dynamic |Sequence|Overlapped|Total |
Sequence   |Total   |Delta   |Total   |Delta   |Cycle   |Cycle     |Cycle |
           |Coverage|Coverage|Coverage|Coverage|Count   |Count     |Count |

1.1.1.1.1 |0.00    |0.00    |0.00    |0.00    |1       |0         |1     |
1.1.1.2.1 |31.83   |31.83   |0.00    |0.00    |183     |0         |184   |
1.2.1.1.1 |31.83   |0.00    |0.00    |0.00    |1       |0         |185   |
1.2.1.2.1 |44.79   |12.96   |0.00    |0.00    |174     |86        |359   |
1.2.1.2.2 |51.72   |6.93    |0.00    |0.00    |88      |86        |447   |
1.2.1.2.3 |55.43   |3.70    |0.00    |0.00    |88      |86        |535   |
1.2.1.2.4 |57.78   |2.35    |0.00    |0.00    |89      |0         |624   |
```

In the log given above:

- `1.1.1.1.1` is the modeinit for scan test section
- `1.1.1.2.1` (scan test) has the relative pattern 1
- `1.2.1.1.1` is the modeinit for logic test section
- `1.2.1.2.1` (first logic test) has the relative pattern 2

- `1.2.1.2.2` has the relative pattern 3
- `1.2.1.2.3` has the relative pattern 4
- `1.2.1.2.4` has the relative pattern 5

You can also specify a combination of odometer value with relative numbers. Any testrange entry in a comma separated list of testranges that contains a period is assumed to be an odometer entry. Any entry without a period is assumed to be a relative test number.

For example, to extract the first 10 sequences along with the sequence 2.4.1.1.1, specify the test range as follows:

```
testrange=1:10,2.4.1.1.1
```

Note: A relative test number cannot be used to identify a modeinit, scan or setup sequence. These sequences must be identified using the odometer value.

To specify a range of experiments in the `testrange` keyword, use a period (.) between the experiment numbers, such as shown below:

```
testrange=1.:2.
```

Fault Simulation of Functional Tests

Fault simulation of functional tests can consist of a challenging set of tasks. Depending on the size of the design, the number of functional patterns, and number of faults to be simulated, the simulation can take large amounts of CPU time, system memory and file space.

Prerequisites for Fault Simulation

1. Build model. Refer to build_model in the *Encounter Test: Reference: Commands* for more information.
2. Build a test mode. Refer to build_testmode in the *Encounter Test: Reference: Commands* for more information.
3. Build the standard fault model. Refer to build_faultmodel in the *Encounter Test: Reference: Commands* for more information.

Encounter Test can read in the following functional pattern languages:

- ASCII format of Encounter Test pattern data (TBDpatt). Refer to "Reading Encounter Test Pattern Data (TBDpatt)" on page 57 for more information.

- STIL test patterns. Refer to "Reading Standard Test Interface Language (STIL)" on page 58 for more information.
- Enhanced VCD (EVCD). Refer to "Reading Extended Value Change Dump (EVCD) File" on page 61 for more information.

For more information, refer to "Reading Test Data and Sequence Definitions" on page 55.

You can import this input into Encounter Test and use it as input to the simulator. This simulation capability provides a means of fault grading functional tests (or other manual patterns) and inserting them into the tester data.

Recommendation for Fault Grading Functional Tests

- If your design is fully synchronous, with well-analyzed clocking races, we recommend using the zero-delay mode of the General Purpose simulator. There are, however, many factors to consider, as described in "Functional Test Guidelines" on page 83. If you encounter pitfalls or conditions resulting from the described factors that are inappropriate, the unit-delay mode can be manipulated to work with some modeling effort.

  Fault simulation of functional tests is typically faster using zero-delay mode instead of unit-delay mode, however in dealing with a large design size, a large number of patterns required for a functional test, and a large number of faults remaining untested after scan chain tests, it may not be feasible to perform full fault simulation within a reasonable time. Based on these conditions, we recommend using the option for Random Fault Selection, `nrandsel` to fault grade your functional tests. This will allow you to obtain an estimate of the combined fault coverage of the scan-based ATPG tests plus the functional tests.

- As mentioned in the next section, importing functional patterns requires the pre-requisite of creating a test mode (`build_testmode`). To create the testmode, you need to define all the clocks that are pulsed within the functional patterns being imported. Encounter Test uses the leading edge of these clocks to divide the imported patterns into the following order of event sequences:

  - Stim Inputs
  - Pulse Clock
  - Observe Outputs

  You may use an existing testmode but note that any initialization patterns created for that test mode will be pre-pended to the functional patterns and that any fixed value (TI) inputs must not be toggled.

- If you want to use the functional patterns as part of your hardware test suite, ensure that all patterns are applied and all the results are observed through the device pins. At times, the functional patterns created by the designer use simulation software to directly load and/or observe internal logic states. However, you cannot use such patterns to test the hardware. Review your functional patterns to ensure that all stimuli are applied through the device pins, including any circuit initialization.

  As all patterns are applied and all the results are observed through the device pins, note that fault simulation and fault detection also happens under these restrictions. In other words, a fault buried deep within a large design must be stimulated from and propagate to a package pin. If the design contains many ranks of logical register, it may take many functional clocks to detect that fault. Because of run time and memory constraints, all fault simulators must decide how many clock cycles they will carry each and every fault before they drop it.

Reducing Fault Count

It is recommended to reduce the number of faults that are targetted in the functional pattern fault simulation as it significantly improves run time.

You can use either of the following ways to reduce the fault count:

- Run scanchain-based ATPG patterns and commit the test patterns before importing the functional patterns
- Create a special reduced fault model that targets the specified logic within the design

Creating and Committng ATPG Patterns

If your design has scan chains, it is strongly recommended to create and commit ATPG patterns before fault simulating the functional patterns:

1. Import the design net list and Verilog models

   ```
   build_model designsource=<netlist_name>.v   techlib=<tech_files>
   ```

2. Build a test mode that defines access to the scan chains:

   ```
   build_testmode testmode=FULLSCAN  assignfile=<name>
   ```

3. Build the standard fault model and assign faults to all the logic:

   ```
   build_faultmodel
   ```

4. Create and commit the scan chain confidence patterns (created by `create_scanchain_tests`) and normal ATPG patterns (created by `create_logic_tests`):

```
create_scanchain_tests testmode=FULLSCAN experiment=chain
commit_tests testmode=FULLSCAN inexperiment=chain
create_logic_tests testmode=FULLSCAN experiment=atpg
commit_tests inexperiment=atpg
```

Now when you import and fault simulate functional patterns, only the remaining faults will be fault simulated.

Note: While importing the functional patterns, you should decide on the test mode to be used. The FULLSCAN test mode can be used if the following criteria are met:

- The FULLSCAN test mode defined TI or TC input pins then the imported functional patterns cannot toggle these inputs. They must be at their TI/TC logic values.
- The functional clocks pulsed in the functional pattern set are defined in the FULLSCAN test mode. You can define more clocks (such as scan shift only) than are used by the functional test.
- The FULLSCAN test mode is compatible with the functional patterns;otherwise, build another test mode.

Creating Reduced Fault Model

You can optionally create a specific targeted fault list if you cannot reduce a large fault list (millions) through scanchain-based ATPG patterns. Usually, functional patterns target specific functions/regions of the design and thus targeting only that logic with the fault list helps improve run time.

```
prepare_fault_subset  -a  testmode=<name> experiment=<name> faultlist=<file>
```

`prepare_fault_subset` reads an input fault list and creates an empty uncommitted vector file. The `-a` parameter allows `prepare_fault_subset` to accept and use an existing experiment (created by `read_vectors`)

To create the input fault list file, you can prepare your own file or edit a fault list file created by Encounter Test.

```
build_faultmodel

report_faults testmode=<name> globalscope=no statusuntested=yes statustested=no
```

For more information, refer to "Building a Fault Model" in the *Encounter Test: Guide 4: Faults f*or more information.

EVCD Flow for Importing and Fault Simulating Functional Patterns

1. Build the model

   ```
   build_model designsource=<netlist_name>.v   techlib=<tech_files>
   ```

2. Build the testmode

   ```
   build_testmode testmode=FULLSCAN  assignfile=<name>
   ```

3. Build the faultmodel

   ```
   build_faultmodel
   ```

4. After that, you can either:

   ❑ Use the total fault list created by Encounter Test

   ❑ Create ATPG patterns. Refer to Creating and Committng ATPG Patterns on page 80 for more information.

   ❑ Prepare an alternate fault model. Refer to Creating Reduced Fault Model on page 81 for more information.

5. Import the EVCD patterns from the functional simulation.

   ```
   read_vectors testmode=FULLSCAN experiment=<name> language=evcd
   importfile=<name>.evcd
   ```

   The test mode can be FULLSCAN test mode or one specifically for importing these patterns.

   The experiment can be new or the one that was created using `prepare_fault_subset`

6. Fault simulate the functional patterns

   ```
   analyze_vectors testmode=FULLSCAN inexperiment=<experiment_name from above>
   experiment=<output_name>  measurelatch=sequential
   ```

   Specifying `measurelatch=sequentia`l propagates faults through measure (scan chain) latches/FF's. This specification is required if a scan chain test mode such as FULLSCAN is used.

7. Commit and write out the test patterns

   ```
   commit_tests testmode=FULLSCAN  inexperiment=<evcd_analyzed> force=yes
   write_vectors testmode=FULLSCAN  language=<stil|wgl|verilog>
   ```

   If you commit the functional patterns then you should write out all the committed patterns. (remember you may have created scan chain and ATPG patterns).

   If you do not commit the test patterns then specify `write_vectors inexperiment=<name>` where *<name>* is the name of the output experiment from `analyze_vectors`.

The following is an example of fault simulation using FULLSCAN testmode:

```
build_model \
designsource=./verilog_source/DLX_TOP.v,./verilog_source/DLX_CORE.v \
techlib=./verilog_lib,./verilog_lib/rf32x8.v,./verilog_lib/rf64x29.v,./
verilog_lib/tsmc13.v
build_testmode modedef=FULLSCAN testmode=FULLSCAN \
assignfile=./verilog_source/FULLSCAN.pinassign
verify_test_structures testmode=FULLSCAN
build_faultmodel includedynamic=no
create_scanchain_tests testmode=FULLSCAN experiment=atpg
create_logic_tests testmode=FULLSCAN experiment=atpg   append=yes
commit_tests testmode=FULLSCAN  inexperiment=atpg
read_vectors testmode=FULLSCAN experiment=func_ptrns \
language=evcd importfile=My_Functional_Patterns.EVCD
analyze_vectors testmode=FULLSCAN   \
inexperiment=func_ptrns  experiment=func_ptrns_out measurelatch=sequential
    ## If you do not want to add the functional patterns to the comitted
    ## ATPG patterns then omit
commit_tests testmode=FULLSCAN inexperiment= func_ptrns_out  force=yes
    ## If you omit the commit step you can reference the functional patterns
    ##  with "inexperiment=func_ptrns_out"  parameter.
write_vectors testmode=FULLSCAN language=<stil|wgl|verilog>
```

The following is an example of fault simulation using non-scan test mode and reduced fault count:

```
build_model \
designsource=./verilog_source/DLX_TOP.v,./verilog_source/DLX_CORE.v \
techlib=./verilog_lib,./verilog_lib/rf32x8.v,./verilog_lib/rf64x29.v,./
verilog_lib/tsmc13.v
build_testmode modedef=FULLSCAN testmode=MY_FUNC_TESTMODE \
assignfile=./verilog_source/MY_FUNC_TESTMODE.pinassign
verify_test_structures   testmode=MY_FUNC_TESTMODE
build_faultmodel includedynamic=no
read_vectors testmode= MY_FUNC_TESTMODE experiment=func_ptrns_subset \
language=evcd importfile=My_Functional_Patterns.EVCD
##  If you report_faults you can then edit the output to create a reduced list.
prepare_fault_subset   -a  testmode= MY_FUNC_TESTMODE       \
experiment=func_ptrns_subset      faultlist=My_EVCD_Fault_List
analyze_vectors testmode= MY_FUNC_TESTMODE        \
inexperiment=func_ptrns_subset  experiment=func_ptrns_subset_out  \
measurelatch=sequential contentionremove=no contentionreport=none
```

Functional Test Guidelines

In developing functional test cases, you must ensure that patterns can be simulated correctly so simulation results will match the chip hardware results on the tester. This requires that the functional test cases have these characteristics:

- Functional Test Case Timing

  A VERY slow clock rate must be used in order to allow each PI change to completely propagate through the logic so the functional simulator results can match the Static simulation results. A Static simulator assumes there is sufficient time between input change events to allow all design change activity to settle before the next input change event. Clock and Data input changes must be done at different, widely separated time intervals so no real-time races between clock and data occurs that would require accurate design delay information to compute the hardware results.

- Pulse Events

  There must be only a single Clock PI (possibly correlated with another PI) in any Pulse Event. When Encounter Test detects two or more pins pulsed in one event, the pins are simulated simultaneously. This function is called a multipulse.

- Clock PI Stim Events

  If any PI that can act as a clock to change a latch state (System Clock, TCK, TRST, System Reset, even a clock gate) is ever stimulated, this PI must be the only PI in this Stim Event. Data PI changes must be done in separate Stim Events to prevent clock versus data races.

  Refer to "1149.1 Boundary Controls" in the *Encounter Test: Guide 2: Testmodes* for additional information on these latch states.

  Encounter Test detects when more than one clock is stimulated or pulsed in any one event, and reports the clock nets that have been changed as a result. This function, called multiclock, is not performed for init procedures, scan chain tests, and macro tests. See "Tester_Loop" in the *Encounter Test: Reference: Test Pattern Formats* for a description of init procedure.

- Bi-Directional I/Os

  The functional test case should be written so as not to create three-state contentions. Typically this would require the product drivers go to Z before any BIDI PI Stims to 1/0, and BIDI PI Stims to Z should occur before the product drivers can go out of Z. These rules can be difficult to follow while developing a functional test case because the chip driver enable state must be known when working with BIDI Stims. Note that it is not impossible to have a simultaneous clock event to enable or disable a chip driver and a Stim Event on a BIDI PI (causing only a transient contention) because Clock and Data PI Events must be separated to achieve race-free simulation results.

Zero vs Unit Delay Simulation with General Purpose Simulator

The choice between zero-delay or unit-delay modes depends on several factors that require analysis and conclusion.

First is the issue of valid simulation results so that the tester does not fail a correctly designed and manufactured chip. If the ATPG patterns or functional tests have been written correctly, (refer to "Functional Test Guidelines" on page 83) and if there are no unpredictable races in the design (that is, no severe TSV violations or data hold-time violations), either a zero-delay or unit-delay mode of simulation can work if the models are developed correctly. A zero-delay simulation predicts the clock always wins the clock/data race. That is, for clock gating or data hold time races, zero-delay simulation will make calculations assuming that the clock goes Off before new data arrives. For a unit-delay simulation model to work, the inherent races among Flip-Flops, latches, and RAMs have to be accounted for in the models, taking clock distribution unit-delay skews into consideration. If gate unit delays are not accurate representations of clock distribution skew (for example, due to heavily loaded tests), you might have to modify the models to make the simulation work.

In addition to balancing the clock distribution unit delays, another way to make the unit-delay simulation model work properly is to insert gates (buffers) into the data inputs or outputs of all latch flip-flops or RAM models. This delays the propagation of new data signals feeding to opposite phase latches until their clocks can turn Off. The number of delay gates required in the latch/flip-flop/RAM models depends on how well-balanced is the clock distribution tree logic, how much logic already exists between latches/flip-flops/RAMs in the design, and whether you need to use the `xwidth` option to predict unknown values for some types of truly unpredictable design races. This unit-delay simulation technique using the `xwidth` option (*Delay Uncertainty Specific Sim option* on the graphical user interface) is a major reason to select unit-delay simulation over zero-delay simulation.

Unit-delay simulation mode allows for more accurate simulation in the presence of structures which present problems for zero-delay simulation provided the model is unit-delay accurate. For zero-delay simulation to function properly, the clock shaping logic must be identified to the simulation engine to prevent the suppression of pulses. As TSV is intended to identify such errors, unit delay is used for rules checking as the identification of TSV violations is required for accurate simulation . For an accurate unit delay model, zero delay simulation does not provide any advantage over unit delay simulation and will yield incorrect results in the presence of incorrectly modeled clock shaping networks.

Using the ZDLY Attribute

The `ZDLY=YES` property specifies that the block should be treated as zero delay when simulating with the General Purpose Simulator in unit delay mode. The `ZDLY` property can be specified only on primitives. This must be applied in the source verilog or library files.

Simulating OPC Logic

Cut points are used to hide from Encounter Test sections of logic that are intractable to test generation and other analysis functions. A common example of such intractable logic is an LBIST controller when the design is being processed in a test mode where the LBIST controller is active (running). Sequential clock generation macros also fall into this category. Logic that is hidden, or blocked, by cut points is referred to as OPC (on-product control or clock) logic. OPC logic is completely ignored by most Encounter Test application programs, and Encounter Test relies upon user-specified sequences (modeinit, custom scan protocol, user test sequences) to process designs that have cut points specified.

Confidence in Encounter Test data that is produced with cut points can be gained by use of the Verify On-Product Clock Sequences tool (see "On Product Clock Sequence Verification" in the *Encounter Test: Guide 3: Test Structures*) if a simulation model (instead of a black box) is available for the OPC logic. Additional confidence may be gained by resimulating the test data without the use of cut points. You may be able to export the test data and re-import it to another test mode that does not have cut points. However, in some cases this may not be possible. For example, if the cut points are necessary for the definition of scan strings, and you define a test mode without any cut points, the simulator would require an expanded form of the tests where each scan operation appears as a long sequence of scan clock pulses. You may or may not want to simulate the scan operations at this detailed level.

To avoid the necessity of defining additional test modes for simulation without cut points, and to cover the case where the detailed simulation of scan operations is not needed, there is a simulation option for ignoring cut points. Using this option, you can generate test data using cut points and resimulate the tests in the same test mode, ignoring the cut points. Specify `useppis=no` on the simulation (`analyze_vectors`) command line to use this feature.

A good verification of an LBIST controller can be achieved by the following steps:

1. Generate LBIST tests (using cut points).
2. Export Encounter Test Pattern data, specifying the options to expand scan operations and unravel LBIST test sequences.
3. Re-import the Encounter Test Pattern data to the same test mode.
4. Use the General Purpose simulator or the Good Machine Delay simulator (if you have a delay model) to simulate the tests. Be sure to turn on the option to Compare at Measure Commands. The simulator will let you know if there are any "miscompares" . between this simulation and the original test data generated by Step 1.

InfiniteX Simulation

InfiniteX simulation can be used to control how High Speed Scan Based Simulation simulates StimPI and StimLatch events when race conditions have been identified by Verify Test Structures. This mode of simulation employs a pessimistic approach to simulating race conditions by introducing Xs into the simulation of a StimPI or StimLatch event to pessimistically simulate the race condition. For example, if Verify Test Structures has identified certain clock-data conditions, the simulator introduces an X into StimPI events where the state of the PI is changing state (0->X->1 as opposed to 0->1 directly).

Invoke InfiniteX simulation by either specifying a value for the `infinitex` keyword in the command environment or by selecting High Speed Scan Based Simulation option to *Use pessimistic simulation* on the GUI. The default for *infinitex* is controlled by whether TSV was previously run. If TSV was not run, standard simulation is performed. If TSV was run, the default behavior for infiniteX simulation is dictated by the following conditions:

- InfiniteX simulation for latches is performed if:

  Verify Test Structures check Analyze potential clock signal races did not complete or was not run

  Or, Analyze potential clock signal races was run and produced message TSV-059.

  For additional information, refer to "Analyze Potential Clock Signal Races" in the *Encounter Test: Reference: Legacy Functions* and "Analyze Potential Clock Signal Races" in the *Encounter Test: Guide 3: Test Structures.*

- InfiniteX simulation for PIs is performed if:

  Verify Test Structures check Analyze test clocks' control of memory elements did not complete or was not run

  Or, Analyze clocks' control of memory elements was run and produced message TSV-008 or message TSV-310.

  For additional information, refer to "Analyze test clocks' control of memory elements" for LSSD in the *Encounter Test: Reference: Legacy Functions* and "Analyze test clocks' control of memory elements" for GSD in the *Encounter Test: Guide 3: Test Structures*.

Note: You can turn off the infiniteX simulation mode by specifying `infinitex=none` for the `create_logic_tests` command.

Pessimistic Simulation of Latches and Flip-Flops

Simulation of latches and flip-flops may be manipulated so that an X state appearing on a clock input will cause the output of the latch or flip-flop to be X, regardless of the state of the corresponding data pin and the current state of the memory element. This technique facilitates creation of test data that will match a simulator which calculates X for a flip-flop whose clock input is Xr.

Use the `latchsimulation=optimistic|pessimistic` keyword or option *Latch/Flip-flop output when clock at X* on GUI *Simulation* forms to specify whether to leave the output unchanged if it is the same as the data input state (`latchsimulation=optimistic`) or to always set output to X (`latchsimulation=pessimistic`), that is, to match the pessimistic flip-flop model.

All latches are simulated the same way in that all or simulated either optimistically or pessimistically based on the selected option.

The following are limitations associated with the pessimistic method of simulation:

- Simulation activity that occurs during Build Test Mode or Verify Test Structures is not affected. This may cause the following conditions:
  - Miscompares as a result of simulation mismatches during the test mode initialization sequence
  - Miscompares due to the failure (of the verification simulator) to recognize a data-muxed fixed-value latch whose clock is at X
- Simulation performed during GUI analysis does not support this option.
- Use of `latchsimulation=pessimistic` may reduce the fault coverage of a given set of tests.

Resolving Internal and External Logic Values Due to Termination

During Test Generation and Simulation, the termination values used on pins and internal nets are defined by the TDR and test simulation options.

For a detailed analysis of the interactions of the terms and parameters used to determine the termination values during Test Generation and Simulation, refer to "Termination Values" in the *Encounter Test: Guide 1: Models.*

Refer to the following for additional related information:

- "TESTER_DESCRIPTION_RULE" in the *Encounter Test: Guide 2: Testmodes*
- "TDR_DEFINITION" in the *Encounter Test: Guide 2: Testmodes*

Overall Simulation Restrictions

Restrictions of Test Simulation using High Speed Scan-Based Simulation are:

- Patterns cannot place a test inhibit (TI) function pin or a test constraint (TC) function pin away from its stability value.
- Patterns cannot turn more than one clock away from its stability value at the same time.

Restrictions of Test Simulation using General Purpose Simulation are:

- Transition fault simulation is not supported.
- Only the following `Test_Section` types are supported:
  - `logic`
  - `flush`
  - `scan`
  - `macro`
  - `driver_receiver`
- Supports only the test type of static.
- Pattern elimination for three-state contention or ineffective tests is not supported.
- Only test sections of Logic, IDDQ, `Driver_Receiver`, ICT (all types), and IOWRAP (all types) can be sorted by sequence.
- SimVision is not supported on Linux-64 bit platforms.
- The capability to allow simulation run time controls to be established from information stored in vector files generated by Encounter ATPG has a limitation that applies to the detection of syntax errors introduced by a user manually editing the `simOptions` keyed data in an Encounter Test vector file. Only invalid keyword values can be detected and reported. Erroneous keywords themselves are not detected as syntax errors but are simply ignored. It is generally recommended that "simOptions" keyed data not be modified.

5 Test Pattern Analysis

This chapter discusses the tools and concepts that can be used to analyze test patterns within Encounter Test.

The sections in the chapter include:

Test Pattern Analysis is the process of:

- Taking data from manual patterns or a test generation process;
- Simulating the patterns to determine if the expected responses match the actual values; and then
- Analyzing any miscompares to determine the cause of the problem.

Debugging Miscompares

Miscompares can be caused by various factors, some of which are given below:

- The model is logically incorrect
- The model does not account for the delay mode requirements of the simulator that generated the original data (or the one that is being used for the re-simulation). For example, if the cell description has some extra blocks to model some logical configuration, the unit delay simulator may find that signals do not get through the logic on time since it assigns a unit of delay to each primitive in the path. This might work better with a zero delay simulation.

- The input patterns are incorrect (either due to an error in the manually generated patterns; or due to a problem with the original simulation).

The following are some recommended considerations while analyzing the patterns:

1. The first thing to consider is where the "expected responses" come from.
   - ❑ If you are writing manual patterns, the expected responses are included in these patterns as an Expect_Event (see "TBDpatt and TBDseqPatt Format" in the *Encounter Test: Reference: Test Pattern Formats*).
   - ❑ If you are analyzing a TBD from an Encounter Test test generation process, the simulation portion of that process creates measures to indicate that the tester should measure a particular set of values on primary outputs and/or latches. When you resimulate the test data, the simulator compares its results with the previous results specified by measure events (Measure_PO and Scan_Unload).
2. The next thing to consider is the analysis technique you plan to use. This choice will influence the type of simulation to use for the re-simulation. Using the Test Simulation application, you may select a variety of simulation techniques. In addition, there is an interactive simulation technique specifically aimed at test pattern analysis (and diagnostics).

Some analysis techniques are given below:

1. Use the following process to view the miscomparing patterns in a waveform display:
   - ❑ Create a Watch List containing all the nets, pins, and/or blocks you want to include in the analysis. This Watch List can be created:
     - ❍ Interactively through the Encounter Test View Schematic window or the Watch List window (refer to Watch List in the *Encounter Test: Reference: GUI*)
     - ❍ Manually (refer to "Using Watch List" on page 93 for more information)
   - ❑ Select either the *General Purpose Simulation* or the *Good Machine Delay Simulation* options from the Simulate Vectors windows since they are the only ones that support this type of analysis. Set the appropriate simulation run parameters to specify the watch list you are using, the specific test data during which simulation should be saved for viewing, and any other desired options.
   - ❑ When the simulation ends, click either the *View Waveforms* icon or *Windows - Show - Waveforms*. The *Select Optional Transition File* dialog is displayed.
   - ❑ Select the transition file (`.trn`) for the test mode and experiment run at the time of simulation and click the *Open* button. A SimVision window is displayed with a set of signals that define the correspondence between the Encounter Test vector format and the waveform window timeline.

- ❑ Use SimVision facilities to select and display design signals of interest. Refer to "SimVision Overview" on page 99 for additional information.

2. To view the miscomparing patterns on the graphical view of the design, use *View Vectors* to select a test sequence analyze and invoke *View Circuit Values* to see the values displayed on the View Schematic window.

See "Viewing Test Data" on page 95 and "General View Circuit Values Information" in the *Encounter Test: Reference: GUI* for more information.

3. If you are satisfied after viewing the patterns and the model and making your own correlation of the data or, if you are limited to this choice by the data you are analyzing, then use the following process:

a. Select your choice of simulation.

b. View the resulting patterns using *View Vectors.*

c. View the logic on the *Encounter Test View Schematic* window and manually analyze the problem.

Using Watch List

You can use a watch list to create and customize a list of design objects for input to various Encounter Test applications using either the graphical interface or by manually specifying it. See "Watch List" in the *Encounter Test: Reference: GUI* for details.

To manually create a watch list, use the following syntax and semantics:

- Each line in the file is a statement, which can be of one of the following types:

  Model Object Statement

  Each Model Object Statement identifies one Encounter Test Model Object by type and name. It also allows you to specify an alias name for the Object. The syntax of the statement is the Model Object name optionally followed by the alias name. The Model Object name in the full name or short name format. The simple name should have a type specified before the name. The types are NET, PIN, or BLOCK. If a type is not specified, then net is assumed first, then pin, and then block. The alias name can be any combination of alpha-numeric or the following special characters: `!#$%&()+,-.:<=>?@[]^_`|~`.

  Note: If an entry is a BLOCK, Encounter Test will create a watch list for all ports/pins on that block. Refer to "expandnets" on page 94 for information on how to identify signals within a BLOCK.

  Facility Start Statement

  The Facility Start statement marks the beginning of a group of Model Objects that will be associated together. The statement syntax is the word FACILITY followed by white space followed by a facility name and ending with an open brace '}'. The facility name must start with an alphabetic character. It may contain alpha-numeric characters or underscores. It cannot end in an underscore nor have more than one underscore repeated. These are the same rules for identifiers in VHDL. The name will always be folded to upper case and is therefore case insensitive. When the same facility name appears more than once in the file, only one facility by that name is created containing all the Model Objects associated with the facility. It is an error to nest facilities. Here is an example Facility Start Statement:

  ```
  FACILITY TARGET  {
  ```

  Facility End Statement

  The Facility End Statement marks the end of the group of Model Objects defined by the previous Facility Start Statement. It is simply a close brace '}' character as the first character of a line. It is an error to have a Facility End without a corresponding Facility Start. It is an error to nest facilities.

  expandnets

  Use this syntax to direct Encounter Test to record net switching for all nets within the specified level or hierarchy. This facility must be named `expandnets`. For example,

  ```
  facility expandnets {
      block xyz
      block abc
  }
  ```

directs Encounter Test to record net switching activities for all nets within block `xyz` and block `abc`.

Comments

The characters '`//`' and '`/*`' begin a comment. Comments are allowed at the end of a statement or on a line by itself. Once a comment is started, it continues to the end of the line.

- An example of a watch list:

```
facility unit_a_buss_byte_0  {
    "unit_a.buss_out[7]"
    "unit_a.buss_out[6]"
    "unit_a.buss_out[5]"
    "unit_a.buss_out[4]"
    "unit_a.buss_out[3]"
    "unit_a.buss_out[2]"
    "unit_a.buss_out[1]"
    "unit_a.buss_out[0]"
}

facility expandnets {
    block unit_b
    block unit_c
}

"sys_clock"
"a_clock"
"b_clock"
"init_a.sys_clock"
```

There can be only one statement per line. Each statement must be on a single line.

Viewing Test Data

View the test pattern data created by Encounter Test by displaying the hierarchy of a Vectors or TBDseq file. To access this tool, select *Window - Show - Vectors* from the main window. Enter the information as described in "View Vectors Window" in the *Encounter Test: Reference: GUI* to display the patterns.

The Vectors hierarchy consists of the following entities:

```
Experiment
  Test_Section
    Tester_Loop
      Test_Procedure
        Test_Sequence
          Pattern
            Event
```

See "Encounter Test Vector Data" on page 48 for information about each level of the hierarchy.

You can perform a variety of tasks using the hierarchy. Its capabilities include:

- Moving interactively up and down a hierarchy.
- Expanding and collapsing levels of a Vectors file.
- Displaying a design view with simulation values.
- Displaying Details about specific objects in a Vectors file. This consists of information also available in a TBDpatt file, including data, an audit summary, and a listing of the contents of the Vectors file for the selected object.
- Creating a minimum Vectors file containing the minimal amount of data needed to perform simulation.
- Displaying Attributes of specific entities in a Vectors file.
- Displaying simulation results requiring special analysis (for example, miscompare data).
- Display of failing patterns based on the currently selected failures. See Reading Failures in the *Encounter Test: Guide 7: Diagnostics* for more information.

For additional information, refer to:

- “Viewing Test Data” in the *Encounter Test: Reference: GUI*.
- “Performing Test Pattern Analysis” on page 100.

Viewing Test Sequence Summary

The `report_vector_summary` program generates a summary of the test sequences during a test. The summary is in the form of a table containing event types for each test sequence and the information on the effectiveness in detecting faults.

The vector information is collected and summarized according to test sequences having similar stream of patterns, events, and clock used. The report considers each unique event stream as a template and displays it as a string of characters, with each character representing a different event type.

If you set the `vectors` option to `yes`, the report displays the short form of vectors in the summary. Set the `key` option to `yes` to print the description for each character used in the template string.

The following is a sample vector summary generated for a test pattern by using the `report_vector_summary` command with `key=yes` and `vectors=yes`:

```
Key for string format of Test Sequence report
-----------------------------------------------------
S - Stim PI
I - Stim PPI
s - Scan Load
a - Skewed Scan Load
P - Pulse Clock (P is followed by name of the clock is being pulsed)
C - Stim Clock
Q - Pulse PPI (P is followed by name of the clock is being pulsed)
M - Measure PO
m - Scan Unload
b - Skewed Scan Unload
c - Channel_Scan
D - Diagnostic Skewed Scan Unload
d - Diagnostic_Scan_Unload
F - Force
O - Product MISR Signature
X - Any other events


    #Sequences    %Det   #Static   #Dynamic   1st Seq  Template
    ----------    ----   -------   --------   ------   --------
#1           1    51.4       338          0        1   scan
#2          23    47.9       315          0        2   sSP(CLOCK)Mm
#3           4     0.6         4          0       25   sSS(CLOCK)MM(CLOCK)
#4           1     0.2         1          0       29   sSMm
#5           2     0.0         0          0        0    init

----Template #1---- scan
Scan_Load
Stim_PI
Stim_PI
Stim_PI
Pulse(CLOCK)
Measure_PO
Stim_PI
Pulse(CLOCK)
Measure_PO
Stim_PI
Pulse(CLOCK)
```

```
Measure_PO
Stim_PI
Pulse(CLOCK)
Measure_PO
Stim_PI
Pulse(CLOCK)
Measure_PO
Scan_Unload
----Template #2---- logic
Scan_Load
Stim_PI
Pulse(CLOCK)
Measure_PO
Scan_Unload
----Template #3---- logic
Scan_Load
Stim_PI
Stim_Clock(CLOCK)
Measure_PO
Stim_Clock(CLOCK)
Scan_Unload
----Template #4---- logic
Scan_Load
Stim_PI
Measure_PO
Scan_Unload
----Template #5---- init
Stim_PI
```

The sample report contains three sections:

- The first section is the key describing the format of the summary. It lists the characters used in short form of the vectors in the summary and the event type that each character represents.
- The second section is the summary of test vectors. The summary contains the following columns:
  - #Sequences - shows the number of sequences with event stream matching the template in the Template column
  - %Det - shows the percentage of test coverage obtained by the sequences

- ❑ #Static - shows the number of static faults detected by the sequences
- ❑ #Dynamic - shows the number of dynamic faults detected by the sequences
- ❑ 1st Seq - shows the location of the first matching sequence in the pattern set
- ❑ Template - shows the string describing the template. For the init, scan, and flush sequences, the summary lists the event type instead of the string. The Key section describes the characters representing the template string.

- The third section describes the vectors in each template listed in the summary.

Refer to report_vector_summary in the *Encounter Test: Reference: Commands* for more information.

SimVision Overview

You can use SimVision to analyze simulation results. Its capabilities include:

- Displaying digital waveforms
- Displaying values on nets during any time period in a simulation
- Arranging signals (move, copy, delete, repeat, show, hide) in the window for easy viewing, enabling better interpretation of results
- Multiple waveform graphs per window
- Multiple windows that allow you to organize data and to view multiple test data segments
- Annotating waves with text
- Performing logical and arithmetic operations on waveforms

A `.dsn` or `.trn` file can be input to SimVision for waveform analysis. The files can be created by either running Encounter Test test generation or simulation applications or from *View Vectors* by selecting a test section with scope data then clicking the custom mouse button to invoke *Create Waveforms for Pin Timing*. Refer to the description for the View Vectors "View Pull-down" in the *Encounter Test: Reference: GUI*.

To view Encounter Test patterns in SimVision, perform the following steps:

1. Click the *View Waveforms* icon on the main toolbar. Start SimVision by entering `simvision` at the command prompt.
2. Open a `.trn` file for the experiment on the resulting **Select Optional Transition File** dialog.

The following is an alternative method:

1. Invoke Simvision using this syntax: `simvision -input showall.sv`.

   `showall.sv` is a Simvision script file that contains:

   ❍ `database open` *`path`*/TBscope*`.testmode.experiment`*.trn

   ❍ `browser new`

   ❍ `waveform new`

   ❍ `set waves [browser find -name *]`

   ❍ `waveform add -signals $waves`

Refer to the *SimVision User Guide* for additional details.

Performing Test Pattern Analysis

For details on using the graphical interface for analyzing and viewing test data, refer to "Viewing Test Data" in the *Encounter Test: Reference: GUI*.

Note: You can perform Test Pattern Analysis only through graphical user interface.

Prerequisite Tasks

- Select an experiment.
- Create a Vectors file by running a test generation or simulation application.

Input Files

An existing Encounter Test experiment, sequence definition files, and failure data (if existing).

Output Files

If *Create Minimum Vectors* is selected on the Test View window, a new Vectors file is created as output.

Index