



# **Tessent Scan and ATPG User's Manual**

Software Version 2013.4

December 2013

---

**© 1999-2013 Mentor Graphics Corporation  
All rights reserved.**

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

**U.S. GOVERNMENT LICENSE RIGHTS:** The software and documentation were developed entirely at private expense and are commercial computer software and commercial computer software documentation within the meaning of the applicable acquisition regulations. Accordingly, pursuant to FAR 48 CFR 12.212 and DFARS 48 CFR 227.7202, use, duplication and disclosure by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in the license agreement provided with the software, except for provisions which are contrary to applicable mandatory federal laws.

**TRADEMARKS:** The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the owner of the Mark, as applicable. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: [www.mentor.com/trademarks](http://www.mentor.com/trademarks).

Mentor Graphics Corporation  
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777  
Telephone: 503.685.7000  
Toll-Free Telephone: 800.592.2210  
Website: [www.mentor.com](http://www.mentor.com)  
SupportNet: [supportnet.mentor.com/](http://supportnet.mentor.com/)

Send Feedback on Documentation: [supportnet.mentor.com/doc\\_feedback\\_form](http://supportnet.mentor.com/doc_feedback_form)

# Table of Contents

---

## Chapter 1

<b>Overview</b>	<b>17</b>
What is Design-for-Test?	17
DFT Strategies	17
Top-Down Design Flow with DFT	18

## Chapter 2

<b>Understanding Scan and ATPG Basics</b>	<b>21</b>
Understanding Scan Design	21
Internal Scan Circuitry	21
Scan Design Overview	22
Understanding Scan	24
Understanding Wrapper Chains	25
Understanding Test Points	27
Test Structure Insertion with Tessent Scan	29
Understanding ATPG	30
The ATPG Process	30
Mentor Graphics ATPG Applications	31
Scan Sequential ATPG with the ATPG Tool	31
Understanding Test Types and Fault Models	32
Test Types	33
Fault Modeling	35
User-Defined Fault Modeling	42
Multiple Detect	51
Embedded Multiple Detect	52
Fault Detection	54
Fault Classes	55
Testability Calculations	65

## Chapter 3

<b>Understanding Common Tool Terminology and Concepts</b>	<b>67</b>
Scan Terminology	67
Scan Cells	67
Master Element	69
Slave Element	69
Shadow Element	69
Copy Element	70
Extra Element	71
Scan Chains	71
Scan Groups	72
Scan Clocks	73
Scan Architectures	73

Mux-DFF .....	73
Clocked-Scan .....	74
LSSD.....	74
Test Procedure Files.....	75
Model Flattening .....	76
Understanding Design Object Naming.....	76
The Flattening Process .....	76
Simulation Primitives of the Flattened Model .....	78
Learning Analysis .....	81
Equivalence Relationships .....	81
Logic Behavior .....	81
Implied Relationships .....	82
Forbidden Relationships .....	83
Dominance Relationships .....	83
ATPG Design Rules Checking .....	83
General Rules Checking .....	84
Procedure Rules Checking .....	84
Bus Mutual Exclusivity Analysis .....	84
Scan Chain Tracing.....	86
Shadow Latch Identification.....	86
Data Rules Checking.....	87
Transparent Latch Identification.....	87
Clock Rules Checking.....	87
RAM Rules Checking.....	87
Bus Keeper Analysis.....	87
Extra Rules Checking .....	88
Scanability Rules Checking .....	88
Constrained/Forbidden/Block Value Calculations .....	89
 <b>Chapter 4</b>	
<b>Understanding Testability Issues .....</b>	<b>91</b>
Synchronous Circuitry .....	92
Synchronous Design Techniques .....	92
Asynchronous Circuitry.....	93
Scannability Checking .....	93
Scannability Checking of Latches .....	94
Support for Special Testability Cases .....	94
Feedback Loops .....	94
Structural Combinational Loops and Loop-Cutting Methods .....	94
Structural Sequential Loops and Handling .....	99
Redundant Logic.....	100
Asynchronous Sets and Resets .....	100
Gated Clocks.....	101
Tri-State Devices .....	102
Non-Scan Cell Handling.....	103
Clock Dividers .....	107
Pulse Generators .....	108
JTAG-Based Circuits .....	110

## Table of Contents

---

Testing RAM and ROM .....	110
Incomplete Designs. ....	117
 <b>Chapter 5</b>	
<b>Inserting Internal Scan</b>	
<b>and Test Circuitry.....</b>	<b>119</b>
The Tessent Scan Process Flow.....	119
Tessent Scan Inputs and Outputs .....	121
Test Structures Supported by Tessent Scan .....	122
Invoking Tessent Scan .....	123
Preparing for Test Structure Insertion .....	123
Selecting the Scan Methodology .....	123
Defining Scan Cell and Scan Output Mapping.....	123
Enabling Test Logic Insertion.....	124
Specifying Clock Signals .....	127
Specifying Existing Scan Information .....	128
Handling Existing Boundary Scan Circuitry .....	130
Changing the System Mode (Running Rules Checking) .....	130
Setting Up a Basic Scan Insertion Run.....	131
Setting Up for Wrapper Chain Identification .....	131
Manually Specifying Control and Observe Points .....	133
Manually Including and Excluding Cells for Scan.....	134
Reporting Scannability Information .....	136
Automatic Recognition of Existing Shift Registers .....	138
The Identification Process.....	140
Reporting Identification Information .....	141
Inserting Test Structures .....	141
Setting Up for Internal Scan Insertion .....	141
Naming Scan Input and Output Ports .....	141
Buffering Test Pins .....	144
Running the Insertion Process .....	144
Merging Scan Chains with Different Shift Clocks.....	145
Saving the New Design and ATPG Setup.....	148
Writing the Netlist.....	148
Writing the Test Procedure File and Dofile for ATPG.....	148
Running Rules Checking on the New Design.....	148
Exiting Tessent Scan.....	149
Inserting Scan Block-by-Block .....	149
Verilog Flow Example .....	149
 <b>Chapter 6</b>	
<b>Generating Test Patterns .....</b>	<b>153</b>
ATPG Basic Tool Flow .....	154
ATPG Tool Inputs and Outputs .....	156
Understanding the ATPG Method .....	157
Performing Basic Operations.....	163
Invoking the Application .....	163
Setting the System Mode .....	163

Setting Up Design and Tool Behavior . . . . .	163
Setting Up the Circuit Behavior . . . . .	163
Setting Up Tool Behavior . . . . .	168
Defining the Scan Data . . . . .	173
Checking Rules and Debugging Rules Violations . . . . .	175
Running Good/Fault Simulation on Existing Patterns . . . . .	175
Fault Simulation . . . . .	176
Good-Machine Simulation . . . . .	178
Running Random Pattern Simulation . . . . .	178
Changing to the Fault System Mode . . . . .	179
Adding the Faults List . . . . .	179
Running the Simulation . . . . .	179
Setting Up the Fault Information for ATPG . . . . .	179
Changing to the Analysis System Mode . . . . .	180
Setting the Fault Type . . . . .	180
Creating the Faults List . . . . .	180
Adding Faults to an Existing List . . . . .	180
Loading Faults from an External List . . . . .	181
Writing Faults to an External File . . . . .	181
Setting the Fault Sampling Percentage . . . . .	181
Setting the Fault Mode . . . . .	182
Setting the Possible-Detect Credit . . . . .	182
Performing ATPG . . . . .	183
Setting Up for ATPG . . . . .	183
Creating Patterns with Default Settings . . . . .	191
Approaches for Improving ATPG Efficiency . . . . .	191
Saving the Test Patterns . . . . .	194
Low-Power ATPG . . . . .	194
Low-Power Capture . . . . .	194
Low-Power Shift . . . . .	195
Setting up Low-Power ATPG . . . . .	195
Creating an IDDQ Test Set . . . . .	197
Generating an IDDQ Test Set . . . . .	198
Specifying Leakage Current Checks . . . . .	199
Net Pair Identification with Calibre for Bridge Fault Test Patterns . . . . .	200
Four-Way Dominant Fault Model . . . . .	200
Top-level Bridging ATPG . . . . .	200
Incremental Multi-Fault ATPG . . . . .	201
The Bridge Parameters File . . . . .	203
Creating a Delay Test Set . . . . .	205
Creating a Transition Delay Test Set . . . . .	206
Transition Fault Detection . . . . .	207
Basic Procedure for Generating a Transition Test Set . . . . .	211
Timing for Transition Delay Tests . . . . .	211
Preventing Pattern Failures Due to Timing Exception Paths . . . . .	214
Creating a Path Delay Test Set . . . . .	220
At-Speed Test Using Named Capture Procedures . . . . .	230
Mux-DFF Example . . . . .	238
Support for Internal Clock Control . . . . .	244

## Table of Contents

---

Generating Test Patterns for Different Fault Models and Fault Grading . . . . .	248
Using Timing-Aware ATPG . . . . .	250
Generating Patterns for a Boundary Scan Circuit . . . . .	260
Dofile and Explanation . . . . .	260
TAP Controller State Machine . . . . .	261
Test Procedure File and Explanation . . . . .	262
Using the MacroTest Capability . . . . .	267
The MacroTest Process Flow . . . . .	268
Qualifying Macros for MacroTest . . . . .	270
When to Use MacroTest . . . . .	271
Defining the Macro Boundary . . . . .	273
Defining Test Values . . . . .	277
Recommendations for Using MacroTest . . . . .	279
MacroTest Examples . . . . .	281
Verifying Test Patterns . . . . .	286
Simulating the Design with Timing . . . . .	286
Debugging Simulation Mismatches . . . . .	287
When, Where, and How Many Mismatches? . . . . .	289
DRC Issues . . . . .	289
Shadow Cells . . . . .	290
Library Problems . . . . .	290
Timing Violations . . . . .	290
Analyzing the Simulation Data . . . . .	291
Analyzing Simulation Mismatches . . . . .	293
Understanding the Simulation Mismatch Analysis Flow . . . . .	293
Automatically Analyzing Simulation Mismatches . . . . .	294
Manually Analyzing Simulation Mismatches . . . . .	297
Analyzing Patterns . . . . .	301
Checking for Clock-Skew Problems with Mux-DFF Designs . . . . .	302

## Chapter 7

<b>Multiprocessing for ATPG and Simulation . . . . .</b>	<b>303</b>
Introduction . . . . .	303
Definition of Multiprocessing Terms . . . . .	303
Using Multiprocessing to Reduce Runtime . . . . .	304
Multiprocessing Requirements . . . . .	304
Procedures for Multiprocessing . . . . .	306
Overview Procedure for Using Multiprocessing . . . . .	306
Troubleshooting SSH Environment and Passphrase Errors . . . . .	307
Disabling Multithreading Functionality . . . . .	309
Adding Threads to the Master . . . . .	309
Adding Processors in Manual Mode . . . . .	310
Adding Processors in Grid Mode . . . . .	312
Adding Processors to the LSF Grid . . . . .	313
Deleting Processors Added in Manual Mode . . . . .	313
Deleting Processors Added in Grid Mode . . . . .	314

<b>Chapter 8</b>	
<b>Scan Pattern Retargeting</b>	<b>315</b>
Overview	315
Tools and Licensing	316
Scan Pattern Retargeting Process	317
Core-Level Pattern Generation	320
Scan Pattern Retargeting of Test Patterns	322
Test Procedure Retargeting	323
Scan Pattern Retargeting Without a Netlist	324
Generating Patterns at the Core Level	328
Retargeting Patterns in Internal Mode	329
Retargeting Patterns Without a Top-Level Netlist	330
Generating Patterns in External Mode	331
Retargeting Example	332
Limitations	336
<b>Chapter 9</b>	
<b>Test Pattern Formatting and Timing</b>	<b>337</b>
Test Pattern Timing Overview	338
Timing Terminology	339
General Timing Issues	339
Generating a Procedure File	340
Defining and Modifying Timeplates	341
Saving Timing Patterns	344
Features of the Formatter	344
Pattern Formatting Issues	344
Saving Patterns in Basic Test Data Formats	347
Saving in ASIC Vendor Data Formats	352
<b>Chapter 10</b>	
<b>Test Pattern File Formats</b>	<b>355</b>
ASCII File Format	355
Header_Data	355
Setup_Data	355
Functional_Chain_Test	358
Scan_Test	359
Scan_Cell	361
BIST Pattern File Format	363
Setup_Data	363
Scan_Test	364
<b>Chapter 11</b>	
<b>Power-Aware DRC and ATPG</b>	<b>367</b>
Power-Aware Overview	367
Assumptions and Limitations	368
Multiple Power Mode Test Flow	368
Power-Aware ATPG for Traditional Fault Models	369
CPF and UPF Parser	370



## Table of Contents

---

Power-Aware ATPG Procedure . . . . .	372
Power-Aware Flow Examples . . . . .	373
Example 1 . . . . .	373
Example 2 . . . . .	374
<b>Chapter 12</b>	
<b>Testing Low-Power Designs . . . . .</b>	<b>377</b>
Low-Power Testing Overview . . . . .	378
Assumptions and Limitations . . . . .	378
Low-Power CPF/UPF Parameters . . . . .	378
Test Insertion . . . . .	380
Low-Power Test Flow . . . . .	380
Scan Insertion with Tessent Scan . . . . .	381
Power-Aware Design Rule Checks . . . . .	386
Low-Power DRCs . . . . .	386
Low-Power DRC Troubleshooting . . . . .	386
Power State-Aware ATPG . . . . .	388
Power Domain Testing . . . . .	388
Low-Power Cell Testing . . . . .	390
<b>Chapter 13</b>	
<b>MTFI File Format . . . . .</b>	<b>391</b>
Introduction . . . . .	391
MTFI Syntax . . . . .	392
MTFI File Example . . . . .	394
MTFI Features . . . . .	394
Support of Fault Classes and Sub-Classes . . . . .	395
Support of Stuck and Transition Fault Information . . . . .	397
Support of N-Detect Values . . . . .	397
Support of Different Fault Types in the Same File . . . . .	399
Support for Hierarchical Fault Accounting . . . . .	399
<b>Chapter 14</b>	
<b>Graybox Overview . . . . .</b>	<b>403</b>
Introduction . . . . .	403
What is a Graybox? . . . . .	403
Graybox Process Overview . . . . .	405
Example dofile for Creating a Graybox Netlist . . . . .	406
Generating Graybox Netlist for EDT Logic Inserted Blocks . . . . .	407
<b>Appendix A</b>	
<b>Clock Gaters . . . . .</b>	<b>409</b>
Basic Clock Gater Cell . . . . .	409
Two Types of Embedding . . . . .	409
Ideal Case (Type-A) . . . . .	410
Potential DRC Violator (Type-B) . . . . .	411
Cascaded Clock Gaters . . . . .	413
Understanding a Level-2 Clock Gater . . . . .	413

---

Example Combinations of Cascaded Clock Gaters .....	414
Summary .....	414
<b>Appendix B</b>	
<b>Debugging State Stability.....</b>	<b>417</b>
Understanding State Stability .....	417
Displaying the State Stability Data.....	417
State Stability Data Format.....	418
State Stability Examples .....	419
<b>Appendix C</b>	
<b>Running Tessent Shell as a Batch Job .....</b>	<b>441</b>
Commands and Variables for the dofile .....	441
Command Line Options.....	442
Scheduling a Batch Job for Execution Later.....	443
<b>Appendix D</b>	
<b>Getting Help .....</b>	<b>445</b>
Documentation.....	445
Mentor Graphics Support.....	445
<b>Index</b>	
<b>Third-Party Information</b>	
<b>End-User License Agreement</b>	

# List of Figures

---

Figure 1-1. Top-Down Design Flow Tasks and Products .....	20
Figure 2-1. Test Concepts .....	21
Figure 2-2. Design Before and After Adding Scan .....	23
Figure 2-3. Scan Representation .....	24
Figure 2-4. Example of Partitioned Design .....	26
Figure 2-5. Wrapper Chains Added to Partition A .....	27
Figure 2-6. Uncontrollable and Unobservable Circuitry .....	28
Figure 2-7. Testability Benefits from Test Point Circuitry .....	28
Figure 2-8. Manufacturing Defect Space for a Design .....	32
Figure 2-9. Internal Faulting Example. ....	35
Figure 2-10. Single Stuck-At Faults for AND Gate .....	37
Figure 2-11. IDDQ Fault Testing. ....	39
Figure 2-12. Transition Fault Detection Process .....	40
Figure 2-13. Fault Detection Process. ....	54
Figure 2-14. Path Sensitization Example. ....	55
Figure 2-15. Example of “Unused” Fault in Circuitry. ....	56
Figure 2-16. Example of “Tied” Fault in Circuitry .....	57
Figure 2-17. Example of “Blocked” Fault in Circuitry .....	58
Figure 2-18. Example of “Redundant” Fault in Circuitry .....	58
Figure 2-19. Fault Class Hierarchy .....	61
Figure 3-1. Common Tool Concepts .....	67
Figure 3-2. Generic Scan Cell .....	68
Figure 3-3. Generic Mux-DFF Scan Cell Implementation .....	68
Figure 3-4. LSSD Master/Slave Element Example .....	69
Figure 3-5. Dependently-clocked Mux-DFF/Shadow Element Example .....	70
Figure 3-6. Independently-clocked Mux-DFF/Shadow Element Example .....	70
Figure 3-7. Mux-DFF/Copy Element Example .....	71
Figure 3-8. Generic Scan Chain. ....	71
Figure 3-9. Generic Scan Group .....	72
Figure 3-10. Scan Clocks Example .....	73
Figure 3-11. Mux-DFF Replacement. ....	74
Figure 3-12. Clocked-Scan Replacement .....	74
Figure 3-13. LSSD Replacement .....	75
Figure 3-14. Design Before Flattening .....	77
Figure 3-15. Design After Flattening. ....	77
Figure 3-16. 2x1 MUX Example .....	79
Figure 3-17. LA, DFF Example. ....	79
Figure 3-18. TSD, TSH Example .....	79
Figure 3-19. PBUS, SWBUS Example .....	80
Figure 3-20. Equivalence Relationship Example. ....	81

Figure 3-21. Example of Learned Logic Behavior . . . . .	82
Figure 3-22. Example of Implied Relationship Learning . . . . .	82
Figure 3-23. Forbidden Relationship Example . . . . .	83
Figure 3-24. Dominance Relationship Example . . . . .	83
Figure 3-25. Bus Contention Example. . . . .	85
Figure 3-26. Bus Contention Analysis. . . . .	85
Figure 3-27. Simulation Model with Bus Keeper . . . . .	88
Figure 3-28. Constrained Values in Circuitry . . . . .	89
Figure 3-29. Forbidden Values in Circuitry. . . . .	89
Figure 3-30. Blocked Values in Circuitry . . . . .	90
Figure 4-1. Testability Issues. . . . .	91
Figure 4-2. Structural Combinational Loop Example . . . . .	94
Figure 4-3. Loop Naturally-Blocked by Constant Value. . . . .	95
Figure 4-4. Cutting Constant Value Loops . . . . .	95
Figure 4-5. Cutting Single Multiple-Fanout Loops . . . . .	96
Figure 4-6. Loop Candidate for Duplication . . . . .	96
Figure 4-7. TIE-X Insertion Simulation Pessimism . . . . .	97
Figure 4-8. Cutting Loops by Gate Duplication . . . . .	97
Figure 4-9. Cutting Coupling Loops . . . . .	98
Figure 4-10. Sequential Feedback Loop . . . . .	99
Figure 4-11. Test Logic Added to Control Asynchronous Reset. . . . .	101
Figure 4-12. Test Logic Added to Control Gated Clock . . . . .	102
Figure 4-13. Tri-state Bus Contention . . . . .	103
Figure 4-14. Requirement for Combinationally Transparent Latches. . . . .	104
Figure 4-15. Example of Sequential Transparency . . . . .	105
Figure 4-16. Clocked Sequential Scan Pattern Events. . . . .	106
Figure 4-17. Clock Divider . . . . .	108
Figure 4-18. Example Pulse Generator Circuitry . . . . .	108
Figure 4-19. Long Path Input Gate Must Go to Gates of the Same Type . . . . .	109
Figure 4-20. Design with Embedded RAM. . . . .	111
Figure 4-21. RAM Sequential Example. . . . .	113
Figure 5-1. Internal Scan Insertion Procedure . . . . .	119
Figure 5-2. Basic Scan Insertion Flow with Tessent Scan. . . . .	120
Figure 5-3. The Inputs and Outputs of Tessent Scan. . . . .	121
Figure 5-4. Test Logic Insertion . . . . .	125
Figure 5-5. Example Report from report_dft_check Command . . . . .	137
Figure 5-6. Lockup Cell Insertion . . . . .	147
Figure 5-7. Hierarchical Design Prior to Scan. . . . .	149
Figure 5-8. Final Scan-Inserted Design . . . . .	151
Figure 6-1. Test Generation Procedure . . . . .	153
Figure 6-2. Overview of ATPG Tool Usage . . . . .	154
Figure 6-3. ATPG Tool Inputs and Outputs . . . . .	156
Figure 6-4. Clock-PO Circuitry . . . . .	159
Figure 6-5. Data Capture Handling Example . . . . .	171
Figure 6-6. Efficient ATPG Flow . . . . .	183

## List of Figures

---

Figure 6-7. Circuitry with Natural “Select” Functionality. . . . .	185
Figure 6-8. Simulation Frames . . . . .	187
Figure 6-9. Waveform Modeling for DFFs and Latches . . . . .	188
Figure 6-10. Flow for Creating a Delay Test Set. . . . .	206
Figure 6-11. Transition Delay . . . . .	207
Figure 6-12. Transition Launch and Capture Events. . . . .	208
Figure 6-13. Events in a Broadside Pattern . . . . .	208
Figure 6-14. Basic Broadside Timing . . . . .	209
Figure 6-15. Events in a Launch Off Shift Pattern . . . . .	209
Figure 6-16. Basic Launch Off Shift Timing. . . . .	209
Figure 6-17. Broadside Timing Example. . . . .	212
Figure 6-18. Launch Off Shift (Skewed) Timing Example . . . . .	213
Figure 6-19. Multicycle Path Example . . . . .	214
Figure 6-20. Setup Time and Hold Time Violations . . . . .	215
Figure 6-21. Across Clock Domain Hold Time Violation. . . . .	216
Figure 6-22. Effect Cone of a Non-specific False Path Definition . . . . .	220
Figure 6-23. Path Delay Launch and Capture Events . . . . .	221
Figure 6-24. Robust Detection Example . . . . .	222
Figure 6-25. Non-robust Detection Example. . . . .	223
Figure 6-26. Functional Detection Example . . . . .	224
Figure 6-27. Example Use of Transition_condition Statement. . . . .	226
Figure 6-28. Example of Ambiguous Path Definition. . . . .	228
Figure 6-29. Example of Ambiguous Path Edges . . . . .	228
Figure 6-30. On-chip Clock Generation . . . . .	232
Figure 6-31. PLL-Generated Clock and Control Signals. . . . .	233
Figure 6-32. Cycles Merged for ATPG . . . . .	235
Figure 6-33. Cycles Expanded for ATPG . . . . .	236
Figure 6-34. Mux-DFF Example Design. . . . .	239
Figure 6-35. Mux-DFF Broadside Timing, Cell to Cell . . . . .	239
Figure 6-36. Broadside Timing, Clock Pulses in Non-adjacent cycles . . . . .	241
Figure 6-37. Mux-DFF Cell to PO Timing . . . . .	242
Figure 6-38. Mux-DFF PI to Cell Timing . . . . .	243
Figure 6-39. Simplified Per-Cycle Clock Control Model . . . . .	244
Figure 6-40. Simplified Sequence Clock Control Model . . . . .	245
Figure 6-41. Clock Control Capture Cycles . . . . .	246
Figure 6-42. Timing Slack. . . . .	250
Figure 6-43. Testcase 1 Logic . . . . .	254
Figure 6-44. Dofile. . . . .	255
Figure 6-45. Reports . . . . .	255
Figure 6-46. Glitch Detection Case . . . . .	259
Figure 6-47. State Diagram of TAP Controller Circuitry . . . . .	262
Figure 6-48. Conceptual View of MacroTest . . . . .	267
Figure 6-49. Basic Scan Pattern Creation Flow with MacroTest . . . . .	269
Figure 6-50. Mismatch Diagnosis Guidelines . . . . .	288
Figure 6-51. Simulation Transcript . . . . .	292

Figure 6-52. Automatic Simulation Mismatch Analysis Flow .....	295
Figure 6-53. Manual Simulation Mismatch Analysis Flow.....	298
Figure 6-54. Clock-Skew Example .....	302
Figure 8-1. Core-level Pattern Generation Process .....	317
Figure 8-2. Scan Pattern Retargeting - Internal Mode.....	318
Figure 8-3. Pattern Generation - External Mode .....	319
Figure 8-4. Merging Patterns for Multiple Views of a Core at the Top Level .....	326
Figure 8-5. Merging Patterns for Top Level and Core at the Top Level.....	327
Figure 8-6. Retargeting of Core-level Patterns .....	332
Figure 9-1. Defining Basic Timing Process Flow .....	337
Figure 10-1. Example Scan Circuit .....	362
Figure 12-1. Scan Chain Insertion by Power Domain.....	380
Figure 12-2. Input Wrapper Cells and Power Domains.....	384
Figure 12-3. Output Wrapper Cells and Power Domains .....	385
Figure 13-1. Using MTFI with Hierarchical Designs .....	401
Figure 14-1. Full Hierarchical Block Netlist .....	404
Figure 14-2. Graybox Version of Block Netlist.....	405
Figure A-1. Basic Clock Gater Cell.....	409
Figure A-2. Two Types of Embedding for the Basic Clock Gater .....	410
Figure A-3. Type-B Clock Gater Causes Tracing Failure.....	412
Figure A-4. Sample EDT Test Procedure Waveforms .....	412
Figure A-5. Two-level Clock Gating.....	413
Figure B-1. Design Used in State Stability Examples.....	421
Figure B-2. Typical Initialization Problem .....	423
Figure B-3. Three-bit Shift Register (Excerpted from Figure D-1).....	425
Figure B-4. Initialization with a Non-Shift Clock .....	426
Figure B-5. Clocking ff20 with a Pulse Generator .....	428

## List of Tables

---

Table 2-1. Test Type/Fault Model Relationship .....	35
Table 2-2. UDFM Keywords .....	43
Table 2-3. Fault Sub-classes .....	62
Table 4-1. RAM/ROM Commands .....	115
Table 5-1. Scan Direction and Active Values .....	126
Table 6-1. ATPG Constraint Conditions .....	186
Table 6-2. Bridge Definition File Keywords .....	203
Table 6-3. Testcase 2 Data .....	251
Table 11-1. Power Data Commands Directly Related to ATPG and DRC .....	370
Table 11-2. Example Design With Four Power domains and Power Modes .....	373
Table 12-1. Power Data Commands Directly Related to ATPG .....	379
Table 13-1. MTFI Command Summary .....	391
Table 14-1. Graybox Command Summary .....	403
Table A-1. Clock Gater Summary .....	414





# Chapter 1

## Overview

---

This manual gives an overview of ASIC/IC Design-for-Test (DFT) strategies and shows the use of Mentor Graphics ASIC/IC DFT products as part of typical DFT design processes. This manual discusses the Tessent products that use Scan and ATPG technology:

- Tessent Scan, which is Tessent Shell operating in “dft -scan” context
- Tessent FastScan, which is Tessent Shell operating in “patterns -scan” context
- Tessent TestKompress (with EDT off), which is Tessent Shell operating in “patterns -scan” context

This manual uses the term “ATPG tool” to refer to any of the following products: Tessent FastScan, Tessent TestKompress (with EDT off), or Tessent Shell operating in the “patterns -scan” context.

For information about contexts in Tessent Shell, refer to “[Contexts and System Modes](#)” in the *Tessent Shell User’s Manual*. For information about any of the commands mentioned in this manual, refer to the *Tessent Shell Reference Manual*.

## What is Design-for-Test?

*Testability* is a design attribute that measures how easy it is to create a program to comprehensively test a manufactured design’s quality. Traditionally, design and test processes were kept separate, with test considered only at the end of the design cycle. But in contemporary design flows, test merges with design much earlier in the process, creating what is called a *design-for-test (DFT)* process flow. Testable circuitry is both *controllable* and *observable*. In a testable design, setting specific values on the primary inputs results in values on the primary outputs that indicate whether or not the internal circuitry works properly. To ensure maximum design testability, designers must employ special DFT techniques at specific stages in the development process.

## DFT Strategies

At the highest level, there are two main approaches to DFT: *ad hoc* and *structured*. The following subsections discuss these DFT strategies.

### Ad Hoc DFT

Ad hoc DFT implies using good design practices to enhance a design's testability without making major changes to the design style. Some ad hoc techniques include:

- Minimizing redundant logic
- Minimizing asynchronous logic
- Isolating clocks from the logic
- Adding internal control and observation points

Using these practices throughout the design process improves the overall testability of your design. However, using structured DFT techniques with the Mentor Graphics DFT tools yields far greater improvement. Thus, the remainder of this document concentrates on structured DFT techniques.

## Structured DFT

Structured DFT provides a more systematic and automatic approach to enhancing design testability. Structured DFT's goal is to increase the controllability and observability of a circuit. Various methods exist for accomplishing this. The most common is the *scan design* technique, which modifies the internal sequential circuitry of the design. You can also use the Built-in Self-Test (BIST) method, which inserts a device's testing function within the device itself. Another method is *boundary scan*, which increases board testability by adding circuitry to a chip. "[Understanding Scan and ATPG Basics](#)" describes these methods in detail.

## Top-Down Design Flow with DFT

[Figure 1-1](#) shows the basic steps and the Mentor Graphics tools you would use during a typical ASIC top-down design flow. This document discusses those steps shown in grey; it also mentions certain aspects of other design steps, where applicable. This flow is just a general description of a top-down design process flow using a structured DFT strategy.

As [Figure 1-1](#) shows, the first task in any design flow is creating the initial RTL-level design, through whatever means you choose. In the Mentor Graphics environment, you may choose to create a high-level Verilog description using ModelSim® or a schematic using Design Architect®. You then verify the design's functionality by performing a functional simulation, using ModelSim or another vendor's Verilog simulator.

At this point in the flow you are ready to insert internal scan circuitry into your design using Tessent Scan. You may then want to re-verify the timing because you added scan circuitry. Once you are sure the design is functioning as desired, you can generate test patterns. You can use the ATPG tool to generate a test pattern set in the appropriate format.

Now you should verify that the design and patterns still function correctly with the proper timing information applied. You can use ModelSim or some other simulator to achieve this goal. You may then have to perform a few additional steps required by your ASIC vendor before handing the design off for manufacture and testing.

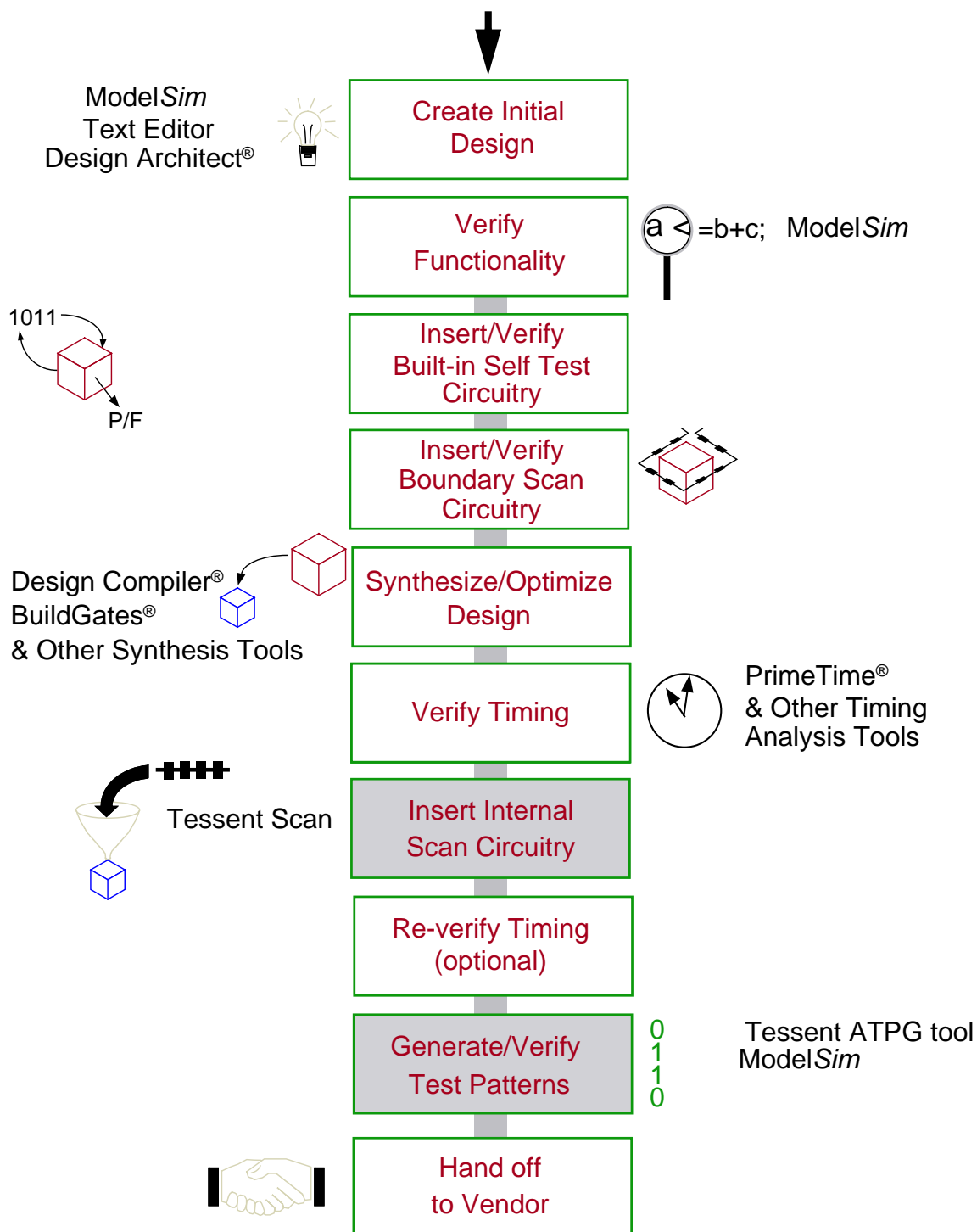
**Note**



It is important for you to check with your vendor early on in your design process for specific requirements and restrictions that may affect your DFT strategies. For example, the vendor's test equipment may only be able to handle single scan chains (see [page 21](#)), have memory limitations, or have special timing requirements that affect the way you generate scan circuitry and test patterns.

---

Figure 1-1. Top-Down Design Flow Tasks and Products



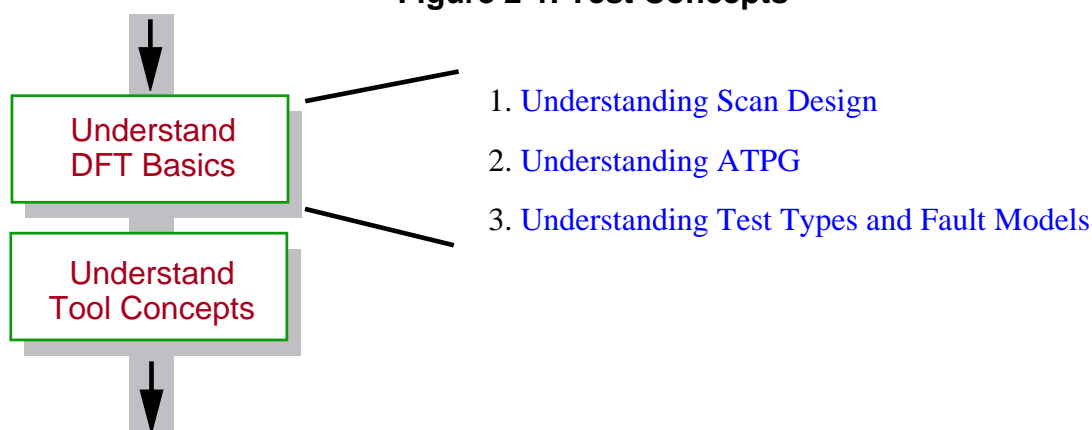
# Chapter 2

## Understanding Scan and ATPG Basics

---

Before you begin the testing process, you must first have an understanding of certain testing concepts. Once you understand these concepts, you can determine the best test strategy for your particular design. [Figure 2-1](#) shows the concepts this section discusses.

**Figure 2-1. Test Concepts**



Scan circuitry facilitates test generation and can reduce external tester usage. There are two main types of scan circuitry: internal scan and boundary scan. *Internal scan* (also referred to as *scan design*) is the internal modification of your design's circuitry to increase its testability. A detailed discussion of internal scan begins in "[Internal Scan Circuitry](#)."

While scan design modifies circuitry within the original design, *boundary scan* adds scan circuitry around the periphery of the design to make internal circuitry on a chip accessible via a standard board interface. The added circuitry enhances board testability of the chip, the chip I/O pads, and the interconnections of the chip to other board circuitry.

## Understanding Scan Design

This section gives you an overview of scan design and how it works. For more detailed information on the concepts presented in this section, refer to the documentation references cited earlier.

### Internal Scan Circuitry

As previously discussed, *internal scan* (or *scan design*) is the internal modification of your design's circuitry to increase its testability. Scan techniques are discussed on [page 24](#).

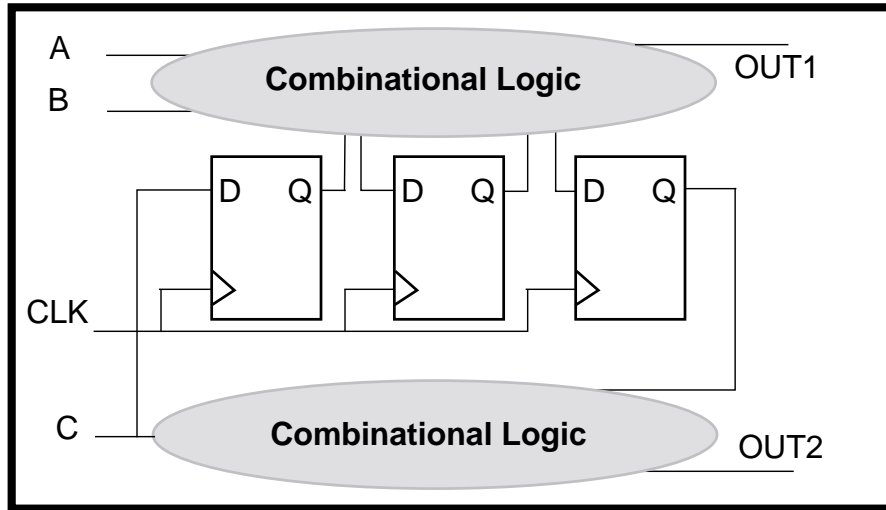
## Scan Design Overview

The goal of scan design is to make a difficult-to-test sequential circuit behave (during the testing process) like an easier-to-test combinational circuit. Achieving this goal involves replacing sequential elements with scannable sequential elements (scan cells) and then stitching the scan cells together into scan registers, or scan chains. You can then use these serially-connected scan cells to shift data in and out when the design is in scan mode.

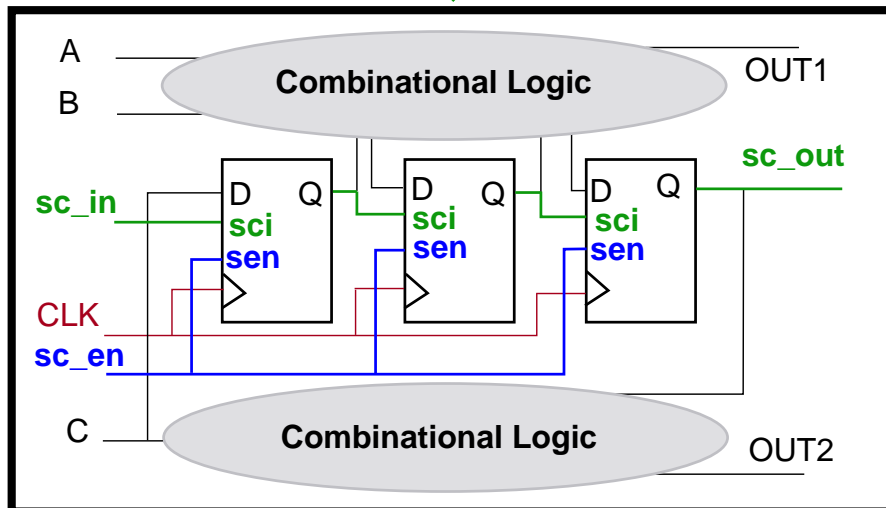
The design shown in [Figure 2-2](#) contains both combinational and sequential portions. Before adding scan, the design had three inputs, A, B, and C, and two outputs, OUT1 and OUT2. This “Before Scan” version is difficult to initialize to a known state, making it difficult to both control the internal circuitry and observe its behavior using the primary inputs and outputs of the design.

**Figure 2-2. Design Before and After Adding Scan**

**Before Scan**



**After Scan**



After adding scan circuitry, the design has two additional inputs, `sc_in` and `sc_en`, and one additional output, `sc_out`. Scan memory elements replace the original memory elements so that when shifting is enabled (the `sc_en` line is active), scan data is read in from the `sc_in` line.

The operating procedure of the scan circuitry is as follows:

1. Enable the scan operation to allow shifting (to initialize scan cells).
2. After loading the scan cells, hold the scan clocks off and then apply stimulus to the primary inputs.

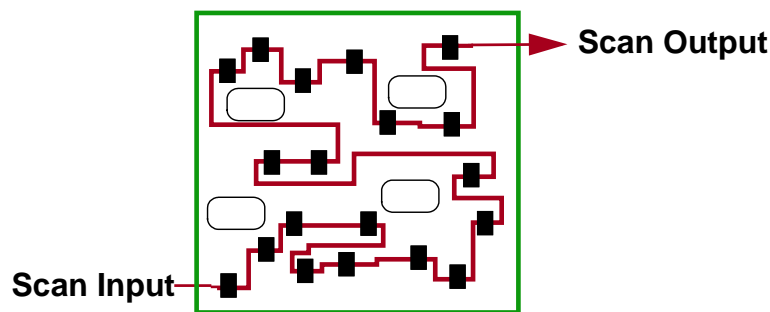
3. Measure the outputs.
4. Pulse the clock to capture new values into scan cells.
5. Enable the scan operation to unload and measure the captured values while simultaneously loading in new values via the shifting procedure (as in step 1).

## Understanding Scan

*Scan* is a scan design methodology that replaces all memory elements in the design with their scannable equivalents and then stitches (connects) them into scan chains. The idea is to control and observe the values in all the design's storage elements so you can make the sequential circuit's test generation and fault simulation tasks as simple as those of a combinational circuit.

Figure 2-3 gives a symbolic representation of a scan design.

**Figure 2-3. Scan Representation**



The black rectangles in Figure 2-3 represent scan elements. The line connecting them is the scan path. Because this is a scan design, all storage elements were converted and connected in the scan path. The rounded boxes represent combinational portions of the circuit.

For information on implementing a scan strategy for your design, refer to “[Test Structures Supported by Tessent Scan](#)” on page 122.

## Scan Benefits

The following are benefits of employing a scan strategy:

- **Highly automated process.**  
Using scan insertion tools, the process for inserting scan circuitry into a design is highly-automated, thus requiring very little manual effort.
- **Highly-effective, predictable method.**  
Scan design is a highly-effective, well-understood, and well-accepted method for generating high test coverage for your design.



- **Ease of use.**  
Using scan methodology, you can insert both scan circuitry and run ATPG without the aid of a test engineer.
- **Assured quality.**  
Scan assures quality because parts containing such circuitry can be tested thoroughly during chip manufacture. If your end products are going to be used in market segments that demand high quality, such as aircraft or medical electronics—and you can afford the added circuitry—then you should take advantage of the scan methodology.

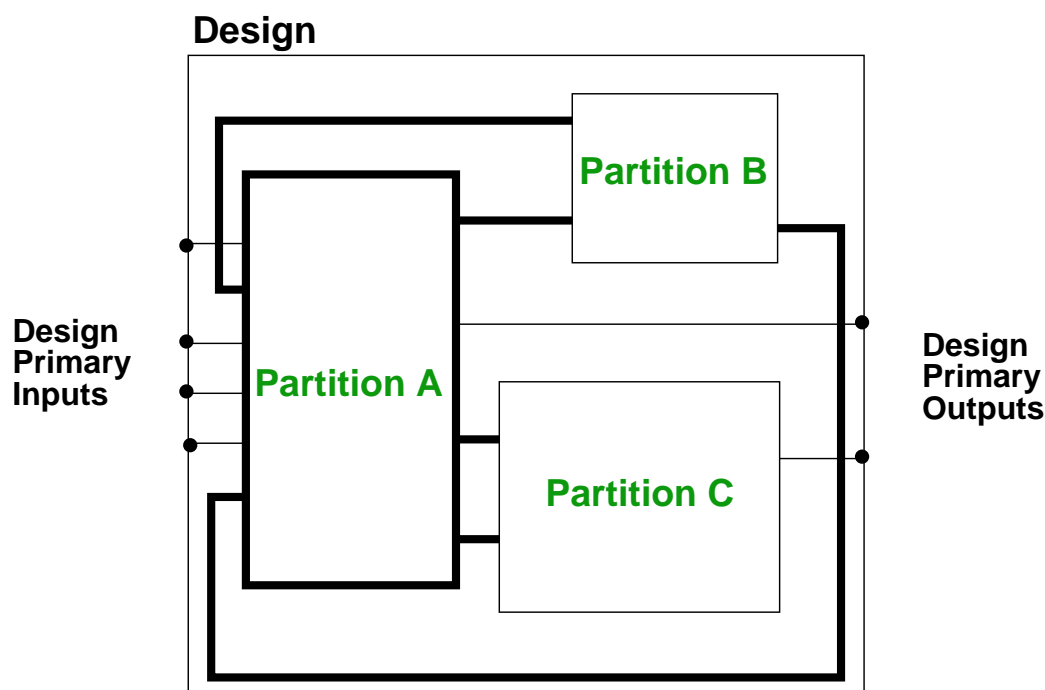
## Understanding Wrapper Chains

The ATPG process on very large, complex designs can often be unpredictable. This problem is especially true for large sequential or partial scan designs. To reduce this unpredictability, a number of hierarchical techniques for test structure insertion and test generation are beginning to emerge. *Creating wrapper chains* is one of these techniques. Large designs that are split into a number of design blocks benefit most from wrapper chains.

Wrapper chains add controllability and observability to the design via a hierarchical wrapper scan chain. A wrapper chain is a series of scan cells connected around the boundary of a design partition that is accessible at the design level. The wrapper chain improves both test coverage and run time by converting sequential elements to scan cells at inputs (outputs) that have low controllability (observability) from outside the block.

The architecture of wrapper chains is illustrated in the following two figures. [Figure 2-4](#) shows a design with three partitions, A, B, and C.

**Figure 2-4. Example of Partitioned Design**



The bold lines in [Figure 2-4](#) indicate inputs and outputs of partition A that are not directly controllable or observable from the design level. Because these lines are not directly accessible at the design level, the circuitry controlled by these pins can cause testability problems for the design.

[Figure 2-5](#) shows how adding wrapper chain structures to partition A increases the controllability and observability (testability) of partition A from the design level.

---

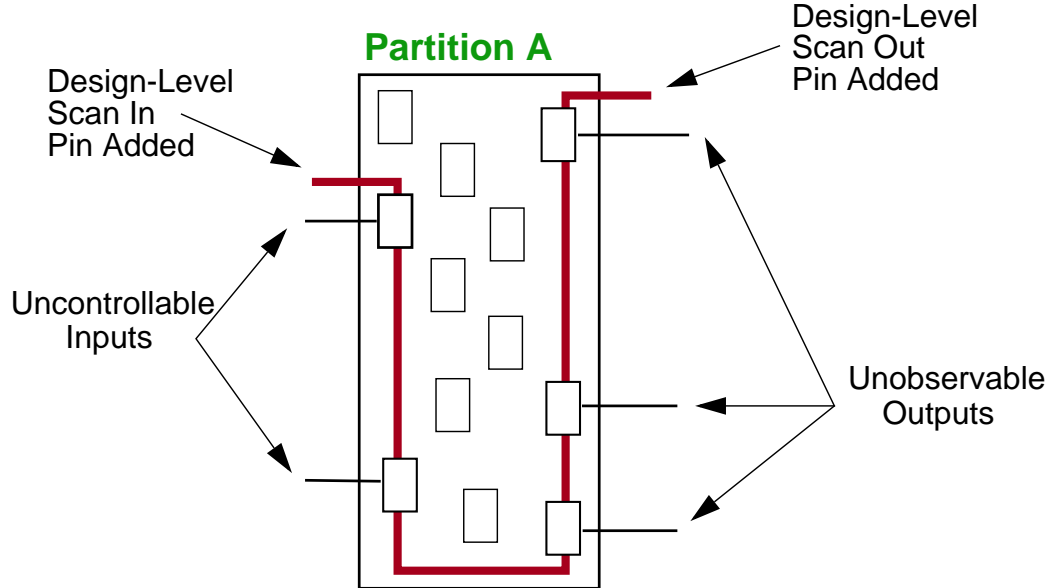
**Note**



Only the first elements that are directly connected to the uncontrollable (unobservable) primary inputs (primary outputs) become part of the wrapper chain.

---

**Figure 2-5. Wrapper Chains Added to Partition A**



The wrapper chain consists of two types of elements: sequential elements connected directly to uncontrolled primary inputs of the partition, and sequential elements connected directly to unobservable (or masked) outputs of the partition. The partition also acquires two design-level pins, scan in and scan out, to give direct access to the previously uncontrollable or unobservable circuitry.

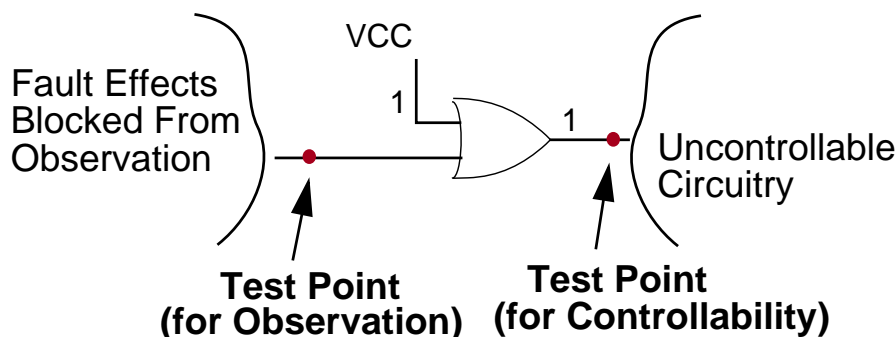
You can also use wrapper chains in conjunction with scan structures. Sequential elements not eligible for wrapper chains become candidates for internal scan.

For information on implementing a scan strategy for your design, refer to [“Setting Up for Wrapper Chain Identification”](#) on page 131.

## Understanding Test Points

A design can contain a number of points that are difficult to control or observe. Sometimes this is true even in designs containing scan. By adding special circuitry at certain locations called test points, you can increase the testability of the design. For example, [Figure 2-6](#) shows a portion of circuitry with a controllability and observability problem.

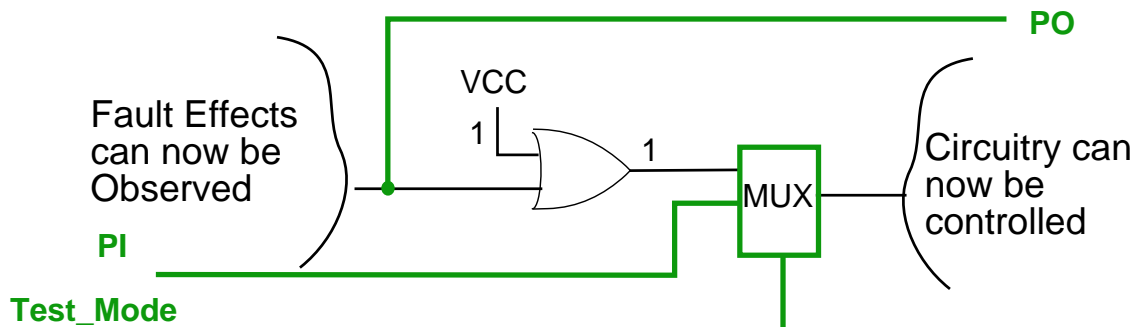
**Figure 2-6. Uncontrollable and Unobservable Circuitry**



In this example, one input of an OR gate is tied to a 1. This blocks the ability to propagate through this path any fault effects in circuitry feeding the other input. Thus, the other input must become a test point to improve observation. The tied input also causes a constant 1 at the output of the OR gate. This means any circuitry downstream from that output is uncontrollable. The pin at the output of the gate becomes a test point to improve controllability. Once identification of these points occurs, added circuitry can improve the controllability and observability problems.

Figure 2-7 shows circuitry added at these test points.

**Figure 2-7. Testability Benefits from Test Point Circuitry**



At the observability test point, an added primary output provides direct observation of the signal value. At the controllability test point, an added MUX controlled by a test\_mode signal and primary input controls the value fed to the associated circuitry.

This is just one example of how test point circuitry can increase design testability. Refer to [“Manually Specifying Control and Observe Points”](#) on page 133 for information on identifying test points and inserting test point circuitry.

Test point circuitry is similar to test logic circuitry. For more information on test logic, refer to [“Enabling Test Logic Insertion”](#) on page 124.

## Test Structure Insertion with Tessent Scan

Tessent Scan, the Mentor Graphics internal scan synthesis tool, can identify sequential elements for conversion to scan cells and then stitch those scan cells into scan chains.

Tessent Scan contains the following features:

- **Verilog format.**  
Reads and writes a Verilog gate-level netlist.
- **Multiple scan types.**  
Supports insertion of three different scan types, or methodologies: mux-DFF, clocked-scan, and LSSD.
- **Multiple test structures.**  
Supports identification and insertion of scan (both sequential ATPG-based and scan sequential procedure-based), wrapper chains, and test points.
- **Scannability checking.**  
Provides powerful scannability checking/reporting capabilities for sequential elements in the design.
- **Design rules checking.**  
Performs design rules checking to ensure scan setup and operation are correct—before scan is actually inserted. This rules checking also guarantees that the scan insertion done by Tessent Scan produces results that function properly in an ATPG tool.
- **Interface to ATPG tools.**  
Automatically generates information for the ATPG tools on how to operate the scan circuitry Tessent Scan creates.
- **Optimal partial scan selection.**  
Provides optimal partial scan analysis and insertion capabilities.
- **Flexible scan configurations.**  
Allows flexibility in the scan stitching process, such as stitching scan cells in fixed or random order, creating either single- or multiple-scan chains, and using multiple clocks on a single-scan chain.
- **Test logic.**  
Provides capabilities for inserting test logic circuitry on uncontrollable set, reset, clock, tri-state enable, and RAM read/write control lines.
- **User specified pins.**  
Allows user-specified pin names for test and other I/O pins.
- **Multiple model levels.**  
Handles gate-level, as well as gate/transistor-level models.

- **Online help.**

Provides online help for every command along with online manuals.

For information about using Tessent Scan to insert scan circuitry into your design, refer to [“Inserting Internal Scan and Test Circuitry”](#) on page 119.

## Understanding ATPG

ATPG stands for Automatic Test Pattern Generation. *Test patterns*, sometimes called *test vectors*, are sets of 1s and 0s placed on primary input pins during the manufacturing test process to determine if the chip is functioning properly. When the test pattern is applied, the Automatic Test Equipment (ATE) determines if the circuit is free from manufacturing defects by comparing the fault-free output—which is also contained in the test pattern—with the actual output measured by the ATE.

## The ATPG Process

The goal of ATPG is to create a set of patterns that achieves a given test coverage, where test coverage is the total percentage of testable faults the pattern set actually detects. (For a more precise definition of test coverage, see [page 65](#).) ATPG consists of two main steps: 1) generating patterns and, 2) performing fault simulation to determine which faults the patterns detect. Mentor Graphics ATPG tools automate these two steps into a single operation or ATPG process. This ATPG process results in patterns you can then save with added tester-specific formatting that enables a tester to load the pattern data into a chip’s scan cells and otherwise apply the patterns correctly.

This section only discusses the generation of test patterns. [“Fault Classes”](#) on page 55 discusses the fault simulation process. The two most typical methods for pattern generation are random and deterministic. Additionally, the ATPG tools can fault simulate patterns from an external set and place those patterns detecting faults in a test set. The following subsections discuss each of these methods.

## Random Pattern Test Generation

An ATPG tool uses *random pattern test generation* when it produces a number of random patterns and identifies only those patterns that detect faults. It then stores only those patterns in the test pattern set. The type of fault simulation used in random pattern test generation cannot replace deterministic test generation because it can never identify redundant faults. Nor can it create test patterns for faults that have a very low probability of detection. However, it can be useful on testable faults terminated by deterministic test generation. As an initial step, using a small number of random patterns can improve ATPG performance.

## Deterministic Test Pattern Generation

An ATPG tool uses *deterministic test pattern generation* when it creates a test pattern intended to detect a given fault. The procedure is to pick a fault from the fault list, create a pattern to detect the fault, fault simulate the pattern, and check to make sure the pattern detects the fault.

More specifically, the tool assigns a set of values to control points that force the fault site to the state opposite the fault-free state, so there is a detectable difference between the fault value and the fault-free value. The tool must then find a way to propagate this difference to a point where it can observe the fault effect. To satisfy the conditions necessary to create a test pattern, the test generation process makes intelligent decisions on how best to place a desired value on a gate. If a conflict prevents the placing of those values on the gate, the tool refines those decisions as it attempts to find a successful test pattern.

If the tool exhausts all possible choices without finding a successful test pattern, it must perform further analysis before classifying the fault. Faults requiring this analysis include redundant, ATPG-untestable, and possible-detected-untestable categories (see [page 55](#) for more information on fault classes). Identifying these fault types is an important by-product of deterministic test generation and is critical to achieving high test coverage. For example, if a fault is proven redundant, the tool may safely mark it as untestable. Otherwise, it is classified as a potentially detectable fault and counts as an untested fault when calculating test coverage.

## External Pattern Test Generation

An ATPG tool uses *external pattern test generation* when the preliminary source of ATPG is a pre-existing set of external patterns. The tool analyzes this external pattern set to determine which patterns detect faults from the active fault list. It then places these effective patterns into an internal test pattern set. The “generated patterns”, in this case, include the patterns (selected from the external set) that can efficiently obtain the highest test coverage for the design.

## Mentor Graphics ATPG Applications

Tessent FastScan and Tessent TestKompress (EDT off) are the Mentor Graphics scan sequential ATPG products, and are the same thing as Tessent Shell operating in “patterns -scan” context. This manual refers to all three of these products as the “ATPG tool.” The following subsections introduce the features of Tessent Shell operating in “patterns -scan” context. “[Generating Test Patterns](#)” on page 153” discusses the ATPG products in greater detail.

## Scan Sequential ATPG with the ATPG Tool

Mentor Graphics ATPG products include the following features:

- Deliver high performance ATPG for designs with structured scan
- Reduce run time with no effect on coverage or pattern count using distributed ATPG

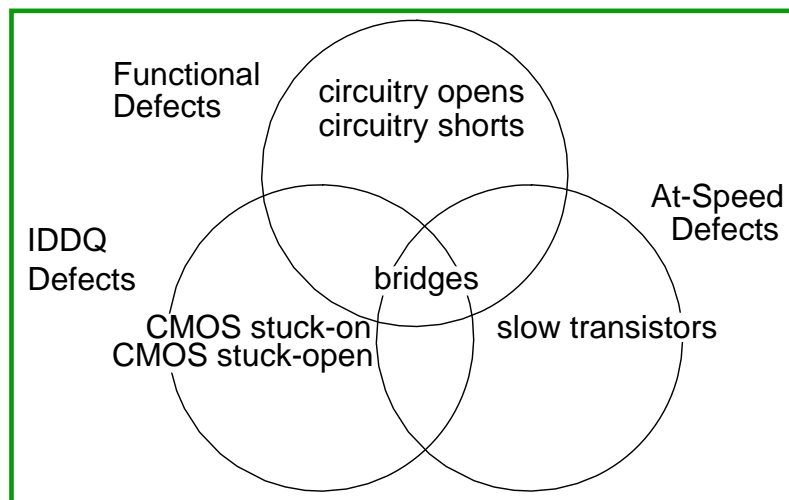
- Maximize test coverage by minimizing the impact of X's caused by false and multicycle paths
- Identify testability problems early using comprehensive design rule checking
- Reduce test validation time with automatic simulation mismatch debugging
- Ensure shorter time to market with integration into all design flows and foundry support
- Have extensive fault model support, including stuck-at, IDDQ, transition, path delay and bridge
- Have on-chip PLL support for accurate at-speed test
- Automate testing small embedded memories and cores with scan
- Supported in the Tessent SoCScan hierarchical silicon test environment

For more information about ATPG functionality, refer to the [Tessent Shell Reference Manual](#).

## Understanding Test Types and Fault Models

A *manufacturing defect* is a physical problem that occurs during the manufacturing process, causing device malfunctions of some kind. The purpose of test generation is to create a set of test patterns that detect as many manufacturing defects as possible. [Figure 2-8](#) gives an example of possible device defect types.

**Figure 2-8. Manufacturing Defect Space for a Design**



Each of these defects has an associated detection strategy. The following subsection discusses the three main types of test strategies.



## Test Types

Figure 2-8 shows three main categories of defects and their associated test types: *functional*, *IDDQ*, and *at-speed*. Functional testing checks the logic levels of output pins for a “0” and “1” response. IDDQ testing measures the current going through the circuit devices. At-speed testing checks the amount of time it takes for a device to change logic states. The following subsections discuss each of these test types in more detail.

### Functional Test

Functional test continues to be the most widely-accepted test type. Functional test typically consists of user-generated test patterns, simulation patterns, and ATPG patterns.

Functional testing uses logic levels at the device input pins to detect the most common manufacturing process-caused problem, static defects (for example, open, short, stuck-on, and stuck-open conditions). Functional testing applies a pattern of 1s and 0s to the input pins of a circuit and then measures the logical results at the output pins. In general, a defect produces a logical value at the outputs different from the expected output value.

### IDDQ Test

IDDQ testing measures quiescent power supply current rather than pin voltage, detecting device failures not easily detected by functional testing—such as CMOS transistor stuck-on faults or adjacent bridging faults. IDDQ testing equipment applies a set of patterns to the design, lets the current settle, then measures for excessive current draw. Devices that draw excessive current may have internal manufacturing defects.

Because IDDQ tests do not have to propagate values to output pins, the set of test vectors for detecting and measuring a high percentage of faults may be very compact. The ATPG tool efficiently creates this compact test vector set.

In addition, IDDQ testing detects some static faults, tests reliability, and reduces the number of required burn-in tests. You can increase your overall test coverage by augmenting functional testing with IDDQ testing.

IDDQ test generation methodologies break down into these categories:

- **Every-vector**  
This methodology monitors the power-supply current for every vector in a functional or stuck-at fault test set. Unfortunately, this method is relatively slow—on the order of 10-100 milliseconds per measurement—making it impractical in a manufacturing environment.
- **Supplemental**  
This methodology bypasses the timing limitation by using a smaller set of IDDQ measurement test vectors (typically generated automatically) to augment the existing test set.

Three test vector types serve to further classify IDDQ test methodologies:

- **Ideal**  
Ideal IDDQ test vectors produce a nearly zero quiescent power supply current during testing of a good device. Most methodologies expect such a result.
- **Non-ideal**  
Non-ideal IDDQ test vectors produce a small, deterministic quiescent power supply current in a good circuit.
- **Illegal**  
If the test vector cannot produce an accurate current component estimate for a good device, it is an illegal IDDQ test vector. You should never perform IDDQ testing with illegal IDDQ test vectors.

IDDQ testing classifies CMOS circuits based on the quiescent-current-producing circuitry contained inside as follows:

- **Fully static**  
Fully static CMOS circuits consume close to zero IDDQ current for all circuit states. Such circuits do not have pullup or pull-down resistors, and there can be one and only one active driver at a time in tri-state buses. For such circuits, you can use any vector for ideal IDDQ current measurement.
- **Resistive**  
Resistive CMOS circuits can have pullup/pull-down resistors and tristate buses that generate high IDDQ current in a good circuit.
- **Dynamic**  
Dynamic CMOS circuits have macros (library cells or library primitives) that generate high IDDQ current in some states. Diffused RAM macros belong to this category.

Some designs have a low current mode, which makes the circuit behave like a fully static circuit. This behavior makes it easier to generate ideal IDDQ tests for these circuits.

The ATPG tool currently supports only the ideal IDDQ test methodology for fully static, resistive, and some dynamic CMOS circuits. The tools can also perform IDDQ checks during ATPG to ensure the vectors they produce meet the ideal requirements. For information on creating IDDQ test sets, refer to [“Creating an IDDQ Test Set”](#).

## At-Speed Test

Timing failures can occur when a circuit operates correctly at a slow clock rate, and then fails when run at the normal system speed. Delay variations exist in the chip due to statistical variations in the manufacturing process, resulting in defects such as partially conducting transistors and resistive bridges.

The purpose of at-speed testing is to detect these types of problems. At-speed testing runs the test patterns through the circuit at the normal system clock speed.

## Fault Modeling

*Fault models* are a means of abstractly representing manufacturing defects in the logical model of your design. Each type of testing—functional, IDDQ, and at-speed—targets a different set of defects.

## Test Types and Associated Fault Models

[Table 2-1](#) associates test types, fault models, and the types of manufacturing defects targeted for detection.

**Table 2-1. Test Type/Fault Model Relationship**

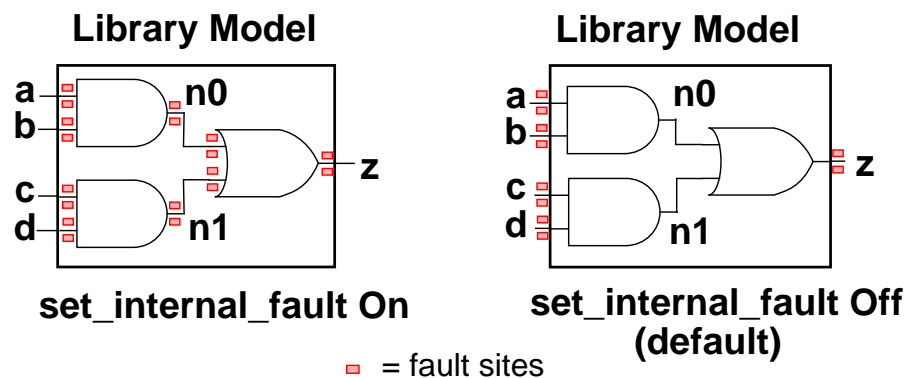
Test Type	Fault Model	Examples of Mfg. Defects Detected
Functional	Stuck-at, toggle	Some opens/shorts in circuit interconnections
IDDQ	Pseudo stuck-at	CMOS transistor stuck-on/some stuck-open conditions, resistive bridging faults, partially conducting transistors
At-speed	Transition, path delay	Partially conducting transistors, resistive bridges

## Fault Locations

By default, faults reside at the inputs and outputs of library models. However, faults can instead reside at the inputs and outputs of gates within library models if you turn internal faulting on.

[Figure 2-9](#) shows the fault sites for both cases.

**Figure 2-9. Internal Faulting Example**



To locate a fault site, you need a unique, hierarchical instance pathname plus the pin name.

You can also use Verilog ``celldefine` statements to extend cell boundaries beyond library models. Using this technique has several implications:

- The default fault population changes. By default, all fault locations are at library boundary pins. However, when the library boundary moves from the ATPG library level up to the ``celldefine` level, the fault locations and fault population change as a result.
- The flattened model can be different because the logic inside ``celldefine` module might be optimized to reduce the flattened model size.
- Hierarchical instance/pin names inside ``celldefine` module are not treated as legal instance/pin names.

## Fault Collapsing

A circuit can contain a significant number of faults that behave identically to other faults. That is, the test may identify a fault, but may not be able to distinguish it from another fault. In this case, the faults are said to be equivalent, and the fault identification process reduces the faults to one equivalent fault in a process known as *fault collapsing*. For performance reasons, early in the fault identification process the ATPG tool singles out a member of the set of equivalent faults and use this “representative” fault in subsequent algorithms. Also for performance reasons, these applications only evaluate the one equivalent fault, or *collapsed fault*, during fault simulation and test pattern generation. The tools retain information on both collapsed and uncollapsed faults, however, so they can still make fault reports and test coverage calculations.

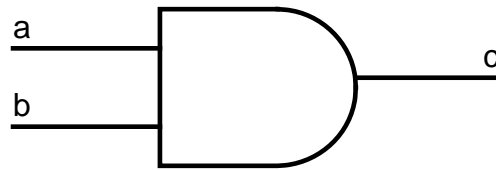
## Supported Fault Model Types

The ATPG tool supports stuck-at, pseudo stuck-at, toggle, and transition fault models. In addition to these, the tool supports static bridge and path delay fault models. The following subsections discuss these supported fault models, along with their fault collapsing rules.

### Functional Testing and the Stuck-At Fault Model

Functional testing uses the *single stuck-at model*, the most common fault model used in fault simulation, because of its effectiveness in finding many common defect types. The stuck-at fault models the behavior that occurs if the terminals of a gate are stuck at either a high (stuck-at-1) or low (stuck-at-0) voltage. The fault sites for this fault model include the pins of primitive instances. [Figure 2-10](#) shows the possible stuck-at faults that could occur on a single AND gate.

**Figure 2-10. Single Stuck-At Faults for AND Gate**



**Possible Errors: 6**

"a" s-a-1, "a" s-a-0

"b" s-a-1, "b" s-a-0

"c" s-a-1, "c" s-a-0

For a single-output, n-input gate, there are  $2(n+1)$  possible stuck-at errors. In this case, with  $n=2$ , six stuck-at errors are possible.

The ATPG tool uses the following fault collapsing rules for the single stuck-at model:

- **Buffer** - input stuck-at-0 is equivalent to output stuck-at-0. Input stuck-at-1 is equivalent to output stuck-at-1.
- **Inverter** - input stuck-at-0 is equivalent to output stuck-at-1. Input stuck-at-1 is equivalent to output stuck-at-0.
- **AND** - output stuck-at-0 is equivalent to any input stuck-at-0.
- **NAND** - output stuck-at-1 is equivalent to any input stuck-at-0.
- **OR** - output stuck-at-1 is equivalent to any input stuck-at-1.
- **NOR** - output stuck-at-0 is equivalent to any input stuck-at-1.
- **Net between single output pin and single input pin** - output pin stuck-at-0 is equivalent to input pin stuck-at-0. Output pin stuck-at-1 is equivalent to input pin stuck-at-1.

## Functional Testing and the Toggle Fault Model

Toggle fault testing ensures that a node can be driven to both a logical 0 and a logical 1 voltage. This type of test indicates the extent of your control over circuit nodes. Because the toggle fault model is faster and requires less overhead to run than stuck-at fault testing, you can experiment with different circuit configurations and get a quick indication of how much control you have over your circuit nodes.

The ATPG tool uses the following fault collapsing rules for the toggle fault model:

- **Buffer** - a fault on the input is equivalent to the same fault value at the output.
- **Inverter** - a fault on the input is equivalent to the opposite fault value at the output.

- **Net between single output pin and multiple input pin** - all faults of the same value are equivalent.

## IDDQ Testing and the Pseudo Stuck-At Fault Model

IDDQ testing, in general, can use several different types of fault models, including node toggle, pseudo stuck-at, transistor leakage, transistor stuck, and general node shorts.

The ATPG tool supports the *pseudo stuck-at* fault model for IDDQ testing. Testing detects a pseudo stuck-at model at a node if the fault is excited and propagated to the output of a cell (library model instance or primitive). Because library models can be hierarchical, fault modeling occurs at different levels of detail.

The pseudo stuck-at fault model detects all defects found by transistor-based fault models—if used at a sufficiently low level. The pseudo stuck-at fault model also detects several other types of defects that the traditional stuck-at fault model cannot detect, such as some adjacent bridging defects and CMOS transistor stuck-on conditions.

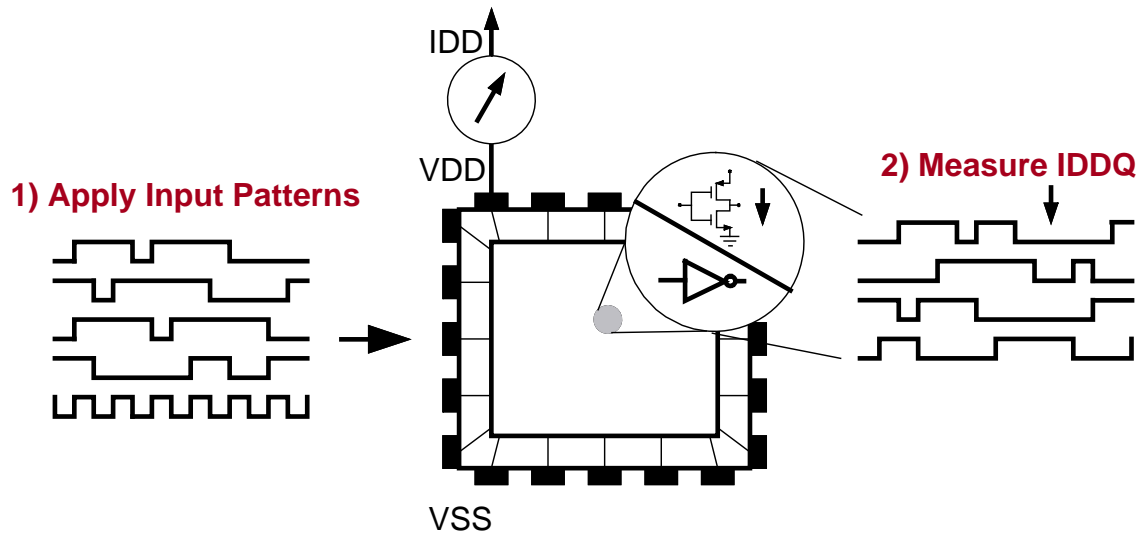
The benefit of using the pseudo stuck-at fault model is that it lets you obtain high defect coverage using IDDQ testing, without having to generate accurate transistor-level models for all library components.

The transistor leakage fault model is another fault model commonly used for IDDQ testing. This fault model models each transistor as a four terminal device, with six associated faults. The six faults for an NMOS transistor include G-S, G-D, D-S, G-SS, D-SS, and S-SS (where G, D, S, and SS are the gate, drain, source, and substrate, respectively).

You can only use the transistor level fault model on gate-level designs if each of the library models contains detailed transistor level information. Pseudo stuck-at faults on gate-level models equate to the corresponding transistor leakage faults for all primitive gates and fanout-free combinational primitives. Thus, without the detailed transistor-level information, you should use the pseudo stuck-at fault model as a convenient and accurate way to model faults in a gate-level design for IDDQ testing.

[Figure 2-11](#) shows the IDDQ testing process using the pseudo stuck-at fault model.

Figure 2-11. IDDQ Fault Testing



The pseudo stuck-at model detects internal transistor shorts, as well as “hard” stuck-ats (a node actually shorted to VDD or GND), using the principle that current flows when you try to drive two connected nodes to different values. While stuck-at fault models require propagation of the fault effects to a primary output, pseudo stuck-at fault models allow fault detection at the output of primitive gates or library cells.

IDDQ testing detects output pseudo stuck-at faults if the primitive or library cell output pin goes to the opposite value. Likewise, IDDQ testing detects input pseudo stuck-at faults when the input pin has the opposite value of the fault and the fault effect propagates to the output of the primitive or library cell.

By combining IDDQ testing with traditional stuck-at fault testing, you can greatly improve the overall test coverage of your design. However, because it is costly and impractical to monitor current for every vector in the test set, you can supplement an existing stuck-at test set with a compact set of test vectors for measuring IDDQ. This set of IDDQ vectors can either be generated automatically or intelligently chosen from an existing set of test vectors. Refer to section “[Creating an IDDQ Test Set](#)” on page 197 for information.

The fault collapsing rule for the pseudo stuck-at fault model is as follows: for faults associated with a single cell, pseudo stuck-at faults are considered equivalent if the corresponding stuck-at faults are equivalent.

[set\\_transition\\_holdpi](#) — Freezes all primary inputs values other than clocks and RAM controls during multiple cycles of pattern generation.

## The Static Bridge Fault Model

Certain pairs of nets in a design have specific characteristics that make them vulnerable to bridging. The *static bridge* fault model is used by the ATPG tool to test against potential bridge

sites (net pairs) extracted from the design. You can load the bridge sites from a bridge definition file or from the Calibre query server output file. For more information, see [“Net Pair Identification with Calibre for Bridge Fault Test Patterns”](#) on page 200.

This model uses a 4-Way Dominant fault model that works by driving one net (dominant) to a logic value and ensuring that the other net (follower) can be driven to the opposite value.

Let sig\_A and sig\_B be two nets in the design. If sig\_A and sig\_B are bridged together, the following faulty relationships exist:

- sig\_A is dominant with a value of 0 (sig\_A=0; sig\_B=1/0)
- sig\_A is dominant with a value of 1 (sig\_A=1; sig\_B=0/1)
- sig\_B is dominant with a value of 0 (sig\_B=0; sig\_A=1/0)
- sig\_B is dominant with a value of 1 (sig\_B=1; sig\_A=0/1)

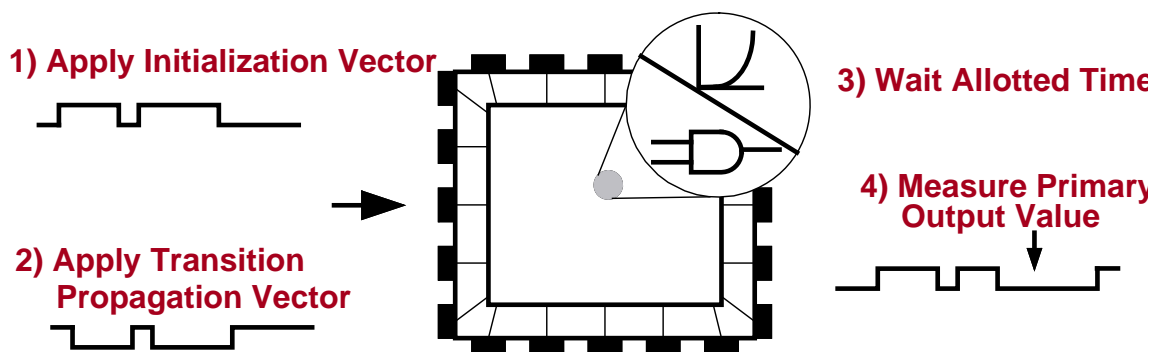
By default, the tool creates test patterns that test each net pair against all four faulty relationships.

## At-Speed Testing and the Transition Fault Model

Transition faults model large delay defects at gate terminals in the circuit under test. The transition fault model behaves as a stuck-at fault for a temporary period of time. The *slow-to-rise* transition fault models a device pin that is defective because its value is slow to change from a 0 to a 1. The *slow-to-fall* transition fault models a device pin that is defective because its value is slow to change from a 1 to a 0.

[Figure 2-12](#) demonstrates the at-speed testing process using the transition fault model. In this example, the process could be testing for a slow-to-rise or slow-to-fall fault on any of the pins of the AND gate.

**Figure 2-12. Transition Fault Detection Process**



A transition fault requires two test vectors for detection: an *initialization vector* and a *transition propagation vector*. The initialization vector propagates the initial transition value to the fault site. The transition vector, which is identical to the stuck-at fault pattern, propagates the final



transition value to the fault site. To detect the fault, the tool applies proper at-speed timing relative to the second vector, and measures the propagated effect at an external observation point.

The tool uses the following fault collapsing rules for the transition fault model:

- **Buffer** - a fault on the input is equivalent to the same fault value at the output.
- **Inverter** - a fault on the input is equivalent to the opposite fault value at the output.
- **Net between single output pin and single input pin** - all faults of the same value are equivalent.

[set\\_fault\\_type](#) — Specifies the fault model for which the tool develops or selects ATPG patterns. The transition option for this command specifies the tool to develop or select ATPG patterns for the transition fault model.

For more information on generating transition test sets, refer to [“Creating a Transition Delay Test Set”](#) on page 206.

## At-Speed Testing and the Path Delay Fault Model

Path delay faults model defects in circuit paths. Unlike the other fault types, path delay faults do not have localized fault sites. Rather, they are associated with testing the combined delay through all gates of specific paths (typically critical paths).

Path topology and edge type identify path delay faults. The path topology describes a user-specified path from beginning, or *launch point*, through a combinational path to the end, or *capture point*. The launch point is either a primary input or a state element. The capture point is either a primary output or a state element. State elements used for launch or capture points are either scan elements or non-scan elements that qualify for clock-sequential handling. A path definition file defines the paths for which you want patterns generated.

The edge type defines the type of transition placed on the launch point that you want to detect at the capture point. A “0” indicates a rising edge type, which is consistent with the slow-to-rise transition fault and is similar to a temporary stuck-at-0 fault. A “1” indicates a falling edge type, which is consistent with the slow-to-fall transition fault and is similar to a temporary stuck-at-1 fault.

The ATPG tool targets multiple path delay faults for each pattern it generates. Within the (ASCII) test pattern set, patterns that detect path delay faults include comments after the pattern statement identifying the path fault, type of detection, time and point of launch event, time and point of capture event, and the observation point. Information about which paths were detected by each pattern is also included.

For more information on generating path delay test sets, refer to [“Creating a Path Delay Test Set”](#) on page 220.

## User-Defined Fault Modeling

The standard fault models supported by the tool are typically sufficient to create a highly efficient pattern set that detects the majority of potential defects. However, a number of defects are not covered by these models and are detected only by accident; these defects require specific conditions that cannot be defined for the existing fault models.

You can use user-defined fault models (UDFMs) to define custom fault models. UDFMs extend the natively-supported fault models (primarily stuck-at and transition) by adding combinational or sequential constraints on other pins/nets. These custom models enable you to generate specific test patterns for process-related defects like intra-cell bridges or any other kind of defect that requires additional constraints. You can also use UDFMs to define required conditions in an additional fault model to reduce the need for functional test patterns.

Specifically, UDFMs provide the following capabilities:

- Support for generating high compact pattern sets
- Close integration with the existing ATPG flow
- Support of static (stuck-at) and delay (transition) fault models
- Definition of test alternatives
- Definition of additional single or multiple-cycle conditions
- Definition on library or hierarchy levels
- Definition on specific instances
- Write and load of fault status information using the MTFI format. For more information, refer to [“MTFI File Format”](#) on page 391.

## Restrictions

The following restrictions apply to UDFMs:

- No support for any kind of fault collapsing.
- No support for fault protection (`set_fault_protection` command).
- No support for multiple detection during pattern generation (`set_multiple_detection` command).
- No support for the analyze fault functionality (`analyze_fault` command).
- Separation of static and delay fault models. Both fault types cannot be handled together in one pattern generation step. The definitions must be handled in different pattern generation runs; therefore, this leads to multiple pattern sets.

---

**Note**

UDFM requires additional memory. The amount of additional memory needed depends on the number of UDFM definitions and their complexity.

---

## UDFM File Format

Input for: read\_fault\_sites

The User-Defined Fault Model (UDFM) file is an ASCII file that models custom fault sites for internal library cells. The UDFM file is created manually or is created using Tessent CellModelGen. UDFM provides a method for expanding native ATPG faults such as stuck-at to exhaustively test library cell models.

### Format

A UDFM file must conform to the following syntax rules:

- Precede each line of comment text with a pair of backslashes (//).
- Keywords are not case-sensitive.
- Use a colon sign to define a value for a keyword.
- Enclose all string values in double quotation marks (" ").
- Enclose all UDFM declarations in braces ({}).
- Separate each entry within the UDFM declaration with a semicolon (;).

### Keywords

In a UDFM file, you use keywords to define the fault models you are creating and their behavior. [Table 2-2](#) specifies version 1.0 of the UDFM syntax.

**Table 2-2. UDFM Keywords**

Keyword	Description
UDFM	Required. Specifies version 1.0 of the UDFM syntax. The syntax version number must precede the UDFM declaration.
UdfmType	Required. Specifies a user-defined identifier for all faults declared within the UDFM file. This identifier allows you to target the group of faults declared in a UDFM file for an ATPG run or other test pattern manipulations. The following commands allow you to use the UdfmType identifier as an argument: add_faults, delete_faults, report_fault_sites, report_faults, and write_faults.
Cell	Optional. Maps a set of defined faults to a specified library cell. When the UDFM is loaded, the defined faults are applied to all instances of the specified library cell. The cell definition supports only input and output ports.

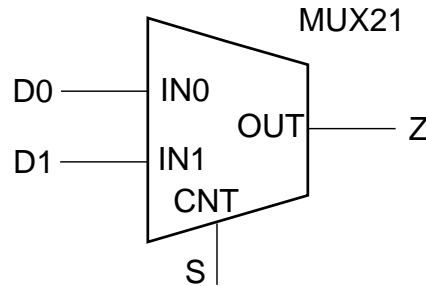
**Table 2-2. UDFM Keywords**

Keyword	Description
Module	Optional. Maps a set of defined faults to a specified module. When the UDFM is loaded, the defined faults are applied to all instances of the specified module. The specified value can be a library cell or a module at any hierarchy in the design.
Instance	Optional. Maps a set of defined faults to a specified instance. When the UDFM is loaded, the defined faults are applied to the specified instance. Complete pathname must be specified either the full hierarchical path (starting with “/”) or relative to the instance path.
Fault	Required. Defines one or more faults. The definition contains two parts: <ul style="list-style-type: none"> <li>• Fault identification string — name used to reference the fault.</li> <li>• List of test alternatives — When more than one test is listed, the fault is tested if any test alternatives are satisfied.</li> </ul>
Test	Required. Specifies the fault conditions to test for. You must minimally specify the defect type with the faulty value and, optionally, a list of conditions. For more information, see the StaticFault or Delay Fault keyword.
Conditions	Optional. Defines the nodes and values that represent all fault conditions. To fulfill a <i>condition</i> , the specified node must be set to the specified value: <ul style="list-style-type: none"> <li>• 0: state low</li> <li>• 1: state high</li> <li>• —: no condition</li> </ul>
StaticFault or DelayFault	One of these keywords is required, except when you also use the “Observation” keyword set to “false.” Specifies the faulty value at the specified location. If this statement applies to a cell, you can specify only cell ports; if it applies to module definitions, you can specify only ports and module internal elements.
Observation	Optional. Allows you to disable fault propagation as required by faults models like toggle bridge. By default, fault propagation is enabled. Valid arguments are “true” or “false.” Note that when you disable observation (observation:false), the definition of the UDFM faulty value (StaticFault or DelayFault) becomes optional. This attribute is supported only on the “UDFMType” level and disables the fault propagation for all fault models that follow.
Properties	Optional. Stores information for future uses or by external user tools.
EncryptedTest	Optional. Automatically used by Tessent CellModelGen to protect the IP. For more information, refer to the <a href="#">Tessent CellModelGen Tool Reference</a> .

## Example UDFM Files

### Intra-cell Bridge Fault Example

The following is an example that tests for a cell internal bridge fault of a 2-to-1 multiplexer (MUX21) from the CMOS123\_std library. This example specifies two tests with the required activation conditions on the input pins for observing on pin Z the effects of the bridge fault.



```

UDFM {
  version : 1;
  Properties {
    "library-name" : "CMOS123_std";
    "create-by"    : "my tool 1.0";
  }
  UdfmType ("intra-cell-bridges") {
    Cell ("MUX21") {
      Fault ("myFlt-N1-Z") {
        Test {
          StaticFault {"Z" : 1;}
          Conditions {"D0" : 0; "D1" : 0; "S" : 0;}
        }
        Test {
          StaticFault {"Z" : 0;}
          Conditions {"D0" : 0; "D1" : 1; "S" : 0;}
        }
      }
    }
  }
}

```

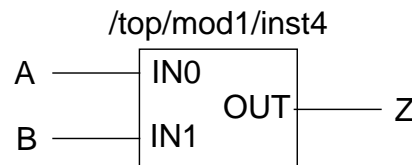
### Delay Fault Example

The following is an example that tests for delay faults on the input pins of a 2-to-1 multiplexer (MUX21, shown in the previous example). This example specifies a test where the D0 input transitions from 1 to 0 while observing that the Z output is faulty if it remains at 1.

```
UDFM {  
  version : 1;  
  UdfmType ("Special-delay") {  
    Cell ("MUX21") {  
      Fault ("myFlt-D0") {  
        Test {  
          DelayFault {"Z" : 1;}  
          Conditions {"D0" : 10; "D1" : 00; "S" : 00;}  
        }  
      }  
    }  
  }  
}
```

### Instance-based Fault Example

The following is an example that tests for a bridge fault on the input pins of a single generic 2-input instance. This example specifies two tests for observing on pin Z the effects of a bridge fault on the input pins.



```
UDFM {  
  version : 1;  
  UdfmType ("intra-cell-bridges") {  
    Instance ("/top/mod1/inst4") {  
      Fault ("f001") {  
        Test{ StaticFault{"Z":0;} Conditions{"A":0;"B":1;} }  
      }  
      Fault ("f002") {  
        Test{ StaticFault{"Z":1;} Conditions{"A":1;"B":1;} }  
      }  
    }  
  }  
}
```

### Toggle Bridges Example

The following is an example that tests for two specific toggle bridge faults within an entire design. This example tests for bridge faults between the following two sets of nets:

Bridge1: /top/mod2/i3/y /top/mod2/i5/a  
Bridge2: /top/mod1/i47/z /top/mod1/iop/c

This example specifies two tests to detect each bridge fault by ensuring that each net pair can be set to different values simultaneously.

```
UDFM {  
  Version:2;  
  UdfmType("toggle-bridges") {  
    Observation:false;  
    Instance("/") {  
      Fault("Bridge1") {  
        Test{Conditions{"top/mod2/i3/y":1; "top/mod2/i5/a":0;}}  
        Test{Conditions{"top/mod2/i3/y":0; "top/mod2/i5/a":1;}}  
      }  
      Fault("Bridge2") {  
        Test{Conditions{"top/mod1/i47/z":1; "top/mod1/iop/c":0;}}  
        Test{Conditions{"top/mod1/i47/z":0; "top/mod1/iop/c":1;}}  
      }  
    }  
  }  
}
```

## Creating a User-Defined Fault Model

The following procedure describes how you can manually define your own fault models.

### Prerequisites

- Text editor

### Procedure

---

**Note**

---

In this procedure, entries added in the current step are shown in **bold**.

---

1. In a text editor, enter the UDFM statement as shown here:

```
UDFM { }
```

All of the additional statements you use to define all fault models will be contained within the curly brackets of this statement.

2. Add the UDFM version number statement next.

```
UDFM {  
    version : 1;  
}
```

3. Add the UdfmType keyword to the ASCII file under the version number statement to create a fault model name.

```
UDFM {  
    version : 1; // Syntax version ensures future compatibility  
    UdfmType ("udfm_fault_model") {  
    }  
}
```

4. Specify the type of object you want to attach the fault to using the Cell, Module, and Instance keywords. For more information on these keywords, see “[UDFM Keywords](#).”

```
UDFM {  
    version : 1;  
    UdfmType ("udfm_fault_model") {  
        Cell ("MUX21") {  
        }  
    }  
}
```



5. Define a unique fault model name using the Fault keyword. You can define multiple faults.

```
UDFM {
  version : 1;
  UdfmType ("udfm_fault_model") {
    Cell ("MUX21") {
      Fault ("myFlt-D0") {
      }
    }
  }
}
```

6. Define how the defect can be tested using the Test, StaticFault or Delay Fault, and Conditions keywords. Notice that the following example also shows how to use the conditions statement for a single assignment or a list of assignments. See [“UDFM Keywords”](#) for the complete list of keywords.

```
UDFM {
  version : 1;
  UdfmType ("udfm_fault_model") {
    Cell ("MUX21") {
      Fault ("myFlt-N1-Z") {
        Test {
          StaticFault {"Z" : 1;}
          Conditions {"D0" : 0; "D1" : 0; "S" : 0;}
        }
        Test {
          StaticFault {"Z" : 0;}
          Conditions {"D0" : 0; "D1" : 1; "S" : 0;}
        }
      }
    }
  }
}
```

7. Save the file using a *.udfm* extension.

## Generating UDFM Test Patterns

UDFM test pattern generation is similar to pattern generation for bridge logic faults. In both cases, fault definitions are accessed from an external file and the internal fault list is built using that data. This procedure describes the differences needed for UDFM pattern generation.

UDFM test pattern generation for static fault models is similar to test pattern generation for stuck-at faults. As a result, most ATPG settings for stuck-at faults can be used for UDFM faults. However, because UDFM fault definitions must be imported, the following commands will work slightly differently than they do for stuck-at fault.

<code>add_faults</code>	<code>delete_fault_sites</code>	<code>report_faults</code>	<code>set_fault_type</code>
<code>delete_faults</code>	<code>read_fault_sites</code>	<code>report_udfm_statistics</code>	<code>write_faults</code>

### Prerequisites

- UDFM file that contains fault definitions.

### Procedure

1. Set the fault type to UDFM using the `set_fault_type` command with the UDFM option:

```
set_fault_type udfm
```

---

**Note**

---

When you change the fault type, the current fault list and internal test pattern set are deleted.

---

2. Load the fault definitions from a specified UDFM file into your current tool session using the `read_fault_sites` command:

```
read_fault_sites <filename>.udfm
```

3. Create the internal fault list using all of the fault definitions from the UDFM file with the `add_faults` command:

```
add_faults -All
```

4. Generate test patterns using the `create_patterns` command:

```
create_patterns
```

For more information, refer to the [Tessent Shell Reference Manual](#).

5. Print out the resulting statistics using the `report_statistics` command.

```
report_statistics
```

For more information, refer to the [Tessent Shell Reference Manual](#).

## Related Topics

[set\\_fault\\_type](#)

## Multiple Detect

The basic idea of multiple detect (n-detect) is to randomly target each fault multiple times. By changing the way the fault is targeted and the other values in the pattern set, the potential to detect a bridge increases. This approach starts with a standard stuck-at or transition pattern set. Each fault is graded for multiple detect. Additional ATPG is then performed and patterns created targeting the faults that have lower than the multiple detect target threshold.

## Bridge Coverage Estimate

Bridge coverage estimate (BCE) is a metric for reporting the ability of multiple detect to statistically detect a bridge defect. If a bridge fault exists between the target fault site and another net then there is a 50% change of detecting the fault with one pattern. If a second randomly different pattern targets the same fault then the probability of detecting the bridge is  $1 - 0.5^2$ . BCE performs this type of calculation for all faults in the target list and will always be lower than the test coverage value.

The following shows the statistics report that is automatically produced when multiple detection or embedded multiple detect is enabled:

Statistics Report Transition Faults			
Fault Classes		#faults (total)	
FU (full)		1114	
DS (det_simulation)		1039	(93.27%)
UU (unused)		30	( 2.69%)
TI (tied)		4	( 0.36%)
RE (redundant)		3	( 0.27%)
AU (atpg_untestable)		38	( 3.41%)
Coverage			
test_coverage		96.47%	
fault_coverage		93.27%	
atpg_effectiveness		100.00%	
#test_patterns		130	
#clock_sequential_patterns		130	
#simulated_patterns		256	
CPU_time (secs)		0.5	
Multiple Detection Statistics			

Detections (N)	DS Faults (Detection == N)	Test Coverage (Detection >= N)
1	0 ( 0.00%)	1039 ( 96.47%)
2	0 ( 0.00%)	1039 ( 96.47%)
3	93 ( 8.35%)	1039 ( 96.47%)
4	56 ( 5.03%)	946 ( 87.84%)
5	54 ( 4.85%)	890 ( 82.64%)
6	56 ( 5.03%)	836 ( 77.62%)
7	50 ( 4.49%)	780 ( 72.42%)
8	46 ( 4.13%)	730 ( 67.78%)
9	54 ( 4.85%)	684 ( 63.51%)
10+	630 ( 56.55%)	630 ( 58.50%)
bridge_coverage_estimate		94.71%

The report includes the BCE value and the number of faults with various detects less than the target detection value.

## Embedded Multiple Detect

Embedded multiple detect (EMD) is a method of improving the multiple detect of a pattern set without increasing the number of patterns within that pattern set. Essentially, EMD produces the same quality patterns as a standard pattern set but in addition adds improved multiple detection for free. The only cost for the extra multiple detection is a longer run time during pattern creation of about 30% to 50%. As a result, EMD is often considered a no-cost additional value and used for ATPG.

When performing ATPG, the tool tries to detect as many previously undetected faults in parallel within the same pattern as possible. However, even though ATPG maximizes the number of previously undetected faults detected per pattern, only a small percentage of scan cells will have specific values necessary for the detection. These *specified bits* need to be loaded in scan cells for that pattern are referred to as the test cube. The remaining scan cells that are not filled with test cube values are randomly filled for fortuitous detection of untargeted faults. EMD uses the same ATPG starting point to produce a test cube but then determines if there are some faults that previously had a low number of detections. For these faults, EMD will put additional scan cell values added to the test cube to improve multiple detection on top of the new detection pattern.

EMD will have a multiple detection that is better than normal ATPG but might not be as high a BCE as n-detect with additional patterns could produce. In a design containing EDT circuitry, the amount of detection will be dependent on the how aggressive the compression is. The more aggressive (higher) compression, the lower the encoding capacity and the fewer test cube bits can be specified per pattern. If a design is targeting 200x compression then the available test cube bits might be mostly filled up for many of the patterns with values for the undetected fault detection. As a result, the additional EMD multiple detection might not be significantly higher than BCE for the standard pattern set.

Standard multiple detect will have a cost of additional patterns but will also have a higher multiple detection than EMD. How much difference between EMD and multiple detect will be dependent on the particular design's pattern set and the level of compression used<sup>1</sup>.

## Enabling Multiple Detect for EMD

You can enable either EMD or multiple detect using the [set\\_multiple\\_detection](#) command. Either approach is supported with stuck-at and transition patterns. You can use the following arguments with the `set_multiple_detection` command:

- **-Guaranteed\_atpg\_detection** — Sets the multiple detect target for each fault. ATPG will try to target each fault the specified number of times. ATPG does not guarantee that the fault will be detected in a completely different path but randomly changes the way it excites and propagates the fault. In addition, the random fill is different so values around the target fault are likely to be randomly different than previous detections.
- **-Desired\_atpg\_detections** — Sets the EMD target. Users often set this target to 5 or a value in that range.
- **-Simulation\_drop\_limit** — This is the accuracy of the BCE calculation. In general, there is no reason to change this value from the default of 10. This means that the BCE simulations stop once a fault is learned to be detected 10 times. A fault multiple detected 10 times will have a BCE and statistical chance of detecting a defect of  $1 - 1/2e10$  or .99902. Which is only a 0.0009% inaccuracy which is slightly conservative but insignificant.

## Logic BIST

Logic BIST has a natural very high multiple detection. The faults that are detected with logic BIST would often have multiple detection well above 10. This is in part due to the very large number of patterns typically used for logic BIST. In addition, many of the hard to detect areas of a circuit are made randomly testable and easier to produce high multiple detect coverage with test logic inserted during logic BIST.

## Multiple Detect and AU Faults

During multiple detect ATPG, the AU (ATPG\_untestable) fault count changes only at the beginning of each ATPG loop rather than during the loop. This is normal behavior when using NCPs (named capture procedures) for ATPG. The tool updates AU faults after going through all the NCPs at the end of the loop.

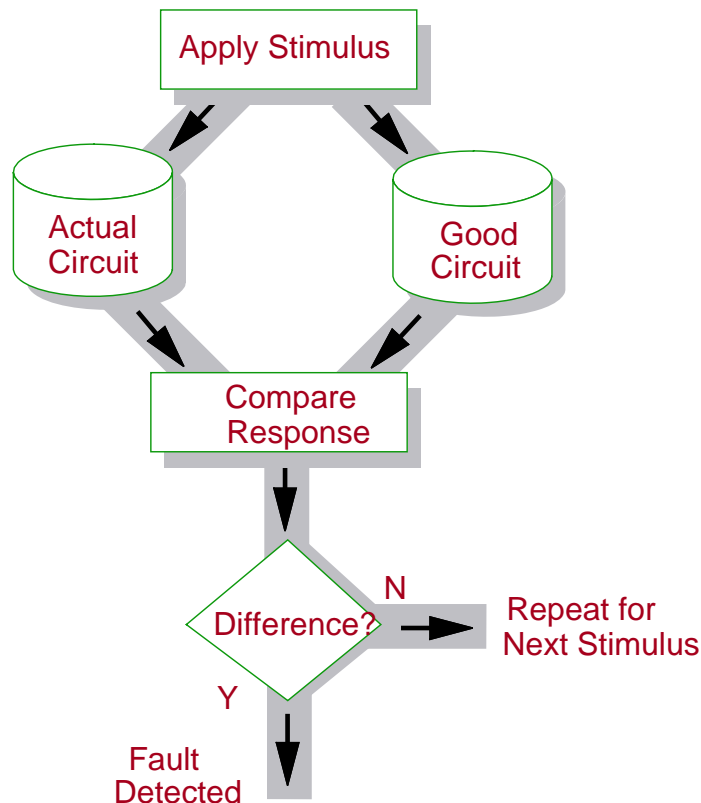
---

1. J. Geuzebroek, et al., "Embedded Multi-Detect ATPG and Its Effect on the Detection of Unmodeled Defects", Proceedings IEEE Int. Test Conference, 2007

## Fault Detection

Figure 2-13 shows the basic fault detection process.

**Figure 2-13. Fault Detection Process**



Faults detection works by comparing the response of a known-good version of the circuit to that of the actual circuit, for a given stimulus set. A fault exists if there is any difference in the responses. You then repeat the process for each stimulus set.

## Path Sensitization and Fault Detection

One common fault detection approach is *path sensitization*. The path sensitization method, which is used by the tool to detect stuck-at faults, starts at the fault site and tries to construct a vector to propagate the fault effect to a primary output. When successful, the tools create a stimulus set (a test pattern) to detect the fault. They attempt to do this for each fault in the circuit's fault universe. Figure 2-14 shows an example circuit for which path sensitization is appropriate.

**Figure 2-14. Path Sensitization Example**

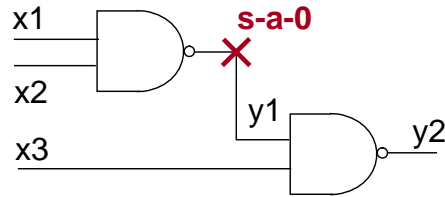


Figure 2-14 has a stuck-at-0 on line y1 as the target fault. The x1, x2, and x3 signals are the primary inputs, and y2 is the primary output. The path sensitization procedure for this example follows:

1. Find an input value that sets the fault site to the opposite of the desired value. In this case, the process needs to determine the input values necessary at x1 and/or x2 that set y1 to a 1, since the target fault is s-a-0. Setting x1 (or x2) to a 0 properly sets y1 to a 1.
2. Select a path to propagate the response of the fault site to a primary output. In this case, the fault response propagates to primary output y2.
3. Specify the input values (in addition to those specified in step 1) to enable detection at the primary output. In this case, in order to detect the fault at y1, the x3 input must be set to a 1.

## Fault Classes

The tool categorizes faults into *fault classes*, based on how the faults were detected or why they could not be detected. Each fault class has a unique name and two character class code. When reporting faults, the tool uses either the class name or the class code to identify the fault class to which the fault belongs.

---

### Note



The tools may classify a fault in different categories, depending on the selected fault type.

---

## Untestable (UT)

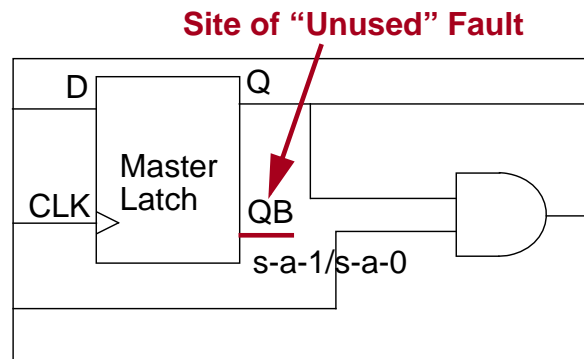
Untestable (UT) faults are faults for which no pattern can exist to either detect or possible-detect them. Untestable faults cannot cause functional failures, so the tools exclude them when calculating test coverage. Because the tools acquire some knowledge of faults prior to ATPG, they classify certain unused, tied, or blocked faults before ATPG runs. When ATPG runs, it immediately places these faults in the appropriate categories. However, redundant fault detection requires further analysis.

The following list discusses each of the untestable fault classes.

- **Unused (UU)**

The unused fault class includes all faults on circuitry unconnected to any circuit observation point and faults on floating primary outputs. For information about UU fault sub-classes, refer to [Table 2-3](#) on page 62. [Figure 2-15](#) shows the site of an unused fault.

**Figure 2-15. Example of “Unused” Fault in Circuitry**



- **Tied (TI)**

The tied fault class includes faults on gates where the point of the fault is tied to a value identical to the fault stuck value. The tied circuitry could be due to:

- Tied signals
- AND and OR gates with complementary inputs
- Exclusive-OR gates with common inputs
- Line holds due to primary input pins held at a constant logic value during test by CT0 or CT1 pin constraints you applied with the [add\\_input\\_constraints](#) command

**Note**

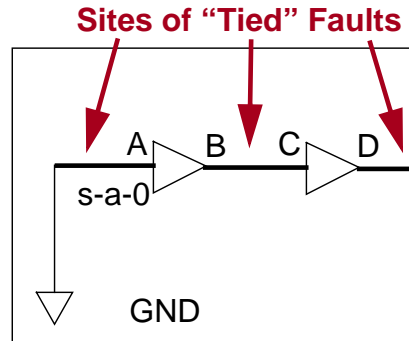


The tools do not use line holds set by the “[add\\_input\\_constraints -C0](#)” (or C1) command to determine tied circuitry. C0 and C1 pin constraints (as distinct from CT0 and CT1 constraints) result in [ATPG\\_untestable \(AU\)](#) faults, not tied faults. For more information, refer to the [add\\_input\\_constraints](#) command.

[Figure 2-16](#) shows the site of a tied fault.



Figure 2-16. Example of “Tied” Fault in Circuitry



Because tied values propagate, the tied circuitry at A causes tied faults at A, B, C, and D.

- **Blocked (BL)**

The blocked fault class includes faults on circuitry for which tied logic blocks all paths to an observable point. The tied circuitry could be due to:

- Tied signals
- AND and OR gates with complementary inputs
- Exclusive-OR gates with common inputs
- Line holds due to primary input pins held at a constant logic value during test by CT0 or CT1 pin constraints you applied with the [add\\_input\\_constraints](#) command.

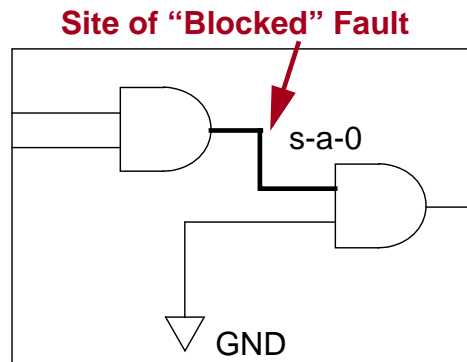
**Note**



The tools do not use line holds set by the “[add\\_input\\_constraints -C0](#)” (or C1) command to determine tied circuitry. C0 and C1 pin constraints (as distinct from CT0 and CT1 constraints) result in [ATPG\\_untestable \(AU\)](#) faults, not blocked faults. For more information, refer to the [add\\_input\\_constraints](#) command.

This class also includes faults on selector lines of multiplexers that have identical data lines. [Figure 2-17](#) shows the site of a blocked fault.

### Figure 2-17. Example of “Blocked” Fault in Circuitry



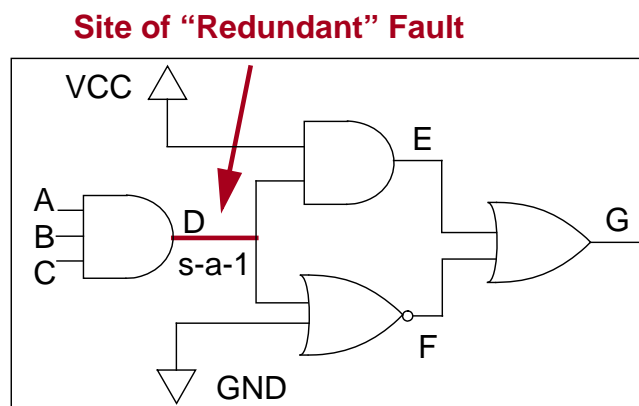
## Note

Tied faults and blocked faults can be equivalent faults.

- **Redundant (RE)**

The redundant fault class includes faults the test generator considers undetectable. After the test pattern generator exhausts all patterns, it performs a special analysis to verify that the fault is undetectable under any conditions. [Figure 2-18](#) shows the site of a redundant fault.

### Figure 2-18. Example of “Redundant” Fault in Circuitry



In this circuit, signal G always has the value of 1, no matter what the values of A, B, and C. If D is stuck at 1, this fault is undetectable because the value of G can never change, regardless of the value at D.

## Testable (TE)

Testable (TE) faults are all those faults that cannot be proven untestable. The testable fault classes include:

- **Detected (DT)**

The detected fault class includes all faults that the ATPG process identifies as detected. The detected fault class contains two groups:

- `det_simulation` (DS) - faults detected when the tool performs fault simulation.
- `det_implication` (DI) - faults detected when the tool performs learning analysis.

The `det_implication` group normally includes faults in the scan path circuitry, as well as faults that propagate ungated to the shift clock input of scan cells. The scan chain functional test, which detects a binary difference at an observation point, guarantees detection of these faults. The tool also classifies scan enable stuck-in-system-mode faults on the multiplexer select line of mux-DFFs as DI.

The tool provides the `update_implication_detections` command, which lets you specify additional types of faults for this category. Refer to the [update\\_implication\\_detections](#) command description in the *Tessent Shell Reference Manual*.

For path delay testing, the detected fault class includes two other groups:

- `det_robust` (DR) - robust detected faults.
- `det_functional` (DF) - functionally detected faults.

For detailed information on the path delay groups, refer to [“Path Delay Fault Detection”](#) on page 221.

- **Posdet (PD)**

The posdet, or possible-detected fault class includes all faults that fault simulation identifies as possible-detected but not hard detected. A possible-detected fault results from a good-machine simulation observing 0 or 1 and the faulty machine observing X. A hard-detected fault results from binary (not X) differences between the good and faulty machine simulations. The posdet class contains two groups:

- `posdet_testable` (PT) - potentially detectable posdet faults. PT faults result when the tool cannot prove the 0/X or 1/X difference is the only possible outcome. A higher abort limit may reduce the number of these faults.
- `posdet_untestable` (PU) - proven ATPG\_untestable during pattern generation and hard undetectable posdet faults. Typically, faults may be classified as PU during ATPG or when the `“compress_patterns -reset_au”` command is used.

By default, the calculations give 50% credit for posdet faults. You can adjust the credit percentage with the [set\\_possible\\_credit](#) command.

- **ATPG\_untestable (AU)**

The ATPG\_untestable fault class includes all faults for which the test generator is unable to find a pattern to create a test, and yet cannot prove the fault redundant. Testable faults become ATPG\_untestable faults because of constraints, or limitations, placed on the ATPG tool (such as a pin constraint or an insufficient sequential depth). These faults may be possible-detectable, or detectable, if you remove some constraint, or change some limitation, on the test generator (such as removing a pin constraint or changing the sequential depth). You *cannot* detect them by increasing the test generator abort limit.

The tools place faults in the AU category based on the type of deterministic test generation method used. That is, different test methods create different AU fault sets. Likewise, the tool can create different AU fault sets even using the same test method. Thus, if you switch test methods (that is, change the fault type) or tools, you should reset the AU fault list using the [reset\\_au\\_faults](#) command.

---

**Note**

During multiple detect ATPG, the AU fault count changes only at the beginning of each ATPG loop rather than during the loop. This is normal behavior when using NCPs (named capture procedures) for ATPG. The tool updates AU faults after going through all the NCPs at the end of the loop.

---

AU faults are categorized into several predefined fault sub-classes, as listed in [Table 2-3](#) on page 62.

- **Undetected (UD)**

The undetected fault class includes undetected faults that cannot be proven untestable or ATPG\_untestable. The undetected class contains groups:

- uncontrolled (UC) - undetected faults, which during pattern simulation, never achieve the value at the point of the fault required for fault detection—that is, they are uncontrollable.
- unobserved (UO) - faults whose effects do not propagate to an observable point.

All testable faults prior to ATPG are put in the UC category. Faults that remain UC or UO after ATPG are aborted, which means that a higher abort limit may reduce the number of UC or UO faults.

---

**Note**

Uncontrolled and unobserved faults can be equivalent faults. If a fault is both uncontrolled and unobserved, it is categorized as UC.

---

## Fault Class Hierarchy

Fault classes are hierarchical. The highest level, Full, includes all faults in the fault list. Within Full, faults are classified into untestable and testable fault classes, and so on, in the manner shown in [Figure 2-19](#).

**Figure 2-19. Fault Class Hierarchy**

- ```

1. Full (FU)
  1.1 TEstable (TE)
    a. DETected (DT)
      i. DET_Simulation (DS)
      ii. DET_Implication (DI)
      iii. DET_Robust (DR)—Path Delay Testing Only
      iv. DET_Functional (DF)—Path Delay Testing Only
    b. POSDET (PD)
      i. POSDET_Untestable (PU)
      ii. POSDET_Testable (PT)
    c. Atpg_untestable (AU)
    d. UNDetected (UD)
      i. UNControlled (UC)
      ii. UNObserved (UO)
  1.2 UNTestable (UT)
    a. UNUsed (UU)
    b. Tied (TI)
    c. Blocked (BL)
    d. Redundant (RE)
  
```

For any given level of the hierarchy, the tool assigns a fault to one—and only one—class. If the tool can place a fault in more than one class of the same level, the tool places the fault in the class that occurs first in the list of fault classes.

## Fault Sub-classes

The DI, AU, UD, and UU fault classes are further categorized into fault sub-classes, which are listed in [Table 2-3](#). For more information about each AU and UD fault sub-classes, click on the hyperlinks in the table.

The DI and AU fault classes can also contain user-defined sub-classes, which you create by grouping a set of faults into one sub-class and assigning a name to the group.

**Table 2-3. Fault Sub-classes**

| Fault Class | Fault Sub-class  | Code | Description                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------|------------------|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DI          | EDT_LOGIC        | EDT  | Faults in EDT logic detected by implication when EDT Finder is On. Supported for stuck and transition fault types. <ul style="list-style-type: none"> <li>Includes faults implicitly tested when applying the EDT patterns.</li> <li>Excludes redundant faults in EDT logic, faults in bypass logic or other dual configurations.</li> <li>Excludes faults already identified as implicitly tested.</li> </ul> |
| AU          | BLACK_BOXES      | BB   | Fault untestable due to black box                                                                                                                                                                                                                                                                                                                                                                              |
|             | EDT_BLOCKS       | EDT  | AU faults in EDT block                                                                                                                                                                                                                                                                                                                                                                                         |
|             | PIN_CONSTRAINTS  | PC   | Tied or blocked by input constraint                                                                                                                                                                                                                                                                                                                                                                            |
|             | TIED_CELLS       | TC   | Tied or blocked by tied non-scan cell                                                                                                                                                                                                                                                                                                                                                                          |
|             | CELL_CONSTRAINTS | CC   | Tied or blocked by cell constraint                                                                                                                                                                                                                                                                                                                                                                             |
|             | FALSE_PATHS      | FP   | Fault untestable due to false path                                                                                                                                                                                                                                                                                                                                                                             |
|             | MULTICYCLE_PATHS | MCP  | Fault untestable due to multicycle path                                                                                                                                                                                                                                                                                                                                                                        |
|             | SEQUENTIAL_DEPTH | SEQ  | Untestable due to insufficient sequential depth                                                                                                                                                                                                                                                                                                                                                                |
| UD          | ATPG_ABORT       | AAB  | Fault aborted by ATPG tool                                                                                                                                                                                                                                                                                                                                                                                     |
|             | UNSUCCESS        | UNS  | Fault undetected due to unsuccessful test                                                                                                                                                                                                                                                                                                                                                                      |
|             | EDT_ABORT        | EAB  | Fault aborted due to insufficient EDT encoding capacity                                                                                                                                                                                                                                                                                                                                                        |
| UU          | CONNECTED        | CON  | Fault in the logic cone but with no observation point                                                                                                                                                                                                                                                                                                                                                          |
|             | FLOATING         | FL   | Fault on the unconnected output of a cell                                                                                                                                                                                                                                                                                                                                                                      |

## AU.BB — BLACK\_BOXES

These are faults that are untestable due to a black box, which includes faults that need to be propagated through a blackbox to reach an observation point, as well as faults whose control or observation requires values from the output(s) of a blackbox. You can use the [report\\_black\\_boxes](#) command to identify the black boxes in your design. Your only potential option for resolving AU.BB faults is to create a model for each black box, although this may not fix the entire problem.

## AU.EDT — EDT\_BLOCKS

These are faults inside an instance identified as an EDT instance using the [set\\_edt\\_instances](#) command. If EDT Finder is turned on, most of those faults (regardless of the use of [set\\_edt\\_instances](#)) are classified as DI.EDT faults.

To further reduce the number of AU.EDT faults, you must either use default naming for EDT blocks, or you must use the [add\\_edt\\_blocks](#) command to explicitly name EDT blocks. Note that the tool excludes AU.EDT faults from the relevant fault coverage by default because the chain and IP test actually checks most of the EDT logic in addition to the scan chains, so other than properly naming EDT blocks, you need not do anything more. For more information about relevant coverage, refer to the description of the [set\\_relevant\\_coverage](#) command in the *Tessent Shell Reference Manual*.

## AU.PC — PIN\_CONSTRAINTS

These are faults that are uncontrollable or that cannot be propagated to an observation point, in the presence of a constraint value. That is, because the tool can't toggle the pin, the tool cannot test the fanout. The only possible solution is to evaluate whether you really need the input constraint, and if not, to remove the constraint. To help troubleshoot, you can use the “[report\\_statistics -detailed\\_analysis](#)” command to report specific pins.

## AU.TC — TIED\_CELLS

These are faults associated with cells that are always 0 (TIE0) or 1 (TIE1) or X (TIEX) during capture. One example is test data registers that are loaded during test\_setup and then constrained so as to preserve that value during the rest of scan test. The only possible solution is to verify that the tied state is really required, and if not, modify the test\_setup procedure. To help troubleshoot, you can use the “[report\\_statistics -detailed\\_analysis](#)” command to report specific cells.

## AU.CC — CELL\_CONSTRAINTS

These are faults that are uncontrollable or that cannot be propagated to an observation point, in the presence of a constraint value. That is, because the tool can't toggle a state within the cell, the tool cannot test the fanout or the faults along the input path that need to propagate through the cell. The only possible solution is to evaluate whether you really need the cell constraint, and if not, to remove the constraint. To help troubleshoot, you can use the “[report\\_statistics -detailed\\_analysis](#)” command to report specific cells.

For more information about cell constraints and their affect on ATPG, refer to the [add\\_cell\\_constraints](#) command description in the *Tessent Shell Reference Manual*.

## AU.FP — FALSE\_PATHS

These are faults that can be tested only through a false path. However, if there is any other path possible that is not false in which the fault exists, then the fault remains in the fault list.

Therefore, many faults along false paths might not be classified as AU.FP. For example, assume that the end of a long false path is connected to both a flop A and a flop B. If flop B isn't in a false path, then the tool can test all the setup faults using flop B and can therefore classify none as AU.FP.

In the case of hold-time false paths, the tool cannot classify them as AU.FP because a test is almost always possible that doesn't cause a hold-time violation. For example, consider a circuit with a hold-time false path that contains a flop A that feeds a buffer connected to flop B. If a pattern places a 1 on flop A's D input and a 0 on Q, then ATPG simulation would capture X in flop B because the tool can't be certain the capture cycle would appropriately capture 0 in flop B. That is, the hold-time violation could cause flop A to update and then propagate to flop B before the clock captures at flop B. However, if flop A's D input is 0 and Q is 0, then the 0 would be properly captured regardless of the hold-time false path. This is why the tool cannot remove the faults in this circuit from the fault list and define them as AU.FP.

Note that the tool excludes AU.FP faults from the relevant fault coverage by default. For more information about relevant coverage, refer to the description of the [set\\_relevant\\_coverage](#) command in the *Tessent Shell Reference Manual*.

## AU.MCP — MULTICYCLE\_PATHS

These are faults that can be tested only through a multicycle path, which means that the fault can't propagate between the launch and capture point within a single clock cycle. For information about resolving multicycle path issues, refer to “[Preventing Pattern Failures Due to Timing Exception Paths](#)” on page 214.

## AU.SEQ — SEQUENTIAL\_DEPTH

These are faults associated with non-scan cells that require multiple clock cycles to propagate to an observe point. One possible solution is to increase the sequential depth using the “[set\\_pattern\\_type](#) -sequential” command. Another solution is to ensure that you have defined all clocks in the circuit using the [add\\_clocks](#) command.

## UD.AAB — ATPG\_ABORT

These are faults that are undetected because the tool reached its abort limit. You can gain additional information about UD.AAB faults using the [report\\_aborted\\_faults](#) command. You can raise the abort limit to increase coverage using the [set\\_abort\\_limit](#) command.

## UD.UNS — UNSUCCESS

These are faults that are undetected for unknown reasons. There is nothing you can do about faults in this sub-class.

## UD.EAB — EDT\_ABORT

These are faults that are undetected for one of these reasons:



- Your design's chain:channel ratio is insufficient to detect the fault. One solution is to run compression analysis on your design to determine the most aggressive chain:channel ratio using the [analyze\\_compression](#) command, and then re-create the EDT logic in your design. Another solution is to insert test points to improve the compressibility of faults.
- The tool cannot compress the test due to the scan cells being clustered (which is reported at the end of ATPG). The solution is to insert test points to improve the compressibility of faults.

## Fault Reporting

When reporting faults with the [report\\_faults](#) command, the tool identifies each fault by three ordered fields:

- fault value (0 for stuck-at-0 or “slow-to-rise” transition faults; 1 for stuck-at-1 or “slow-to-fall” transition faults)
- two-character fault class code
- pin pathname of the fault site

If the tools report uncollapsed faults, they display faults of a collapsed fault group together, with the representative fault first followed by the other members (with EQ fault codes).

## Testability Calculations

Given the fault classes explained in the previous sections, the tool makes the following calculations:

- **Test Coverage**

Test coverage, which is a measure of test quality, is the percentage of faults detected from among all *testable* faults. Typically, this is the number of most concern when you consider the testability of your design.

The tool calculates test coverage using the formula:

$$\frac{\#DT + (\#PD * \text{posdet\_credit})}{\#testable} \times 100$$

In this formula, `posdet_credit` is the user-selectable detection credit (the default is 50%) given to possible detected faults with the [set\\_possible\\_credit](#) command.

- **Fault Coverage**

Fault coverage consists of the percentage of faults detected from among all faults that the test pattern set tests—treating untestable faults the same as undetected faults.

The tool calculates fault coverage using the formula:

$$\#DT + (\#PD * \text{posdet\_credit})$$

$$\frac{\text{---}}{\#full} \times 100$$

- **ATPG Effectiveness**

ATPG effectiveness measures the ATPG tool's ability to either create a test for a fault, or prove that a test cannot be created for the fault under the restrictions placed on the tool.

The tool calculates ATPG effectiveness using the formula:

$$\frac{\#DT + \#UT + \#AU + \#PU + (\#PT * \text{posdet\_credit})}{\#full} \times 100$$

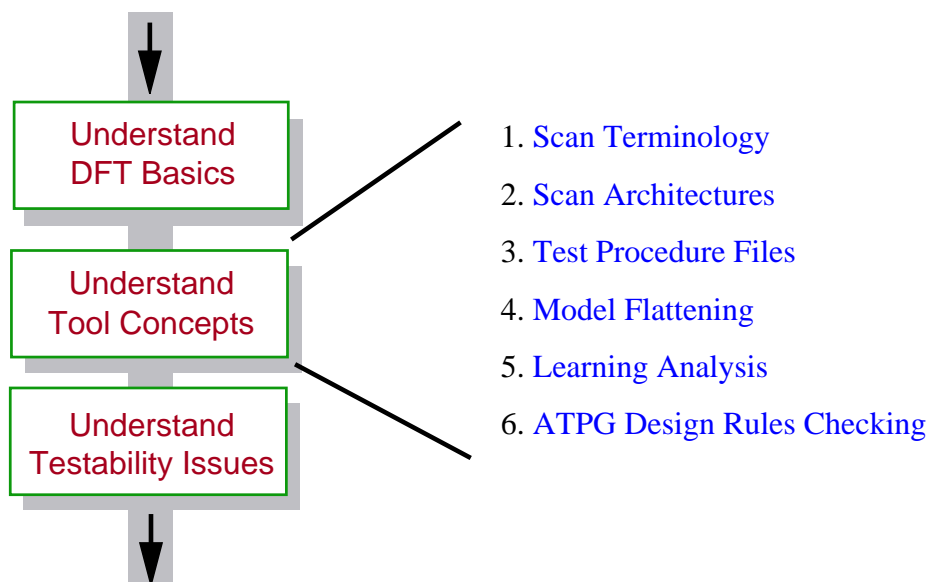
# Chapter 3

## Understanding Common Tool Terminology and Concepts

---

Now that you understand the basic ideas behind DFT, scan design, and ATPG, you can concentrate on the Mentor Graphics DFT tools and how they operate. Tessent Scan and the ATPG tools not only work toward a common goal (to improve test coverage), they also share common terminology, internal processes, and other tool concepts, such as how to view the design and the scan circuitry. [Figure 3-1](#) shows the range of subjects common to these tools.

**Figure 3-1. Common Tool Concepts**



The following subsections discuss common terminology and concepts associated with scan insertion and ATPG using Tessent Scan and ATPG tools.

## Scan Terminology

This section introduces the scan terminology common to Tessent Scan and the ATPG tools.

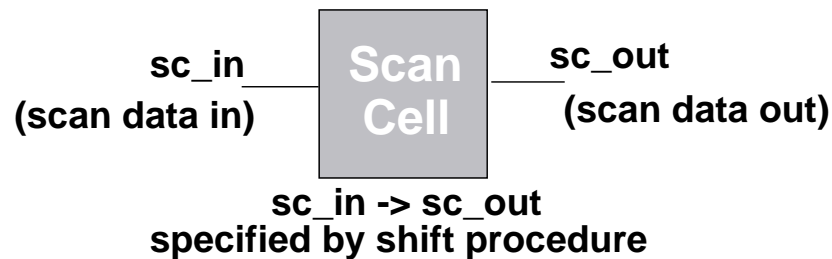
## Scan Cells

A *scan cell* is the fundamental, independently-accessible unit of scan circuitry, serving both as a control and observation point for ATPG and fault simulation. You can think of a scan cell as a black box composed of an input, an output, and a procedure specifying how data gets from the

input to the output. The circuitry inside the black box is not important as long as the specified procedure shifts data from input to output properly.

Because scan cell operation depends on an external procedure, scan cells are tightly linked to the notion of test procedure files. “[Test Procedure Files](#)” on page 75 discusses test procedure files in detail. [Figure 3-2](#) illustrates the black box concept of a scan cell and its reliance on a test procedure.

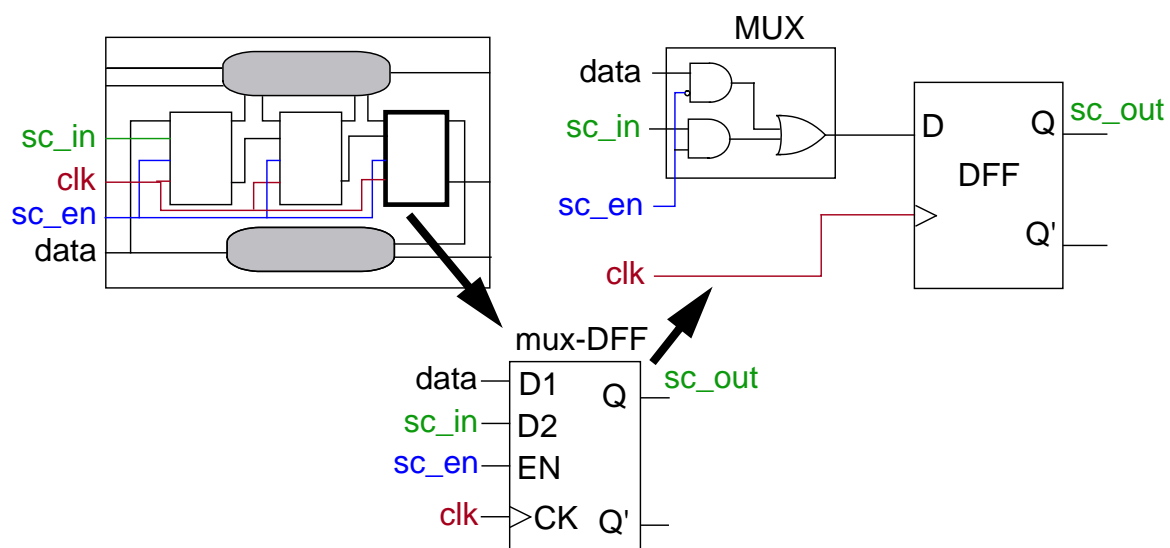
**Figure 3-2. Generic Scan Cell**



A scan cell contains at least one memory element (flip-flop or latch) that lies in the scan chain path. The cell can also contain additional memory elements that may or may not be in the scan chain path, as well as data inversion and gated logic between the memory elements.

[Figure 3-3](#) gives one example of a scan cell implementation (for the mux-DFF scan type).

**Figure 3-3. Generic Mux-DFF Scan Cell Implementation**



Each memory element may have a set and/or reset line in addition to clock-data ports. The ATPG process controls the scan cell by placing either normal or inverted data into its memory elements. The scan cell observation point is the memory element at the output of the scan cell. Other memory elements can also be observable, but may require a procedure for propagating

their values to the scan cell's output. The following subsections describe the different memory elements a scan cell may contain.

## Master Element

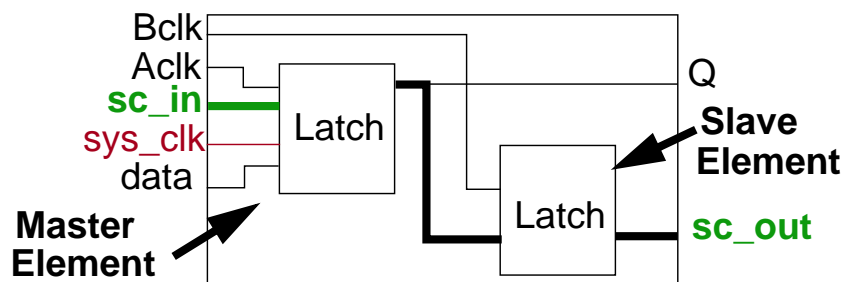
The *master element*, the primary memory element of a scan cell, captures data directly from the output of the previous scan cell. Each scan cell must contain one and only one master element. For example, Figure 3-3 shows a mux-DFF scan cell, which contains only a master element. However, scan cells can contain memory elements in addition to the master. Figures 3-4 through 3-7 illustrate examples of master elements in a variety of other scan cells.

The **shift** procedure in the test procedure file controls the master element. If the scan cell contains no additional independently-clocked memory elements in the scan path, this procedure also observes the master. If the scan cell contains additional memory elements, you may need to define a separate observation procedure (called **master\_observe**) for propagating the master element's value to the output of the scan cell.

## Slave Element

The *slave element*, an independently-clocked scan cell memory element, resides in the scan chain path. It cannot capture data directly from the previous scan cell. When used, it stores the output of the scan cell. The **shift** procedure both controls and observes the slave element. The value of the slave may be inverted relative to the master element. Figure 3-4 shows a slave element within a scan cell.

**Figure 3-4. LSSD Master/Slave Element Example**



In the example of Figure 3-4, Aclk controls scan data input. Activating Aclk, with sys\_clk (which controls system data) held off, shifts scan data into the scan cell. Activating Bclk propagates scan data to the output.

## Shadow Element

The *shadow element*, either dependently- or independently-clocked, resides outside the scan chain path. It can be inside or outside of a scan cell. Figure 3-5 gives an example of a scan cell with a dependently-clocked, non-observable shadow element with a non-inverted value.

**Figure 3-5. Dependently-clocked Mux-DFF/Shadow Element Example**

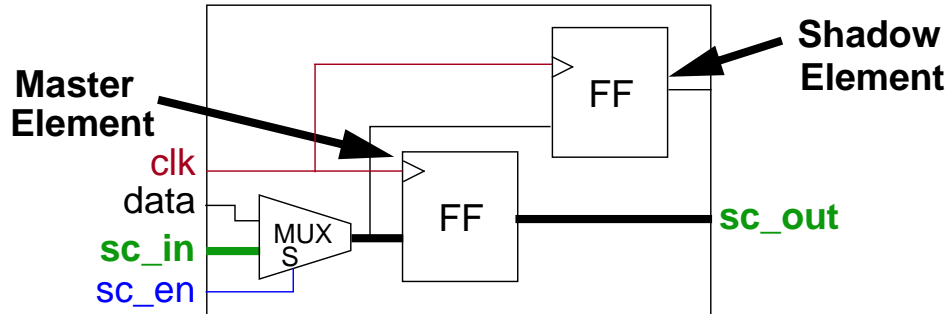
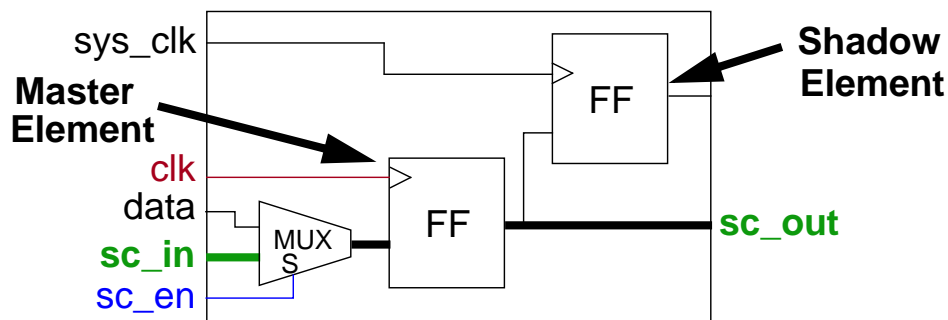


Figure 3-6 shows a similar example where the shadow element is independently-clocked.

**Figure 3-6. Independently-clocked Mux-DFF/Shadow Element Example**



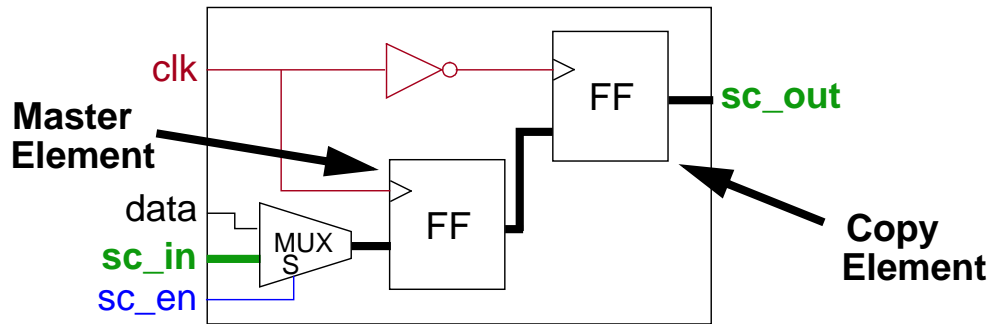
You load a data value into the dependently-clocked shadow element with the **shift** procedure. If the shadow element is independently clocked, you use a separate procedure called **shadow\_control** to load it. You can optionally make a shadow observable using the **shadow\_observe** procedure. A scan cell may contain multiple shadows but only one may be observable, because the tools allow only one **shadow\_observe** procedure. A shadow element's value may be the inverse of the master's value.

The definition of a shadow element is based on the shadow having the same (or inverse) value as the master element it shadows. A variety of interconnections of the master and shadow will accomplish this. In Figure 3-5, the shadow's data input is connected to the master's data input, and both FFs are triggered by the same clock edge. The definition would also be met if the shadow's data input was connected to the master's output and the shadow was triggered on the trailing edge, the master on the leading edge, of the same clock.

## Copy Element

The *copy element* is a memory element that lies in the scan chain path and can contain the same (or inverted) data as the associated master or slave element in the scan cell. Figure 3-7 gives an example of a copy element within a scan cell in which a master element provides data to the copy.

Figure 3-7. Mux-DFF/Copy Element Example



The clock pulse that captures data into the copy's associated scan cell element also captures data into the copy. Data transfers from the associated scan cell element to the copy element in the second half of the same clock cycle.

During the **shift** procedure, a copy contains the same data as that in its associated memory element. However, during system data capture, some types of scan cells allow copy elements to capture different data. When the copy's value differs from its associated element, the copy becomes the observation point of the scan cell. When the copy holds the same data as its associated element, the associated element becomes the observation point.

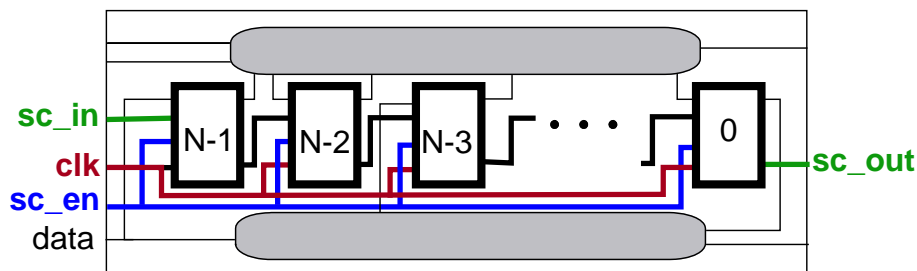
## Extra Element

The *extra element* is an additional, independently-clocked memory element of a scan cell. An extra element is any element that lies in the scan chain path between the master and slave elements. The **shift** procedure controls data capture into the extra elements. These elements are not observable. Scan cells can contain multiple extras. Extras can contain inverted data with respect to the master element.

## Scan Chains

A *scan chain* is a set of serially linked scan cells. Each scan chain contains an external input pin and an external output pin that provide access to the scan cells. Figure 3-8 shows a scan chain, with scan input "sc\_in" and scan output "sc\_out".

Figure 3-8. Generic Scan Chain

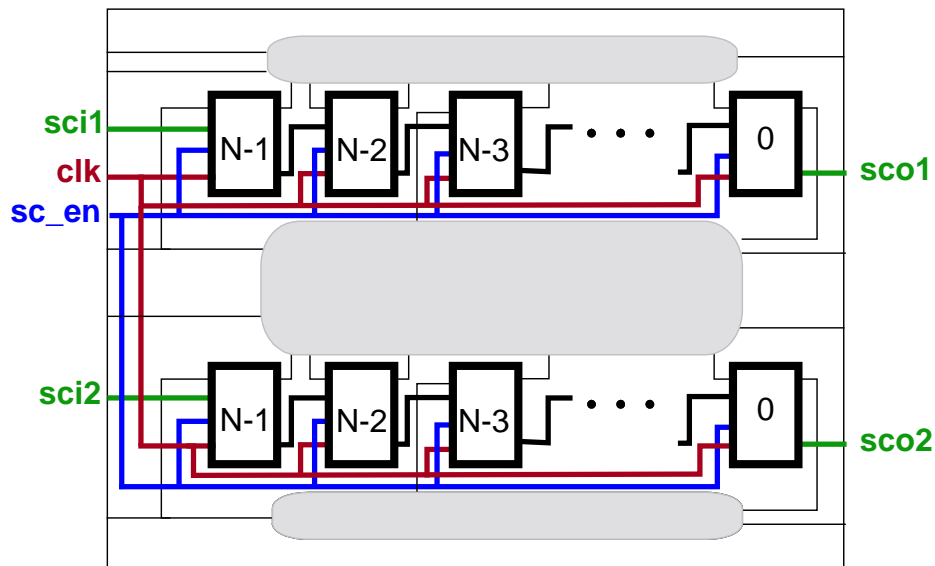


The scan chain length ( $N$ ) is the number of scan cells within the scan chain. By convention, the scan cell closest to the external output pin is number 0, its predecessor is number 1, and so on. Because the numbering starts at 0, the number for the scan cell connected to the external input pin is equal to the scan chain length minus one ( $N-1$ ).

## Scan Groups

A *scan chain group* is a set of scan chains that operate in parallel and share a common test procedure file. The test procedure file defines how to access the scan cells in all of the scan chains of the group. Normally, all of a circuit's scan chains operate in parallel and are thus in a single scan chain group.

**Figure 3-9. Generic Scan Group**



You may have two clocks, A and B, each of which clocks different scan chains. You often can clock, and therefore operate, the A and B chains concurrently, as shown in [Figure 3-9](#). However, if two chains share a single scan input pin, these chains cannot be operated in parallel. Regardless of operation, all defined scan chains in a circuit must be associated with a scan group. A scan group is a concept used by Mentor Graphics DFT and ATPG tools.

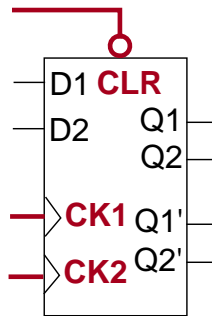
Scan groups are a way to group scan chains based on operation. All scan chains in a group must be able to operate in parallel, which is normal for scan chains in a circuit. However when scan chains cannot operate in parallel, such as in the example above (sharing a common scan input pin), the operation of each must be specified separately. This means the scan chains belong to different scan groups.



## Scan Clocks

*Scan clocks* are external pins capable of capturing values into scan cell elements. Scan clocks include set and reset lines, as well as traditional clocks. Any pin defined as a clock can act as a capture clock during ATPG. [Figure 3-10](#) shows a scan cell whose scan clock signals are shown in bold.

**Figure 3-10. Scan Clocks Example**



In addition to capturing data into scan cells, scan clocks, in their off state, ensure that the cells hold their data. Design rule checks ensure that clocks perform both functions. A clock's *off-state* is the primary input value that results in a scan element's clock input being at its inactive state (for latches) or state prior to a capturing transition (for edge-triggered devices). In the case of [Figure 3-10](#), the off-state for the CLR signal is 1, and the off-states for CK1 and CK2 are both 0.

## Scan Architectures

You can choose from a number of different scan types, or *scan architectures*. Tessent Scan supports the insertion of mux-DFF (mux-scan), clocked-scan, and LSSD architectures. Additionally, Tessent Scan supports all standard scan types, or combinations thereof, in designs containing pre-existing scan circuitry. You can use the [set\\_scan\\_type](#) command (see [page 123](#)) to specify the type of scan architecture you want inserted in your design.

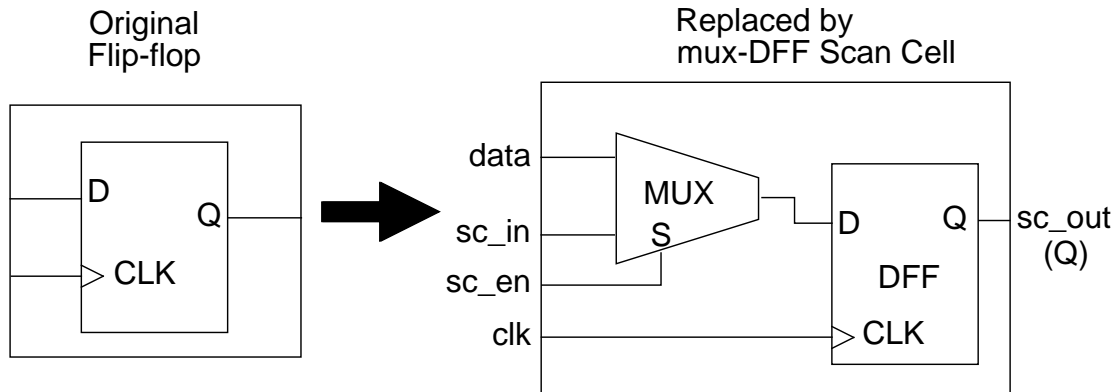
Each scan style provides different benefits. Mux-DFF or clocked-scan are generally the best choice for designs with edge-triggered flip-flops. Additionally, clocked-scan ensures data hold for non-scan cells during scan loading. LSSD is most effective on latch-based designs.

The following subsections detail the mux-DFF, clocked-scan, and LSSD architectures.

### Mux-DFF

A mux-DFF cell contains a single D flip-flop with a multiplexed input line that allows selection of either normal system data or scan data. [Figure 3-11](#) shows the replacement of an original design flip-flop with mux-DFF circuitry.

**Figure 3-11. Mux-DFF Replacement**

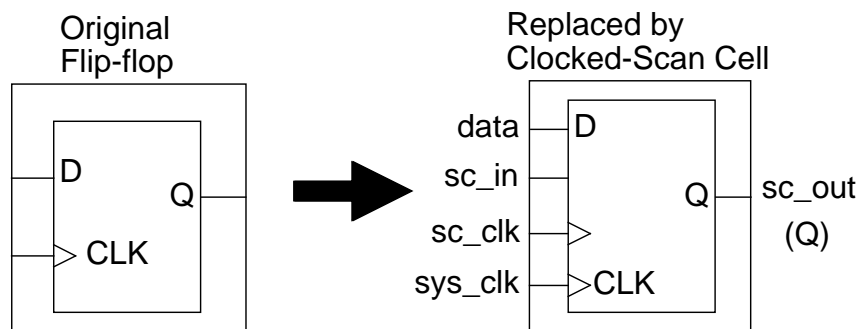


In normal operation ( $sc\_en = 0$ ), system data passes through the multiplexer to the D input of the flip-flop, and then to the output Q. In scan mode ( $sc\_en = 1$ ), scan input data ( $sc\_in$ ) passes to the flip-flop, and then to the scan output ( $sc\_out$ ).

## Clocked-Scan

The clocked-scan architecture is very similar to the mux-DFF architecture, but uses a dedicated test clock to shift in scan data instead of a multiplexer. Figure 3-12 shows an original design flip-flop replaced with clocked-scan circuitry.

**Figure 3-12. Clocked-Scan Replacement**

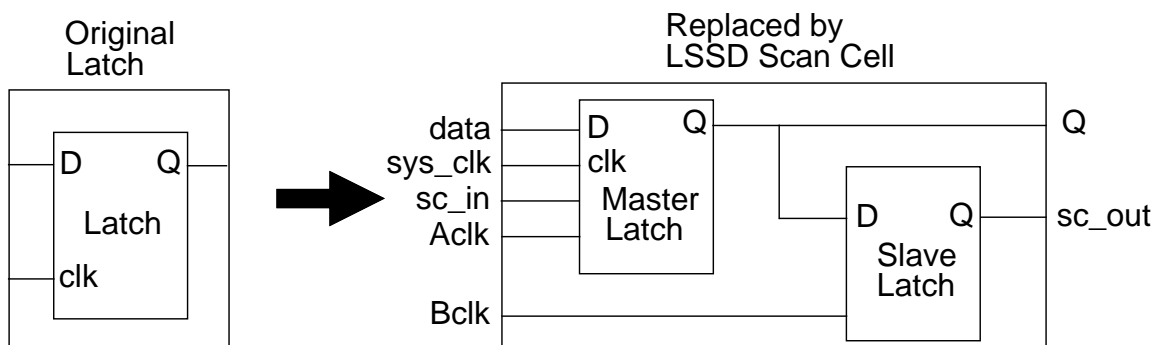


In normal operation, the system clock ( $sys\_clk$ ) clocks system data ( $data$ ) into the circuit and through to the output (Q). In scan mode, the scan clock ( $sc\_clk$ ) clocks scan input data ( $sc\_in$ ) into the circuit and through to the output ( $sc\_out$ ).

## LSSD

LSSD, or Level-Sensitive Scan Design, uses three independent clocks to capture data into the two polarity hold latches contained within the cell. Figure 3-13 shows the replacement of an original design latch with LSSD circuitry.

**Figure 3-13. LSSD Replacement**



In normal mode, the master latch captures system data (data) using the system clock (sys\_clk) and sends it to the normal system output (Q). In test mode, the two clocks (Aclk and Bclk) trigger the shifting of test data through both master and slave latches to the scan output (sc\_out).

There are several varieties of the LSSD architecture, including single latch, double latch, and clocked LSSD.

## Test Procedure Files

Test procedure files describe, for the ATPG tool, the scan circuitry operation within a design. Test procedure files contain cycle-based procedures and timing definitions that tell the ATPG tool how to operate the scan structures within a design. In order to use the scan circuitry in your design, you must do the following:

- Define the scan circuitry for the tool.
- Create a test procedure file to describe the scan circuitry operation. Tessent Scan can create test procedure files for you.
- Perform DRC process. This occurs when you exit from setup mode.

Once the scan circuitry operation passes DRC, the ATPG tool processes assume the scan circuitry works properly.

If your design contains scan circuitry, the ATPG tool requires a test procedure file. You must create one before running ATPG.

For more information about the test procedure file format, see [“Test Procedure File”](#) in the *Tessent Shell User’s Manual*, which describes the syntax and rules of test procedure files, give examples for the various types of scan architectures, and outline the checking that determines whether the circuitry is operating correctly.

## Model Flattening

To work properly, the ATPG tool and Tessent Scan must use their own internal representations of the design. The tools create these internal design models by flattening the model and replacing the design cells in the netlist (described in the library) with their own primitives. The tools flatten the model when you initially attempt to exit setup mode, just prior to design rules checking. The ATPG tool also provides the `create_flat_model` command, which allows flattening of the design model while still in setup mode.

If a flattened model already exists when you exit setup mode, the tools will only reflatten the model if you have since issued commands that would affect the internal representation of the design. For example, adding or deleting primary inputs, tying signals, and changing the internal faulting strategy are changes that affect the design model. With these types of changes, the tool must re-create or re-flatten the design model. If the model is undisturbed, the tool keeps the original flattened model and does not attempt to reflatten.

For a list of the specific Tessent Scan commands that cause flattening, refer to the [set\\_system\\_mode](#) description in the *Tessent Shell Reference Manual*.

- [create\\_flat\\_model](#) — Creates a primitive gate simulation representation of the design.
- [report\\_flattener\\_rules](#) — Displays either a summary of all the flattening rule violations or the data for a specific violation.
- [set\\_flattener\\_rule\\_handling](#) — Specifies how the tool handles flattening violations.

## Understanding Design Object Naming

Tessent Scan and the ATPG tool use special terminology to describe different objects in the design hierarchy. The following list describes the most common:

- *Instance* — A specific occurrence of a library model or functional block in the design.
- *Hierarchical instance* — An instance that contains additional instances and/or gates underneath it.
- *Module* — A Verilog functional block (module) that can be repeated multiple times. Each occurrence of the module is a hierarchical instance.

## The Flattening Process

The flattened model contains only simulation primitives and connectivity, which makes it an optimal representation for the processes of fault simulation and ATPG. Figure 3-14 shows an example of circuitry containing an AND-OR-Invert cell and an AND gate, before flattening.

**Figure 3-14. Design Before Flattening**

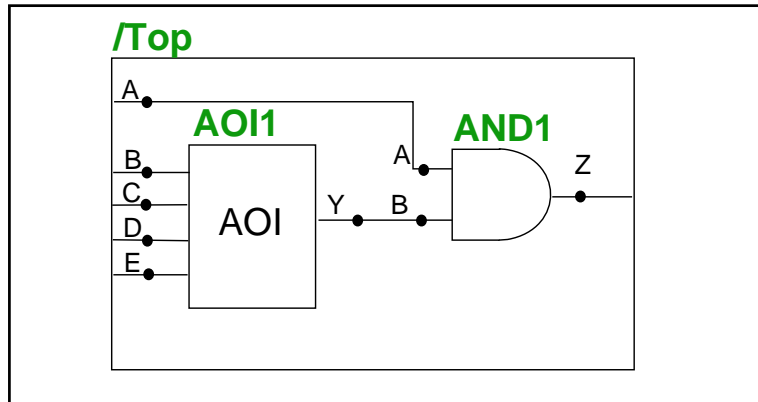
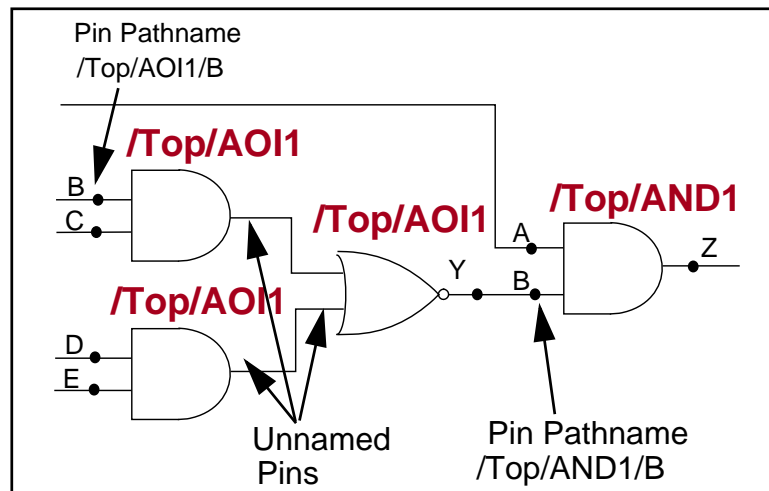


Figure 3-15 shows this same design once it has been flattened.

**Figure 3-15. Design After Flattening**



After flattening, only naming preserves the design hierarchy; that is, the flattened netlist maintains the hierarchy through instance naming. Figures 3-14 and 3-15 show this hierarchy preservation. */Top* is the name of the hierarchy's top level. The simulation primitives (two AND gates and a NOR gate) represent the flattened instance *AOI1* within */Top*. Each of these flattened gates retains the original design hierarchy in its naming—in this case, */Top/AOI1*.

The tools identify pins from the original instances by hierarchical pathnames as well. For example, */Top/AOI1/B* in the flattened design specifies input pin B of instance *AOI1*. This naming distinguishes it from input pin B of instance *AND1*, which has the pathname */Top/AND1/B*. By default, pins introduced by the flattening process remain unnamed and are not valid fault sites. If you request gate reporting on one of the flattened gates, the NOR gate for example, you will see a system-defined pin name shown in quotes. If you want internal faulting in your library cells, you must specify internal pin names within the library model. The flattening process then retains these pin names.

You should be aware that in some cases, the design flattening process can appear to introduce new gates into the design. For example, flattening decomposes a DFF gate into a DFF simulation primitive, the Q and Q' outputs require buffer and inverter gates, respectively. If your design wires together multiple drivers, flattening would add wire gates or bus gates. Bidirectional pins are another special case that requires additional gates in the flattened representation.

## Simulation Primitives of the Flattened Model

Tessent Scan and the ATPG tool select from a number of simulation primitives when they create the flattened circuitry. The simulation primitives are multiple-input (zero to four), single-output gates, except for the RAM, ROM, LA, and DFF primitives. The following list describes these simulation primitives:

- **PI, PO** - primary inputs are gates with no inputs and a single output, while primary outputs are gates with a single input and no fanout.
- **BUF** - a single-input gate that passes the values 0, 1, or X through to the output.
- **FB\_BUF** - a single-input gate, similar to the BUF gate, that provides a one iteration delay in the data evaluation phase of a simulation. The tools use the FB\_BUF gate to break some combinational loops and provide more optimistic behavior than when TIEX gates are used.

---

### Note



There can be one or more loops in a feedback path. In analysis mode, you can display the loops with the [report\\_loops](#) command. In setup mode, use [report\\_feedback\\_paths](#).

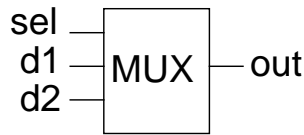
---

The default loop handling is simulation-based, with the tools using the FB\_BUF to break the combinational loops. In setup mode, you can change the default with the [set\\_loop\\_handling](#) command. Be aware that changes to loop handling will have an impact during the flattening process.

- **ZVAL** - a single-input gate that acts as a buffer unless Z is the input value. When a Z is the input value, the output is an X. You can modify this behavior with the [set\\_z\\_handling](#) command.
- **INV** - a single-input gate whose output value is the opposite of the input value. The INV gate cannot accept a Z input value.
- **AND, NAND** - multiple-input gates (two to four) that act as standard AND and NAND gates.
- **OR, NOR** - multiple-input (two to four) gates that act as standard OR and NOR gates.
- **XOR, XNOR** - 2-input gates that act as XOR and XNOR gates, except that when either input is an X, the output is an X.

- **MUX** - a 2x1 mux gate whose pins are order dependent, as shown in [Figure 3-16](#).

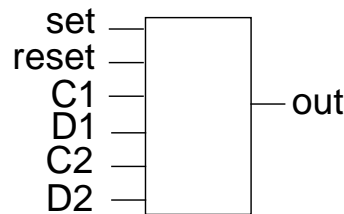
**Figure 3-16. 2x1 MUX Example**



The sel input is the first defined pin, followed by the first data input and then the second data input. When sel=0, the output is d1. When sel=1, the output is d2.

- **LA, DFF** - state elements, whose order dependent inputs include set, reset, and clock/data pairs, as shown in [Figure 3-17](#).

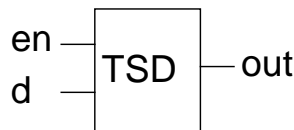
**Figure 3-17. LA, DFF Example**



Set and reset lines are always level sensitive, active high signals. DFF clock ports are edge-triggered while LA clock ports are level sensitive. When set=1, out=1. When reset=1, out=0. When a clock is active (for example C1=1), the output reflects its associated data line value (D1). If multiple clocks are active and the data they are trying to place on the output differs, the output becomes an X.

- **TLA, STLA, STFF** - special types of learned gates that act as, and pass the design rule checks for, transparent latch, sequential transparent latch, or sequential transparent flip-flop. These gates propagate values without holding state.
- **TIE0, TIE1, TIEX, TIEZ** - zero-input, single-output gates that represent the effect of a signal tied to ground or power, or a pin or state element constrained to a specific value (0,1,X, or Z). The rules checker may also determine that state elements exhibit tied behavior and replace them with the appropriate tie gates.
- **TSD, TSH** - a 2-input gate that acts as a tri-state™ driver, as shown in [Figure 3-18](#).

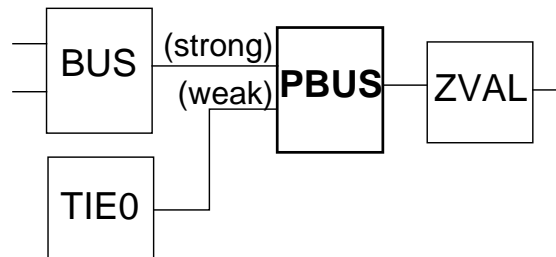
**Figure 3-18. TSD, TSH Example**



When en=1, out=d. When en=0, out=Z. The data line, d, cannot be a Z. The ATPG tool uses the TSD gate for the same purpose.

- **SW, NMOS** - a 2-input gate that acts like a tri-state driver but can also propagate a Z from input to output. The ATPG tool uses the SW gate uses the NMOS gate for the same purpose.
- **BUS** - a multiple-input (up to four) gate whose drivers must include at least one TSD or SW gate. If you bus more than four tri-state drivers together, the tool creates cascaded BUS gates. The last bus gate in the cascade is considered the dominant bus gate.
- **WIRE** - a multiple-input gate that differs from a bus in that none of its drivers are tri-statable.
- **PBUS, SWBUS** - a 2-input pull bus gate, for use when you combine strong bus and weak bus signals together, as shown in [Figure 3-19](#).

**Figure 3-19. PBUS, SWBUS Example**



The strong value always goes to the output, unless the value is a Z, in which case the weak value propagates to the output. These gates model pull-up and pull-down resistors. The ATPG tool uses the PBUS gate.

- **ZHOLD** - a single-input buskeeper gate (see [page 87](#) for more information on buskeepers) associated with a tri-state network that exhibits sequential behavior. If the input is a binary value, the gate acts as a buffer. If the input value is a Z, the output depends on the gate's hold capability. There are three ZHOLD gate types, each with a different hold capability:
  - **ZHOLD0** - When the input is a Z, the output is a 0 if its previous state was 0. If its previous state was a 1, the output is a Z.
  - **ZHOLD1** - When the input is a Z, the output is a 1 if its previous state was a 1. If its previous state was a 0, the output is a Z.
  - **ZHOLD0,1** - When the input is a Z, the output is a 0 if its previous state was a 0, or the output is a 1 if its previous state was a 1.

In all three cases, if the previous value is unknown, the output is X.

- **RAM, ROM**- multiple-input gates that model the effects of RAM and ROM in the circuit. RAM and ROM differ from other gates in that they have multiple outputs.



- **OUT** - gates that convert the outputs of multiple output gates (such as RAM and ROM simulation gates) to a single output.

## Learning Analysis

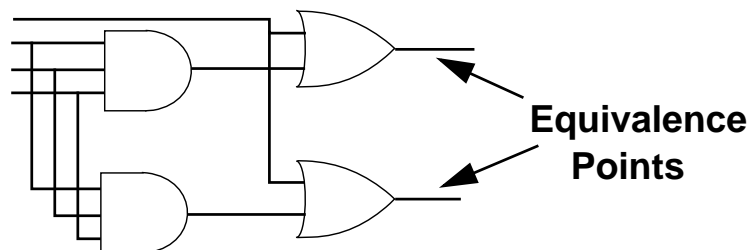
After design flattening, the ATPG tool performs extensive analysis on the design to learn behavior that may be useful for intelligent decision making in later processes, such as fault simulation and ATPG. You have the ability to turn learning analysis off, which may be desirable if you do not want to perform ATPG during the session. For more information on turning learning analysis off, refer to the [set\\_static\\_learning](#) description in the *Tessent Shell Reference Manual*.

The ATPG tools perform static learning only once—after flattening. Because pin and ATPG constraints can change the behavior of the design, static learning does not consider these constraints. Static learning involves gate-by-gate local simulation to determine information about the design. The following subsections describe the types of analysis performed during static learning.

## Equivalence Relationships

During this analysis, simulation traces back from the inputs of a multiple-input gate through a limited number of gates to identify points in the circuit that always have the same values in the good machine. [Figure 3-20](#) shows an example of two of these equivalence points within some circuitry.

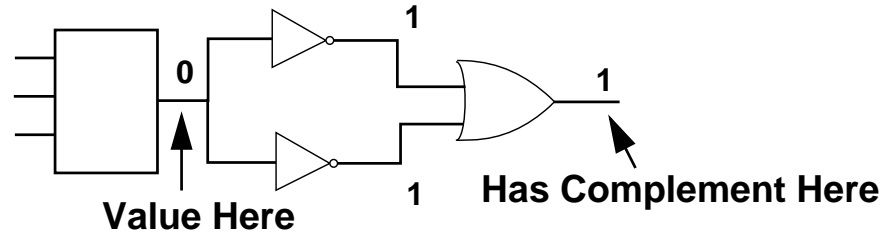
**Figure 3-20. Equivalence Relationship Example**



## Logic Behavior

During logic behavior analysis, simulation determines a circuit's functional behavior. For example, [Figure 3-21](#) shows some circuitry that, according to the analysis, acts as an inverter.

**Figure 3-21. Example of Learned Logic Behavior**



During gate function learning, the tool identifies the circuitry that acts as gate types TIE (tied 0, 1, or X values), BUF (buffer), INV (inverter), XOR (2-input exclusive OR), MUX (single select line, 2-data-line MUX gate), AND (2-input AND), and OR (2-input OR). For AND and OR function checking, the tool checks for busses acting as 2-input AND or OR gates. The tool then reports the learned logic gate function information with the messages:

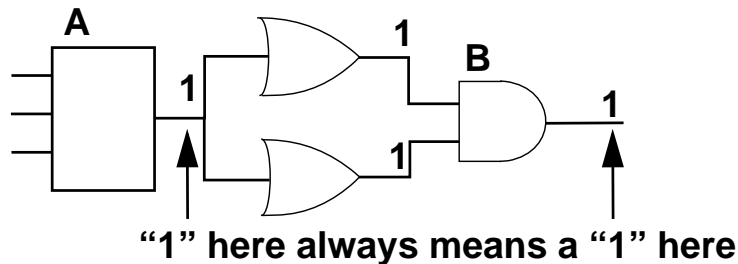
```
Learned gate functions:  #<gatetype>=<number> ...  
Learned tied gates:     #<gatetype>=<number> ...
```

If the analysis process yields no information for a particular category, it does not issue the corresponding message.

## Implied Relationships

This type of analysis consists of contrapositive relation learning, or learning implications, to determine that one value implies another. This learning analysis simulates nearly every gate in the design, attempting to learn every relationship possible. [Figure 3-22](#) shows the implied learning the analysis derives from a piece of circuitry.

**Figure 3-22. Example of Implied Relationship Learning**

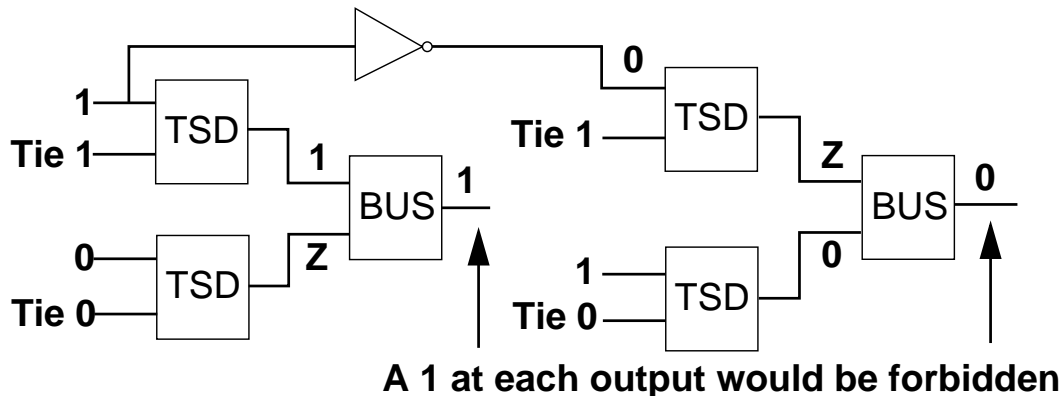


The analysis process can derive a very powerful relationship from this circuitry. If the value of gate A=1 implies that the value of gate B=1, then B=0 implies A=0. This type of learning establishes circuit dependencies due to reconvergent fanout and busses, which are the main obstacles for ATPG. Thus, implied relationship learning significantly reduces the number of bad ATPG decisions.

## Forbidden Relationships

During forbidden relationship analysis, which is restricted to bus gates, simulation determines that one gate cannot be at a certain value if another gate is at a certain value. [Figure 3-23](#) shows an example of such behavior.

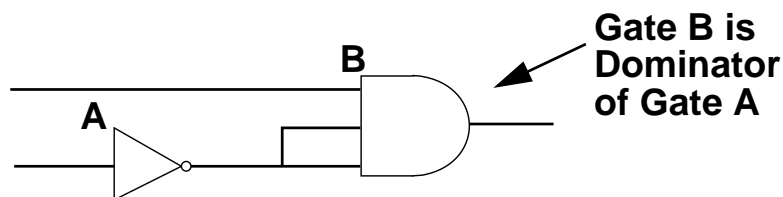
**Figure 3-23. Forbidden Relationship Example**



## Dominance Relationships

During dominance relationship analysis, simulation determines which gates are dominators. If all the fanouts of a gate go to a second gate, the second gate is the dominator of the first. [Figure 3-24](#) shows an example of this relationship.

**Figure 3-24. Dominance Relationship Example**



## ATPG Design Rules Checking

Tessent Scan and the ATPG tool perform design rules checking (DRC) after design flattening. While not all of the tools perform the exact same checks, design rules checking generally consists of the following processes, done in the order shown:

1. [General Rules Checking](#)
2. [Procedure Rules Checking](#)
3. [Bus Mutual Exclusivity Analysis](#)
4. [Scan Chain Tracing](#)

5. [Shadow Latch Identification](#)
6. [Data Rules Checking](#)
7. [Transparent Latch Identification](#)
8. [Clock Rules Checking](#)
9. [RAM Rules Checking](#)
10. [Bus Keeper Analysis](#)
11. [Extra Rules Checking](#)
12. [Scanability Rules Checking](#)
13. [Constrained/Forbidden/Block Value Calculations](#)

## General Rules Checking

General rules checking searches for very-high-level problems in the information defined for the design. For example, it checks to ensure the scan circuitry, clock, and RAM definitions all make sense. General rules violations are errors and you cannot change their handling. The “[General Rules](#)” section in the *Tessent Shell Reference Manual* describes the general rules in detail.

## Procedure Rules Checking

Procedure rules checking examines the test procedure file. These checks look for parsing or syntax errors and ensure adherence to each procedure’s rules. Procedure rules violations are errors and you cannot change their handling. The “[Procedure Rules](#)” section in the *Tessent Shell Reference Manual* describes the procedure rules in detail.

## Bus Mutual Exclusivity Analysis

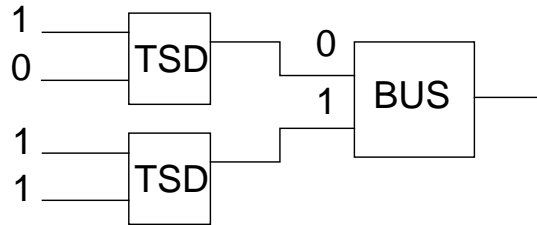
Buses in circuitry can cause the following problems for ATPG:

- Bus contention during ATPG
- Testing stuck-at faults on tri-state drivers of buses.

This section addresses the first concern, that ATPG must place buses in a non-contending state. For information on how to handle testing of tri-state devices, see “[Tri-State Devices](#)” on page 102.

[Figure 3-25](#) shows a bus system that can have contention.

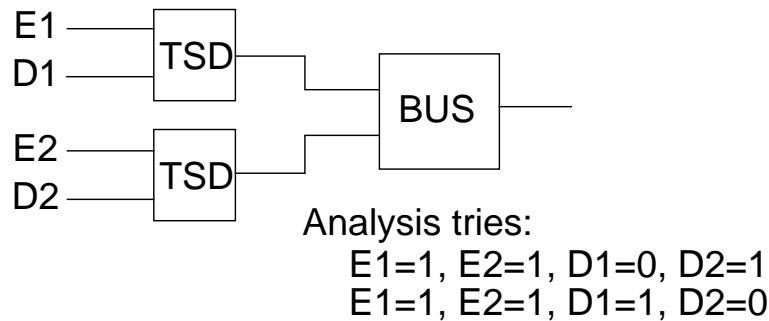
**Figure 3-25. Bus Contention Example**



Many designs contain buses, but good design practices usually prevent bus contention. As a check, the learning analysis for buses determines if a contention condition can occur within the given circuitry. Once learning determines that contention cannot occur, none of the later processes, such as ATPG, ever check for the condition.

Buses in a Z-state network can be classified as dominant or non-dominant and strong or weak. Weak buses and pull buses are allowed to have contention. Thus the process only analyzes strong, dominant buses, examining all drivers of these gates and performing full ATPG analysis of all combinations of two drivers being forced to opposite values. [Figure 3-26](#) demonstrates this process on a simple bus system.

**Figure 3-26. Bus Contention Analysis**



If ATPG analysis determines that either of the two conditions shown can be met, the bus fails bus mutual-exclusivity checking. Likewise, if the analysis proves the condition is never possible, the bus passes these checks. A third possibility is that the analysis aborts before it completes trying all of the possibilities. In this circuit, there are only two drivers, so ATPG analysis need try only two combinations. However, as the number of drivers increases, the ATPG analysis effort grows significantly.

You should resolve bus mutual-exclusivity before ATPG. Extra rules E4, E7, E9, E10, E11, E12, and E13 perform bus analysis and contention checking. Refer to [“Extra Rules”](#) in the *Tessent Shell Reference Manual* for more information on these bus checking rules.

## Scan Chain Tracing

The purpose of scan chain tracing is for the tool to identify the scan cells in the chain and determine how to use them for control and observe points. Using the information from the test procedure file (which has already been checked for general errors during the procedure rules checks) and the defined scan data, the tool identifies the scan cells in each defined chain and simulates the operation specified by the **load\_unload** procedure to ensure proper operation. Scan chain tracing takes place during the trace rules checks, which trace back through the sensitized path from output to input. Successful scan chain tracing ensures that the tools can use the cells in the chain as control and observe points during ATPG.

Trace rules violations are either errors or warnings, and for most rules you cannot change the handling. The “[Scan Chain Trace Rules](#)” section in the *Tessent Shell Reference Manual* describes the trace rules in detail.

## Shadow Latch Identification

Shadows are state elements that contain the same data as an associated scan cell element, but do not lie in the scan chain path. So while these elements are technically non-scan elements, their identification facilitates the ATPG process. This is because if a shadow element's content is the same as the associated element's content, you always know the shadow's state at that point. Thus, a shadow can be used as a control point in the circuit.

If the circuitry allows, you can also make a shadow an observation point by writing a **shadow\_observe** test procedure. The section entitled “[Shadow Element](#)” on page 69 discusses shadows in more detail.

The DRC process identifies shadow latches under the following conditions:

1. The element must not be part of an already identified scan cell.
2. Plus any one of the following:
  - At the time the clock to the shadow latch is active, there must be a single sensitized path from the data input of the shadow latch up to the output of a scan latch. Additionally the final shift pulse must occur at the scan latch no later than the clock pulse to the shadow latch (strictly before, if the shadow is edge triggered).
  - The shadow latch is loaded before the final shift pulse to the scan latch is identified by tracing back the data input of the shadow latch. In this case, the shadow will be a shadow of the next scan cell closer to scan out than the scan cell identified by tracing. If there is no scan cell close to scan out, then the sequential element is not a valid shadow.
  - The shadow latch is sensitized to a scan chain input pin during the last shift cycle. In this case, the shadow latch will be a shadow of the scan cell closest to scan in.

## Data Rules Checking

Data rules checking ensures the proper transfer of data within the scan chain. Data rules violations are either errors or warnings, however, you can change the handling. The “[Scan Cell Data Rules](#)” section in the *Tessent Shell Reference Manual* describes the data rules in detail.

## Transparent Latch Identification

Transparent latches are latches that can propagate values but do not hold state. A basic scan pattern contains the following events:

**Latch must behave as transparent between events 2 and 3** →

1. Load scan chain
2. Force values on primary inputs
3. Measure values on primary outputs
4. Pulse the capture clock
5. Unload the scan chain

Between the PI force and PO measure, the tool constrains all pins and sets all clocks off. Thus, for a latch to qualify as transparent, the analysis must determine that it can be turned on when clocks are off and pins are constrained. TLA simulation gates, which rank as combinational, represent transparent latches.

## Clock Rules Checking

After the scan chain trace, clock rules checking is the next most important analysis. Clock rules checks ensure data stability and capturability in the chain. Clock rules violations are either errors or warnings, however, you can change the handling. The “[Clock Rules](#)” section in the *Tessent Shell Reference Manual* describes the clock rules in detail.

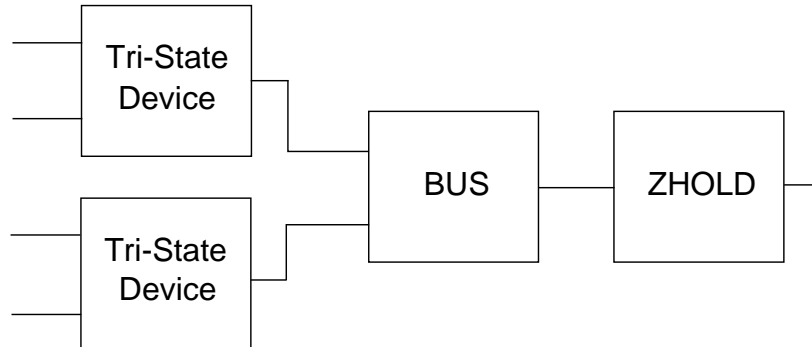
## RAM Rules Checking

RAM rules checking ensures consistency with the defined RAM information and the chosen testing mode. RAM rules violations are all warnings, however, you can change their handling. The “[RAM Rules](#)” section in the *Tessent Shell Reference Manual* describes the RAM rules in detail.

## Bus Keeper Analysis

Bus keepers model the ability of an undriven bus to retain its previous binary state. You specify bus keeper modeling with a **bus\_keeper** attribute in the model definition. When you use the **bus\_keeper** attribute, the tool uses a ZHOLD gate to model the bus keeper behavior during design flattening. In this situation, the design’s simulation model becomes that shown in [Figure 3-27](#):

**Figure 3-27. Simulation Model with Bus Keeper**



Rules checking determines the values of ZHOLD gates when clocks are off, pin constraints are set, and the gates are connected to clock, write, and read lines. ZHOLD gates connected to clock, write, and read lines do not retain values unless the clock off-states and constrained pins result in binary values.

During rules checking, if a design contains ZHOLD gates, messages indicate when ZHOLD checking begins, the number and type of ZHOLD gates, the number of ZHOLD gates connected to clock, write, and read lines, and the number of ZHOLD gates set to a binary value during the clock off-state condition.

---

**Note**



Only the ATPG tool requires this type of analysis, because of the way it “flattens” or simulates a number of events in a single operation.

---

For information on the bus\_keeper model attribute, refer to “[Attributes](#)” in the *Tessent Cell Library Manual*.

## Extra Rules Checking

Excluding rule E10, which performs bus mutual-exclusivity checking, most extra rules checks do not have an impact on Tessent Scan and the ATPG tool processes. However, they may be useful for enforcing certain design rules. By default, most extra rules violations are set to ignore, which means they are not even checked during DRC. However, you may change the handling. For more information, refer to “[Extra Rules](#)” in the *Tessent Shell Reference Manual* for more information.

## Scannability Rules Checking

Each design contains a certain number of memory elements. Tessent Scan examines all these elements and performs scannability checking on them, which consists mainly of the audits performed by rules S1, S2, S3, and S4. Scannability rules are all warnings, and you cannot



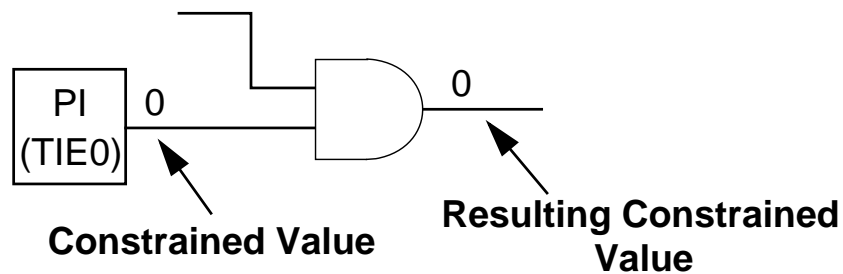
change their handling, except for the S3 rule. For more information, refer to “[Scanability Rules \(S Rules\)](#)” in the *Tessent Shell Reference Manual*.

## Constrained/Forbidden/Block Value Calculations

This analysis determines constrained, forbidden, and blocked circuitry. The checking process simulates forward from the point of the constrained, forbidden, or blocked circuitry to determine its effects on other circuitry. This information facilitates downstream processes, such as ATPG.

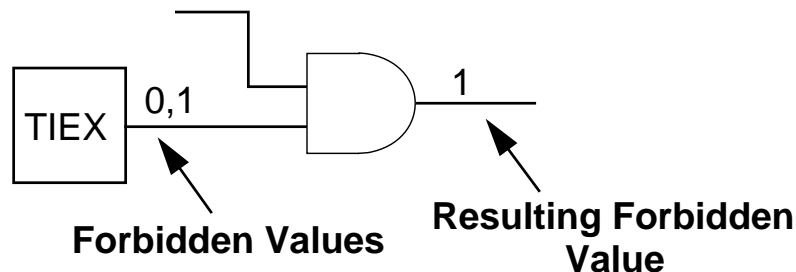
[Figure 3-28](#) gives an example of a tie value gate that constrains some surrounding circuitry.

**Figure 3-28. Constrained Values in Circuitry**



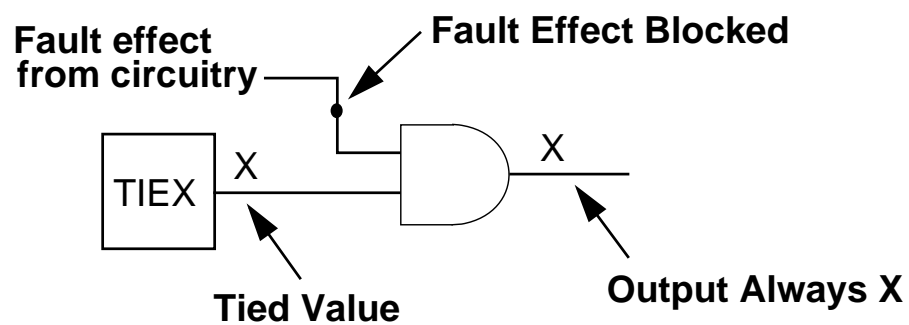
[Figure 3-29](#) gives an example of a tied gate, and the resulting forbidden values of the surrounding circuitry.

**Figure 3-29. Forbidden Values in Circuitry**



[Figure 3-30](#) gives an example of a tied gate that blocks fault effects in the surrounding circuitry.

**Figure 3-30. Blocked Values in Circuitry**



# Chapter 4

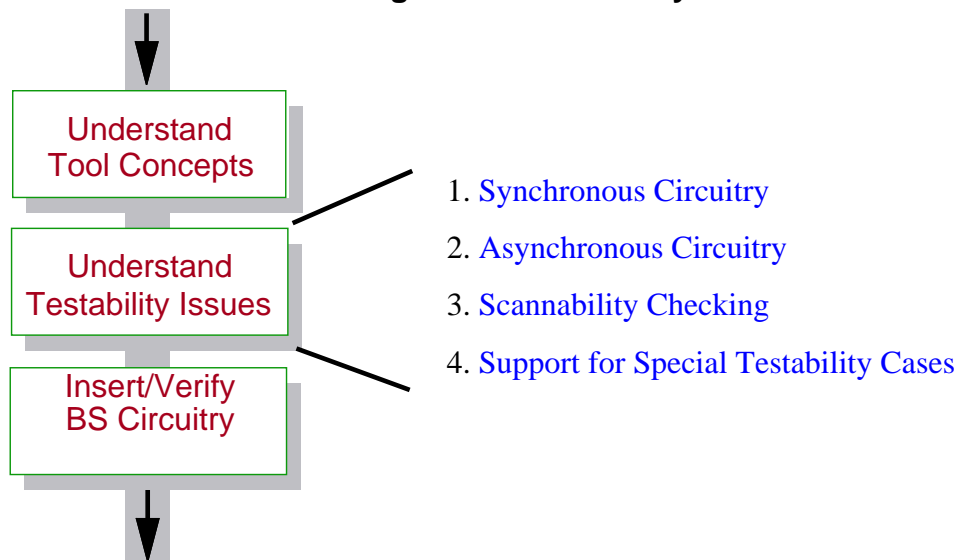
## Understanding Testability Issues

---

Testability naturally varies from design to design. Some features and design styles make a design difficult, if not impossible, to test, while others enhance a design's testability.

Figure 4-1 shows the testability issues this section discusses.

**Figure 4-1. Testability Issues**



The following subsections discuss these design features and describe their effect on the design's testability.

## Synchronous Circuitry

Using synchronous design practices, you can help ensure that your design will be both testable and manufacturable. In the past, designers used asynchronous design techniques with TTL and small PAL-based circuits. Today, however, designers can no longer use those techniques because the organization of most gate arrays and FPGAs necessitates the use of synchronous logic in their design.

A synchronous circuit operates properly and predictably in all modes of operation, from static DC up to the maximum clock rate. Inputs to the circuit do not cause the circuit to assume unknown states. And regardless of the relationship between the clock and input signals, the circuit avoids improper operation.

Truly synchronous designs are inherently testable designs. You can implement many scan strategies, and run the ATPG process with greater success, if you use synchronous design techniques. Moreover, you can create most designs following these practices with no loss of speed or functionality.

## Synchronous Design Techniques

Your design's level of synchronicity depends on how closely you observe the following techniques:

- The system has a minimum number of clocks—optimally only one.
- You register all design inputs and account for metastability. That is, you should treat the metastability time as another delay in the path. If the propagation delay plus the metastability time is less than the clock period, the system is synchronous. If it is greater than or equal to the clock period, you need to add an extra flip-flop to ensure the proper data enters the circuit.
- No combinational logic drives the set, reset, or clock inputs of the flip-flops.
- No asynchronous signals set or reset the flip-flops.
- Buffers or other delay elements do not delay clock signals.
- Do not use logic to delay signals.
- Do not assume logic delays are longer than routing delays.

If you adhere to these design rules, you are much more likely to produce a design that is manufacturable, testable, and operates properly over a wide range of temperature, voltage, and other circuit parameters.

## Asynchronous Circuitry

A small percentage of designs need some asynchronous circuitry due to the nature of the system. Because asynchronous circuitry is often very difficult to test, you should place the asynchronous portions of your design in one block and isolate it from the rest of the circuitry. In this way, you can still utilize DFT techniques on the synchronous portions of your design.

## Scannability Checking

Tessent Scan performs the scannability checking process on a design's sequential elements. For the tool to insert scan circuitry into a design, it must replace existing sequential elements with their scannable equivalents. Before beginning substitution, the original sequential elements in the design must pass *scannability checks*; that is, the tool determines if it can convert sequential elements to scan elements without additional circuit modifications. Scannable sequential elements pass the following checks:

1. When all clocks are off, all clock inputs (including set and reset inputs) of the sequential element must be in their inactive state (initial state of a capturing transition). This prevents disturbance of the scan chain data before application of the test pattern at the primary input. If the sequential element does not pass this check, its scan values could become unstable when the test tool applies primary input values. This checking is a modification of rule C1. For more information on this rule, refer to “C1” in the *Tessent Shell Reference Manual*.
2. Each clock input (*not* including set and reset inputs) of the sequential element must be capable of capturing data when a single clock primary input goes active while all other clocks are inactive. This rule ensures that this particular storage element can capture system data. If the sequential element does not meet this rule, some loss of test coverage could result. This checking is a modification of rule C7. For more information on this rule, refer to “C7” in the *Tessent Shell Reference Manual*.

When a sequential element passes these checks, it becomes a *scan candidate*, meaning that Tessent Scan can insert its scan equivalent into the scan chain. However, even if the element fails to pass one of these checks, it may still be possible to convert the element to scan. In many cases, you can add additional logic, called *test logic*, to the design to remedy the situation. For more information on test logic, refer to “[Enabling Test Logic Insertion](#)” on page 124.

---

### Note



If TIE0 and TIE1 nonscan cells are scannable, they are considered for scan. However, if these cells are used to hold off sets and resets of other cells so that another cell can be scannable, you must use the [add\\_nonscan\\_instances](#) command to make them nonscan.

---

## Scannability Checking of Latches

By default, Tessent Scan performs scannability checking on all flip-flops and latches. When latches do not pass scannability checks, Tessent Scan considers them non-scan elements and then classifies them into one of the categories explained in “Non-Scan Cell Handling” on page 103. However, if you want Tessent Scan to perform transparency checking on the non-scan latches, you must turn off checking of rule D6 prior to scannability checking. For more information on this rule, refer to “D6” in the *Tessent Shell Reference Manual*.

## Support for Special Testability Cases

The following subsections explain certain design features that can pose design testability problems and describe how Mentor Graphics DFT tools handle these situations.

### Feedback Loops

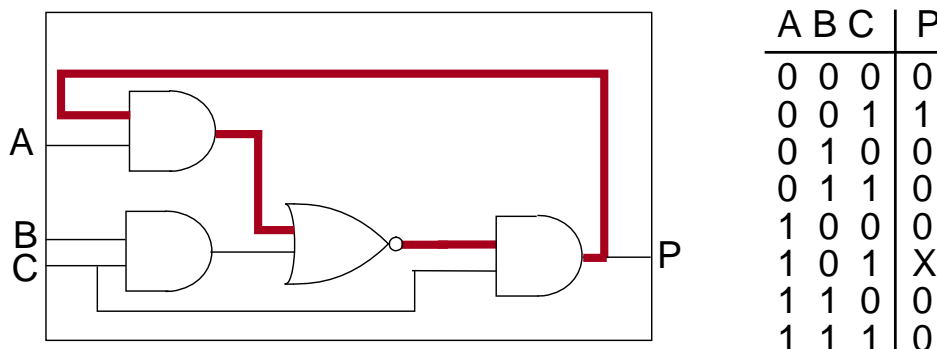
Designs containing loop circuitry have inherent testability problems. A *structural loop* exists when a design contains a portion of circuitry whose output, in some manner, feeds back to one of its inputs. A *structural combinational loop* occurs when the feedback loop, the path from the output back to the input, passes through only combinational logic. A *structural sequential loop* occurs when the feedback path passes through one or more sequential elements.

The ATPG tool and Tessent Scan all provide some common loop analysis and handling. However, loop treatment can vary depending on the tool. The following subsections discuss the treatment of structural combinational and structural sequential loops.

### Structural Combinational Loops and Loop-Cutting Methods

Figure 4-2 shows an example of a structural combinational loop. Notice that the A=1, B=0, C=1 state causes unknown (oscillatory) behavior, which poses a testability problem.

**Figure 4-2. Structural Combinational Loop Example**



The flattening process, which each tool runs as it attempts to exit setup mode, identifies and cuts, or breaks, all structural combinational loops. The tools classify and cut each loop using the appropriate methods for each category.

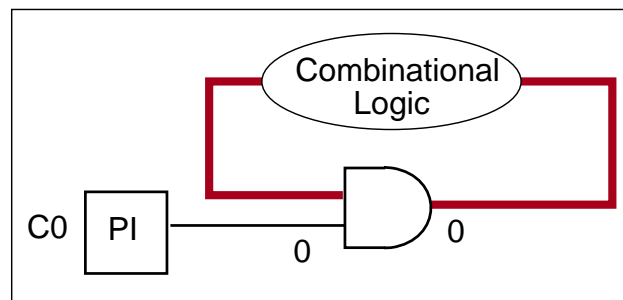
The following list presents the loop classifications, as well as the loop-cutting methods established for each. The order of the categories presented indicates the least to most pessimistic loop cutting solutions.

#### 1. Constant value

This loop cutting method involves those loops blocked by tied logic or pin constraints. After the initial loop identification, the tools simulate TIE0/TIE1 gates and constrained inputs. Loops containing constant value gates as a result of this simulation, fall into this category.

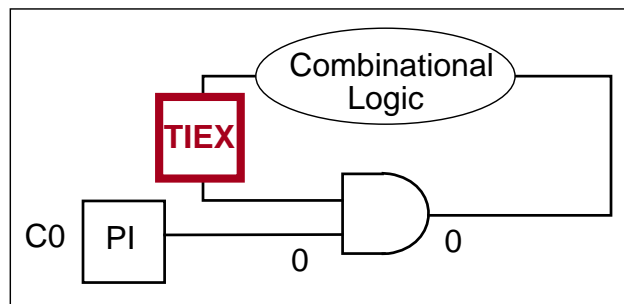
Figure 4-3 shows a loop with a constrained primary input value that blocks the loop's feedback effects.

**Figure 4-3. Loop Naturally-Blocked by Constant Value**



These types of loops lend themselves to the simplest and least pessimistic breaking procedures. For this class of loops, the tool inserts a TIE-X gate at a non-constrained input (which lies in the feedback path) of the constant value gate, as Figure 4-4 shows.

**Figure 4-4. Cutting Constant Value Loops**



This loop cutting technique yields good circuit simulation that always matches the actual circuit behavior, and thus, the tools employ this technique whenever possible. The tools can use this loop cutting method for blocked loops containing AND, OR, NAND, and

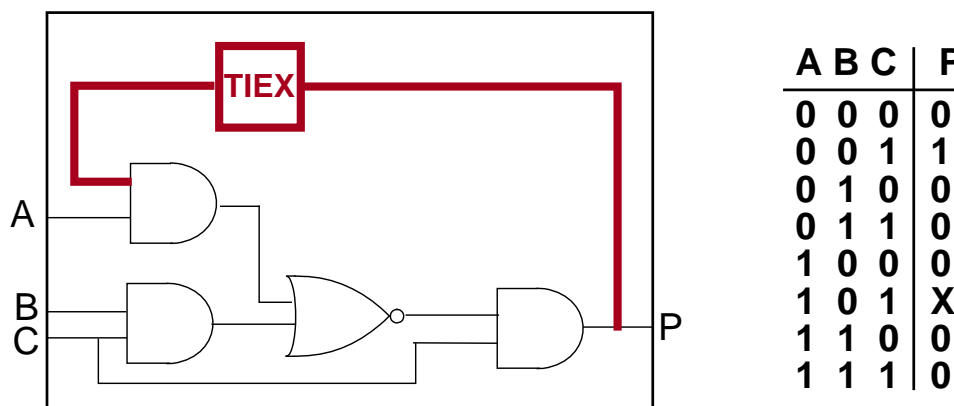
NOR gates, as well as MUX gates with constrained select lines and tri-state drivers with constrained enable lines.

## 2. Single gate with “multiple fanout”

This loop cutting method involves loops containing only a single gate with multiple fanout.

Figure 4-2 on page 94 shows the circuitry and truth table for a single multiple-fanout loop. For this class of loops, the tool cuts the loop by inserting a TIE-X gate at one of the fanouts of this “multiple fanout gate” that lie in the loop path, as Figure 4-5 shows.

**Figure 4-5. Cutting Single Multiple-Fanout Loops**

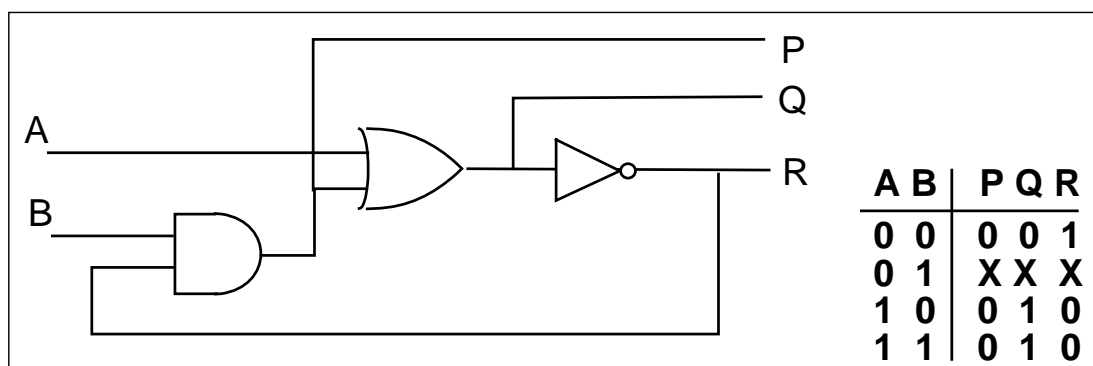


## 3. Gate duplication for multiple gate with multiple fanout

This method involves duplicating some of the loop logic—when it proves practical to do so. The tools use this method when it can reduce the simulation pessimism caused by breaking combinational loops with TIE-X gates. The process analyzes a loop, picks a connection point, duplicates the logic (inserting a TIE-X gate into the copy), and connects the original circuitry to the copy at the connection point.

Figure 4-6 shows a simple loop that the tools would target for gate duplication.

**Figure 4-6. Loop Candidate for Duplication**







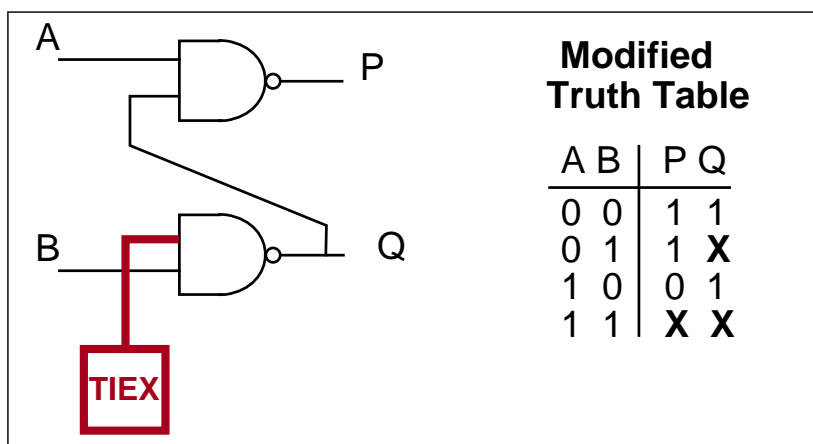
Additionally, this technique can prove costly in terms of gate count as the loop size increases. Also, the tools cannot use this method on complex or *coupled loops*—those loops that connect with other loops (because gate duplication may create loops as well).

#### 4. Coupling loops

The tools use this technique to break loops when two or more loops share a common gate. This method involves inserting a TIE-X gate at the input of one of the components within a loop. The process selects the cut point carefully to ensure the TIE-X gate cuts as many of the coupled loops as possible.

For example, assume the SR latch shown in [Figure 4-6](#) was part of a larger, more complex, loop coupling network. In this case, loop circuitry duplication would turn into an iterative process that would never converge. So, the tools would have to cut the loop as shown in [Figure 4-9](#).

**Figure 4-9. Cutting Coupling Loops**



The modified truth table shown in [Figure 4-9](#) demonstrates that this method yields the most pessimistic simulation results of all the loop-cutting methods. Because this is the most pessimistic solution to the loop cutting problem, the tools only use this technique when they cannot use any of the previous methods.

## ATPG-Specific Combinational Loop Handling Issues

By default, the ATPG tool performs parallel pattern simulation of circuits containing combinational feedback networks. This is controlled by using the [set\\_loop\\_handling](#) command.

A learning process identifies feedback networks after flattening, and an iterative simulation is used in the feedback network. For an iterative simulation, the ATPG tool inserts FB\_BUF gates to break the combinational loops.

The ATPG tool also has the ability to insert TIE-X gates to break the combinational loops. The gate duplication option reduces the impact that a TIE-X gate places on the circuit to break combinational loops. By default, this duplication switch is off.

### Note



The `set_loop_handling` command replaces functionality previously available by the Set Loop Duplication command.

## Tessent Scan-Specific Combinational Loop Handling Issues

Tessent Scan identifies combinational loops during flattening. By default, it performs TIE-X insertion using the methods specified in “[Structural Combinational Loops and Loop-Cutting Methods](#)” on page 94 to break all loops detected by the initial loop analysis. You can turn loop duplication off using the `set_loop_duplication` command.

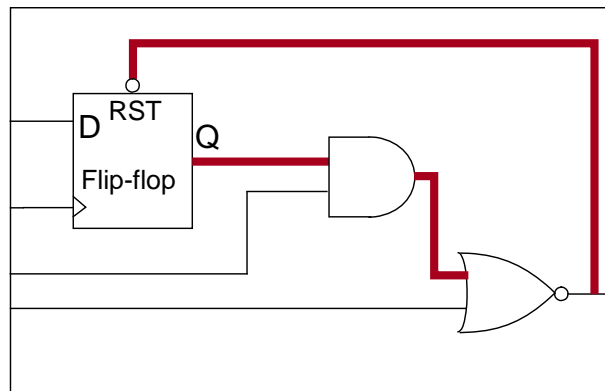
You can report on loops using the `report_loops` or the `report_feedback_paths` commands. While both involved with loop reporting, these commands behave somewhat differently. Refer to the [Tessent Shell Reference Manual](#) for details. You can write all identified structural combinational loops to a file using the `write_loops` command.

You can use the loop information Tessent Scan provides to handle each loop in the most desirable way. For example, assuming you wanted to improve the test coverage for a coupling loop, you could use the `add_control_points/add_observe_points` commands within Tessent Scan to insert a test point to control or observe values at a certain location within the loop.

## Structural Sequential Loops and Handling

Sequential feedback loops occur when the output of a latch or flip-flop feeds back to one of its inputs, either directly or through some other logic. [Figure 4-10](#) shows an example of a structural sequential feedback loop.

**Figure 4-10. Sequential Feedback Loop**



### Note



The tools model RAM and ROM gates as combinational gates, and thus, they consider loops involving only combinational gates and RAMs (or ROMs) as combinational loops—not sequential loops.

The following sections provide tool-specific issues regarding sequential loop handling.

## ATPG-Specific Sequential Loop Handling

While the ATPG tool can suffer some loss of test coverage due to sequential loops, these loops do not cause the tool the extensive problems that combinational loops do. By its very nature, the ATPG tool re-models the non-scan sequential elements in the design using the simulation primitives described in “[ATPG Handling of Non-Scan Cells](#)” on page 104. Each of these primitives, when inserted, automatically breaks the loops in some manner.

Within the ATPG tool, sequential loops typically trigger C3 and C4 design rules violations. When one sequential element (a source gate) feeds a value to another sequential element (a sink gate), the tool simulates old data at the sink. You can change this simulation method using the [set\\_capture\\_handling](#) command. For more information on the C3 and C4 rules, refer to “[Clock Rules](#)” in the *Tessent Shell Reference Manual*. For more information about the [set\\_capture\\_handling](#) command, refer to its description in the *Tessent Shell Reference Manual*.

## Redundant Logic

In most cases, you should avoid using redundant logic because a circuit with redundant logic poses testability problems. First, classifying redundant faults takes a great deal of analysis effort. Additionally, redundant faults, by their nature, are untestable and therefore lower your fault coverage. [Figure 2-18](#) on page 58 gives an example of redundant circuitry.

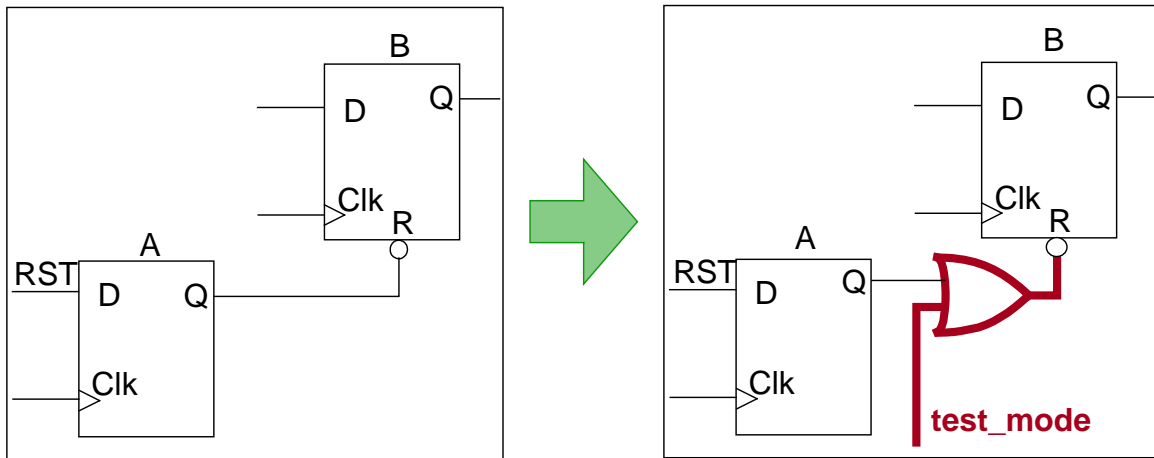
Some circuitry requires redundant logic; for example, circuitry to eliminate race conditions or circuitry which builds high reliability into the design. In these cases, you should add test points to remove redundancy during the testing process.

## Asynchronous Sets and Resets

Scannability checking treats sequential elements driven by uncontrollable set and reset lines as unscannable. You can remedy this situation in one of two ways: you can add test logic to make the signals controllable, or you can use initialization patterns during test to control these internally-generated signals. Tessent Scan provides capabilities to aid you in both solutions.

[Figure 4-11](#) shows a situation with an asynchronous reset line and the test logic added to control the asynchronous reset line.

**Figure 4-11. Test Logic Added to Control Asynchronous Reset**



In this example, Tessent Scan adds an OR gate that uses the `test_mode` (not `scan_enable`) signal to keep the reset of flip-flop B inactive during the testing process. You would then constrain the `test_mode` signal to be a 1, so flip-flop B could never be reset during testing. To insert this type of test logic, you can use the Tessent Scan command `set_test_logic` (see [page 124](#) for more information).

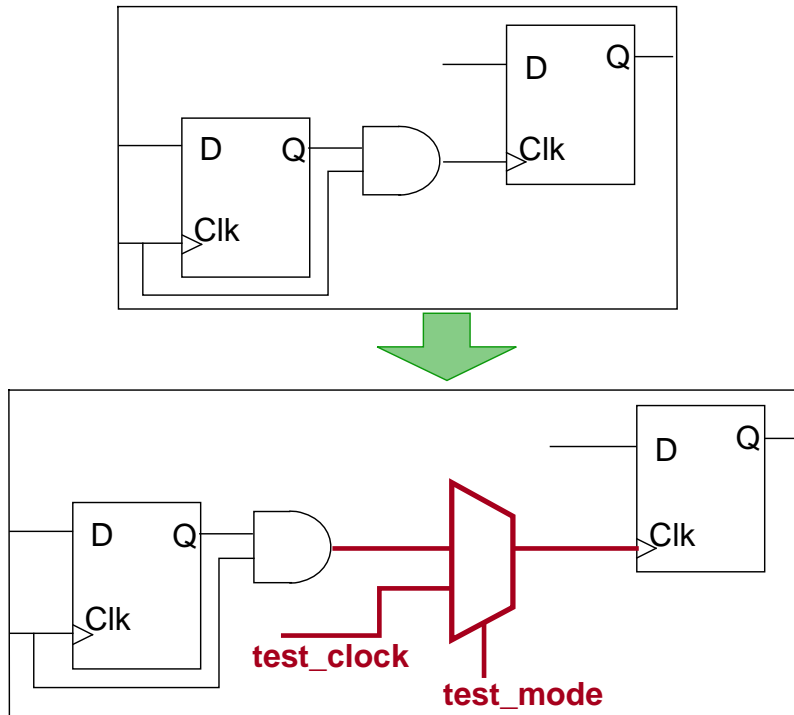
Tessent Scan also allows you to specify an initialization sequence in the test procedure file to avoid the use of this additional test logic. For additional information, refer to the [add\\_scan\\_groups](#) description in the *Tessent Shell Reference Manual*.

## Gated Clocks

Primary inputs typically cannot control the gated clock signals of sequential devices. In order to make some of these sequential elements scannable, you may need to add test logic to modify their clock circuitry.

For example, [Figure 4-12](#) shows an example of a clock that requires some test logic to control it during test mode.

**Figure 4-12. Test Logic Added to Control Gated Clock**

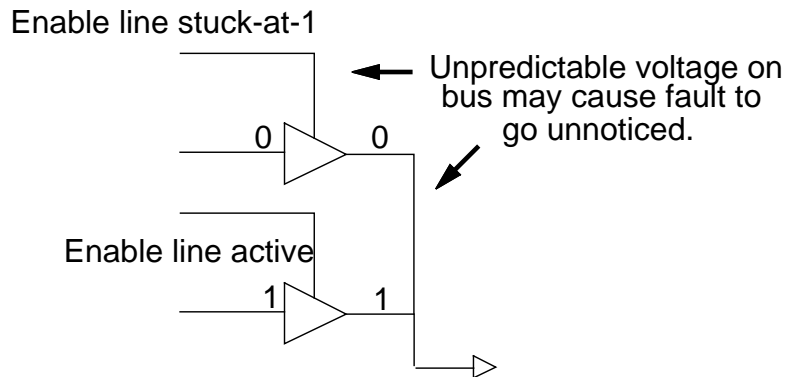


In this example, Tessent Scan makes the element scannable by adding a test clock, for both scan loading/unloading and data capture, and multiplexing it with the original clock signal. It also adds a signal called `test_mode` to control the added multiplexer. The `test_mode` signal differs from the `scan_mode` or `scan_enable` signals in that it is active during the entire duration of the test—not just during scan chain loading/unloading. To add this type of test logic into your design, you can use the `set_test_logic` and `set_scan_insertion` commands. For more information about these commands, refer to pages [124](#) and [142](#), respectively.

## Tri-State Devices

Tri-state™ buses are another testability challenge. Faults on tri-state bus enables can cause one of two problems: *bus contention*, which means there is more than one active driver, or *bus float*, which means there is no active driver. Either of these conditions can cause unpredictable logic values on the bus, which allows the enable line fault to go undetected. [Figure 4-13](#) shows a tri-state bus with bus contention caused by a stuck-at-1 fault.

**Figure 4-13. Tri-state Bus Contention**



Tessent Scan can add gating logic that turns off the tri-state devices during scan chain shifting. The tool gates the tri-state device enable lines with the `scan_enable` signal so they are inactive and thus prevent bus contention during scan data shifting. To insert this type of gating logic, you can use the Tessent Scan command `set_tristate_gating` (see [page 124](#) for more information).

In addition, the ATPG tool lets you specify the fault effect of bus contention on tri-state nets. This capability increases the testability of the enable line of the tri-state drivers. Refer to the [set\\_net\\_dominance](#) description in the *Tessent Shell Reference Manual* for details.

## Non-Scan Cell Handling

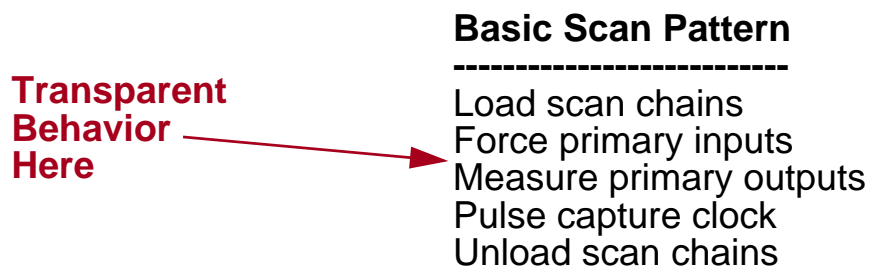
During rules checking and learning analysis, the ATPG tool learns the behavior of all state elements that are not part of the scan circuitry. This learning involves how the non-scan element behaves after the scan loading operation. As a result of the learning analysis, the ATPG tool categorizes each of the non-scan cells.

## ATPG Handling of Non-Scan Cells

The ATPG tool places non-scan cells in one of the following categories:

- **TIEX** — In this category, the ATPG tool considers the output of a flip-flop or latch to always be an X value during test. This condition may prevent the detection of a number of faults.
- **TIE0** — In this category, the ATPG tool considers the output of a flip-flop or latch to always be a 0 value during test. This condition may prevent the detection of a number of faults.
- **TIE1** — In this category, the ATPG tool considers the output of a flip-flop or latch to always be a 1 value during test. This condition may prevent the detection of a number of faults.
- **Transparent (combinational)** — In this category, the non-scan cell is a latch, and the latch behaves transparently. When a latch behaves transparently, it acts, in effect, as a buffer—passing the data input value to the data output. The TLA simulation gate models this behavior. [Figure 4-14](#) shows the point at which the latch must exhibit transparent behavior.

**Figure 4-14. Requirement for Combinationally Transparent Latches**



Transparency occurs if the clock input of the latch is inactive during the time between the force of the primary inputs and the measure of the primary outputs. If your latch is set up to behave transparently, you should not experience any significant fault detection problems (except for faults on the clock, set, and reset lines). However, only in limited cases do non-scan cells truly behave transparently. For the tool to consider the latch transparent, it must meet the following conditions:

- The latch must not create a potential feedback path, unless the path is broken by scan cells or non-scan cells (other than transparent latches).
- The latch must have a path that propagates to an observable point.
- The latch must be able to pass a data value to the output when all clocks are off.
- The latch must have clock, set, and reset signals that can be set to a determined value.



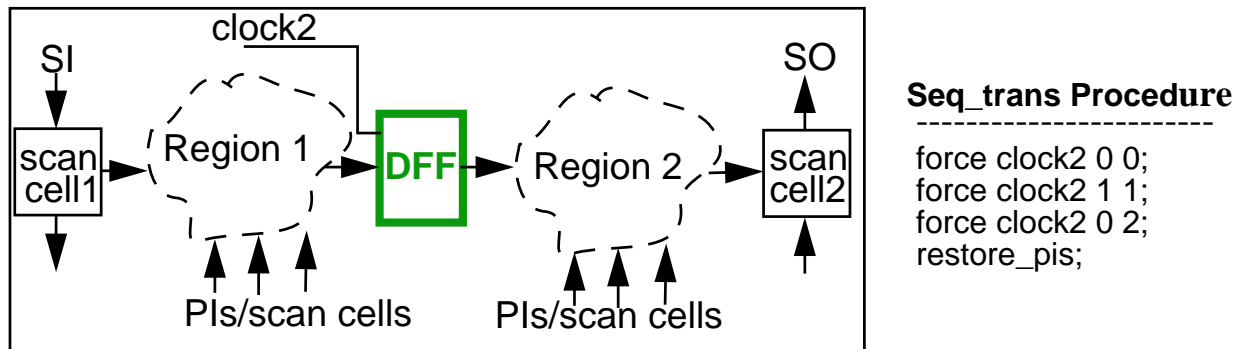
For more information on the transparent latch checking procedure, refer to “D6” in the *Tessent Shell Reference Manual*.

- **Sequential transparent** — Sequential transparency extends the notion of transparency to include non-scan elements that can be forced to behave transparently at the same point in which natural transparency occurs. In this case, the non-scan element can be either a flip-flop, a latch, or a RAM read port. A non-scan cell behaves as sequentially transparent if, given a sequence of events, it can capture a value and pass this value to its output, without disturbing critical scan cells.

Sequential transparent handling of non-scan cells lets *you* describe the events that place the non-scan cell in transparent mode. You do this by specifying a procedure, called **seq\_transparent**, in your test procedure file. This procedure contains the events necessary to create transparent behavior of the non-scan cell(s). After the tool loads the scan chain, forces the primary inputs, and forces all clocks off, the **seq\_transparent** procedure pulses the clocks of all the non-scan cells or performs other specified events to pass data through the cell “transparently”.

Figure 4-15 shows an example of a scan design with a non-scan element that is a candidate for sequential transparency.

**Figure 4-15. Example of Sequential Transparency**



The DFF shown in Figure 4-15 behaves sequentially transparent when the tool pulses its clock input, clock2. The sequential transparent procedure shows the events that enable transparent behavior.

**Note** To be compatible with combinational ATPG, the value on the data input line of the non-scan cell must have combinational behavior, as depicted by the combinational Region 1. Also, the output of the state element, in order to be useful for ATPG, must propagate to an observable point.

Benefits of sequential transparent handling include more flexibility of use compared to transparent handling, and the ability to use this technique for creating “structured partial scan” (to minimize area overhead while still obtaining predictable high test coverage).

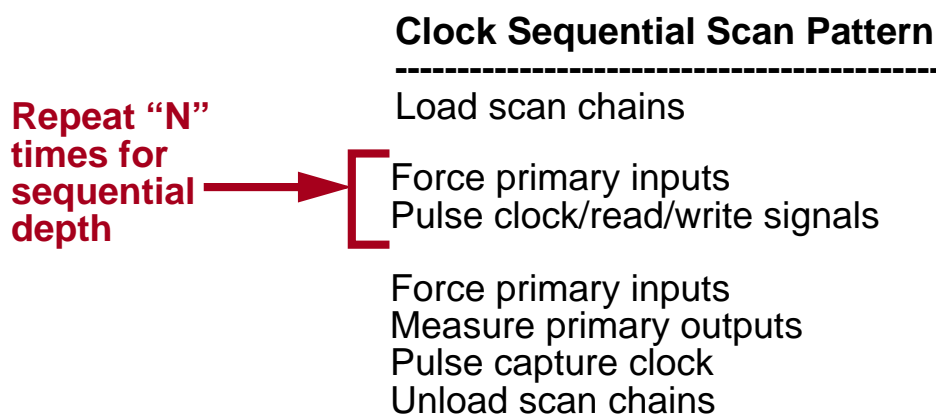
Also, the notion of sequential transparency supports the design practice of using a cell called a *transparent slave*. A transparent slave is a non-scan latch that uses the slave clock to capture its data. Additionally, you can define and use up to 32 different, uniquely-named **seq\_transparent** procedures in your test procedure file to handle the various types of non-scan cell circuitry in your design.

Rules checking determines if non-scan cells qualify for sequential transparency via these procedures. Specifically, the cells must satisfy rules P5, P6, P41, P44, P45, P46, D3, and D9. For more information on these rules, refer to “[Design Rule Checking](#)” in the *Tessent Shell Reference Manual*. Clock rules checking treats sequential transparent elements the same as scan cells.

Limitations of sequential transparent cell handling include the following:

- Impaired ability to detect AC defects (transition fault type causes sequential transparent elements to appear as tie-X gates).
  - Cannot make non-scan cells clocked by scan cells sequentially transparent without **condition** statements.
  - Limited usability of the sequential transparent procedure if applying it disturbs the scan cells (contents of scan cells change during the **seq\_transparent** procedure).
  - Feedback paths to non-scan cells, unless broken by scan cells, prevent treating the non-scan cells as sequentially transparent.
- **Clock sequential** — If a non-scan cell obeys the standard scan clock rules—that is, if the cell holds its value with all clocks off—the tool treats it as a clock sequential cell. In this case, after the tool loads the scan chains, it forces the primary inputs and pulses the clock/write/read lines multiple times (based on the sequential depth of the non-scan cells) to set up the conditions for a test. A normal observe cycle then follows. [Figure 4-16](#) shows a clock sequential scan pattern.

**Figure 4-16. Clocked Sequential Scan Pattern Events**



This technique of repeating the primary input force and clock pulse allows the tool to keep track of new values on scan cells and within feedback paths.

When DRC performs scan cell checking, it also checks non-scan cells. When the checking process completes, the rules checker issues a message indicating the number of non-scan cells that qualify for clock sequential handling.

You instruct the tool to use clock sequential handling by selecting the `-Sequential` option to the `set_pattern_type` command. During test generation, the tool generates test patterns for target faults by first attempting combinational, and then RAM sequential techniques. If unsuccessful with these techniques, the tool performs clock sequential test generation if you specify a non-zero sequential depth.

#### Note



Setting the `-Sequential` switch to either 0 (the default) or 1 results in patterns with a maximum sequential depth of one, but the tool creates clock sequential patterns only if the setting is 1 or higher.

To report on clock sequential cells, you use the `report_nonscan_cells` command. For more information on setting up and reporting on clock sequential test generation, refer to the `set_pattern_type` and `report_nonscan_cells` descriptions in the *Tessent Shell Reference Manual*.

Limitations of clock sequential non-scan cell handling include:

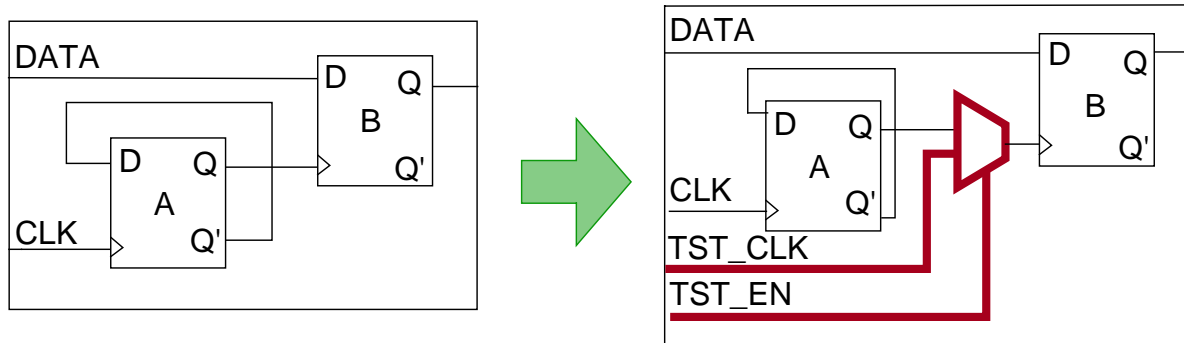
- The maximum allowable sequential depth is 255 (a typical depth would range from 2 to 5).
- Copy and shadow cells cannot behave sequentially.
- The tool cannot detect faults on clock/set/reset lines.
- You cannot use the read-only mode of RAM testing with clock sequential pattern generation.
- The tool simulates cells that capture data on a trailing clock edge (when data changes on the leading edge) using the original values on the data inputs.
- Non-scan cells that maintain a constant value after load\_unload simulation are treated as tied latches.
- This type of testing has high memory and performance costs.

## Clock Dividers

Some designs contain uncontrollable clock circuitry; that is, internally-generated signals that can clock, set, or reset flip-flops. If these signals remain uncontrollable, Tessent Scan will not consider the sequential elements controlled by these signals “scannable”. And consequently, they could disturb sequential elements during scan shifting. Thus, the system cannot convert these elements to scan.

Figure 4-17 shows an example of a sequential element (B) driven by a clock divider signal and with the appropriate circuitry added to control the divided clock signal.

**Figure 4-17. Clock Divider**



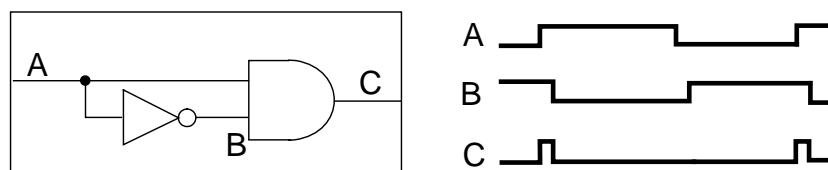
Tessent Scan can assist you in modifying your circuit for maximum controllability (and thus, maximum scannability of sequential elements) by inserting special circuitry, called *test logic*, at these nodes when necessary. Tessent Scan typically gates the uncontrollable circuitry with chip-level test pins. In the case of uncontrollable clocks, Tessent Scan adds a MUX controlled by the test\_clk and test\_en signals.

For more information on test logic, refer to “Enabling Test Logic Insertion” on page 124.

## Pulse Generators

A pulse generator is circuitry that creates a pulse at its output when active. Figure 4-18 gives an example of pulse generator circuitry.

**Figure 4-18. Example Pulse Generator Circuitry**

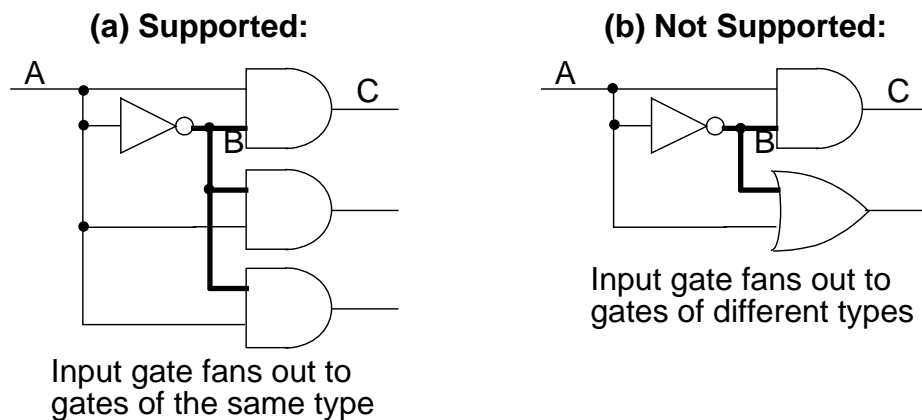


When designers use this circuitry in clock paths, there is no way to create a stable on state. Without a stable on state, the fault simulator and test generator have no way to capture data into the scan cells. Pulse generators also find use in write control circuitry, a use that impedes RAM testing.

By default, the ATPG tool identifies the reconvergent pulse generator sink (PGS) gates, or simply “pulse generators”, during the learning process. For the tools to provide support, a “pulse generator” must satisfy the following requirements:

- The “pulse generator” gate must have a connection (at C in [Figure 4-18](#)) to a clock input of a memory element or a write line of a RAM.
- The “pulse generator” gate must be an AND, NAND, OR, or NOR gate.
- Two inputs of the “pulse generator” gate must come from one reconvergent source gate.
- The two reconvergent paths may only contain inverters and buffers.
- There must be an inversion difference in the two reconvergent paths.
- The two paths must have different lengths (propagation times).
- In the long path, the inverter or buffer that connects to the “pulse generator” input must only go to gates of the same gate type as shown in (a) in [Figure 4-19](#). A fanout to gates of different types as in (b) in the figure is not supported. The tools model this input gate as tied to the non-controlling value of the “pulse generator” gate (TIE1 for AND and NAND gates, TIE0 for OR and NOR gates).

**Figure 4-19. Long Path Input Gate Must Go to Gates of the Same Type**



The ATPG tool provides two commands that deal with pulse generators: `set_pulse_generators`, which controls the identification of the “pulse generator” gates, and `report_pulse_generators`, which displays the list of “pulse generator” gates. Refer to the *Tessent Shell Reference Manual* for information about the `set_pulse_generators` and `report_pulse_generators` commands.

Additionally, rules checking includes some checking for “pulse generator” gates. Specifically, Trace rules #16 and #17 check to ensure proper usage of “pulse generator” gates. Refer to “[T16](#)” and “[T17](#)” in the *Tessent Shell Reference Manual* for more details on these rules.

The ATPG tool supports pulse generators with multiple timed outputs. For detailed information about this support, refer to “[Pulse Generators with User Defined Timing](#)” in the *Tessent Cell Library User’s Manual*.

## JTAG-Based Circuits

Boundary scan circuitry, as defined by IEEE standard 1149.1, can result in a complex environment for the internal scan structure and the ATPG process. The two main issues with boundary scan circuitry are 1) connecting the boundary scan circuitry with the internal scan circuitry, and 2) ensuring that the boundary scan circuitry is set up properly during ATPG.

## Testing RAM and ROM

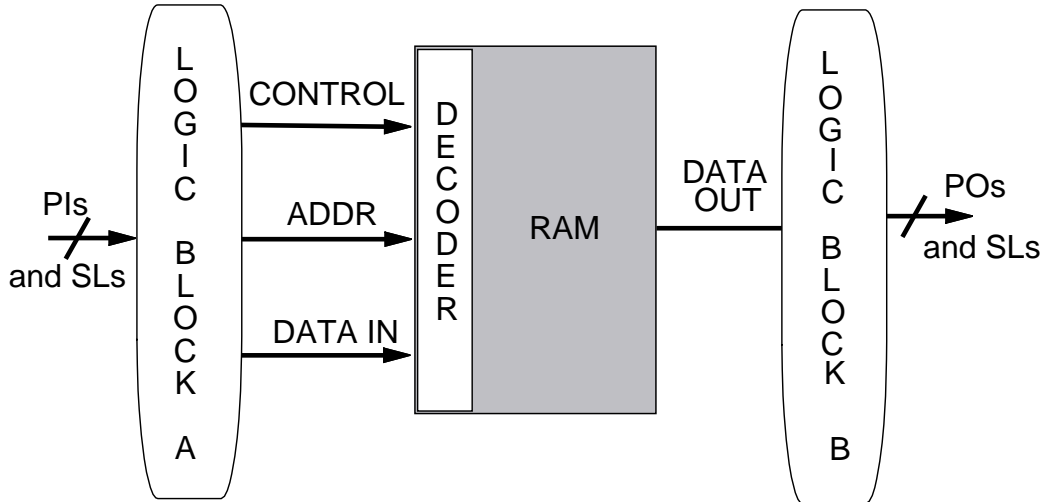
The three basic problems of testing designs that contain RAM and ROM are 1) modeling the behavior, 2) passing rules checking to allow testing, and 3) detecting faults during ATPG. The “[RAM and ROM](#)” section in the *Tessent Cell Library User’s Manual* discusses modeling RAM and ROM behavior. The “[RAM Rules](#)” section in the *Tessent Shell Reference Manual* discusses RAM rules checking. This section primarily discusses the techniques for detecting faults in circuits with RAM and ROM during ATPG. The “[RAM Summary Results and Test Capability](#)” section of the *Tessent Shell Reference Manual* discusses displayed DRC summary results upon completion of RAM rules checking.

The ATPG tool does *not* test the internals of the RAM/ROM, although MacroTest (separately licensed but available in the ATPG tool) lets you create tests for small memories such as register files by converting a functional test sequence or algorithm into a sequence of scan tests. For large memories, built-in test structures within the chip itself are the best methods of testing the internal RAM or ROM.

However, the ATPG tool needs to model the behavior of the RAM/ROM so that tests can be generated for the logic on either side of the embedded memory. This allows the tool to generate tests for the circuitry around the RAM/ROM, as well as the read and write controls, data lines, and address lines of the RAM/ROM unit itself.

[Figure 4-20](#) shows a typical configuration for a circuit containing embedded RAM.

**Figure 4-20. Design with Embedded RAM**



ATPG must be able to operate the illustrated RAM to observe faults in logic block A, as well as to control the values in logic block B to test faults located there. The ATPG tool has unique strategies for operating the RAMs.

## RAM/ROM Support

The tool treats a ROM as a strictly combinational gate. Once a ROM is initialized, it is a simple task to generate tests because the contents of the ROM do not change. Testing RAM however, is more of a challenge, because of the sequential behavior of writing data to and reading data from the RAM.

The tool supports the following strategies for propagating fault effects through the RAM:

- **Read-only mode** — The tool assumes the RAM is initialized prior to scan test and this initialization must not change during scan. This assumption allows the tool to treat a RAM as a ROM. As such, there is no requirement to write to the RAM prior to reading, so the test pattern only performs a read operation. Important considerations for read-only mode test patterns are as follows:
  - The read-only testing mode of RAM only tests for faults on data out and read address lines, just as it would for a ROM. The tool does not test the write port I/O.
  - To use read-only mode, the circuit must pass rules A1 and A6.
  - Values placed on the RAM are limited to initialized values.
  - Random patterns can be useful for all RAM configurations.
  - You must define initial values and assume responsibility that those values are successfully placed on the correct RAM memory cells. The tool does not perform

any audit to verify this is correct, nor will the patterns reflect what needs to be done for this to occur.

- Because the tester may require excessive time to fully initialize the RAM, it is allowed to do a partial initialization.
- **Pass-through mode** — The tool has two separate pass-through testing modes:
  - **Static pass-through** — To detect faults on data input lines, you must write a known value into some address, read that value from the address, and propagate the effect to an observation point. In this situation, the tool handles RAM transparently, similar to the handling of a transparent latch. This requires several simultaneous operations. The write and read operations are both active and thus writing to and reading from the same address. While this is a typical RAM operation, it allows testing faults on the data input and data output lines. It is not adequate for testing faults on read and write address lines.
  - **Dynamic pass-through** — This testing technique is similar to static pass-through testing except one pulse of the write clock performs both the write and read operation (if the write and read control lines are complementary). While static pass-through testing is comparable to transparent latch handling, dynamic pass-through testing compares to sequential transparent testing.
- **Sequential RAM test mode** — This is the recommended approach to RAM testing. While the previous testing modes provide techniques for detecting some faults, they treat the RAM operations as combinational. Thus, they are generally inadequate for generating tests for circuits with embedded RAM. In contrast, this testing mode tries to separately model all events necessary to test a RAM, which requires modeling sequential behavior. This enables testing of faults that require detection of multiple pulses of the write control lines. These faults include RAM address and write control lines.

RAM sequential testing requires its own specialized pattern type. RAM sequential patterns consist of one scan pattern with multiple scan chain loads. A typical RAM sequential pattern contains the events shown in [Figure 4-21](#).

---

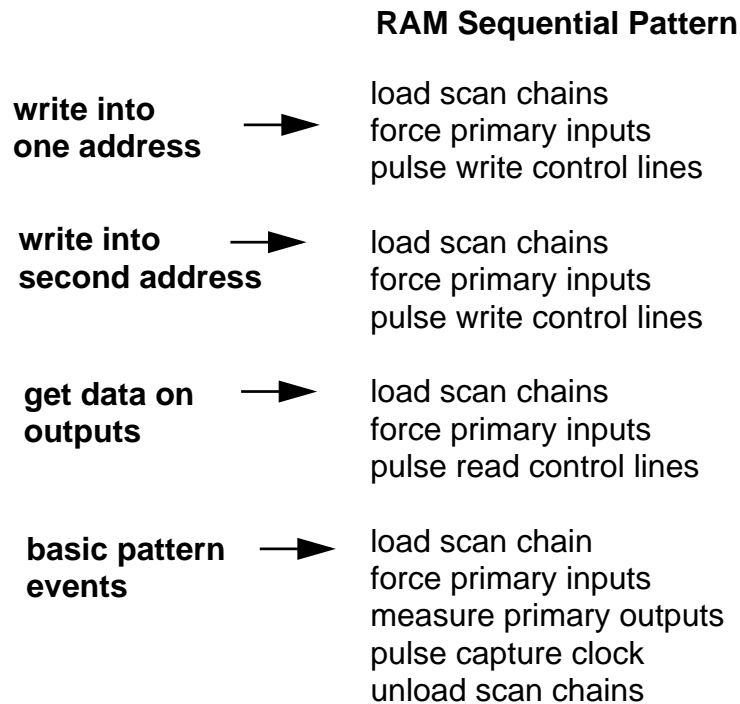
**Note**

For RAM sequential testing, the RAM's read\_enable/write\_enable control(s) can be generated internally. However, the RAM's read/write clock should be generated from a PI. This ensures RAM sequencing is synchronized with the RAM sequential patterns.

---



**Figure 4-21. RAM Sequential Example**



In this example of an address line test, assume that the MSB address line is stuck at 0. The first write would write data into an address whose MSB is 0 to match the faulty value, such as 0000. The second write operation would write different data into a different address (the one obtained by complementing the faulty bit). For this example, it would write into 1000. The read operation then reads from the first address, 0000. If the highest order address bit is stuck-at-0, the 2nd write would have overwritten the original data at address 0, and faulty circuitry data would be read from that address in the 3rd step.

Another technique that may be useful for detecting faults in circuits with embedded RAM is clock sequential test generation. It is a more flexible technique, which effectively detects faults associated with RAM. [“Clock Sequential Patterns”](#) on page 160 discusses clock sequential test generation in more detail.

## Common Read and Clock Lines

Ram\_sequential simulation supports RAMs whose read line is common with a scan clock. The tool assumes that the read and capture operation can occur at the same time and that the value captured into the scan cell is a function of the value read out from the RAM.

If the clock that captures the data from the RAM is the same clock which is used for reading, the tool issues a C6 clock rules violation. This indicates that you must set the clock timing so that the scan cell can successfully capture the newly read data.

If the clock that captures the data from the RAM is not the same clock that is used for reading, you will likely need to turn on multiple clocks to detect faults. The default “set\_clock\_restriction On” command is conservative, so the tool will not allow these patterns, resulting in a loss in test coverage. If you issue the “set\_clock\_restriction Off” command, the tool allows these patterns, but there is a risk of inaccurate simulation results because the simulator does not propagate captured data effects.

## Common Write and Clock Lines

The tool supports common write and clock lines. The following shows the support for common write and clock lines:

- You can define a pin as both a write control line and a clock if the off-states are the same value. the tool then displays a warning message indicating that a common write control and clock has been defined.
- The rules checker issues a C3 clock rule violation if a clock can propagate to a write line of a RAM, and the corresponding address or data-in lines are connected to scan latches which has a connection to the same clock.
- The rules checker issues a C3 clock rule violation if a clock can propagate to a read line of a RAM, and the corresponding address lines are connected to scan latches which has a connection to the same clock.
- The rules checker issues a C3 clock rule violation if a clock can capture data into a scan latch that comes from a RAM read port that has input connectivity to latches which has a connection to the same clock.
- If you set the simulation mode to Ram\_sequential, the rules checker will not issue an A2 RAM rule violation if a clock is connected to a write input of a RAM. Any clock connection to any other input (including the read lines) will continue to be a violation.
- If a RAM write line is connected to a clock, you cannot use the dynamic pass through test mode.
- Patterns which use a common clock and write control for writing into a RAM will be in the form of ram\_sequential patterns. This requires you to set the simulation mode to Ram\_sequential.
- If you change the value of a common write control and clock line during a test procedure, you must hold all write, set, and reset inputs of a RAM off. The tool considers failure to satisfy this condition as an A6 RAM rule violation and disqualifies the RAM from being tested using read\_only and ram\_sequential patterns.

## RAM/ROM Support Commands

The tool requires certain knowledge about the design prior to test generation. For circuits with RAM, you must define write controls, and if the RAM has data hold capabilities, you must also define read controls. Just as you must define clocks so the tool can effectively write scan

patterns, you must also define these control lines so it can effectively write patterns for testing RAM. And similar to clocks, you must define these signals in setup mode, prior to rules checking. The commands in [Table 4-1](#) support the testing of designs with RAM and/or ROM.

**Table 4-1. RAM/ROM Commands**

| Command Name                                   | Description                                                                                                              |
|------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| <a href="#">add_read_controls</a>              | Defines a PI as a read control and specifies its off value.                                                              |
| <a href="#">add_write_controls</a>             | Defines a PI as a write control and specifies its off value.                                                             |
| <a href="#">create_initialization_patterns</a> | Creates RAM initialization patterns and places them in the internal pattern set.                                         |
| <a href="#">delete_read_controls</a>           | Removes the read control line definitions from the specified primary input pins.                                         |
| <a href="#">delete_write_controls</a>          | Removes the write control line definitions from the specified primary input pins.                                        |
| <a href="#">read_modelfile</a>                 | Initializes the specified RAM or ROM gate using the memory states contained in the specified modelfile.                  |
| <a href="#">report_read_controls</a>           | Displays all of the currently defined read control lines.                                                                |
| <a href="#">report_write_controls</a>          | Displays all of the currently defined write control lines.                                                               |
| <a href="#">set_pattern_type</a>               | Specifies whether the ATPG simulation run uses combinational or sequential RAM test patterns.                            |
| <a href="#">set_ram_initialization</a>         | Specifies whether to initialize RAM and ROM gates that do not have initialization files.                                 |
| <a href="#">set_ram_test</a>                   | Sets the RAM testing mode to either <code>read_only</code> , <code>pass_thru</code> , or <code>static_pass_thru</code> . |
| <a href="#">write_modelfile</a>                | Writes all internal states for a RAM or ROM gate into the file that you specify.                                         |

## Basic ROM/RAM Rules Checking

The rules checker performs the following audits for RAMs and ROMs:

- The checker reads the RAM/ROM initialization files and checks them for errors. If you selected random value initialization, the tool gives random values to all RAM and ROM gates without an initialized file. If there are no initialized RAMs, you cannot use the read-only test mode. If any ROM is not initialized, an error condition occurs. A ROM must have an initialization file but it may contain all Xs. Refer to the [read\\_modelfile](#) description in the *Tessent Shell Reference Manual* for details on initialization of RAM/ROM.
- The RAM/ROM instance name given must contain a single RAM or ROM gate. If no RAM or ROM gate exists in the specified instance, an error condition occurs.

- If you define write control lines and there are no RAM gates in the circuit, an error condition occurs. To correct this error, delete the write control lines.
- When the write control lines are off, the RAM set and reset inputs must be off and the write enable inputs of all write ports must be off. You cannot use RAMs that fail this rule in read-only test mode. If any RAM fails this check, you cannot use dynamic pass-through. If you defined an initialization file for a RAM that failed this check, an error condition occurs. To correct this error, properly define all write control lines or use lineholds (pin constraints).
- A RAM gate must not propagate to another RAM gate. If any RAM fails this check, you cannot use dynamic pass-through.
- A defined scan clock must not propagate directly (unbroken by scan or non-scan cells) to a RAM gate. If any RAM fails this check, you cannot use dynamic pass-through.
- The tool checks the write and read control lines for connectivity to the address and data inputs of all RAM gates. It gives a warning message for all occurrences and if connectivity fails, there is a risk of race conditions for all pass-through patterns.
- A RAM that uses the edge-triggered attribute must also have the **read\_off** attribute set to hold. Failure to satisfy this condition results in an error condition when the design flattening process is complete.
- If the RAM rules checking identifies at least one RAM that the tool can test in read-only mode, it sets the RAM test mode to read-only. Otherwise, if the RAM rules checking passes all checks, it sets the RAM test mode to dynamic pass-through. If it cannot set the RAM test mode to read-only or dynamic pass-through, it sets the test mode to static pass-through.
- A RAM with the **read\_off** attribute set to hold must pass Design Rule A7 (when read control lines are off, place read inputs at 0). The tool treats RAMs that fail this rule as:
  - a TIE-X gate, if the read lines are edge-triggered.
  - a **read\_off** value of X, if the read lines are not edge-triggered.
- The read inputs of RAMs that have the **read\_off** attribute set to hold must be at 0 during all times of all test procedures, except the **test\_setup** procedure.
- The read control lines must be off at time 0 of the **load\_unload** procedure.
- A clock cone stops at read ports of RAMs that have the **read\_off** attribute set to hold, and the effect cone propagates from its outputs.

For more information on the RAM rules checking process, refer to “[RAM Rules](#)” in the *Tessent Shell Reference Manual*.

## Incomplete Designs

The ATPG tool and Tessent Scan can read incomplete Verilog designs due to their ability to generate black boxes. The Verilog parser can blackbox any instantiated module or instance that is not defined in either the ATPG library or the design netlist. The tool issues a warning message for each blackboxed module similar to the following:

```
// WARNING: Following modules are undefined:
//      ao21
//      and02
// Use "add_black_box -auto" to treat these as black boxes.
```

If the tool instantiates an undefined module, it generates a module declaration based on the instantiation. If ports are connected by name, the tool uses those port names in the generated module. If ports are connected by position, the parser generates the port names. Calculating port directions is problematic and must be done by looking at the other pins on the net connected to the given instance pin. For each instance pin, if the connected net has a non-Z-producing driver, the tool considers the generated module port an input, otherwise the port is an output. The tool never generates inout ports since they cannot be inferred from the other pins on the net.

Modules that are automatically blackboxed default to driving X on their outputs. Faults that propagate to the black box inputs are classified as ATPG\_untestable (AU). To change the output values driven, refer to the [add\\_black\\_box](#) description in the *Tessent Shell Reference Manual*.



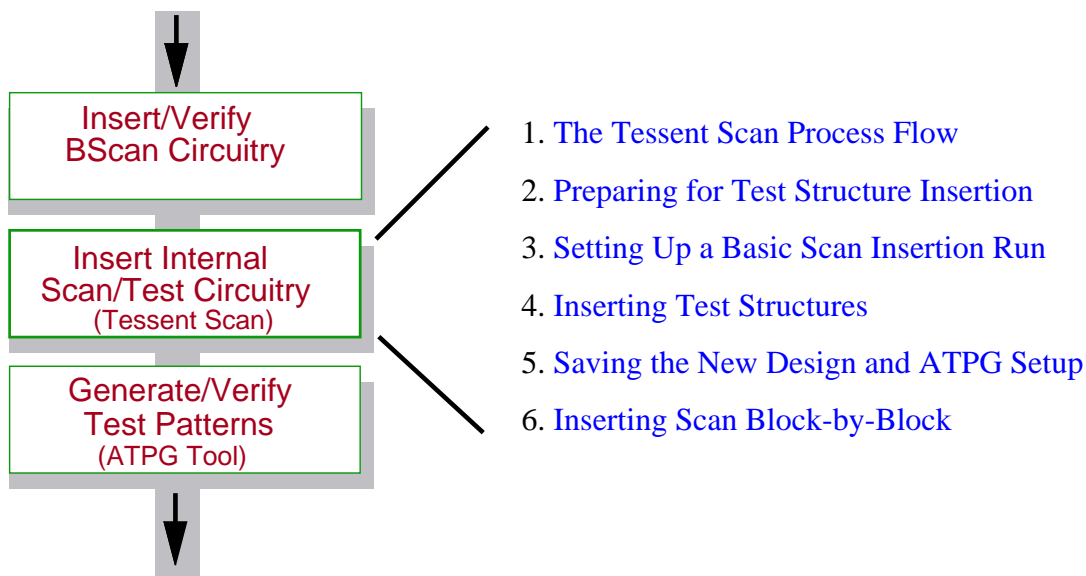
# Chapter 5

## Inserting Internal Scan and Test Circuitry

---

Figure 5-1 shows the process of inserting scan and other test circuitry with Tessent Scan or Tessent Shell operating in dft -scan context.

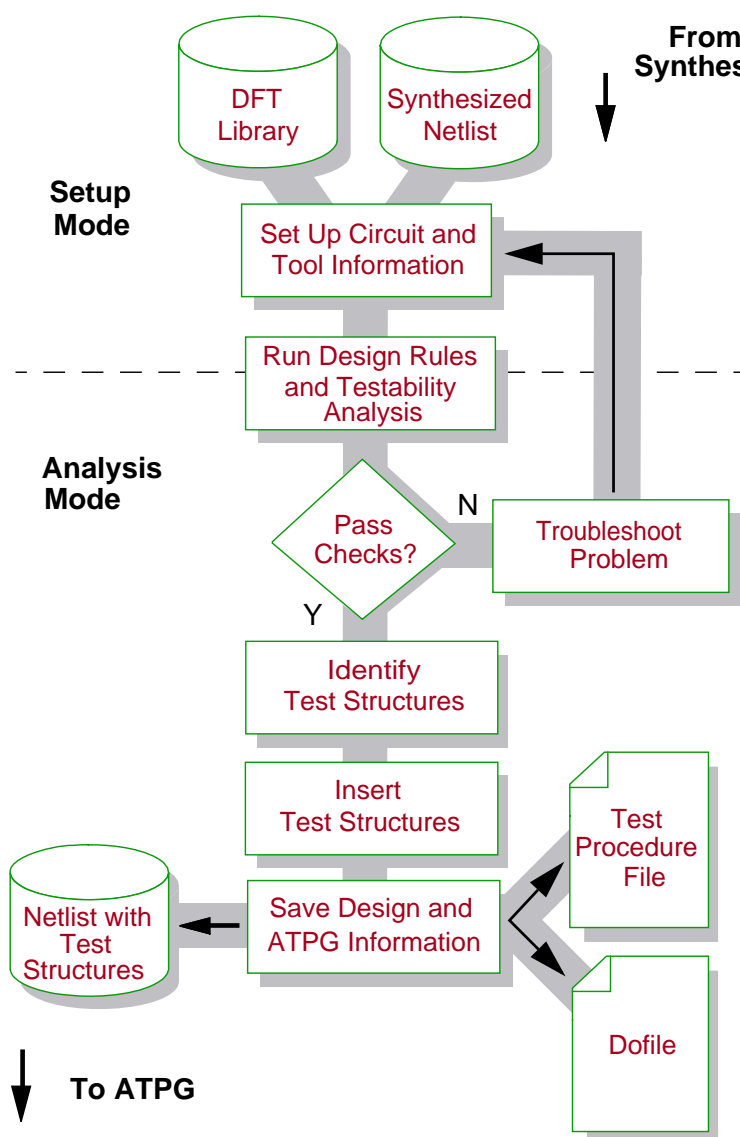
**Figure 5-1. Internal Scan Insertion Procedure**



## The Tessent Scan Process Flow

Figure 5-2 shows the basic flow for synthesizing scan circuitry with Tessent Scan.

Figure 5-2. Basic Scan Insertion Flow with Tessent Scan



You start with a DFT library and a synthesized design netlist. The library is the same one that the ATPG tool uses. [“Tessent Scan Inputs and Outputs”](#) on page 121 describes the netlist formats you can use with Tessent Scan. The design netlist you use as input may be an individual block of the design, or the entire design.

After invoking the tool, your first task is to set up information about the design—this includes both circuit information and information about the test structures you want to insert. [“Preparing for Test Structure Insertion”](#) on page 123 describes the procedure for this task. The next task after setup is to run rules checking and testability analysis, and debug any violations that you encounter. [“Changing the System Mode \(Running Rules Checking\)”](#) on page 130 documents the procedure for this task.



### Note



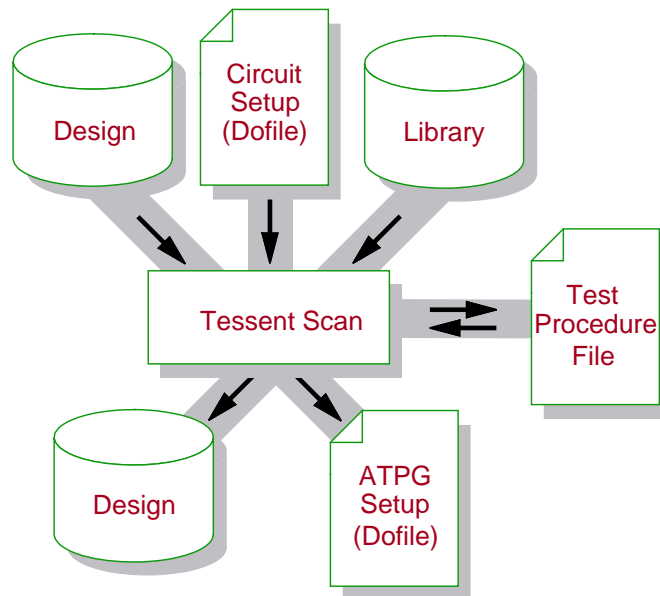
To catch design violations early in the design process, you should run and debug design rules on each block as it is synthesized.

After successfully completing rules checking, you will be in the analysis system mode. At this point, if you have any existing scan you want to remove, you can do so. “[Handling Existing Boundary Scan Circuitry](#)” on page 130 describes the procedure for doing this. You can then set up specific information about the scan or other testability circuitry you want added and identify which sequential elements you want converted to scan. “[Setting Up a Basic Scan Insertion Run](#)” on page 131 describes the procedure for accomplishing this. Finally, with these tasks completed, you can insert the desired test structures into your design. “[Inserting Test Structures](#)” on page 141 describes the procedure for this insertion.

## Tessent Scan Inputs and Outputs

Figure 5-3 shows the inputs used and the outputs produced by Tessent Scan.

**Figure 5-3. The Inputs and Outputs of Tessent Scan**



Tessent Scan uses the following inputs:

- **Design (netlist)** — The supported design data format is gate-level Verilog.
- **Circuit Setup (or Dofile)** — This is the set of commands that gives the tool information about the circuit and how to insert test structures. You can issue these commands interactively in the tool session or place them in a dofile.
- **Library** — The design library contains descriptions of all the cells the design uses. The library also includes information that the tool uses to map non-scan cells to scan cells

and to select components for added test logic circuitry. The tool uses the library to translate the design data into a flat, gate-level simulation model on which it runs its internal processes.

- **Test Procedure File** — This file defines the stimulus for shifting scan data through the defined scan chains. This input is only necessary on designs containing preexisting scan circuitry or requiring test setup patterns.

Tessent Scan produces the following outputs:

- **Design (Netlist)** — This netlist contains the original design modified with the inserted test structures. The output netlist format is gate-level Verilog.
- **ATPG Setup (Dofile and Test Procedure File)** — The tool can automatically create a dofile and test procedure file that you can supply to the ATPG tool. These files contain the circuit setup information that you specified to the tool as well as information on the test structures that the tool inserted into the design. The tool creates these files for you when you issue the [write\\_atpg\\_setup](#) command.

---

**Note**

The timeplate in the test procedure file defines a generic waveform for all of the clocks in the design (including set and reset signals that were defined using the [add\\_clocks](#) command) and not just for those clocks that are required for scan chain shifting. However, the generated shift procedure only pulses clocks that drive scan cells. Internal clock nets that can be mapped to a top level pin (by tracing a sensitized path) are not included in the dofile and test procedure file, but internal clocks that cannot be traced (such as PLL outputs) are included.

---

- **Test Procedure File** — When you issue the [write\\_atpg\\_setup](#) command, the tool writes a simple test procedure file for the scan circuitry it inserted into the design. You use this file with the downstream ATPG tool.

## Test Structures Supported by Tessent Scan

Tessent Scan can identify and insert a variety of test structures, including several different scan architectures and test points.

The following list briefly describes the test structures the tool supports:

- **Scan** — A flow where the tool converts all sequential elements that pass scannability checking into scan cells. [“Understanding Scan”](#) on page 24 discusses the full scan style.
- **Wrapper chains** — A flow where the tool identifies sequential elements that interact with input and output pins. These memory elements are converted into scan chains, and the remaining sequential elements are not affected. For more information, see [“Understanding Wrapper Chains”](#) on page 25.

- **Scan and/or Wrapper** — A flow where the tool converts into scan cells those sequential elements that interact with primary input and output pins, and then stitches the scan cells into dedicated wrapper chains. The tool converts the remaining sequential elements into scan cells and stitches them into separate chains, which are called core chains.
- **Test points** — A flow where the tool inserts control and observe points at user specified locations. [“Understanding Test Points”](#) on page 27 discusses the test points method.

Tessent Scan provides the ability to insert test points at user specified locations. If both scan and test points are enabled during an identification run, the tool performs scan identification followed by test point identification.

## Invoking Tessent Scan

You access Tessent Scan functionality by invoking Tessent Shell and then setting the context to dft -scan:

```
% tessent -shell  
  
SETUP> set_context dft -scan
```

The tool invokes in setup mode, ready for you to begin loading or working on your design. You use this setup mode to define the circuit and scan data which is the next step in the process.

## Preparing for Test Structure Insertion

The following subsections discuss the steps you would typically take to prepare for the insertion of test structures into your design. When the tool invokes, you are in setup mode. All of the setup steps shown in the following subsections occur in setup mode.

### Selecting the Scan Methodology

If you want to insert scan circuitry into your design, you must select the type of architecture for the scan circuitry. You use the [set\\_scan\\_type](#) command to specify the type of scan architecture you want to insert. Your choices are Mux\_scan, Clocked\_scan, or Lssd. For more information, refer to [“Scan Architectures”](#) on page 73.

### Defining Scan Cell and Scan Output Mapping

The tool uses the default mapping defined within the ATPG library. Each scan model in the library describes how the non-scan models map to scan model in the scan\_definition section of the model. For more information on the default mapping of the library model, refer to [“Defining Cell Information”](#) in the *Tessent Cell Library Manual*.

You have the option to customize the scan cell and the cell's scan output mapping behavior. You can change the mapping for an individual instance, all instances under a hierarchical instance, all instances in all occurrences of a module in the design, or all occurrences of the model in the entire design, using the [set\\_cell\\_model\\_mapping](#) command.

For example, you can map the fd1 nonscan model to the fd1s scan model for all occurrences of the model in the design by entering:

```
set_cell_model_mapping -new_model fd1s -model fd1
```

The following example maps the fd1 nonscan model to the fd1s scan model for all matching instances in the “counter” module and for all occurrences of that module in the design:

```
set_cell_model_mapping -new_model fd1s -model fd1 -module counter
```

Additionally, you can change the scan output pin of the scan model in the same manner as the scan cell. Within the scan\_definition section of the model, the scan\_out attribute defines which pin is used as the scan output pin. During the scan stitching process, the tool selects the output pin based on the lowest fanout count of each of the possible pins. If you have a preference as to which pin to use for a particular model or instance, you can also issue the [set\\_cell\\_model\\_mapping](#) command to define that pin.

For example, if you want to use “qn” instead of “q” for all occurrences of the fd1s scan model in the design, enter:

```
set_cell_model_mapping -new_model fd1s -output qn
```

## Enabling Test Logic Insertion

*Test logic* is circuitry that the tool adds to improve the testability of a design. If so enabled, the tool inserts test logic during scan insertion based on the analysis performed during the design rules and scannability checking processes.

Test logic provides a useful solution to a variety of common problems. First, some designs contain uncontrollable clock circuitry; that is, internally-generated signals that can clock, set, or reset flip-flops. If these signals remain uncontrollable, the tool will not consider the sequential elements controlled by these signals scannable. Second, you might want to prevent bus contention caused by tri-state devices during scan shifting.

Tessent Scan can assist you in modifying your circuit for maximum controllability (and thus, maximum scannability of sequential elements) and bus contention prevention by inserting test logic circuitry at these nodes when necessary.

---

### Note

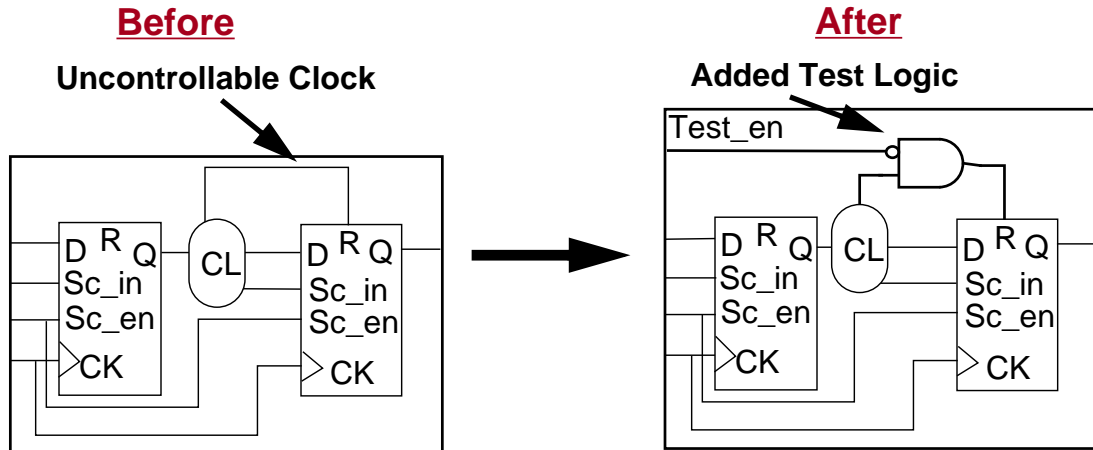


Tessent Scan does not attempt to add test logic to user-defined non-scan instances or models; that is, those specified by the [add\\_nonscan\\_instances](#) or the [add\\_nonscan\\_models](#) commands.

---

Tessent Scan typically gates the uncontrollable circuitry with a chip-level test pin. [Figure 5-4](#) shows an example of test logic circuitry.

**Figure 5-4. Test Logic Insertion**



You can specify which signals to insert test logic on with the [set\\_test\\_logic](#).

You can add test logic to all uncontrollable (set, reset, clock, or RAM write control) signals during the scan insertion process. By default, Tessent Scan does not add test logic. You must explicitly enable the use of test logic with this command.

In adding the test logic circuitry, the tool performs some basic optimizations in order to reduce the overall amount of test logic needed. For example, if the reset line to several flip-flops is a common internally-generated signal, the tool gates it at its source before it fans out to all the flip-flops.

#### Note



You must turn the appropriate test logic on if you want the tool to consider latches as scan candidates. Refer to “D6” in the *Tessent Shell Reference Manual* for more information on scan insertion with latches.

If your design uses bidirectional pins as scan I/Os, the tool controls the scan direction for the bidirectional pins for correct shift operation.

This can be specified with [set\\_bidi\\_gating](#). If the enable signal of the bidirectional pin is controlled by a primary input pin, then the tool adds a “force” statement for the enable pin in the new load\_unload procedure to enable/disable the correct direction. Otherwise, the tool inserts gating logic to control the enable line. The gate added to the bidirectional enable line is either a 2-input AND or OR. By default, no bidirectional gating is inserted and you must make sure that the inserted scan chains function properly by sensitizing the enable lines of any bidirectional ports in the scan path.

There are four possible cases between the scan direction and the active values of a tri-state driver, as shown in [Table 5-1](#). The second input of the gate is controlled from the scan\_enable signal, which might be inverted. You will need to specify AND and OR models through the cell\_type keyword in the ATPG library or use the [add\\_cell\\_models](#) command.

**Table 5-1. Scan Direction and Active Values**

| Driver      | Scan Direction | Gate Type |
|-------------|----------------|-----------|
| active high | input          | AND       |
| active high | output         | OR        |
| active low  | input          | OR        |
| active low  | output         | AND       |

If you enable the gating of bidirectional pins, the tool controls all bidirectional pins. The bidirectional pins not used as scan I/Os are put into input mode (Z state) during scan shifting by either “force” statements in the new load\_unload procedure or by using gating logic.

The tool adds a “force Z” statement in the test procedure file for the output of the bidirectional pin if it is used as scan output pin. This ensures that the bus is not driven by the tristate drivers of both bidirectional pin and the tester at the same time.

## Specifying the Models to use for Test Logic

When adding test logic circuitry, the tool uses a number of gates from the library. The **cell\_type** attribute in the library model descriptions tells the tool which components are available for use as test logic. If the library does not contain this information, you can instead specify which library models to use with the [add\\_cell\\_models](#) command.

---

### Note



Tessent Scan treats any Verilog module enclosed in `celldefine / `endcelldefine directives as a library cell and prevents any logic changes to these modules.

---

## Issues Concerning Test Logic Insertion and Test Clocks

Because inserting test logic actually adds circuitry to the design, you should first try to increase circuit controllability using other options. These options might include such things as performing proper circuit setup or, potentially, adding test points to the circuit prior to scan. Additionally, you should re-optimize a design to ensure that fanout resulting from test logic is correctly compensated and passes electrical rules checks.

In some cases, inserting test logic requires the addition of multiple test clocks. Analysis run during DRC determines how many test clocks the tool needs to insert. The [report\\_scan\\_chains](#) command reports the test clock pins used in the scan chains.

## Related Test Logic Commands

- [delete\\_cell\\_models](#) — Deletes the information specified by the `add_cell_models` command.
- [report\\_cell\\_models](#) — Displays a list of library cell models to be used for adding test logic circuitry.
- [report\\_test\\_logic](#) — Displays a list of test logic added during scan insertion.

## Specifying Clock Signals

Tessent Scan must be aware of the circuit clocks to determine which sequential elements are eligible for scan. The tool considers clocks to be any signals that have the ability to alter the state of a sequential device (such as system clocks, sets, and resets). Therefore, you need to tell the tool about these “clock signals” by adding them to the clock list with the [add\\_clocks](#) command.

With this command, you must specify the off-state for pins you add to the clock list. The off-state is the state in which clock inputs of latches are inactive. For edge-triggered devices, the off state is the clock value prior to the clock’s capturing transition.

For example, you might have two system clocks, called “clk1” and “clk2”, whose off-states are 0 and a global reset line called “rst\_1” whose off-state is 1 in your circuit. You can specify these as clock lines as follows:

```
SETUP> add_clocks 0 clk1 clk2
```

```
SETUP> add_clocks 1 rst_1
```

You can specify multiple clock pins with the same command if they have the same off-state. You must define clock pins prior to entering analysis mode. Otherwise, none of the non-scan sequential elements will successfully pass through scannability checks. Although you can still enter analysis mode without specifying the clocks, the tool will not be able to convert elements that the unspecified clocks control.

### Note



If you are unsure of the clocks within a design, you can use the [analyze\\_control\\_signals](#) command to identify and then define all the clocks. It also defines the other control signals in the design.

- [delete\\_clocks](#) — Deletes primary input pins from the clock list.
- [report\\_clocks](#) — Displays a list of all clocks.
- [report\\_primary\\_inputs](#) — Displays a list of primary inputs.
- [write\\_primary\\_inputs](#) — Writes a list of primary inputs to a file.

## Specifying Existing Scan Information

You may have a design that already contains some existing internal scan circuitry. For example, one block of your design may be reused from another design, and thus, may already contain its own scan chain. You may also have used a third-party tool to insert scan before invoking the tool. If either of these is your situation, there are several ways in which you may want to handle the existing scan data, including, leaving the existing scan alone, deleting the existing scan, and adding additional scan circuitry.

---

### Note



If you are performing block-by-block scan synthesis, you should refer to [“Inserting Scan Block-by-Block”](#) on page 149.

---

If your design contains existing scan chains that you want to use, you must specify this information to the tool while you are in setup mode; that is, before design rules checking. If you do not specify existing scan circuitry, the tool treats all the scan cells as non-scan cells and performs non-scan cell checks on them to determine if they are scan candidates.

Common methodologies for handling existing scan circuitry include:

- Remove the existing scan chain(s) from the design and reverse the scan insertion process. the tool will replace the scan cells with their non-scan equivalent cells. The design can then be treated as you would any other new design to which you want to add scan circuitry. This technique is often used when re-stitching scan cells based on placement and routing results.
- Add additional scan chains based on the non-scan cells while leaving the original scan chains intact.
- Stitch together existing scan cells that were previously unstitched.

The remainder of this section includes details related to these methodologies.

## Specifying Existing Scan Groups

A scan chain group consists of a set of scan chains that are controlled through the same procedures; that is, the same test procedure file controls the operation of all chains in the group. If your design contains existing scan chains, you must specify the scan group to which they belong, as well as the test procedure file that controls the group. To specify an existing scan group, use the [add\\_scan\\_groups](#) command.

For example, you can specify a group name of “group1” controlled by the test procedure file “group1.test\_proc” as follows:

```
SETUP> add_scan_groups group1 group1.test_proc
```

For information on creating test procedure files, refer to [“Test Procedure Files”](#) on page 75.



## Specifying Existing Scan Chains

After specifying the existing scan group, you need to communicate to the tool which scan chains belong to this group. To specify existing scan chains, use the [add\\_scan\\_chains](#) command.

You need to specify the scan chain name, the scan group to which it belongs, and the primary input and output pins of the scan chain. For example, assume your design has two existing scan chains, “chain1” and “chain2”, that are part of “group1”. The scan input and output pins of chain1 are “sc\_in1” and “sc\_out1”, and the scan input and output pins of chain2 are “sc\_in2” and “sc\_out2”, respectively. You can specify this information as follows:

```
SETUP> add_scan_chains chain1 group1 sc_in1 sc_out1
```

```
SETUP> add_scan_chains chain2 group1 sc_in2 sc_out2
```

## Specifying Existing Scan Cells

If the design has existing scan cells that are not stitched together in a scan chain, you need to identify these cells for Tessent Scan. (You cannot define scan chains if the scan cells are not stitched together.) This situation can occur if scan cells are used in the functional design to provide actual timing. Tessent Scan can insert scan cells without stitching if you use the - Connect {Tied | Loop | Buffer} arguments to the [insert\\_test\\_logic](#) command.

Additionally, defining these existing scan cells prevents the tool from performing possibly undesirable default actions, such as scan cell mapping and generation of unnecessary mux gates.

## New Scan Cell Mapping

If you have existing scan cells, you must identify them as such to prevent the tool from classifying them as replaceable by new scan cells. One or the other of the following criteria is necessary for the tool to identify existing scan cells and not map them to new scan cells:

1. Declare the “data\_in = <port\_name>” in the scan\_definition section of the scan cell’s model in the ATPG library.

If you have a hierarchy of scan cell definitions, where one library cell can have another library cell as its scan version, using the data\_in declaration in a model causes the tool to consider that model as the end of the scan definition hierarchy, so that no mapping of instances of that model will occur.

#### Note



It is not recommended that you create a hierarchy of scan cell model definitions. If, for instance, your `data_in` declaration is in the `scan_definitions` section of the third model in the definitions hierarchy, but the tool encounters an instance of the first model in the hierarchy, it will replace the first model with the second model in the hierarchy, not the desired third model. If you have such a hierarchy, you can use the [set\\_cell\\_model\\_mapping](#) command to point to the desired model. `set_cell_model_mapping` overrides the mapping defined in the library model.

---

2. The scan enable port of the instance of the cell model must be either dangling or tied (0 or 1) or pre-connected to a global scan enable pin(s). In addition, the scan input port must be dangling or tied or connected to the cell's scan output port as a self loop or a self loop with (multiple) buffers or inverters.

Dangling implies that there are no connected fan-ins from other pins except tied pins or tied nets. To identify an existing (global) scan enable, use the [set\\_scan\\_insertion](#) command.

Issue the `set_scan_insertion` command before the [insert\\_test\\_logic](#) command.

## Additional Mux Gates

Another consequence of not specifying existing scan cells is the addition of unnecessary multiplexers, creating an undesirable area and routing overhead.

If you use criteria (a) as the means of preventing scan cell mapping, the tool also checks the scan enable and scan in ports. If either one is driven by system logic, then the tool inserts a new mux gate before the data input and uses it as a mux in front of the preexisting scan cell. (This is only for mux-DFF scan; this mux is not inserted for LSSD or `clocked_scan` types of scan.)

If you use a combination of criteria (a) and (b), or just criteria (b), as the means of preventing scan cell mapping, the tool will not insert a mux gate before the data input.

Once the tool can identify existing scan cells, they can be stitched into scan chains in the normal scan insertion process.

## Handling Existing Boundary Scan Circuitry

If your design contains boundary scan circuitry and existing internal scan circuitry, you must integrate the boundary scan circuitry with the internal test circuitry. You must ensure proper connection of the scan chains' `scan_in` and `scan_out` ports to the TAP controller.

## Changing the System Mode (Running Rules Checking)

Tessent Scan performs model flattening, learning analysis, rules checking, and scannability checking when you try to exit the setup system mode. [“Understanding Common Tool](#)

[Terminology and Concepts](#)” on page 67 explains these processes in detail. If you are finished with all the setup you need to perform, you can change the system mode by entering the [set\\_system\\_mode](#) command as follows:

```
SETUP> set_system_mode analysis
```

If an error occurs during the rules checking process, the application remains in setup mode, where you must correct the error. You can clearly identify and easily resolve the cause of many errors. Other errors, such as those associated with proper clock definitions and test procedure files, can be more complex. “[Troubleshooting Rules Violations](#)” in the *Tessent Shell Reference Manual* discusses the procedure for debugging rules violations. You can also use DFTVisualizer to visually investigate the causes of DRC violations. For more information, refer to “[DFTVisualizer](#)” in the *Tessent Shell User’s Manual*.

## Setting Up a Basic Scan Insertion Run

Tessent Scan’s default mode of operation is to replace all candidate non-scan cells with scan cells and then stitch them into scan chains based on a number of internal criteria including minimizing the number of clock domain crossings in the scan path, and also minimizing the routing overhead.

The actual netlist processing is triggered by the [insert\\_test\\_logic](#) command. The default operation of the [insert\\_test\\_logic](#) command is to perform scan substitution and stitching.

## Setting Up for Wrapper Chain Identification

Tessent Scan provides the ability to improve performance and test coverage of hierarchical designs by reducing the visibility of the internal core scan chains of the design submodules. The only requirements for testing submodules at the top-level is that the logic inputs must be controllable using scan chains (i.e. must be driven by registers) and the logic outputs must be observable through scan chains (i.e. must be registered). These requirements can be met using *wrapper scan chains* which are scan chains around the periphery of the submodules that connect to each input and output (except user-defined clocks, scan-related I/O pins, and user-excluded pins) of the submodules to be tested. Wrapper chains comprising the wrapper cells reachable from a submodule’s inputs are treated as the Input type wrapper chains and wrapper chains comprising the wrapper cells reachable from a submodule’s outputs are treated as the Output type wrapper chains. The Input and Output wrapper chains are used to provide proper test coverage of hierarchical designs during the INTEST and EXTEST modes.

In INTEST mode, all inputs to submodules are controllable using the Input wrapper scan chains and all outputs are observable through the Output wrapper scan chains. This provides the ability to independently generate a complete set of ATPG patterns for submodules.

In EXTEST mode, all outputs from submodules are controllable using the Output wrapper scan chains and all inputs are observable through the Input wrapper scan chains. This provides the

ability to test/optimize the logic surrounding the submodules at the top-level without requiring internal visibility into the submodules (i.e. core scan chains are not needed).

## Generating Wrapper Chains

Tessent Scan generates separate Input and Output wrapper chains based on the following conditions:

- The scan cells comprising the wrapper scan chains are identified as the Input and Output wrapper cells using the [set\\_wrapper\\_chains](#) and [analyze\\_wrapper\\_cells](#) commands and are stitched into the separate Input and Output wrapper chains using the [insert\\_test\\_logic](#) command.
  - When you issue the [analyze\\_wrapper\\_cells](#) command, Tessent Scan performs the wrapper cells analysis and identifies the Input and Output wrapper cells, which will be stitched into the separate Input and Output wrapper chains.
  - If the [set\\_wrapper\\_chains](#) command was not issued, the [analyze\\_wrapper\\_cells](#) command will perform the default wrapper cells identification similar to issuing the [set\\_wrapper\\_chains](#) command with no arguments.
  - If the [analyze\\_wrapper\\_cells](#) command is issued in conjunction with the “[set\\_wrapper\\_chains -io\\_registration on](#)” command or the [set\\_registered\\_io](#) command, the dedicated wrapper cells added to the primary Inputs will be stitched into the appropriate Input wrapper chains, while the dedicated wrapper cells added to the primary Outputs will be stitched into the appropriate Output wrapper chains.
- When the [set\\_registered\\_io](#) command is issued and the [analyze\\_wrapper\\_cells](#) command is not issued, the dedicated wrapper cells added to the primary I/Os are treated as Core cells and are stitched into the appropriate Core scan chains.
  - Dedicated wrapper cells are normally analyzed and inserted into scan chains when you issue the [insert\\_test\\_logic](#) command.
  - For backward compatibility, the [report\\_wrapper\\_cells](#) command will trigger the dedicated wrapper cells analysis when the [set\\_registered\\_io](#) command was issued, but the [analyze\\_wrapper\\_cells](#) command was not.
- If you issue the [insert\\_test\\_logic](#) command after the [set\\_wrapper\\_chains](#) command, but have not issued the [analyze\\_wrapper\\_cells](#) command, you will get an error message stating that the wrapper chains analysis is required prior to insertion.

```
Error: Wrapper cells information is out of date:
      use 'analyze_wrapper_cells' to re-analyze wrapper cells prior to
      insertion.
```

- If you use the [add\\_sub\\_chains](#) command with the “-type input\_wrapper” or “-type output\_wrapper” options to explicitly specify sub-chains to be of wrapper type **input\_wrapper** or **output\_wrapper**, the sub-chains will be stitched into the appropriate Input or Output wrapper chains only if the [analyze\\_wrapper\\_chains](#) command was

previously issued; otherwise they will be stitched into the Core chains and will be controlled by the Core scan enable signals.

- The following commands can affect the I/O registration of dedicated wrapper cells and the identification of shared wrapper cells:
  - `add_nonscan_instances`, `delete_nonscan_instances`
  - `add_nonscan_models`, `delete_nonscan_models`
  - `add_scan_instances`, `delete_scan_instances`
  - `add_scan_models`, `delete_scan_models`

If you use any of these commands in conjunction with the [set\\_registered\\_io](#) and [set\\_wrapper\\_chains](#) commands, you will get the following warning message:

```
Warning: Wrapper cells information is out of date:
        use 'analyze_wrapper_cells' to re-analyze wrapper cells prior to
        insertion.
```

## Constraining Input Partition Pins

Input partition pins are block input pins that you cannot directly control from chip-level primary inputs. Referring to [Figure 2-4](#) and [Figure 2-5](#) on pages 26 and 27, the input partition pins are those inputs that come into Block A from Block B. Because these are uncontrollable inputs, you must constrain them to an X value using the [add\\_input\\_constraints](#) command.

## Masking Output Partition Pins

Output partition pins are block output pins that you cannot directly observe from chip-level primary outputs. Referring to [Figure 2-4](#) and [Figure 2-5](#) on pages 26 and 27, the output partition pins are those outputs that go to Block B and Block C. Because these are unobservable outputs, you must mask them with the [add\\_output\\_masks](#) command.

To ensure that masked primary outputs drive inactive values during the testing of other partitions, you can specify that the primary outputs hold a 0 or 1 value during test mode. Special cells called output hold-0 or output hold-1 wrapper cells serve this purpose. By default, the tool uses regular output wrapper cells.

## Manually Specifying Control and Observe Points

You can manually specify which control and observe points to add using the [add\\_control\\_points/add\\_observe\\_points](#) commands. Tessent Scan considers the test points you manually specify to be user-class test points.

When using this command, the *tp\_pin\_pathname* argument specifies the pin pathname of the location where you want to add a control or observe point. If the location is to be a control point, you specify the Control argument with the name of the model to insert (which you define with

[add\\_cell\\_models](#) or the `cell_type` attribute in the library description) and pin(s) to which you want to connect the added gate.

If the location is to be an observe point, you must specify the primary output in which to connect the observe point. You can also specify whether to add a scan cell at the control or observe point.

The following locations in the design are possible candidates for placing control or observe test points:

- Any site in the fanout cone of a declared clock (defined with the [add\\_clocks](#) command).
- The outputs of scanned latches or flip-flops.
- The internal gates of library cells. Only gates driving the top library boundary can have test points.
- Notest points which are set using the [add\\_notest\\_points](#) command.
- The outputs of primitives that can be tri-state.
- The primary inputs for control or observation points.
- The primary outputs for observation points. A primary output driver which also fans out to internal logic could have a control point added, if needed.
- No control points at unobservable sites.
- No observation points at uncontrollable sites.

## Manually Including and Excluding Cells for Scan

Regardless of what type of scan you want to insert, you can manually specify instances or models to either convert or not convert to scan. Tessent Scan uses lists of scan cell candidates and non-scan cells when it selects which sequential elements to convert to scan. You can add specific instances or models to either of these lists. When you manually specify instances or models to be in these lists, these instances are called *user-class* instances. *System-class* instances are those Tessent Scan selects. The following subsections describe how you accomplish this.

## Handling Cells Without Scan Replacements

When Tessent Scan switches from setup to analysis mode, it issues warnings when it encounters sequential elements that have no corresponding scan equivalents. Tessent Scan treats elements without scan replacements as non-scan models and automatically adds them as system-class elements to the non-scan model list. You can display the non-scan model list using the [report\\_nonscan\\_models](#) or [report\\_dft\\_check](#) command.

In many cases, a sequential element may not have a scan equivalent of the currently selected scan type. For example, a cell may have an equivalent mux-DFF scan cell but not an equivalent LSSD scan cell. If you set the scan type to LSSD, Tessent Scan places these models in the non-scan model list. However, if you change the scan type to mux-DFF, Tessent Scan updates the non-scan model list, in this case removing the models from the non-scan model list.

## Specifying Non-Scan Components

Tessent Scan keeps a list of which components it must exclude from scan identification and replacement. To exclude particular instances from the scan identification process, you use the [add\\_nonscan\\_instances](#) command.

For example, you can specify that I\$155/I\$117 and /I\$155/I\$37 are sequential instances you *do not* want converted to scan cells by specifying:

```
SETUP> add_nonscan_instances /I$155/I$117 /I$155/I$37
```

Another method of eliminating some components from consideration for scan cell conversion is to specify that certain models should not be converted to scan. To exclude all instances of a particular model type, you can use the [add\\_nonscan\\_models](#) command.

For example, the following command would exclude all instances of the dff\_3 and dff\_4 components from scan cell conversion.

```
SETUP> add_nonscan_models dff_3 dff_4
```

---

### Note



Tessent Scan automatically treats sequential models without scan equivalents as non-scan models, adding them to the nonscan model list.

---

## Specifying Scan Components

After you decide which specific instances or models you do *not* want included in the scan conversion process, you are ready to identify those sequential elements you *do* want converted to scan. The instances you add to the scan instance list are called user-class instances.

To include particular instances in the scan identification process, use the [add\\_scan\\_instances](#) command. This command lets you specify individual instances, hierarchical instances (for which all lower-level instances are converted to scan), or control signals (for which all instances controlled by the signals are converted to scan).

For example, the following command ensures the conversion of instances /I\$145/I\$116 and /I\$145/I\$138 to scan cells when Tessent Scan inserts scan circuitry.

```
SETUP> add_scan_instances /I$145/I$116 /I$145/I$138
```

To include all instances of a particular model type for conversion to scan, use the [add\\_scan\\_models](#) command. For example, the following command ensures the conversion of all instances of the component models dff\_1 and dff\_2 to scan cells when Tessent Scan inserts scan circuitry.

```
SETUP> add_scan_models dff_1 dff_2
```

## Related Scan and Nonscan Commands

- [delete\\_nonscan\\_instances](#) — Deletes instances from the non-scan instance list.
- [delete\\_nonscan\\_models](#) — Deletes models from the non-scan model list.
- [delete\\_scan\\_instances](#) — Deletes instances from the scan instance list.
- [delete\\_scan\\_models](#) — Deletes models from the scan model list.
- [report\\_nonscan\\_models](#) — Displays the models in the non-scan instance list.
- [report\\_sequential\\_instances](#) — Displays information and testability data for sequential instances.
- [report\\_scan\\_models](#) — Displays models in the scan model list.

## Reporting Scannability Information

Scannability checking is a modified version of clock rules checking that determines which non-scan sequential instances to consider for scan. You may want to examine information regarding the scannability status of all the non-scan sequential instances in your design. To display this information, you use the [report\\_dft\\_check](#) command. This command displays the results of scannability checking for the specified non-scan instances, for either the entire design or the specified (potentially hierarchical) instance.



When you perform a `report_dft_check` command there is typically a large number of nonscan instances displayed, as shown in the sample report in [Figure 5-5](#).

**Figure 5-5. Example Report from `report_dft_check` Command**

|                                 |                        |            |               |                                |
|---------------------------------|------------------------|------------|---------------|--------------------------------|
| SCANNABLE                       | IDENTIFIED             | CLK0_7     | /I_3          | dff (156)                      |
| SCANNABLE                       | IDENTIFIED             | CLK0_7     | /I_2          | dff (157)                      |
| SCANNABLE                       | IDENTIFIED             | CLK0_7     | /I_235        | dff (158)                      |
| SCANNABLE                       | IDENTIFIED             | CLK0_7     | /I_237        | dff (159)                      |
| SCANNABLE                       | IDENTIFIED             | CLK0_7     | /I_236        | dff (160)                      |
| SCANNABLE                       | IDENTIFIED             | Test-logic | /I_265        | dff (161)                      |
| Clock #1: F /I_265/clock        |                        |            |               |                                |
| SCANNABLE                       | IDENTIFIED             | Test-logic | /I_295        | dff (162)                      |
| Clock #1: F /I_295/clock        |                        |            |               |                                |
| SCANNABLE                       | IDENTIFIED             | Test-logic | /I_298        | dff (163)                      |
| Clock #1: F /I_298/clock        |                        |            |               |                                |
| SCANNABLE                       | IDENTIFIED             | Test-logic | /I_296        | dff (164)                      |
| Clock #1: F /I_296/clock        |                        |            |               |                                |
| SCANNABLE                       | IDENTIFIED             | Test-logic | /I_268        | dff (165)                      |
| Clock #1: F /I_268/clock        |                        |            |               |                                |
| SCANNABLE                       | IDENTIFIED             | CLK0_7     | /I_4          | dff (166)                      |
| SCANNABLE                       | IDENTIFIED             | CLK0_7     | /I_1          | dff (167)                      |
| SCANNABLE                       | <b>DEFINED-NONSCAN</b> | Test-logic | <b>/I_266</b> | <b>dfscc (168) Stable-high</b> |
| <b>Clock #1: F /I_266/clock</b> |                        |            |               |                                |
| SCANNABLE                       | DEFINED-NONSCAN        | CLK0_7     | /I_238        | dfscc (169)                    |
| SCANNABLE                       | DEFINED-NONSCAN        | Test-logic | /I_297        | dfscc (170) Stable-high        |
| Clock #1: F /I_297/clock        |                        |            |               |                                |
| SCANNABLE                       | DEFINED-NONSCAN        | Test-logic | /I_267        | dfscc (171) Stable-high        |
| Clock #1: F /I_267/clock        |                        |            |               |                                |

The fields at the end of each line in the nonscan instance report provide additional information regarding the classification of a sequential instance. Using the instance `/I_266` (highlighted in maroon), the “Clock” statement indicates a problem with the clock input of the sequential instance. In this case, when the tool does a trace back of the clock, the signal doesn’t trace back to a defined clock. The message indicates that the signal traced connects to the clock input of this non-scan instance, and doesn’t trace back to a primary input defined as a clock. If several nodes are listed (similarly for “Reset” and “Set”), it means that the line is connected to several endpoints (sequential instances or primary inputs).

This “Clock # 1 F /I\_266/clock” issue can be resolved by either defining the specified input as a clock or allowing Tessent Scan to add a test clock for this instance.

- [report\\_control\\_signals](#) — Displays control signal information.
- [report\\_statistics](#) — Displays a statistics report.
- [report\\_sequential\\_instances](#) — Displays information and testability data for sequential instances.

## Automatic Recognition of Existing Shift Registers

Tessent Scan automatically identifies the shift register structures in the input netlist and tries to preserve their original connections (shift orders) when they are stitched into scan chains. In this attempt, Tessent Scan converts only the first flip-flop of a shift register into a scan cell (if originally a nonscan cell), and preserves the remaining flip-flops in the shift register as nonscan (or replaces them with nonscan cells if originally scan cells).

This approach has several potential benefits such as the reduction of scan cells in the design, and therefore fewer scan path muxes, and the better locality of scan path connections due to the preservation of functional connections.

The following sections describe the process by which Tessent Scan identifies and converts scan cells.

### Identifying the Shift Registers

Tessent Scan identifies shift registers by performing a structural backward tracing in the flattened model of the design. The tracing starts from the data pin of the `_dff` primitive of a sequential cell and ends at the output pin of the `_dff` primitive of another sequential cell. The tracing continues only on a single sensitized path. The sensitization path of a combinational logic gate in the tracing path is checked based on the state-stability values calculated by the tool during DRC rule checking. When “-IGNORE\_CONSTRAINT\_values ON” is specified with the `set_shift_register_identification` command, the state-stability values are ignored during tracing. As a result, multiple-input gates (AND, OR, MUX, etc.) between sequential cells cannot be traced during shift register identification. Single-input gates (BUF and INV), on the other hand, can always be traced. This switch is Off, by default.

The identification occurs when switching to analysis mode, after DRC rule checking is completed, as indicated in the following transcript.

```
// -----  
// Begin shift register identification process for 9971 sequential  
// instances.  
// -----  
// Number of shift register flops recorded for scan insertion: 3798  
// (38.09%)  
// Number of shift registers recorded for scan insertion: 696  
// Longest shift register has 15 flops.  
// Shortest shift register has 2 flops.  
// Potential number of nonscan flops to be converted to scan cells: 696  
// Potential number of scan cells to be converted to nonscan flops: 25
```

Shift register identification assumes the following:

1. Flip-flops that constitute a shift register reside under the same hierarchical instance.
2. Flip-flops that constitute a shift register use the same clock signal.

3. Multiple clock edges are allowed in the shift register structure as long as no lockup cells are required (no TE-LE transitions occur). When lockup cells are required (as in LE-TE transitions), the tool breaks the shift register at this location.
4. Both nonscan and scan flip-flops are considered for identification. However, every nonscan flip-flop should have a mapping scan flip-flop model in the ATPG library and every scan flip-flop should have a mapping nonscan flip-flop model in the ATPG library. In addition, a scan flip-flop should satisfy the following requirements:
  - Its scan input pin is not functionally driven (either dangling or tied to a constant, or looped back from Q/QB output).
  - Its scan enable pin is not functionally driven, and is tied to a constant signal to sensitize the data input pin of the sequential cell such that this input is preserved as the shift path. The scan enable pin is not considered functionally driven if a global scan enable pin (defined using `set_scan_insertion -SEN`) is the driver.
5. Shift registers with multiple branches are identified such that each branch is a separate shift register. The flip-flops on the trunk are included in one of the branches.
6. Shift registers with sequential feedback loops are identified such that the first cell of the shift register is determined by the tool either randomly or based on the availability of the scan cell in the loop.

## Stitching Shift Register Cells Into Scan Chains

Nonscan flip-flops are replaced with their scan-equivalent flip-flops before stitching them into scan chain structures. The identified shift register flip-flops are handled as follows.

1. The first flip-flop is replaced with the equivalent scan flip-flop. If the first flip-flop is originally a scan flip-flop, it is preserved as is.
2. All remaining flip-flops are preserved as nonscan. If they are originally scan flip-flops, they are converted into nonscan flip-flops.

After performing the scan chain insertion, the `report_shift_registers` command may list fewer and shorter shift registers than what were originally identified by switching to DFT mode. The following lists the main reasons why originally identified shift registers can be modified by the scan chain stitching.

1. All scan cell candidates are sorted based on clock, edge, and hierarchy before stitching. The sorting process preserves the cells of the shift registers in exactly the same order as they are functionally connected. Sorted scan cell candidates are then distributed over scan chains. When placing a shift register into a scan chain, it may be cut to a desired length to satisfy the specified/calculated scan chain length.
2. The first or the last cell in a shift register may be removed from the shift register if it is positioned to be at the clock transition between the cells in the scan chain, or positioned to be the first or last cell in the scan chain. These modifications are performed to move

shift register cells from the START and STOP declarations in a scanDEF file that will be generated with a `write_scan_order` command. It is unlikely that the first or the last cell in a shift register will be positioned to be the first or last cell in the scan chain, as the tool when distributing scan cells to chains tries to avoid placing shift registers at the beginning or tail of a scan chain. This is likely to happen only when the percentage of shift registers are very high in the design compared to floating flops, and the tool doesn't have a choice to place any other potential scan cell other than a shift register at the beginning/end of a chain.

The stitching of the flip-flops inside a shift register structure is skipped but the following connections are performed:

1. The scan input pin of the first flip-flop.
2. The scan enable pin of the first flip-flop.
3. The scan output pin of the last flip-flop. The scan output pin on the last flip-flop will be determined by checking the load on the Q and QN ports.

## Reporting Shift Registers

You may want to report `_shift_registers` in the design. To display this information, you use the [report\\_shift\\_registers](#) command. This command reports the identified shift registers in the design after switching to analysis mode. The tool tries to preserve the original connections inside the identified shift. For each identified shift register, this command reports the following information:

- Length
- Hierarchical path where the shift register flip-flops reside
- First and last flip-flop instance name unless the `-verbose` switch is specified in which case all flip-flops in the shift registers are reported

Scan cells can also be reported by the [report\\_scan\\_cells](#) command after the [insert\\_test\\_logic](#) command is executed. If any shift registers are identified in the netlist, a column is added to the report. The column contains a tool-assigned shift register ID number and a cell number that indicates the order in which the flip-flops are originally connected in the shift register structures.

## The Identification Process

Once you complete the proper setup, the identification process for any of the test structures is done automatically when you switch to dft or analysis mode.

During the identification process, a number of messages may be issued about the identified structures.

To identify the dedicated and shared wrapper cells, you can use the [analyze\\_wrapper\\_cells](#) command.

**Note**

If you want to start the selection process anew each time, you must use the [reset\\_state](#) command to clear the existing scan candidate list.

## Reporting Identification Information

If you want a statistical report on all aspects of scan cell identification, you can enter the [report\\_statistics](#) Tessent Scan command.

This command lists the total number of sequential instances, user-defined non-scan instances, user-defined scan instances, system-identified scan instances, scannable instances with test logic, and the scan instances in preexisting chains identified by the rules checker.

The [report\\_sequential\\_instances](#) command displays information and testability data for sequential instances.

## Inserting Test Structures

Typically, after identifying the test structures you want, you perform some test synthesis setup and then insert the structures into the design. The additional setup varies somewhat depending on the type of test structure you select for insertion. The following logically-ordered subsections discuss how to perform these tasks.

### Setting Up for Internal Scan Insertion

As part of the internal scan insertion setup, you may want to set some scan chain parameters, such as the scan input and output port names, and the enable and clock ports. If you specify a port name that matches an existing port of the design, the existing port is used as the scan port. If the specified port name does not exist, Tessent Scan creates a new port with the specified name. If you use an existing, connected output port, Tessent Scan also inserts a mux at the output to select data from either the scan chain or the design, depending on the value of the scan enable signals.

### Naming Scan Input and Output Ports

Before Tessent Scan stitches the identified scan instances into a scan chain, it needs to know the names of various pins, such as the scan input and scan output. If the pin names you specify are existing pins, Tessent Scan will connect the scan circuitry to those pins. If the pin names you specify do not exist, Tessent Scan adds these pins to the design. By default, Tessent Scan adds pins for chainX scan ports and names them scan\_inX and scan\_outX (where X represents the number of the chain).

To give scan ports specific names (other than those created by default), you can use the [add\\_scan\\_pins](#) command.

You must specify the scan chain name, the scan input pin, and the scan output pin. Additionally, you may specify the name of the scan chain clock. For existing pins, you can specify top module pins or pins of lower level instances.

After the scan cells are partitioned and grouped into potential scan chains (before scan chain insertion occurs) Tessent Scan considers some conditions in assigning scan pins to scan chains:

1. Whether the potential scan chain has all or some of the scan cells driven by the specified clock ([add\\_scan\\_pins](#) -Clock). If yes, then the scan chain is assigned to the specified scan input and output pins.
2. Whether the output of the scan candidate is directly connected to a declared output pin. If yes, then the scan input and output pins are assigned to the scan chain containing the scan cell candidate.
3. Any scan chains not assigned to scan input/output pins using conditions 1 and 2 are assigned based on the order in which you declared the scan input/output pins using the [add\\_scan\\_pins](#) command.

If a fixed-order file is specified along with the -Fixed option in the [insert\\_test\\_logic](#) command, conditions 1 and 2 are ignored and the chain\_id in the fixed-order file is then sorted in increasing order. The chain with the smallest chain\_id receives the first specified scan input/output pins. The chain with the second smallest chain\_id receives the second specified scan input/output pins, and so on. If you did not specify enough scan input/output pins for all scan chains, then Tessent Scan creates new scan input/output pins for the remaining scan chains.

For information on the format of the fixed-order file, refer to the [insert\\_test\\_logic](#) description in the *Tessent Shell Reference Manual*.

- [delete\\_scan\\_pins](#) — Deletes scan chain inputs, outputs, and clock names.
- [report\\_scan\\_pins](#) — Displays scan chain inputs, outputs, and clock names.
- [set\\_scan\\_pins](#) — Specifies the index or bus naming conventions for scan input and output pins.

## Naming the Enable and Clock Ports

The enable and clock parameters include the pin names of the scan enables, test enable, test clock, new scan clock, scan master clock, and scan slave clock. Additionally, you can specify the names of the set and reset ports and the RAM write and read ports in which you want to add test logic, along with the type of test logic to use. You do this using the [set\\_scan\\_insertion](#) command. If you do not specify this command, the default pin names are scan\_en, scan\_en\_in, scan\_en\_out, test\_en, test\_clk, scan\_clk, scan\_mclk, scan\_sclk, scan\_set, scan\_reset, write\_clk,

and read\_clk, respectively. If you want to specify the names of existing pins, you can specify top module pins or dangling pins of lower level modules.

---

**Note**

If Tessent Scan adds more than one test clock, it names the first test clock the specified or default <name> and names subsequent test clocks based on this name plus a unique number.

---

The -Muxed and -Disabled switches specify whether Tessent Scan uses an AND gate or MUX gate when performing the gating. If you specify the -Disabled option, then for gating purposes Tessent Scan ANDs the test enable signal with the set and reset to disable these inputs of flip-flops. If you specify the -Muxed option, then for muxing purposes Tessent Scan uses any set and reset pins defined as clocks to multiplex with the original signal. You can specify the -Muxed and -Disabled switches for individual pins by successively issuing the [set\\_scan\\_insertion](#) command.

If Tessent Scan writes out a test procedure file, it places the scan enable at 1 (0) if you specify -Active High (Low).

---

**Note**

If the test enable and scan enable have different active values, you must specify their active values using different commands. Set the test enable active value using the [set\\_scan\\_insertion](#) command. Set the scan enable active value using the [set\\_scan\\_enable](#) command.

---

After setting up for internal scan insertion, refer to “[Running the Insertion Process](#)” on page 144 to complete insertion of the internal scan circuitry.

## Attaching Head and Tail Registers to the Scan Chain

A *head register* is a **non-scan** DFF connected at the beginning of a scan chain. A *tail register* is a **scan** DFF connected at the end of the scan chain. You can have Tessent Scan identify certain cells as the head and tail registers for a scan chain of mux-DFF scan type, and connect them at the beginning and the end of the chain. The head and tail cells are not inserted by Tessent Scan; they are only identified in the original design.

During test logic insertion, Tessent Scan attaches the non-scan head register’s output to the beginning of the scan chain, performs scan replacement on the tail register, and then attaches the scan tail register’s input to the end of the scan chain. If there is no scan replacement in the ATPG library for the tail register, a MUX is added to include the tail DFF into the scan chain.



**Note**



No design rule checks are performed from the scan\_in pin to the output of the head register and from the output of the tail register to the scan\_out pin. You are responsible for making those paths transparent for scan shifting.

---

**Note**



Tessent Scan does not determine the associated top-level pins that are required to be identified for the add\_scan\_chains command. You are responsible for adding this information to the dofile that Tessent Scan creates using the write\_atpg\_setup command. You must also provide the pin constraints that cause the correct behavior of the head and tail registers.

---

You can specify head and tail registration of a scan chain via the [add\\_scan\\_pins](#) command. You need to specify the head register output pin as the scan input pin, and the tail register input pin as the scan output pin, along with the -Registered switch.

## Buffering Test Pins

When the tool inserts scan into a design, the test pins (such as scan enable, test enable, test clock, scan clock, scan master clock, and scan slave clock) may end up driving a lot of fanouts. If you want the tool to limit the number of fanouts and insert buffer trees instead, you can use the [add\\_buffer\\_insertion](#) command.

- [delete\\_buffer\\_insertion](#) — Deletes added buffer insertion information.
- [report\\_buffer\\_insertion](#) — Displays inserted buffer information.

## Running the Insertion Process

The [insert\\_test\\_logic](#) command inserts all of the previously identified test structures into the design. This includes internal scan (sequential and scan-sequential types), wrapper chain, test logic, and test points.

When you issue this command for scan insertion (assuming appropriate prior setup), the tool converts all identified scannable memory elements to scan elements and then stitches them into one or more scan chains. If you select wrapper chains for insertion, the tool converts the non-scan cells identified for wrapper chains to wrapper cells and stitches them into scan chains separate from internal scan chains.

The scan circuitry insertion process may differ depending on whether you insert scan cells and connect them up front, or insert and connect them after layout data is available. the tool allows you to insert scan using both methods.

The insert\_test\_logic command has a number of different options, most of which apply primarily to internal scan insertion.



- If you are using specific cell ordering, you can specify a filename of user-identified instances (in either a fixed or random order) for the stitching order.
- The `-Max_length` option lets you specify a maximum length to the chains.
- The `-NOlimit` switch allows an unlimited chain length.
- The `-NNumber` option lets you specify the number of scan chains for the design.
- The `-Clock` switch lets you choose whether to merge two or more clocks on a single chain.
- The `-Edge` switch lets you choose whether to merge stable high clocks with stable low clocks on chains.

The subsection that follows, “[Merging Scan Chains with Different Shift Clocks](#)“, discusses some of the issues surrounding merging chains with different clocks.

- The `-CConnect` option lets you specify whether to connect the scan cells and scan-specific pins (`scan_in`, `scan_enable`, `scan_clock`, etc.) to the scan chain (which is the default mode), or just replace the scan candidates with scan equivalent cells. If you want to use layout data, you should replace scan cells (using the `-connect off` switch), perform layout, obtain a placement order file, and then connect the chain in the appropriate order (using the `-filename filename -fixed` options). This option is affected by the settings in the [set\\_test\\_logic](#) command. The other options to the `-CConnect` switch specify how to handle the input/output scan pins when not stitching the scan cells into a chain.
- The `-Scan`, `-Test_point`, and `-Ram` switches let you turn scan insertion, test point insertion and RAM gating on or off.
- The `-Verilog` switch causes the tool to insert buffer instances, rather than use the “assign” statement, for scan output pins that also fan out as functional outputs.

If you do not specify any options, the tool stitches the identified instances into default scan chain configurations.

---

#### Note



Because the design is significantly changed by the action of this command, the original flattened, gate-level simulation model created when you entered the analysis system mode is deleted.

---

## Merging Scan Chains with Different Shift Clocks

You can merge scan cells with different shift clocks into the same scan chain using the `-clock merge` option of the [insert\\_test\\_logic](#) command. To avoid synchronization problems, the tool places the scan cells using the same shift clock adjacent to each other in the chain and then inserts lockup cells between the scan cells that use different shift clocks. Lockup cells are inserted on the scan path to synchronize clock domains in adjacent scan cells and do not interfere with the functional operation of the circuit.

### Note



You can use clock groups to minimize the lockup cell insertion. The tool places the scan cells with shift clocks from different clock groups in separate scan chains. Lockup latches are then inserted only between the cells with different shift clocks in the same clock group. For more information, refer to the [add\\_clock\\_groups](#) description in the *Tessent Shell Reference Manual*.

---

Use this procedure to merge scan cells with different shift clocks.

## Procedure

1. Define latch and inverter library models to use for lockup cells. For example:

```
add_cell_models dlat1a -type dlat enable data -active high|Low
add_cell_models inv01 -type inv
```

The cell type for the latch must be either a DLAT or DFF and the active edge specified must reflect the overall inversion of the clock pin inside the model.

The inverter is used along with the lockup cell model, when necessary, for correct lockup cell insertion. The inverter model can also be defined with `cell_type` attribute in the model definition.

For more information, see the [add\\_cell\\_models](#) description in the *Tessent Shell Reference Manual*.

2. Set the tool to insert lockup cells. For example:

```
set_lockup_cell on -type dlat
```

You can also use a DFF model for a lockup cell, in which case you need to define a DFF model using the Add Cell Model and use “-type dff” switch for the `set_lockup_cell` command.

For more information, see the [set\\_lockup\\_cell](#) description in the *Tessent Shell Reference Manual*.

### Note

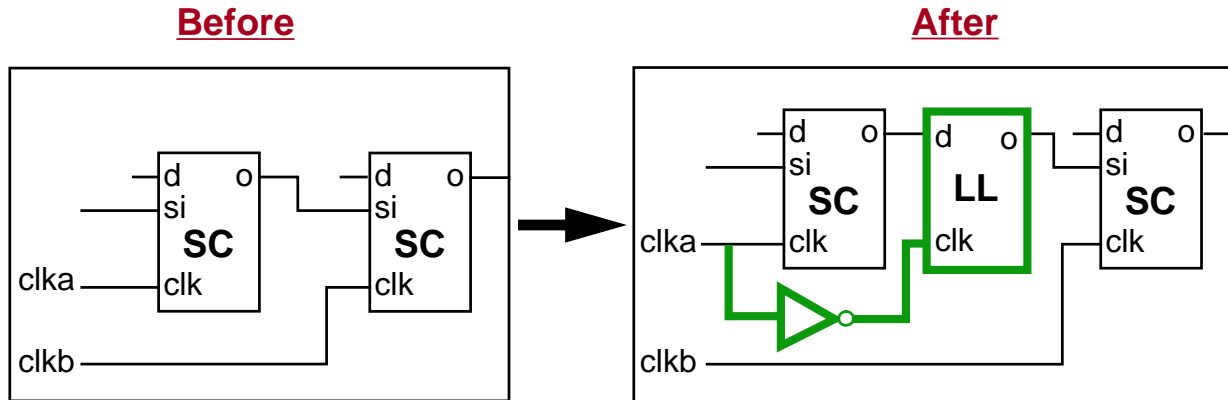


You can also specify the exact lockup cell locations in scan chains using a cell order file that specifies the stitching order of scan cells in scan chains. For more information, see the [insert\\_test\\_logic](#) description in the *Tessent Shell Reference Manual*.

---

[Figure 5-6](#) illustrates lockup cell insertion. The extra inverter on the clock line of the lockup cell provides a half a cycle delay to ensure synchronization of the clock domains.

**Figure 5-6. Lockup Cell Insertion**



## Examples

The following example defines two different groups of clocks, identifies a latch model and inverter model to use for lockup cells, enables lockup cell insertion, and performs the insertions. The *-Clock Merge* option combines the scan cells associated with each of the specified clock groups into a scan chain when the test logic is inserted.

```
add_clocks 0 clk1 clk2 clk3
add_clocks 1 clk4 clk5 clk6
add_clock_groups group1 clk1 clk2 clk3
add_clock_groups group2 clk4 clk5 clk6
add_cell_models dff04 -type dff clk data
add_cell_models inv -type inv
set_lockup_cell on -type dff
insert_test_logic -scan on -clock merge
```

The tool creates two scan chains, one for each clock group and inserts lockup cells between the clock domains of the same clock group, {clk1, clk2, clk3} or {clk4, clk5, clk6}.

- [delete\\_clock\\_groups](#) — Deletes the specified clock groups.
- [report\\_clock\\_groups](#) — Reports the added clock groups.
- [report\\_dft\\_check](#) — Displays and writes the scannability check status for all non-scan instances.
- [report\\_scan\\_cells](#) — Displays a list of all scan cells.
- [report\\_scan\\_chains](#) — Displays scan chain information.
- [report\\_scan\\_groups](#) — Displays scan chain group information.

## Saving the New Design and ATPG Setup

After test structure insertion, the tool releases the current flattened model and has a new hierarchical netlist in memory. Thus, you should save this new version of your design. Additionally, you should save any design information that the ATPG process might need.

### Writing the Netlist

You can save the netlist for your new design by issuing the `write_design` command.

### Issues with the New Version of the Netlist

The following lists some important issues concerning netlist writing:

- Tessent Scan is not intended for use as a robust netlist translation tool. Thus, you should always write out the netlist in the same format in which you read the original design.
- If a design contains only one instantiation of a module, and the tool modifies the instance by adding test structures, the instantiation retains the original module name.
- When the tool identically modifies two or more instances of the same module, all modified instances retain the original module name. This generally occurs for scan designs.
- If a design contains multiple instantiations of a module, and the tool modifies them differently, the tool derives new names for each instance based on the original module name.
- Tessent Scan assigns “net” as the prefix for new net names and “uu” as the prefix for new instance names. It then compares new names with existing names (in a case-insensitive manner) to check for naming conflicts. If it encounters naming conflicts, it changes the new name by appending an index number.

### Writing the Test Procedure File and Dofile for ATPG

If you plan to use the ATPG tool, you can use Tessent Scan to create a dofile (for setting up the scan information) and a test procedure file (for operating the inserted scan circuitry). For information on the new test procedure file format, see “[Test Procedure File](#)” in the *Tessent Shell User’s Manual*.

To create test procedure files, issue the `write_atpg_setup` command.

### Running Rules Checking on the New Design

You can verify the correctness of the added test circuitry by running the full set of rules checks on the new design. To do this, return to setup mode after scan insertion, delete the circuit setup, run the dofile produced for ATPG, and then return to analysis mode. This enables rules

checking on the added scan circuitry to ensure it operates properly before you go to the ATPG process.

For example, if the tool adds a single scan chain and writes out an ATPG setup file named *scan\_design.dofile*, enter something like the following:

```
> set_system_mode setup  
> delete_clocks -all  
> dofile scan_design.dofile  
> set_system_mode analysis
```

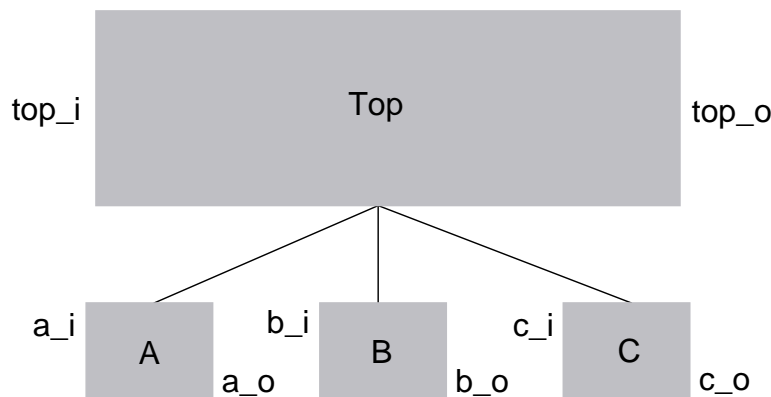
## Exiting Tessent Scan

When you finish the Tessent Scan session, enter the exit command.

## Inserting Scan Block-by-Block

Scan insertion is “block-by-block” when the tool first inserts scan into lower-level hierarchical blocks and then connects them together at a higher level of hierarchy. For example, [Figure 5-7](#) shows a module (Top) with three submodules (A, B, and C).

**Figure 5-7. Hierarchical Design Prior to Scan**



Using block-by-block scan insertion, the tool inserts scan (referred to as “sub-chains”) into blocks A, B, and C, prior to insertion in the Top module. When A, B, and C already contain scan, inserting scan into the Top module is equivalent to inserting any scan necessary at the top level, and then connecting the existing scan circuitry in A, B, and C at the top level.

## Verilog Flow Example

The following shows the basic procedure for adding scan circuitry block-by-block, as well as the input and results of each step.

### 1. Insert scan into block A.

- a. Invoke Tessent Scan on *a.hdl*.  
Assume that the module interface is:

```
A(a_i, a_o)
```

- b. Insert scan.  
Set up the circuit, run rules checking, insert the desired scan circuitry.
- c. Write out scan-inserted netlist.  
Write the scan-inserted netlist to a new filename, such as *a\_scan.hdl*. The new module interface may differ, for example:

```
A(a_i, a_o, sc_i, sc_o, sc_en)
```

- d. Write out the subchain dofile.  
Use the [write\\_subchain\\_setup](#) command to write a dofile called *a.do* for the scan-inserted version of A. The `write_subchain_setup` command uses the [add\\_sub\\_chains](#) command to specify the scan circuitry in the individual module of the design. Assuming that you use the mux-DFF scan style and the design block contains 7 sequential elements converted to scan, the subchain setup dofile could appear as follows:

```
DFT> add_sub_chains /user/jdoe/designs/design1/A chain1 sc_i sc_o  
7 mux_scan sc_en
```

- e. Exit the tool.
2. **Insert scan into block B.**  
Follow the same procedure as in block A.
3. **Insert scan into block C.**  
Follow the same procedure as in blocks A and B.
4. **Concatenate the individual scan-inserted netlists into one file.**

```
$ cat top.hdl a_scan.hdl b_scan.hdl c_scan.hdl > all.hdl
```

**5. Stitch together the chains in blocks A, B, and C.**

- a. Invoke Tessent Scan on *all.hdl*.  
Assume at this point that the module interface is:

```
TOP(top_i, top_o)  
A(a_i, a_o, sc_i, sc_o, sc_en)  
B(b_i, b_o, sc_i, sc_o, sc_en)  
C(c_i, c_o, sc_i, sc_o, sc_en)
```

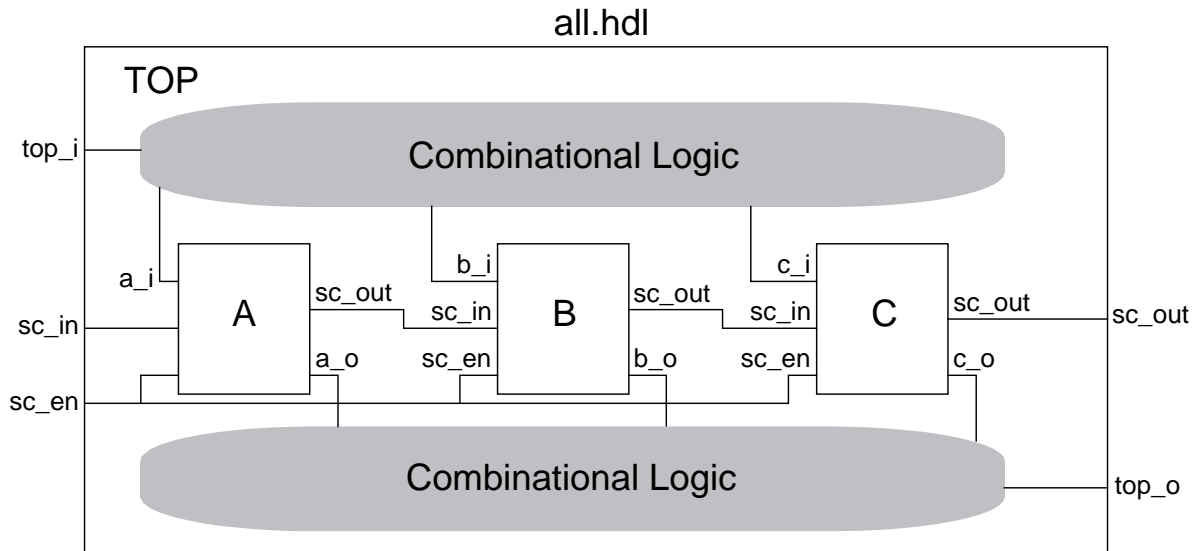
- b. Run each of the scan subchain dofiles (*a.do*, *b.do*, *c.do*).
- c. Insert the desired scan circuitry into the *all.hdl* design.
6. Write out the netlist and exit.

At this point the module interface is:

```
TOP(top_i, top_o, sc_i, sc_o, sc_en)
A(a_i, a_o, sc_i, sc_o, sc_en)
B(b_i, b_o, sc_i, sc_o, sc_en)
C(c_i, c_o, sc_i, sc_o, sc_en)
```

Figure 5-8 shows a schematic view of the design with scan connected in the Top module.

**Figure 5-8. Final Scan-Inserted Design**







# Chapter 6

## Generating Test Patterns

---

Figure 6-1 shows the process for generating test patterns for your design.

**Figure 6-1. Test Generation Procedure**

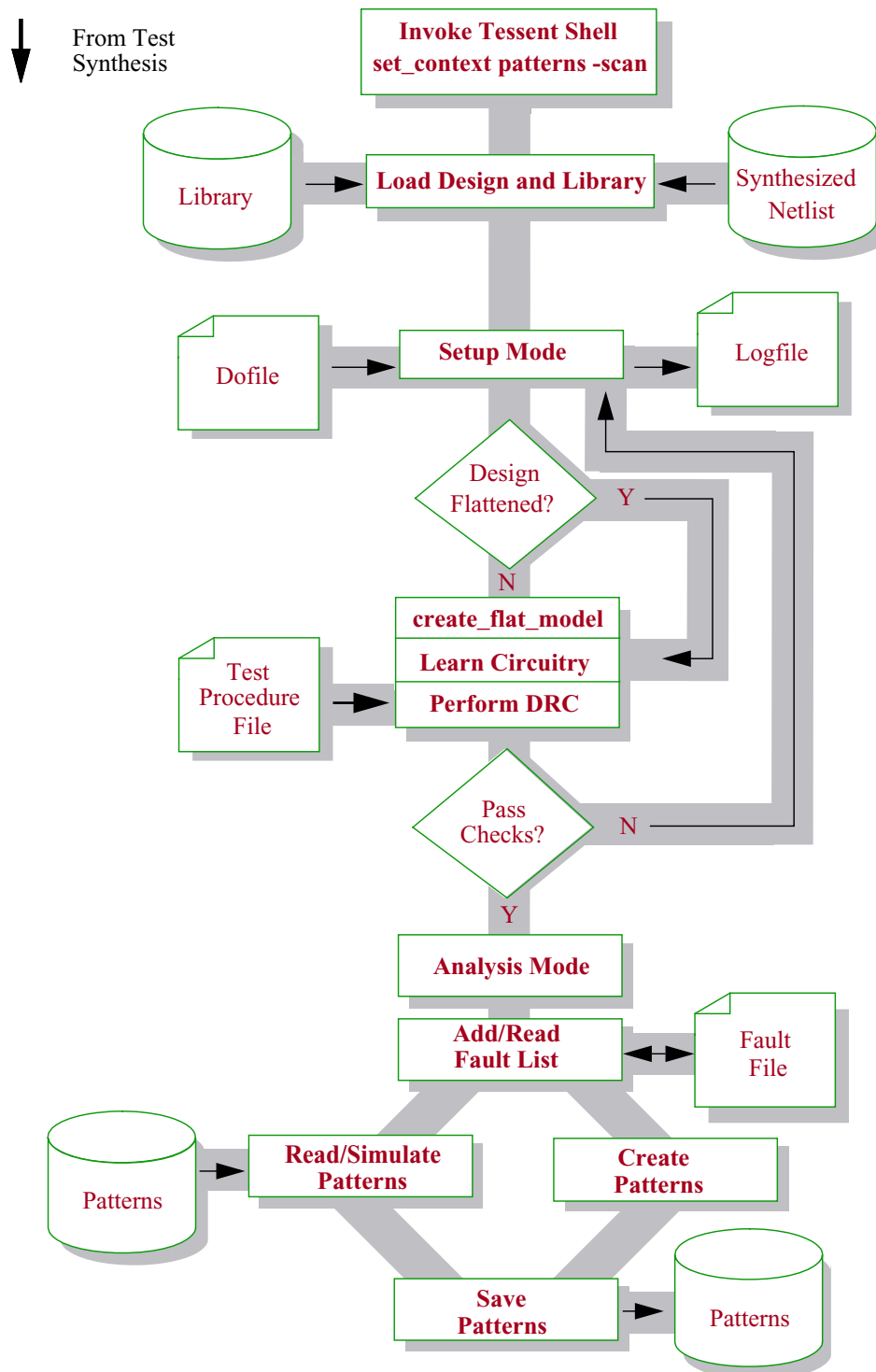


This section discusses each of the tasks outlined in Figure 6-1. You use the ATPG tool (and possibly ModelSim, depending on your test strategy) to perform these tasks.

## ATPG Basic Tool Flow

Figure 6-2 shows the basic process flow for the ATPG tool.

Figure 6-2. Overview of ATPG Tool Usage



The following list describes the basic process for using the ATPG tool:

1. Invoke Tessent Shell using the “tessent -shell” command. Set the context to “patterns -scan” using the set\_context command, which allows you to access ATPG functionality.
2. The ATPG tool requires a structural (gate-level) design netlist and a DFT library, which you accomplish with the [read\\_cell\\_library](#) and [read\\_verilog](#) commands, respectively. “[ATPG Tool Inputs and Outputs](#)” on page 156 describes which netlist formats you can use with the ATPG tool. Every element in the netlist must have an equivalent description in the specified DFT library. The “[Design Library](#)” section in the *Tessent Cell Library Manual* gives information on the DFT library. The tool reads in the library and the netlist, parsing and checking each.
3. After reading the library and netlist, the tool goes into setup mode. Within setup mode, you perform several tasks, using commands either interactively or through the use of a dofile. You can set up information about the design and the design’s scan circuitry. “[Setting Up Design and Tool Behavior](#)” on page 163 documents this setup procedure. Within setup mode, you can also specify information that influences simulation model creation during the design flattening phase.
4. After performing all the desired setup, you can exit setup mode, which triggers a number of operations. If this is the first attempt to exit setup mode, the tool creates a flattened design model. This model may already exist if a previous attempt to exit setup mode failed or you used the create\_flat\_model command. “[Model Flattening](#)” on page 76 provides more details about design flattening.
5. Next, the tool performs extensive learning analysis on this model. “[Learning Analysis](#)” on page 81 explains learning analysis in more detail.
6. Once the tool creates a flattened model and learns its behavior, it begins design rules checking. The “[Design Rule Checking](#)” section in the *Tessent Shell Reference Manual* gives a full discussion of the design rules.
7. Once the design passes rules checking, the tool enters analysis mode, where you can perform simulation on a pattern set for the design. For more information, refer to “[Good-Machine Simulation](#)” on page 178 and “[Fault Simulation](#)” on page 176.
8. At this point, you may want to create patterns. You can also perform some additional setup steps, such as adding the fault list. “[Setting Up the Fault Information for ATPG](#)” on page 179 details this procedure. You can then run ATPG on the fault list. During the ATPG run, the tool also performs fault simulation to verify that the generated patterns detect the targeted faults.

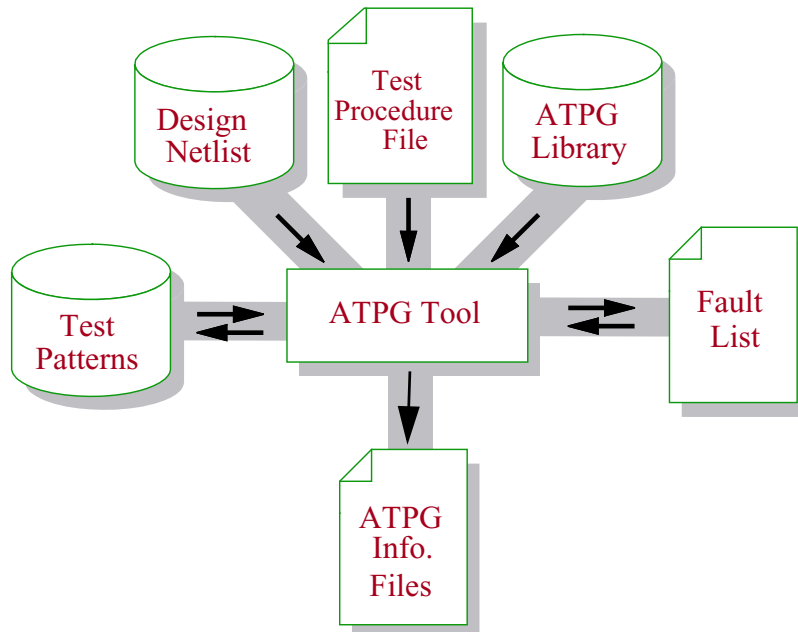
In either case (full or partial scan), you can run ATPG under different constraints, or augment the test vector set with additional test patterns, to achieve higher test coverage. “[Performing ATPG](#)” on page 183 covers this subject.

After generating a test set with the ATPG tool, you should apply timing information to the patterns and verify the design and patterns before handing them off to the vendor. “[Verifying Test Patterns](#)” on page 286 documents this operation.

## ATPG Tool Inputs and Outputs

Figure 6-3 shows the inputs and outputs of the ATPG tool.

**Figure 6-3. ATPG Tool Inputs and Outputs**



The ATPG tool uses the following inputs:

- **Design** — The supported design data format is gate-level Verilog. Other inputs also include 1) a cell model from the design library and 2) a previously-saved, flattened model.
- **Test Procedure File** — This file defines the operation of the scan circuitry in your design. You can generate this file by hand, or Tessent Scan can create this file automatically when you issue the command `write_atpg_setup`.
- **Library** — The design library contains descriptions of all the cells used in the design. The tool uses the library to translate the design data into a flat, gate-level simulation model for use by the fault simulator and test generator.
- **Fault List** — The tool can read in an external fault list. The tool uses this list of faults and their current status as a starting point for test generation.
- **Test Patterns** — The tool can read in externally generated test patterns and use those patterns as the source of patterns to be simulated.

The tool produces the following outputs:

- **Test Patterns** — The tool generates files containing test patterns. They can generate these patterns in a number of different simulator and ASIC vendor formats. “[Test Pattern Formatting and Timing](#)” on page 337 discusses the test pattern formats in more detail.
- **ATPG Information Files** — These consist of a set of files containing information from the ATPG session. For example, you can specify creation of a log file for the session.
- **Fault List** — This is an ASCII-readable file that contains internal fault information in the standard Mentor Graphics fault format.

## Understanding the ATPG Method

To understand how the ATPG tool operates, you should understand the basic ATPG process, timing model, and basic pattern types that the tool produces. The following subsections discuss these topics.

### Basic ATPG Process

The ATPG tool has default values set, so when you start ATPG for the first time (by issuing the `create_patterns` command), the tool performs an efficient combination of random pattern fault simulation and deterministic test generation on the target fault list. “[The ATPG Process](#)” on page 30 discusses the basics of random and deterministic pattern generation.

### Random Pattern Generation Using the ATPG Tool

The tool first performs random pattern fault simulation for each capture clock, stopping when a simulation pattern fails to detect at least 0.5% of the remaining faults. The tool then performs random pattern fault simulation for patterns without a capture clock, as well as those that measure the primary outputs connected to clock lines.

---

**Note**

ATPG constraints and circuitry that can have bus contention are not optimal conditions for random pattern generation. If you specify ATPG constraints, The tool does not perform random pattern generation.

---

### Deterministic Test Generation Using the ATPG Tool

Some faults have a very low chance of detection using a random pattern approach. Thus, after it completes the random pattern simulation, the tool performs deterministic test generation on selected faults from the current fault list. This process consists of creating test patterns for a set of (somewhat) randomly chosen faults from the fault list.

During this process, the tool identifies and removes redundant faults from the fault list. After it creates enough patterns for a fault simulation pass, it displays a message that indicates the number of redundant faults, the number of ATPG untestable faults, and the number of aborted faults that the test generator identifies. The tool then once again invokes the fault simulator, removing all detected faults from the fault list and placing the effective patterns in the test set. The tool then selects another set of patterns and iterates through this process until no faults remain in the current fault list, except those aborted during test generation (that is, those in the UC or UO categories).

## ATPG Tool Timing Model

The tool uses a cycle-based timing model, grouping the test pattern events into test cycles. The ATPG tool simulator uses the non-scan events: **force\_pi**, **measure\_po**, **capture\_clock\_on**, **capture\_clock\_off**, **ram\_clock\_on**, and **ram\_clock\_off**. The tool uses a fixed test cycle type for ATPG; that is, you cannot modify it.

The most commonly used test cycle contains the events: **force\_pi**, **measure\_po**, **capture\_clock\_on**, and **capture\_clock\_off**. The test vectors used to read or write into RAMs contain the events **force\_pi**, **ram\_clock\_on**, and **ram\_clock\_off**. You can associate real times with each event via the timing file.

## ATPG Tool Pattern Types

The ATPG tool has several different types of testing modes. That is, it can generate several different types of patterns depending on the style and circuitry of the design and the information you specify. By default, the tool generates basic scan patterns, which assume a full-scan design methodology. The following subsections describe basic scan patterns, as well as the other types of patterns that the tool can generate.

### Basic Scan Patterns

As mentioned, the tool generates basic scan patterns by default. A scan pattern contains the events that force a single set of values to all scan cells and primary inputs (**force\_pi**), followed by observation of the resulting responses at all primary outputs and scan cells (**measure\_po**). The tool uses any defined scan clock to capture the data into the observable scan cells (**capture\_clock\_on**, **capture\_clock\_off**). Scan patterns reference the appropriate test procedures to define how to control and observe the scan cells. The tool requires that each scan pattern be independent of all other scan patterns. The basic scan pattern contains the following events:

1. Load values into scan chains.
2. Force values on all non-clock primary inputs (with clocks off and constrained pins at their constrained values).
3. Measure all primary outputs (except those connected to scan clocks).
4. Pulse a capture clock or apply selected clock procedure.

## 5. Unload values from scan chains.

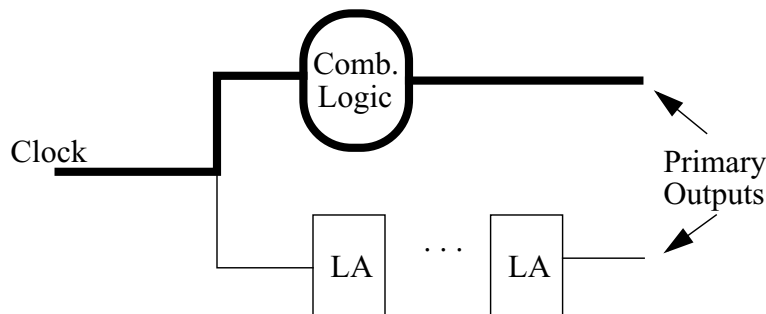
While the list shows the loading and unloading of the scan chain as separate events, more typically the loading of a pattern occurs simultaneously with the unloading of the preceding pattern. Thus, when applying the patterns at the tester, you have a single operation that loads in scan values for a new pattern while unloading the values captured into the scan chains for the previous pattern.

Because the ATPG tool is optimized for use with scan designs, the basic scan pattern contains the events from which the tool derives all other pattern types.

## Clock PO Patterns

Figure 6-4 shows that in some designs, a clock signal may go to a primary output through some combinational logic.

**Figure 6-4. Clock-PO Circuitry**



The tool considers any pattern that measures a PO with connectivity to a clock, regardless of whether or not the clock is active, a clock PO pattern. A normal scan pattern has all clocks off during the force of the primary inputs and the measure of the primary outputs. However, in the clocked primary output situation, if the clock is off, a condition necessary to test a fault within this circuitry might not be met and the fault may go undetected. In this case, in order to detect the fault, the pattern must turn the clock on during the force and measure. This does not happen in the basic scan pattern. The tool allows this within a clock PO pattern, to observe primary outputs connected to clocks.

Clock PO patterns contain the following events:

1. Load values into the scan chains.
2. Force values on all primary inputs, (potentially) including clocks (with constrained pins at their constrained values).
3. Measure all primary outputs that are connected to scan clocks.

The tool generates clock PO patterns whenever it learns that a clock connects to a primary output and if it determines that it can only detect faults associated with the circuitry by using a

clock PO pattern. If you do not want the tool to generate clock PO patterns, you can turn off the capability as follows:

```
SETUP> set_pattern_type off
```

## Clock Sequential Patterns

The ATPG tool's clock sequential pattern type handles limited sequential circuitry, and can also help in testing designs with RAM. This kind of pattern contains the following events:

1. Load the scan chains.
2. Apply the clock sequential cycle.
  - a. Force values on all primary inputs, except clocks (with constrained pins at their constrained values).
  - b. Pulse the write lines, read lines, capture clock, and/or apply selected clock procedure.
  - c. Repeat steps a and b for a total of "N" times, where N is the clock sequential depth - 1.
3. Apply the capture cycle.
  - a. Force pi.
  - b. Measure po.
  - c. Pulse capture clock.
4. Unload the scan chains as you load the next pattern.

To instruct the tool to generate clock sequential patterns, you must set the sequential depth to some number greater than one, using the [set\\_pattern\\_type](#) command as follows:

```
SETUP> set_pattern_type -sequential 2
```

A depth of zero indicates combinational circuitry. A depth greater than one indicates limited sequential circuitry. You should, however, be careful of the depth you specify. You should start off using the lowest sequential depth and analyzing the run results. You can perform several runs, if necessary, increasing the sequential depth each time. Although the maximum allowable depth limit is 255, you should typically limit the value you specify to five or less, for performance reasons.

## Multiple Load Patterns

The tool can optionally include multiple scan chain loads in a clock sequential pattern. By creating patterns that use multiple loads, the tool can:



- take advantage of a design's non-scan sequential cells that are capable of retaining their state through a scan load operation
- test through a RAM/ROM

You enable the multiple load capability by using “-multiple\_load on” with the [set\\_pattern\\_type](#) command and setting the sequential depth to some number greater than one. When you activate this capability, you allow the tool to include a scan load before any pattern cycle.

---

**Note**

An exception is at-speed sequences in named capture procedures. A load may not occur between the at-speed launch and capture cycles. For more information, see the description of the “load” cycle type in [“Defining Internal and External Modes”](#) on page 232.

---

Generally, multiple load patterns require a sequential depth for every functional mode clock pulsed. A minimum sequential depth of 4 is required to enable the tool to create the multiple cycle patterns necessary for RAM testing. The patterns are very similar to RAM sequential patterns, but for many designs will give better coverage than RAM sequential patterns. This method also supports certain tool features (MacroTest, dynamic compression, split-capture cycle, clock-off simulation) not supported by RAM sequential patterns.

## RAM Sequential Patterns

To propagate fault effects through RAM, and to thoroughly test the circuitry associated with a RAM, the tool generates a special type of pattern called RAM sequential. RAM sequential patterns are single patterns with multiple loads, which model some sequential events necessary to test RAM operations. The multiple load events include two address writes and possibly a read (if the RAM has data hold). This type of pattern contains the following events:

1. Load scan cells.
2. Force primary inputs.
3. Pulse write line(s).
4. Repeat steps 1 through 3 for a different address.
5. Load scan cells.
6. Force primary inputs.
7. Pulse read lines (optional, depending on the RAM's data hold attribute).
8. Load scan cells.
9. Force primary inputs
10. Measure primary outputs.

11. Pulse capture clock.
12. Unload values from scan cells.

The following example explains the operations depicted in this type of pattern. Assume you want to test a stuck-at-1 fault on the highest order bit of the address lines. You could do this by writing some data, D, to location 1000. You could then write different data, D', to location 0000. If a stuck-at-1 fault were present on the highest address bit, the faulty machine would overwrite location 1000 with the value D'. Next, you would attempt to read from address location 1000. With the stuck-at-1 fault on the address line, you would read D'.

Conversely, if the fault on the highest order bit of the address line is a stuck-at-0 fault, you would want to write the initial data, D, to location 0000. You would then write different data, D', to location 1000. If a stuck-at-0 fault were present on the highest address bit, the faulty machine would overwrite location 0000 with the value D'. Next, you would attempt to read from address location 0000. With the stuck-at-0 fault on the address line, you would read D'.

You can instruct the tool to generate RAM sequential patterns by issuing the [set\\_pattern\\_type](#) command as follows:

```
SETUP> set_pattern_type -ram_sequential on
```

## Sequential Transparent Patterns

Designs containing some non-scan latches can use basic scan patterns if the latches behave transparently between the time of the primary input force and the primary output measure. A latch behaves transparently if it passes rule D6.

For latches that do not behave transparently, a user-defined procedure can force some of them to behave transparently between the primary input force and primary output measure. A test procedure, which is called **seq\_transparent**, defines the appropriate conditions necessary to force transparent behavior of some latches. The events in sequential transparent patterns include:

1. Load scan chains.
2. Force primary inputs.
3. Apply **seq\_transparent** procedure(s).
4. Measure primary outputs.
5. Unload scan chains.

For more information on sequential transparent procedures, refer to “[Scan and Clock Procedures](#)” in the *Tessent Shell User's Manual*.

## Performing Basic Operations

This section describes the most basic operations you may need to perform with the ATPG tool.

### Invoking the Application

You access ATPG functionality by invoking Tessent Shell and then setting the context to “patterns -scan”:

```
% tessent -shell  
SETUP> set_context patterns -scan
```

### Setting the System Mode

When Tessent Shell invokes, the tool assumes the first thing you want to do is set up circuit behavior, so it automatically puts you in setup mode. To change the system mode to analysis, use the [set\\_system\\_mode](#) command.

## Setting Up Design and Tool Behavior

The first real task you must perform in the basic ATPG flow is to set up information about design behavior and existing scan circuitry. The following subsections describe how to accomplish this setup.

### Setting Up the Circuit Behavior

The ATPG tool provides a number of commands that let you set up circuit behavior. You must execute these commands while in setup mode. A convenient way to execute the circuit setup commands is to place these commands in a dofile, as explained previously in “[Running Tessent Shell as a Batch Job](#)”. The following subsections describe typical circuit behavior set up tasks.

### Defining Equivalent or Inverted Primary Inputs

Within the circuit application environment, often multiple primary inputs of the circuit being tested must always have the same (equivalent) or opposite values. Specifying pin equivalences constrains selected primary input pins to equivalent or inverted values relative to the last entered primary input pin. To add pin equivalences, use the [add\\_input\\_constraints](#) command.

- [delete\\_input\\_constraints](#) — Deletes the specified pin equivalences.
- [report\\_input\\_constraints](#) — Displays the specified pin equivalences.

## Adding Primary Inputs and Outputs

In some cases, you may need to change the test pattern application points (primary inputs) or the output value measurement points (primary outputs). When you add previously undefined primary inputs, they are called user class primary inputs, while the original primary inputs are called system class primary inputs.

To add `_primary_inputs` to a circuit, at setup mode prompt, use the [add\\_primary\\_inputs](#) command. When you add previously undefined primary outputs, they are called user class primary outputs, while the original primary outputs are called system class primary outputs.

To add `_primary_outputs` to a circuit, at the setup mode prompt, use the [add\\_primary\\_outputs](#) command.

- [delete\\_primary\\_inputs](#) — Deletes the specified types of primary inputs.
- [report\\_primary\\_inputs](#) — Reports the specified types of primary inputs.
- [delete\\_primary\\_outputs](#) — Deletes the specified types of primary outputs.
- [report\\_primary\\_outputs](#) — Reports the specified types of primary outputs.

## Using Bidirectional Pins as Primary Inputs or Outputs

During pattern generation, the ATPG tool automatically determines the mode of bidirectional pins (bidi) and avoids creating patterns that drive values on these pins when they are not in input mode. In some situations, however, you might prefer to have the tool treat a bidirectional pin as a PI or PO. For example, some testers require more memory to store bidirectional pin data than PI or PO data. Treating each bidi as a PI or PO when generating and saving patterns will reduce the amount of memory required to store the pin data on these testers.

From the tool's perspective, a bidi consists of several gates and includes an input port and an output port. You can use the commands, [report\\_primary\\_inputs](#) and [report\\_primary\\_outputs](#), to view PIs and POs. Pins that are listed by both commands are bidirectional pins.

Certain other PI-specific and PO-specific commands accept a bidi pinname argument, and enable you to act on just the applicable port functionality (input or output) of the bidi. For example, you can use the [delete\\_primary\\_inputs](#) command with a bidirectional pin argument to remove the input port of the bidi from the design interface. From then on, the tool will treat that pin as a PO. You can use the [delete\\_primary\\_outputs](#) command similarly to delete the output port of a bidi from the design interface, so the tool treats that bidi as a PI.

### Note



Altering the design's interface will result in generated patterns that are different than those the tool would generate for the original interface. It also prevents verification of the saved patterns using the original netlist interface. If you want to be able to verify saved patterns by performing simulation using the original netlist interface, you must use the commands described in the following subsections instead of the `delete_primary_inputs/outputs` commands.

## Setting Up a Bidirectional Pin as a Primary Output for ATPG Only

With the `add_input_constraints` command, you can get the tool to treat a bidi as a PO during ATPG only, without altering the design interface within the tool. You do this by constraining the input part of the bidi to a constant high impedance (CZ) state. The generated patterns will then contain PO data for the bidi, and you will be able to verify saved patterns by performing simulation using the original design netlist.

## Setting Up a Bidirectional Pin as a Primary Input for ATPG Only

With the `add_output_masks` command, you can get the tool to treat a bidi as a PI during ATPG only, without altering the design interface. This command blocks observability of the output part of the bidi. The generated patterns will then contain PI data for the bidi, and you will be able to verify saved patterns by performing simulation using the original design netlist.

## If the Bidirectional Pin Control Logic is Unknown

Sometimes the control logic for a bidi is unknown. In this situation, you can model the control logic as a black box. If you want the tool to treat the bidi as a PI, model the output of the black box to be 0. If you want the bidi treated as a PO, model the output of the black box to be 1.

## If the Bidirectional Pin has a Pull-up or Pull-down Resistor

Using default settings, The ATPG tool generates a known value for a bidirectional pad having pull-up or pull-down resistors. In reality, however, the pull-up or pull-down time is typically very slow and will result in simulation mismatches when a test is carried out at high speed. To prevent such mismatches, Mentor Graphics recommends you use the `add_input_constraints` command. This command changes the tool's simulation of the I/O pad so that instead of a known value, an X is captured for all observation points that depend on the pad. The X masks the observation point, preventing simulation mismatches.

## Examples of Setting Up Bidirectional Pins as PIs or POs

The following examples demonstrate the use of the commands described in the preceding sections about bidirectional pins (bidis). Assume the following pins exist in an example design:

- Bidirectional pins: `/my_inout[0].../my_inout[2]`

- Primary inputs (PIs): /clk, /rst, /scan\_in, /scan\_en, /my\_en
- Primary outputs (POs): /my\_out[0].../my\_out[4]

You can view the bidis by issuing the following two commands:

```
SETUP> report_primary_inputs

SYSTEM: /clk
SYSTEM: /rst
SYSTEM: /scan_in
SYSTEM: /scan_en
SYSTEM: /my_en
SYSTEM: /my_inout[2]
SYSTEM: /my_inout[1]
SYSTEM: /my_inout[0]
SETUP> report_primary_outputs

SYSTEM: /x_out[4]
SYSTEM: /x_out[3]
SYSTEM: /x_out[2]
SYSTEM: /x_out[1]
SYSTEM: /x_out[0]
SYSTEM: /my_inout[2]
SYSTEM: /my_inout[1]
SYSTEM: /my_inout[0]
```

Pins listed in the output of both commands (shown in bold font) are pins the tool will treat as bidis during test generation. To force the tool to treat a bidi as a PI or PO, you can remove the definition of the unwanted input or output port. The following example removes the input port definition, then reports the PIs and POs. You can see the tool now only reports the bidis as POs, which reflects how those pins will be treated during ATPG:

```
SETUP> delete_primary_inputs /my_inout[0] /my_inout[1] /my_inout[2]
SETUP> report_primary_inputs

SYSTEM: /clk
SYSTEM: /rst
SYSTEM: /scan_in
SYSTEM: /scan_en
SYSTEM: /my_en

SETUP> report_primary_outputs

SYSTEM: /x_out[4]
SYSTEM: /x_out[3]
SYSTEM: /x_out[2]
SYSTEM: /x_out[1]
SYSTEM: /x_out[0]
SYSTEM: /my_inout[2]
SYSTEM: /my_inout[1]
SYSTEM: /my_inout[0]
```

Because the preceding approach alters the design's interface within the tool, it may not be acceptable in all cases. Another approach, explained earlier, is to have the tool treat a bidi as a PI or PO during ATPG only, without altering the design interface. To obtain PO treatment for a

bidi, constrain the input part of the bidi to the high impedance state. The following command does this for the /my\_inout[0] bidi:

```
SETUP> add_input_constraints /my_inout[0] -cz
```

To have the tool treat a bidi as a PI during ATPG only, direct the tool to mask (ignore) the output part of the bidi. The following example does this for the /my\_inout[0] and /my\_inout[1] pins:

```
SETUP> add_output_masks /my_inout[0] /my_inout[2]
SETUP> report_output_masks
```

```
TIEX /my_inout[0]
TIEX /my_inout[2]
```

The “TIEX” in the output of “report\_output\_masks” indicates the two pins are now tied to X, which blocks their observability and prevents the tool from using them during ATPG.

## Tying Undriven Signals

Within your design, there could be several *undriven* nets, which are input signals not tied to fixed values. When you read a netlist, the application issues a warning message for each undriven net or floating pin in the module. The ATPG tool must “virtually” tie these pins to a fixed logic value during ATPG. If you do not specify a value, the application uses the default value X, which you can change with the [set\\_tied\\_signals](#) command.

To add\_tied\_signals, at the setup mode prompt, use the [add\\_tied\\_signals](#) command.

This command assigns a fixed value to every named floating net or pin in every module of the circuit under test.

- [set\\_tied\\_signals](#) — Sets default for tying unspecified undriven signals.
- [delete\\_tied\\_signals](#) — Deletes the current list of specified tied signals.
- [report\\_tied\\_signals](#) — Displays current list of specified tied nets and pins.

## Constraining Primary Inputs

The tool can constrain primary inputs during the ATPG process. To add a pin constraint to a specific pin, use the [add\\_input\\_constraints](#) command.

You can specify one or more primary input pin pathnames to be constrained to one of the following formats: constant 0 (C0), constant 1 (C1), high impedance (CZ), or unknown (CX).

For detailed information on the tool-specific usages of this command, refer to [add\\_input\\_constraints](#) in the *Tessent Shell Reference Manual*.

## Masking Primary Outputs

Your design may contain certain primary output pins that have no strobe capability. Or in a similar situation, you may want to mask certain outputs from observation for design trade-off experimentation. In these cases, you could mask these primary outputs using the [add\\_output\\_masks](#) command.

---

### Note



The tool places faults they can only detect through masked outputs in the AU category—not the UO category.

---

## Adding Slow Pads

While running tests at high speed, as might be used for path delay test patterns, it is not always safe to assume that the loopback path from internal registers, via the I/O pad back to internal registers, can stabilize within a single clock cycle. Assuming that the loopback path stabilizes within a single clock cycle may cause problems verifying ATPG patterns or may lead to yield loss during testing.

To prevent a problem caused by this loopback, use the [add\\_input\\_constraints](#) command to modify the simulated behavior of the bidirectional I/O pin, on a pin by pin basis.

For a slow pad, the simulation of the I/O pad changes so that the value propagated into the internal logic is X whenever the primary input is not driven. This causes an X to be captured for all observation points dependent on the loopback value.

- [delete\\_input\\_constraints](#) — Resets the specified I/O pin back to the default simulation mode.
- [report\\_input\\_constraints](#) — Displays all I/O pins marked as slow.

## Setting Up Tool Behavior

In addition to specifying information about the design to the ATPG tool, you can also set up how you want the ATPG tool to handle certain situations and how much effort to put into various processes. The following subsections discuss the typical tool setup.

- [set\\_learn\\_report](#) — Enables access to certain data learned during analysis.
- [set\\_loop\\_handling](#) — Specifies the method in which to break loops.
- [set\\_pattern\\_buffer](#) — Enables the use of temporary buffer files for pattern data.
- [set\\_possible\\_credit](#) — Sets credit for possibly-detected faults.
- [set\\_pulse\\_generators](#) — Specifies whether to identify pulse generator sink gates during learning analysis. The pulse generator learning is restricted to the 2-input AND/OR type



of reconvergence structure. For more comprehensive pulse generator support, use the `_pulse_generator` primitive—see “[Pulse Generators with User Defined Timing](#)” in the *Tessent Cell Library Manual* for complete information.

## Checking Bus Contention

If you use contention checking on tri-state driver busses and multiple-port flip-flops and latches, the tool rejects (from the internal test pattern set) patterns generated by the ATPG process that can cause bus contention. To set contention checking, you use the [set\\_contention\\_check](#) command.

By default, contention checking is on, as are the switches `-Warning` and `-Bus`, causing the tool to check tri-state driver buses and issue a warning if bus contention occurs during simulation. For more information on the different contention checking options, refer to the [set\\_contention\\_check](#) description in the *Tessent Shell Reference Manual*.

To display the current status of contention checking, use the [report\\_environment](#) command.

- [analyze\\_bus](#) — Analyzes the selected buses for mutual exclusion.
- [set\\_bus\\_handling](#) — Specifies how to handle contention on buses.
- [set\\_driver\\_restriction](#) — Specifies whether only a single driver or multiple drivers can be on for buses or ports.
- [report\\_bus\\_data](#) — Reports data for either a single bus or a category of buses.
- [report\\_gates](#) — Reports netlist information for the specified gates.

## Setting Multi-Driven Net Behavior

When you specify the fault effect of bus contention on tri-state nets with the `set_net_dominance` command, you are giving the tool the ability to detect some faults on the enable lines of tri-state drivers that connect to a tri-state bus. At the setup mode prompt, you use the [set\\_net\\_dominance](#) command.

The three choices for bus contention fault effect are And, Or, and Wire (unknown behavior), Wire being the default. The Wire option means that any different binary value results in an X state. The truth tables for each type of bus contention fault effect are shown on the references pages for the [set\\_net\\_dominance](#) description in the *Tessent Shell Reference Manual*.

On the other hand, if you have a net with multiple non-tri-state drivers, you may want to specify this type of net’s output value when its drivers have different values. Using the [set\\_net\\_resolution](#) command, you can set the net’s behavior to And, Or, or Wire (unknown behavior). The default Wire option requires all inputs to be at the same state to create a known output value. Some loss of test coverage can result unless the behavior is set to And (wired-and) or Or (wired-or). To set the multi-driver net behavior, at the setup mode prompt, you use the `set_net_resolution` command.

## Setting Z-State Handling

If your tester has the ability to distinguish the high impedance (Z) state, you should use the Z state for fault detection to improve your test coverage. If the tester can distinguish a high impedance value from a binary value, certain faults may become detectable which otherwise would at best be possibly detected (pos\_det). This capability is particularly important for fault detection in the enable line circuitry of tri-state drivers.

The default for the ATPG tool is to treat a Z state as an X state. If you want to account for Z state values during simulation, you can issue the [set\\_z\\_handling](#) command.

Internal Z handling specifies how to treat the high impedance state when the tri-state network feeds internal logic gates. External handling specifies how to treat the high impedance state at the circuit primary outputs. The ability of the tester normally determines this behavior.

To set the internal or external Z handling, use the `set_z_handling` command at the setup mode prompt.

For internal tri-state driver nets, you can specify the treatment of high impedance as a 0 state, a 1 state, and an unknown state.

---

### Note



This command is not necessary if the circuit model already reflects the existence of a pull gate on the tri-state net.

---

For example, to specify that the tester does not measure high impedance, enter the following:

```
SETUP> set_z_handling external X
```

For external tri-state nets, you can also specify that the tool measures high impedance as a 0 state and distinguished from a 1 state (0), measures high impedance as a 1 state and distinguished from a 0 state (1), measures high impedance as unique and distinguishable from both a 1 and 0 state (Z).

## Controlling the Learning Process

The ATPG tool performs extensive learning on the circuit during the transition from setup to some other system mode. This learning reduces the amount of effort necessary during ATPG. The tool allows you to control this learning process.

For example, the tool lets you turn the learning process off or change the amount of effort put into the analysis. You can accomplish this for combinational logic using the [set\\_static\\_learning](#) command.

By default, static learning is on and the simulation activity limit is 1000. This number ensures a good trade-off between analysis effort and process time. If you want the ATPG tool to perform maximum circuit learning, you should set the activity limit to the number of gates in the design.

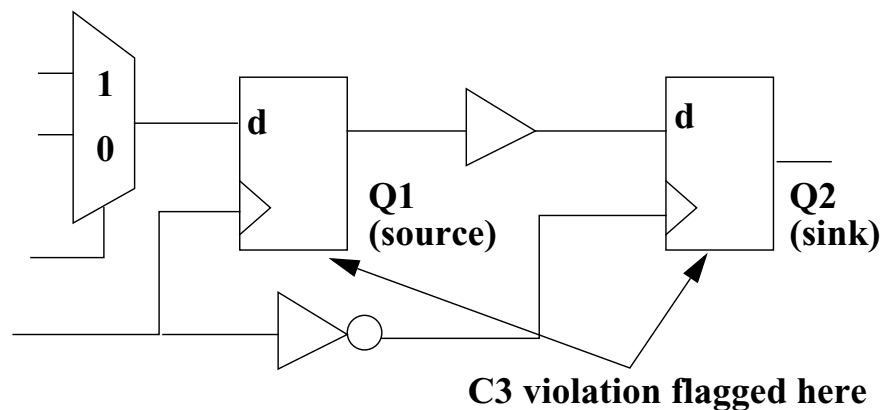
By default, state transition graph extraction is on. For more information on the learning process, refer to “[Learning Analysis](#)” on page 81.

## Setting the Capture Handling

The ATPG tool evaluates gates only once during simulation, simulating all combinational gates before sequential gates. This default simulation behavior correlates well with the normal behavior of a synchronous design, if the design model passes design rules checks—particularly rules C3 and C4. However, if your design fails these checks, you should examine the situation to see if your design would benefit from a different type of data capture simulation.

For example, examine the design of [Figure 6-5](#). It shows a design fragment which fails the C3 rules check.

**Figure 6-5. Data Capture Handling Example**



The rules checker flags the C3 rule because Q2 captures data on the trailing edge of the same clock that Q1 uses. The ATPG tool considers sequential gate Q1 as the data *source* and Q2 as the data *sink*. By default, the tool simulates Q2 capturing old data from Q1. However, this behavior most likely does not correspond to the way the circuit really operates. In this case, the C3 violation should alert you that simulation could differ from real circuit operation.

To allow greater flexibility of capture handling for these types of situations, the tool provides some commands that alter the default simulation behavior. The [set\\_split\\_capture\\_cycle](#) command, for example, effects whether or not the tool updates simulation data between clock edges. When set to “on”, the tool is able to determine correct capture values for trailing edge and level-sensitive state elements despite C3 and C4 violations. If you get these violations, issue the [set\\_split\\_capture\\_cycle ON](#) command.

You can select modified capture handling for level sensitive or trailing edge gates. For these types of gates, you select whether you want simulation to use old data, new data, or X values.

The [set\\_capture\\_handling](#) command changes the data capture handling globally for all the specified types of gates that fail C3 and C4. If you want to selectively change capture handling, you can use the [add\\_capture\\_handling](#) command.

You can specify the type of data to capture, whether the specified gate(s) is a source or sink point, and the gates or objects (identified by ID number, pin names, instance names, or cell model names) for which to apply the special capture handling.

---

**Note**

When you change capture handling to simulate new data, the tool only performs new data simulation for one additional level of circuitry. That is, sink gates capture new values from their sources. However, if the sources are also sinks that are set to capture new data, the tool does not simulate this effect.

---

For more information about [set\\_capture\\_handling](#) or [add\\_capture\\_handling](#), refer to the *Tessent Shell Reference Manual*. For more information about C3 and C4 rules violations, refer to “[Clock Rules](#)” in the *Tessent Shell Reference Manual*.

- [delete\\_capture\\_handling](#) — Removes special data capture handling for the specified objects.
- [set\\_drc\\_handling](#) — Specifies violation handling for a design rules check.

## Setting Transient Detection

You can set how the tool handles zero-width events on the clock lines of state elements. The tool lets you turn transient detection on or off with the [set\\_transient\\_detection](#) command.

With transient detection off, DRC simulation treats all events on state elements as valid. Because the simulator is a zero delay simulator, it is possible for DRC to simulate zero-width monostable circuits with ideal behavior, which is rarely matched in silicon. The tool treats the resulting zero-width output pulse from the monostable circuit as a valid clocking event for other state elements. Thus, state elements change state although their clock lines show no clocking event.

With transient detection on, the tool sets state elements to a value of X if the zero-width event causes a change of state in the state elements. This is the default behavior upon invocation of the tool.

## Defining the Scan Data

You must define the scan clocks and scan chains before the application performs rules checking (which occurs upon exiting setup mode). The following subsections describe how to define the various types of scan data.

### Defining Scan Clocks

The tool considers any signals that capture data into sequential elements (such as system clocks, sets, and resets) to be scan clocks. Therefore, to take advantage of the scan circuitry, you need to define these “clock signals” by adding them to the clock list.

You must specify the *off-state* for pins you add to the clock list. The off-state is the state in which clock inputs of latches are inactive. For edge-triggered devices, the off-state is the clock value prior to the clock’s capturing transition. You add clock pins to the list by using the [add\\_clocks](#) command.

You can constrain a clock pin to its off-state to suppress its usage as a capture clock during the ATPG process. The constrained value must be the same as the clock off-state, otherwise an error occurs. If you add an equivalence pin to the clock list, all of its defined equivalent pins are also automatically added to the clock list.

- [delete\\_clocks](#) — Deletes the specified pins from the clock list.
- [report\\_clocks](#) — Reports all defined clock pins.

### Defining Scan Groups

A scan group contains a set of scan chains controlled by a single test procedure file. You must create this test procedure file prior to defining the scan chain group that references it. To define scan groups, you use the [add\\_scan\\_groups](#) command.

- [delete\\_scan\\_groups](#) — Deletes specified scan groups and associated chains.
- [report\\_scan\\_groups](#) — Displays current list of scan chain groups.

### Defining Scan Chains

After defining scan groups, you can define the scan chains associated with the groups. For each scan chain, you must specify the name assigned to the chain, the name of the chain’s group, the scan chain input pin, and the scan chain output pin. To define scan chains and their associated scan groups, you use the [add\\_scan\\_chains](#) command.

#### Note



Scan chains of a scan group can share a common scan input pin, but this condition requires that both scan chains contain the same data after loading.

---

- [delete\\_scan\\_chains](#) — Deletes the specified scan chains.
- [report\\_scan\\_chains](#) — Displays current list of scan chains.

## Setting the Clock Restriction

You can specify whether or not to allow the test generator to create patterns that have more than one non-equivalent capture clock active at the same time. To set the clock restriction, you use the [set\\_clock\\_restriction](#) command.

#### Note



If you choose to turn off the clock restriction, you should verify the generated pattern set using a timing simulator—to ensure there are no timing errors.

---

## Adding Constraints to Scan Cells

The tool can constrain scan cells to a constant value (C0 or C1) during the ATPG process to enhance controllability or observability. Additionally, the tools can constrain scan cells to be either uncontrollable (CX), unobservable (OX), or both (XX).

You identify a scan cell by either a pin pathname or a scan chain name plus the cell's position in the scan chain.

To add constraints to scan cells, you use the [add\\_cell\\_constraints](#) command.

If you specify the pin pathname, it must be the name of an output pin directly connected (through only buffers and inverters) to a scan memory element. In this case, the tool sets the scan memory element to a value such that the pin is at the constrained value. An error condition occurs if the pin pathname does not resolve to a scan memory element.

If you identify the scan cell by chain and position, the scan chain must be a currently-defined scan chain and the position is a valid scan cell position number. The scan cell closest to the scan-out pin is in position 0. The tool constrains the scan cell's MASTER memory element to the selected value. If there are inverters between the MASTER element and the scan cell output, they may invert the output's value.

- [delete\\_cell\\_constraints](#) — Deletes the constraints from the specified scan cells.
- [report\\_cell\\_constraints](#) — Reports all defined scan cell constraints.

## Adding Nofault Settings

Within your design, you may have instances that should not have internal faults included in the fault list. You can label these parts with a *nofault* setting. To add a nofault setting, you use the [add\\_nofaults](#) command.

You can specify that the listed pin pathnames, or all the pins on the boundary and inside the named instances, are not allowed to have faults included in the fault list.

- [delete\\_nofaults](#) — Deletes the specified nofault settings.
- [report\\_nofaults](#) — Displays all specified nofault settings.

## Checking Rules and Debugging Rules Violations

If an error occurs during the rules checking process, the application remains in setup mode so you can correct the error. You can easily resolve the cause of many such errors; for instance, those that occur during parsing of the test procedure file. Other errors may be more complex and difficult to resolve, such as those associated with proper clock definitions or with shifting data through the scan chain.

The ATPG tool performs model flattening, learning analysis, and rules checking when you try to exit setup mode. Each of these processes is explained in detail in “[Understanding Common Tool Terminology and Concepts](#)” on page 67. To change from setup to one of the other system modes, you enter the [set\\_system\\_mode](#) command.

---

### Note



The ATPG tool does not require the Drc mode because it uses the same internal design model for all of its processes.

---

The “[Troubleshooting Rules Violations](#)” section in the *Tessent Shell Reference Manual* discusses some procedures for debugging rules violations. The Debug window of DFTVisualizer is especially useful for analyzing and debugging certain rules violations. The “[Attributes and DFTVisualizer](#)” section in the *Tessent User’s Manual* discusses DFTVisualizer in detail.

## Running Good/Fault Simulation on Existing Patterns

The purpose of fault simulation is to determine the fault coverage of the current pattern source for the faults in the active fault list. The purpose of “good” simulation is to verify the simulation model. Typically, you use the good and fault simulation capabilities of the ATPG tool to grade existing hand- or ATPG-generated pattern sets.

## Fault Simulation

The following subsections discuss the procedures for setting up and running fault simulation using the ATPG tool.

### Preparing for Fault Simulation

Fault simulation runs in analysis mode without additional setup. You enter analysis mode using the following command:

```
SETUP> set_system_mode analysis
```

### Setting the Fault Type

By default, the fault type is stuck-at. If you want to simulate patterns to detect stuck-at faults, you do not need to issue this command.

If you wish to change the fault type to toggle, pseudo stuck-at (IDDQ), transition, path delay, or bridge, you can issue the [set\\_fault\\_type](#) command.

Whenever you change the fault type, the application deletes the current fault list and current internal pattern set.

### Creating the Faults List

Before you can run fault simulation, you need an active fault list from which to run. You create the faults list using the [add\\_faults](#) command.

Typically, you would create this list using all faults as follows:

```
> add_faults -all
```

[“Setting Up the Fault Information for ATPG”](#) on page 179 provides more information on creating the fault list and specifying other fault information.

### Setting the Pattern Source

You can have the tools perform simulation and test generation on a selected pattern source, which you can change at any time. To use an external pattern source, you use the [read\\_patterns](#) command.

---

#### Note



You may notice a slight drop in test coverage when using an external pattern set as compared to using generated patterns. This is an artificial drop.

---



The ATPG tool can perform simulation with a select number of random patterns. Refer to the *Tessent Shell Reference Manual* for additional information about these application-specific [simulate\\_patterns](#) command options.

- [set\\_capture\\_clock](#) — Specifies the capture clock for random pattern simulation.
- [set\\_random\\_clocks](#) — Specifies the selection of clock\_sequential patterns for random pattern simulation.
- [set\\_random\\_patterns](#) — Specifies the number of random patterns to be simulated.

## Executing Fault Simulation

You execute the fault simulation process by using the [simulate\\_patterns](#) command. You can repeat this command as many times as you want for different pattern sources.

- [report\\_faults](#) — Displays faults for selected fault classes.
- [report\\_statistics](#) — Displays a statistics report.

## Writing the Undetected Faults List

Typically, after performing fault simulation on an external pattern set, you will want to save the faults list. You can then use this list as a starting point for ATPG. To save the faults, you use the [write\\_faults](#) command.

Refer to “[Writing Faults to an External File](#)” on page 181 or the [write\\_faults](#) description in the *Tessent Shell Reference Manual* for command option details.

To read the faults back in for ATPG, go to analysis mode (using [set\\_system\\_mode](#)) and enter the [read\\_faults](#) command.

## Debugging the Fault Simulation

To debug your fault simulation, you can write a list of pin values that differ between the faulty and good machine. Do this using the [add\\_lists](#) and [set\\_list\\_file](#) commands.

The [add\\_lists](#) command specifies which pins you want reported. The [set\\_list\\_file](#) command specifies the name of the file in which to place simulation values for the selected pins. The default behavior is to write pin values to standard output.

## Resetting Circuit and Fault Status

You can reset the circuit status and status of all testable faults in the fault list to undetected. Doing so lets you re-run the fault simulation using the current fault list, which does not cause deletion of the current internal pattern set. To reset the testable faults in the current fault list, enter the [reset\\_state](#) command.

## Good-Machine Simulation

Given a test vector, you use *good machine simulation* to predict the logic values in the good (fault-free) circuit at all the circuit outputs. The following subsections discuss the procedures for running good simulation on existing hand- or ATPG-generated pattern sets using the ATPG tool.

### Preparing for Good-Machine Simulation

Good-machine simulation runs in analysis mode without additional setup. You enter analysis mode using the following command:

```
SETUP> set_system_mode analysis
```

### Specifying an External Pattern Source

By default, simulation runs using an internal ATPG-generated pattern source. To run simulation using an external set of patterns, enter the following command:

```
ANALYSIS> read_patterns filename
```

### Debugging the Good Machine Simulation

You can debug your good machine simulation in several ways. If you want to run the simulation and save the values of certain pins in batch mode, you can use the [add\\_lists](#) and [set\\_list\\_file](#) commands.

The `add_lists` command specifies which pins to report. The `set_list_file` command specifies the name of the file in which you want to place simulation values for the selected pins.

If you prefer to perform interactive debugging, you can use the [simulate\\_patterns](#) and [report\\_gates](#) commands to examine internal pin values.

### Resetting Circuit Status

In analysis mode, you can reset the circuit status by using the [reset\\_state](#) command.

## Running Random Pattern Simulation

The following subsections show the typical procedure for running random pattern simulation.

## Changing to the Fault System Mode

You run random pattern simulation in the analysis system mode. If you are not already in the analysis system mode, use the [set\\_system\\_mode](#) command as in the following example:

```
SETUP> set_system_mode analysis
```

## Adding the Faults List

To generate the faults list and eliminate all untestable faults, use the [add\\_faults](#) and [delete\\_faults](#) commands together as in the following example:

```
> add_faults -all  
> delete_faults -untestable
```

In this example, the `delete_faults` command with the `-untestable` switch removes faults from the fault list that are untestable using random patterns.

## Running the Simulation

To run the random pattern simulation, enter the “[simulate\\_patterns](#) -source random” command.

After the simulation run, you can display the undetected faults with the [report\\_faults](#) command. Some of the undetected faults may be redundant. You can run ATPG on the undetected faults to identify those that are redundant.

## Setting Up the Fault Information for ATPG

Prior to performing test generation, you must set up a list of all faults the application has to evaluate. The tool can either read the list in from an external source, or generate the list itself. The type of faults in the fault list vary depending on the fault model and your targeted test type. For more information on fault modeling and the supported models, refer to “[Fault Modeling](#)” on page 35.

After the application identifies all the faults, it implements a process of structural equivalence fault collapsing from the original uncollapsed fault list. From this point on, the application works on the collapsed fault list. The results, however, are reported for both the uncollapsed and collapsed fault lists. Executing any command that changes the fault list causes the tool to discard all patterns in the current internal test pattern set due to the probable introduction of inconsistencies. Also, whenever you re-enter setup mode, it deletes all faults from the current fault list. The following subsections describe how to create a fault list and define fault related information.

## Changing to the Analysis System Mode

You can enter the fault list commands from analysis system mode. You switch from setup to the analysis mode using the [set\\_system\\_mode](#) command.

For example, assuming your circuit passes rules checking with no violations, you can exit the setup system mode and enter the analysis system mode as follows:

```
SETUP> set_system_mode analysis
```

## Setting the Fault Type

By default, the fault type is stuck-at. If you want to generate patterns to detect stuck-at faults, you *do not* need to issue the [set\\_fault\\_type](#) command.

If you wish to change the fault type to toggle, pseudo stuck-at (IDDQ), transition, or path delay, you can issue the [set\\_fault\\_type](#) command.

Whenever you change the fault type, the application deletes the current fault list and current internal pattern set.

## Creating the Faults List

The application creates the internal fault list the first time you [add\\_faults](#) or load in external faults. Typically, you would create a fault list with all possible faults of the selected type, although you can place some restrictions on the types of faults in the list. To create a list with all faults of the given type, enter the [add\\_faults](#) command using the -All switch as in the following example:

```
ANALYSIS> add_faults -all
```

If you do not want all possible faults in the list, you can use other options of the [add\\_faults](#) command to restrict the added faults. You can also specify no-faulted instances to limit placing faults in the list. You flag instances as “Nofault” while in setup mode. For more information, refer to “[Adding Nofault Settings](#)” on page 175.

When the tool first generates the fault list, it classifies all faults as uncontrolled (UC).

- [delete\\_faults](#) — Deletes the specified faults from the current fault list.
- [report\\_faults](#) — Displays the specified types of faults.

## Adding Faults to an Existing List

To add new faults to the current fault list, enter the [add\\_faults](#) command.

You must enter either a list of object names (pin pathnames or instance names) or use the -All switch to indicate the pins whose faults you want added to the fault list. You can use the -Stuck-at switch to indicate which stuck faults on the selected pins you want added to the list. If you do not use the Stuck-at switch, the tool adds both stuck-at-0 and stuck-at-1 faults. The tool initially places faults added to a fault list in the undetected-uncontrolled (UC) fault class.

## Loading Faults from an External List

You can place faults from a previous run (from an external file) into the internal fault list. To read faults from an external file into the current fault list, enter the [read\\_faults](#) command.

The applications support external fault files in the 3, 4, or 6 column formats. The only data they use from the external file is the first column (stuck-at value) and the last column (pin pathname), unless you use the -Restore option.

The -Retain option causes the application to retain the fault class (second column of information) from the external fault list. The -Delete option deletes all faults in the specified file from the internal faults list. The -DELETE\_Equivalent option, in the ATPG tool, deletes from the internal fault list all faults listed in the file, as well as all their equivalent faults.

---

### Note



In the ATPG tool, the *filename* specified cannot have fault information lines with comments appended to the end of the lines or fault information lines greater than five columns. The tool will not recognize the line properly and will not add the fault on that line to the fault list.

---

## Writing Faults to an External File

You can write all or only selected faults from a current fault list into an external file. You can then edit or load this file to create a new fault list. To write\_faults to a file, enter the [write\\_faults](#) command. You must specify the name of the file you want to write.

## Setting the Fault Sampling Percentage

By reducing the fault sampling percentage (which by default is 100%), you can decrease the process time to evaluate a large circuit by telling the application to process only a fraction of the total collapsed faults. To set the fault sampling percentage, use the [set\\_fault\\_sampling](#) command.

You must specify a percentage (between 1 and 100) of the total faults you want processed.

## Setting the Fault Mode

You can specify use of either the collapsed or uncollapsed fault list for fault counts, test coverages, and fault reports. The default is to use uncollapsed faults.

To set the fault mode, you use the [set\\_fault\\_mode](#) command.

## Setting the Possible-Detect Credit

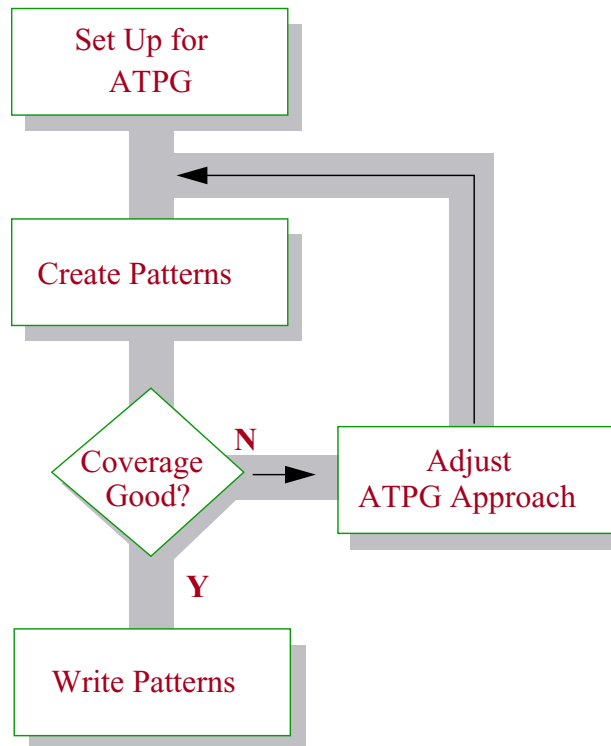
Before reporting test coverage, fault coverage, and ATPG effectiveness, you should specify the credit you want given to possible-detected faults. To set the credit to be given to possible-detected faults, use the [set\\_possible\\_credit](#) command.

The selected credit may be any positive integer less than or equal to 100, the default being 50%.

## Performing ATPG

Obtaining the optimal test set in the least amount of time is a desirable goal. [Figure 6-6](#) outlines how to most effectively meet this goal.

**Figure 6-6. Efficient ATPG Flow**



The first step in the process is to perform any special setup you may want for ATPG. This includes such things as setting limits on the pattern creation process itself. The second step is to create patterns with default settings (see [page 191](#)). This is a very fast way to determine how close you are to your testability goals. You may even obtain the test coverage you desire from your very first run. However, if your test coverage is not at the required level, you may have to troubleshoot the reasons for the inadequate coverage and create additional patterns using other approaches (see [page 191](#)).

## Setting Up for ATPG

Prior to ATPG, you may need to set certain criteria that aid the test generators in the test generation process. If you just want to generate patterns quickly in the ATPG tool using default settings, the recommended method for pattern creation is using the [create\\_patterns](#) command.

If the initial patterns are unsatisfactory, then run the `create_patterns` command a second time. If, however, you are still unable to create a satisfactory pattern set, then use the [set\\_pattern\\_type](#) command in conjunction with the `create_patterns` command using the following sequence:

```
ANALYSIS> set_pattern_type -sequential 2
```

```
ANALYSIS> create_patterns
```

A reasonable practice is creating patterns using these two commands with the sequential depth set to 2. This is described in more detail in [“Creating Patterns with Default Settings”](#) on page 191.

## Defining ATPG Constraints

*ATPG constraints* are similar to pin constraints and scan cell constraints. Pin constraints and scan cell constraints restrict the values of pins and scan cells, respectively. ATPG constraints place restrictions on the acceptable kinds of values at any location in the circuit. For example, you can use ATPG constraints to prevent bus contention or other undesirable events within a design. Additionally, your design may have certain conditions that can never occur under normal system operation. If you want to place these same constraints on the circuit during ATPG, use ATPG constraints.

During deterministic pattern generation, only the restricted values on the constrained circuitry are allowed. Unlike pin and scan cell constraints, which are only available in setup mode, you can define ATPG constraints in any system mode after design flattening. If you want to set ATPG constraints prior to performing design rules checking, you must first create a flattened model of the design using the [create\\_flat\\_model](#) command.

ATPG constraints are useful when you know something about the way the circuit behaves that you want the ATPG process to examine. For example, the design may have a portion of circuitry that behaves like a bus system; that is, only one of various inputs may be on, or selected, at a time. Using ATPG constraints, combined with a defined ATPG function, you can specify this information to the ATPG tool. ATPG functions place artificial Boolean relationships on circuitry within your design. After defining the functionality of a portion of circuitry with an ATPG function, you can then constrain the value of the function as desired with an ATPG constraint. This is more useful than just constraining a point in a design to a specific value.



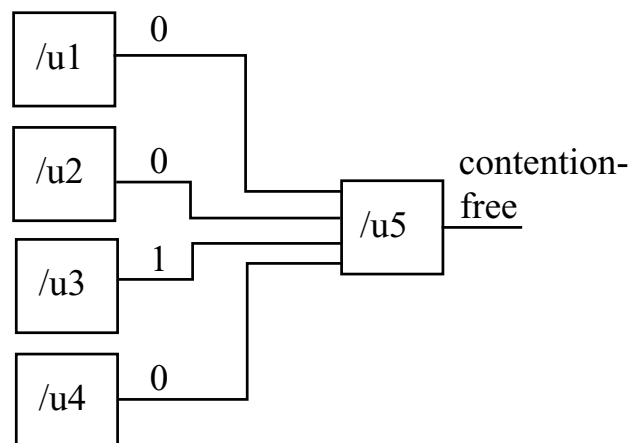
To define ATPG functions, use the [add\\_atpg\\_functions](#) command. When using this command, you specify a name, a function type, and an object to which the function applies.

You can specify ATPG constraints with the [add\\_atpg\\_constraints](#) command. When using this command, you specify a value, an object, a location, and a type.

Test generation considers all current constraints. However, design rules checking considers only static constraints. You can only add or delete static constraints in setup mode. Design rules checking generally does not consider dynamic constraints, but there are some exceptions detailed in the [set\\_drc\\_handling](#) command reference description (see the `Atpg_analysis` and `ATPGC` options). You can add or delete dynamic constraints at any time during the session. By default, ATPG constraints are dynamic.

[Figure 6-7](#) and the following commands give an example of how you use ATPG constraints and functions together.

**Figure 6-7. Circuitry with Natural “Select” Functionality**



The circuitry of [Figure 6-7](#) includes four gates whose outputs are the inputs of a fifth gate. Assume you know that only one of the four inputs to gate /u5 can be on at a time, such as would be true of four tri-state enables to a bus gate whose output must be contention-free. You can specify this using the following commands:

```
ANALYSIS> add_atpg_functions sel_func1 select1 /u1/o /u2/o /u3/o /u4/o
```

```
ANALYSIS> add_atpg_constraints 1 sel_func1
```

These commands specify that the “select1” function applies to gates /u1, /u2, /u3, and /u4 and the output of the select1 function should always be a 1. Deterministic pattern generation must

ensure these conditions are met. The conditions causing this constraint to be true are shown in [Table 6-1](#). When this constraint is true, gate /u5 will be contention-free.

**Table 6-1. ATPG Constraint Conditions**

| /u1 | /u2 | /u3 | /u4 | sel_func1 | /u5             |
|-----|-----|-----|-----|-----------|-----------------|
| 0   | 0   | 0   | 1   | 1         | contention-free |
| 0   | 0   | 1   | 0   | 1         | contention-free |
| 0   | 1   | 0   | 0   | 1         | contention-free |
| 1   | 0   | 0   | 0   | 1         | contention-free |

Given the defined function and ATPG constraint you placed on the circuitry, the ATPG tool only generates patterns using the values shown in [Table 6-1](#).

Typically, if you have defined ATPG constraints, the tools do not perform random pattern generation during ATPG. However, using the ATPG tool you can perform random pattern simulation using [simulate\\_patterns](#) command. In this situation, the tool rejects patterns during fault simulation that do not meet the currently-defined ATPG constraints.

- [analyze\\_atpg\\_constraints](#) — Analyzes a given constraint for either its ability to be satisfied or for mutual exclusivity.
- [analyze\\_restrictions](#) — Performs an analysis to automatically determine the source of the problems from a failed ATPG run.
- [delete\\_atpg\\_constraints](#) — Removes the specified constraint from the list.
- [delete\\_atpg\\_functions](#) — Removes the specified function definition from the list.
- [report\\_atpg\\_constraints](#) — Reports all ATPG constraints in the list.
- [report\\_atpg\\_functions](#) — Reports all defined ATPG functions.

## Excluding Power and Ground Ports from the Pattern Set

You can constrain power and ground ports during circuit setup and DRC, and also exclude those ports from the pattern set. The first step is to specify the power and ground ports by setting the value of the “function” attribute on a top-level input or inout port. For example, the following command designates the “vcc” port as a power input:

```
SETUP> set_attribute_value vcc -name function -value power
```

Note that the port can only be an input or inout port on a top-level module, and the only allowable values are “power” and “ground.” Also, the port cannot be an IJTAG port in the ICL file, if present. And you cannot change the “function” attribute after reading in the flat model.

Setting the value to “power” has the effect of adding an input constraint of CT1 on that port; and setting the value to “ground” has the effect of adding an input constraint of CT0 on that port.

The only way to remove these inferred constraints is by using the `reset_attribute_value` command. That is, the `delete_input_constraints` and `delete_pin_constraints` commands do not work in this situation.

After specifying the power and ground ports, you can write patterns that exclude those ports using the “`write_patterns -parameter_list`” command. Also, the parameter file has a keyword `ALL_EXCLUDE_POWER_GROUND` that allows you to control whether power and ground ports are excluded from tester pattern file formats, such as STIL and WGL. For more information, refer to the `ALL_EXCLUDE_POWER_GROUND` keyword description in the “[Parameter File Format and Keywords](#)” section of the *Tessent Shell Reference Manual*.

## Setting ATPG Limits

Normally, there is no need to limit the ATPG process when creating patterns. There may be an occasional special case, however, when you want the tool to terminate the ATPG process if CPU time, test coverage, or pattern (cycle) count limits are met. To set these limits, use the [set\\_atpg\\_limits](#) command.

## Understanding Event Simulation for DFFs and Latches

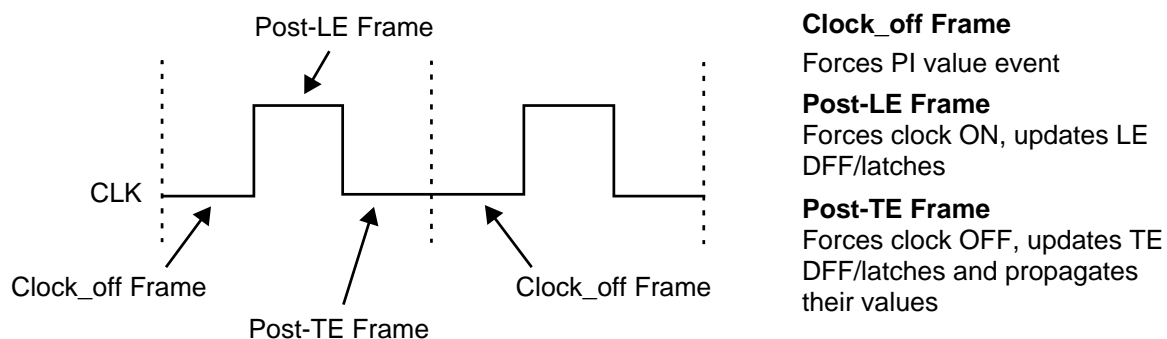
The following explains how the tool’s simulation kernel models DFFs and latches. The kernel simulates clock pulses as “010” or “101.” The three-digit notation refers to the state of the three frames that comprise a clock cycle: `clock_off` frame, `post-LE` frame, and `post-TE` frame. There are always at least three events per cycle. For more information, refer to [Figure 6-8](#).

The kernel distinguishes only between “edge-triggered” and “level-sensitive” sequential elements:

- Edge-triggered elements update on 0→1 transition.
- Level-sensitive elements update on `clock=1`.

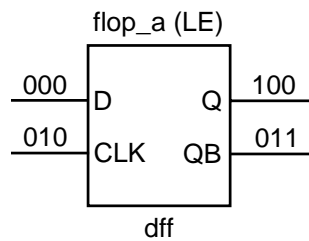
The kernel updates all elements immediately in the same frame.

**Figure 6-8. Simulation Frames**

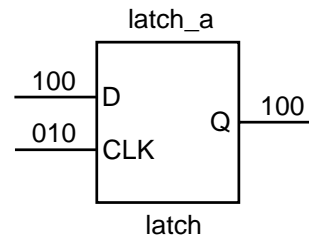


[Figure 6-9](#) explains how the simulation kernel models waveforms with DFFs and latches.

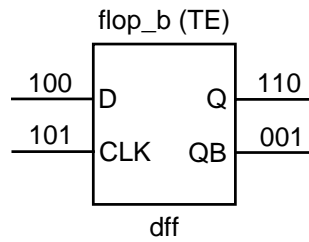
**Figure 6-9. Waveform Modeling for DFFs and Latches**



1. The first frame of the CLK input forces the DFF to output its init value of 1 on the Q pin.
2. The second frame of the CLK input forces the CLK ON due to the 0→1 edge, the DFF evaluates its D input and outputs Q=0 for the second frame.
3. The third frame of the CLK input forces the CLK OFF, the DFF performs no evaluation and holds state Q=0 for the third frame.



1. The first frame of the CLK input forces the latch to output its init value of 1 on the Q pin.
2. The second frame of the CLK input forces the level-sensitive CLK ON, the latch evaluates its D input and outputs Q=0 for the second frame.
3. The third frame of the CLK input forces the CLK OFF, the latch performs no evaluation and holds state Q=0 for the third frame.



1. The first frame of the CLK input forces the DFF to output its init value of 1 on the Q pin.
2. The second frame of the CLK input forces the CLK OFF, the DFF holds state and outputs Q=1 for the second frame.
3. The third frame of the CLK input forces the CLK ON due to the 0→1 edge, the DFF evaluates the third frame of its D input and outputs Q=0.

## Displaying Event Simulation Data for a Gate

You can display event simulation data for a gate using the [set\\_gate\\_report](#) and [report\\_gates](#) commands. For examples of how to do this refer to “Example 5” in the [set\\_gate\\_report](#) command description in the *Tessent Shell Reference Manual*.

## Using a Flattened Model to Save Time and Memory

To save time and memory, you can flatten your design and subsequently use the flattened model instead of the design netlist with the tool.

### Note



Before you save a flattened version of your design, ensure you have specified all necessary settings accurately. Some design information, such as that related to hierarchy, is lost when you flatten the design. Consequently, commands that require this information do not operate with the flattened netlist. Some settings, once incorporated in the flattened netlist, cannot be changed (for example, a tied constraint you apply to a primary input pin).

When you reinvoke the tool, you use this flattened netlist by specifying the “-flat” switch. The tool reinvokes in the same mode (setup or analysis) the tool was in when you saved the flattened model.

You flatten your design with the tool using one of the following methods depending on the tool’s current mode:

- **analysis mode** — The tool automatically creates the flat model when you change from setup to analysis mode using the [set\\_system\\_mode](#) command. After model flattening, you save the flattened design using the [write\\_flat\\_model](#) command.
- **setup mode** — You can manually create a flattened model in setup mode using the [create\\_flat\\_model](#) command. After model flattening, you save the flattened design using the [write\\_flat\\_model](#) command.

You can read a flat model into the tool in setup mode using the [read\\_flat\\_model](#) command.

An advantage of using a flattened netlist rather a regular netlist, is that you save memory and have room for more patterns.

## Creating a Pattern Buffer Area

To reduce demands on virtual memory when you are running the tool with large designs, use the [set\\_pattern\\_buffer](#) command with the ATPG tool. The tool will then store runtime pattern data in temporary files rather than in virtual memory.

## Using Fault Sampling to Save Processing Time

Another command, [set\\_fault\\_sampling](#), enables you to perform quick evaluation runs of large designs prior to final pattern generation. Intended for trial runs only, you can use this command to reduce the processing time when you want a quick estimate of the coverage to expect with your design.

## Setting Up for Checkpointing

The term “checkpointing” refers to when the tool automatically saves test patterns at regular periods, called checkpoints, throughout the pattern creation process. This is useful when ATPG takes a long time and there is a possibility it could be interrupted accidentally. For example, if a system failure occurs during ATPG, checkpointing enables you to recover and continue the run from close to the interruption point. You do not have to redo the entire pattern creation process from the beginning. The continuation run uses the data saved at the checkpoint, just prior to the interruption, saving you the time required to recreate the patterns that would otherwise have been lost.

The [set\\_checkpointing\\_options](#) command turns the checkpoint functionality on or off, specifies the time period between each write of the test patterns, as well as the name of the pattern file to which the tool writes the patterns.

## Example Checkpointing

Suppose a large design takes several days for the ATPG tool to process. You do not want to restart pattern creation from the beginning if a system failure ends ATPG one day after it begins. The following dofile segment defines a checkpoint interval of 90 minutes and enables checkpointing.

```
set_checkpointing_options on -pattern_file my_checkpoint_file -period 90 \  
-replace -pattern_format ascii -faultlist_file my_checkpoint_fault_file
```

If you need to perform a continuation run, invoking on a flattened model can be much faster than reflattening the netlist (see [“Using a Flattened Model to Save Time and Memory”](#) on page 188 for more information). After the tool loads the design, but before you continue the interrupted run, be sure to set all the same constraints you used in the interrupted run. The next dofile segment uses checkpoint data to resume the interrupted run:

```
# Load the fault population stored by the checkpoint.  
#  
# The ATPG process can spend a great deal of time proving  
# faults to be redundant (RE) or ATPG untestable (AU). By  
# loading the fault population using the -restore option, the  
# status of these fault sites will be restored. This will  
# save the time required to reevaluate these fault sites.  
read_faults my_checkpoint_fault_file -restore  
#  
# The report_statistics command shows if the fault coverage  
# is at the same level as at the last checkpoint the tool  
# encountered.  
report_statistics  
#  
# Set the pattern source to the pattern set that was stored  
# by the checkpoint. Then fault simulate these patterns.  
# During the fault simulation, the external patterns will be  
# copied into the tool's internal pattern set. Then, by  
# setting the pattern source back to the internal pattern  
# set, additional patterns can be added during a subsequent  
# ATPG run. This sequence is accomplished with the following  
# segment of the dofile.  
#  
# Fault grade the checkpoint pattern set.  
read_patterns my_checkpoint_file  
#  
# Reset the fault status to assure that the patterns  
# simulated do detect faults. When the pattern set is fault  
# simulated, if no faults are detected, the tool will not  
# retain the patterns in the internal pattern set.  
reset_state  
simulate_patterns  
report_statistics  
#  
# Create additional ATPG patterns  
create_patterns
```

After it executes the above commands, the tool should be at the same fault grade and number of patterns as when it last saved checkpoint data during the interrupted run. To complete the pattern creation process, you can now use the [create\\_patterns](#) command as described in the next section.

## Creating Patterns with Default Settings

You execute an optimal ATPG process that includes highly efficient pattern compression, by using the [create\\_patterns](#) command:

```
ANALYSIS> create_patterns
```

Review the transcript for any command or setting changes the tool suggests and implement those that will help you achieve your test goals. Refer to the [create\\_patterns](#) command description in the *Tessent Shell Reference Manual* for more information.

If the design has multiple clocks or non-scan sequential elements, consider issuing the following command before “create\_patterns”:

```
ANALYSIS> set_pattern_type -sequential 2
```

If the results do not meet your requirements, consider increasing the -sequential setting to 3, or as high as 4. The [set\\_pattern\\_type](#) command reference page provides details on the use of this command and can help you decide if you need it. Also, you can use the [report\\_sequential\\_fault\\_depth](#) command to quickly assess the upper limits of coverage possible under optimal test conditions for various sequential depths. This command displays an estimate of the maximum test coverage possible at different sequential depth settings.

If the first pattern creation run gives inadequate coverage, refer to “[Approaches for Improving ATPG Efficiency](#)” on page 191. To analyze the results if pattern creation fails, use the [analyze\\_atpg\\_constraints](#) command and the [analyze\\_restrictions](#) command.

## Approaches for Improving ATPG Efficiency

If you are not satisfied with the test coverage after initially creating patterns, or if the resulting pattern set is unacceptably large, you can make adjustments to several system defaults to improve results in another ATPG run. The following subsections provide helpful information and strategies for obtaining better results during pattern creation.

## Understanding the Reasons for Low Test Coverage

There are two basic reasons for low test coverage:

- Constraints on the tool
- Abort conditions

A high number of faults in the [ATPG\\_untestable \(AU\)](#) or PU fault categories indicates the problem lies with tool constraints. PU faults are a type of possible-detected, or [Posdet \(PD\)](#), fault. A high number of UC and UO faults, which are both [AU faults are categorized into several predefined fault sub-classes, as listed in Table 2-3 on page 62.](#) faults, indicates the problem lies with abort conditions. If you are unfamiliar with these fault categories, refer to “[Fault Classes](#)” on page 55.

When trying to establish the cause of low test coverage, you should examine the messages the tool prints during the deterministic test generation phase. These messages can alert you to what might be wrong with respect to [Redundant \(RE\)](#) faults, [ATPG\\_untestable \(AU\)](#) faults, and aborts. If you do not like the progress of the run, you can terminate the process with CTRL-C.

If a high number of aborted faults (UC or UO) appears to cause the problem, you can set the abort limit to a higher number, or modify some command defaults to change the way the application makes decisions. The number of aborted faults is high if reclassifying them as [Detected \(DT\)](#) or [Posdet \(PD\)](#) would result in a meaningful improvement in test coverage. In the tool’s coverage calculation (see “[Testability Calculations](#)” on page 65), these reclassified faults would increase the numerator of the formula. You can quickly estimate how much improvement would be possible using the formula and the fault statistics from your ATPG run. The following subsections discuss several ways to handle aborted faults.

---

**Note**

Changing the abort limit is not always a viable solution for a low coverage problem. The tool cannot detect [ATPG\\_untestable \(AU\)](#) faults, the most common cause of low test coverage, even with an increased abort limit. Sometimes you may need to analyze why a fault, or set of faults, remain undetected to understand what you can do.

---

Also, if you have defined several ATPG constraints or have specified `set_contention_check On -Atpg`, the tool may not abort because of the fault, but because it cannot satisfy the required conditions. In either of these cases, you should analyze the buses or ATPG constraints to ensure the tool *can* satisfy the specified requirements.

## Analyzing a Specific Fault

You can report on all faults in a specific fault category with the [report\\_faults](#) command. You can analyze each fault individually, using the pin pathnames and types listed by `report_faults`, with the [analyze\\_fault](#) command.

The `analyze_fault` command runs ATPG on the specified fault, displaying information about the processing and the end results. The application displays different data depending on the circumstances. You can optionally display relevant circuitry in DFTVisualizer using the `-Display` option. See the [analyze\\_fault](#) description in the *Tessent Shell Reference Manual* for more information.



You can also report data from the ATPG run using the `report_testability_data` command within the ATPG tool for a specific category of faults. This command displays information about connectivity surrounding the problem areas. This information can give you some ideas as to where the problem might lie, such as with RAM or clock PO circuitry. Refer to the [report\\_testability\\_data](#) description in the *Tessent Shell Reference Manual* for more information.

## Reporting on Aborted Faults

During the ATPG process, the tool may terminate attempts to detect certain faults given the ATPG effort required. The tools place these types of faults, called *aborted faults*, in the [AU faults are categorized into several predefined fault sub-classes, as listed in Table 2-3 on page 62.](#) fault class, which includes the UC and UO sub-classes. You can determine why these faults are undetected by using the `report_aborted_faults` command.

## Setting the Abort Limit

If the fault list contains a number of aborted faults, the tools may be able to detect these faults if you change the abort limit. You can increase the abort limit for the number of backtracks, test cycles, or CPU time and recreate patterns. To set the abort limit using the ATPG tool, use the `set_abort_limit` command.

The default for combinational ATPG is 30. The clock sequential abort limit defaults to the limit set for combinational. Both the `report_environment` command and a message at the start of deterministic test generation indicate the combinational and sequential abort limits. If they differ, the sequential limit follows the combinational abort limit.

The application classifies any faults that remain undetected after reaching the limits as *aborted faults*—which it considers undetected faults.

## Related Command

[report\\_aborted\\_faults](#) — Displays and identifies the cause of aborted faults.

## Setting Random Pattern Usage

The ATPG tool lets you specify whether to use random test generation processes when creating uncompressed patterns. In general, if you use random patterns, the test generation process runs faster and the number of test patterns in the set is larger. If not specified, the default is to use random patterns in addition to deterministic patterns. If you use random patterns exclusively, test coverage is typically very low. To set random pattern usage for ATPG, use the `set_random_atpg` command.

---

### Note



The `create_patterns` command does not use random patterns when generating compressed patterns.

---

## Changing the Decision Order

Prior to ATPG, the tool learns which inputs of multiple input gates it can most easily control. It then orders these inputs from easiest to most difficult to control. Likewise, the tool learns which outputs can most easily observe a fault and orders these in a similar manner. Then during ATPG, the tool uses this information to generate patterns in the simplest way possible.

This facilitates the ATPG process, however, it minimizes random pattern detection. This is not always desirable, as you typically want generated patterns to randomly detect as many faults as possible. To maximize random pattern detection, the tool provides the [set\\_decision\\_order](#) command to allow flexible selection of control inputs and observe outputs during pattern generation.

## Saving the Test Patterns

To save generated test patterns, enter the [write\\_patterns](#) command. For more information on the test data formats, refer to “[Saving Timing Patterns](#)” on page 344.

## Low-Power ATPG

Low-power ATPG allows you to create test patterns that minimize the amount of switching activity during test to reduce power consumption. Excessive power consumption can overwhelm the circuit under test, causing it to malfunction.

Low-power ATPG controls switching activity during scan load/unload and capture as described in the following topics:

- [Low-Power Capture](#)
- [Low-Power Shift](#)

---

### Note



Low-power constraints are directly related to the number of test patterns generated in an application. For example, using stricter low-power constraints results in more test patterns.

---

## Low-Power Capture

Low-power capture employs the clock gaters in a design to achieve the power target. Clock gaters controlling untargeted portions of the design are turned off, while clock gaters controlling targeted portions are turned on.

Power is controlled most effectively in designs that employ clock gaters, and especially multiple levels of clock gaters (hierarchy), to control a majority of the state elements.

This low-power feature is available using the *Capture* option of the [set\\_power\\_control](#) command.

## Low-Power Shift

Low-power shift minimizes the switching activity during shift with a constant fill algorithm where random values in scan chains are replaced with constant values as they are shifted through the core; A repeat fill heuristic is used to generate the constant values. This low-power feature is available in the ATPG tool using the [set\\_power\\_control](#) command during test pattern generation.

## Setting up Low-Power ATPG

Use this procedure to enable low-power for capture and shift before test patterns are generated. Setting up low-power ATPG is an iterative process that includes the setup, generation, and analysis of test patterns.

### Prerequisites

- Gate-level netlist with scan chains inserted.
- DFT strategy for your design. A test strategy helps define the most effective testing process for your design.

### Procedure

1. Invoke Tessent Shell:

```
% tessent -shell my_gate_scan.v -library my_lib.atpg \  
-logfile log/atpg_cg.log -replace
```

2. Set the context to “patterns -scan”:

```
SETUP> set_context patterns -scan
```

3. Set up test patterns.
4. Exit setup mode and run DRC:

```
SETUP> set_system_mode analysis
```

5. Correct any DRC violations.
6. Turn on low-power capture. For example:

```
ANALYSIS> set_power_control capture -switching_threshold_percentage 30 \  
-rejection_threshold_percentage 35
```

Switching during the capture cycle is minimized to 30% and any test patterns that exceed a 35% rejection threshold are discarded.

7. Turn on low-power shift. For example

```
ANALYSIS> set_power_control shift on -switching_threshold_percentage 20 \  
-rejection_threshold_percentage 25
```

Switching during scan chain loading is minimized to 20% and any test patterns that exceed a 25% rejection threshold are discarded.

8. Create test patterns:

```
ANALYSIS> create_patterns
```

Test patterns are generated and the test pattern statistics and power metrics display.

9. Analyze reports, adjust power and test pattern settings until power and test coverage goals are met. You can use the [report\\_power\\_metrics](#) command to report the capture power usage associated with a specific instance or set of modules.
10. Save test patterns. For example:

```
ANALYSIS> write_patterns ../generated/patterns_edt_p.stil -stil -replace
```

## Related Topics

[Low-Power Capture](#)

[set\\_power\\_control](#)

[Low-Power Shift](#)

## Creating an IDDQ Test Set

The ATPG tool supports the pseudo stuck-at fault model for IDDQ testing. This fault model allows detection of most of the common defects in CMOS circuits (such as resistive shorts) without costly transistor level modeling. “IDDQ Test” on page 33 introduces IDDQ testing.

Additionally, the tool supports supplemental IDDQ test generation. The tool creates a supplemental IDDQ test set when it generates an original set of IDDQ patterns based on the pseudo stuck-at fault model. Before running the supplemental IDDQ process, you must first set the fault type to IDDQ with the `set_fault_type` command.

During selective or supplemental IDDQ test generation, the tool classifies faults at the inputs of sequential devices such as scan cells or non-scan cells as **Blocked (BL)** faults. This is because the diversity of flip-flop and latch implementations means the pseudo stuck-at fault model cannot reliably guide ATPG to create a good IDDQ test. In contrast, a simple combinational logic gate has one common, fully complementary implementation (a NAND gate, for example, has two parallel pFETs between its output and Vdd and two series nFETs between its output and Vss), so the tool can more reliably declare pseudo stuck-at faults as detected. The switch level implementation of a flip-flop varies so greatly that assuming a particular implementation is highly suspect. The tool therefore takes a pessimistic view and reports coverage lower than it actually is, because it is unlikely such defects will go undetected for all IDDQ patterns.

Using the ATPG tool, you can generate IDDQ patterns using several user-specified checks. These checks can help ensure that the IDDQ test vectors do not increase IDDQ in the good circuit. The following subsections describe IDDQ test generation, and user-specified checks in more detail.

## Determining When to Perform the Measures

The pre-existing external test set may or may not target IDDQ faults. For example, you can run ATPG using the stuck-at fault type and then select patterns from this set for IDDQ testing. If the pattern set does not target IDDQ faults, it will not contain statements that specify IDDQ measurements. IDDQ test patterns must contain statements that tell the tester to make an IDDQ measure. In the Text format, this IDDQ measure statement, or *label*, appears as follows:

```
measure IDDQ ALL <time>;
```

By default, the ATPG tool places these statements at the end of patterns (cycles) that can contain IDDQ measurements. You can manually add these statements to patterns (cycles) within the external pattern set.

When you want to select patterns from an external set, you must specify which patterns can contain an IDDQ measurement. If the pattern set contains no IDDQ measure statements, you can specify that the tools assume the tester can make a measurement at the end of each pattern or cycle. If the pattern set already contains IDDQ measure statements (if you manually added these statements), you can specify that simulation should only occur for those patterns that

already contain an IDDQ measure statement, or label. To set this measurement information, use the [set\\_iddq\\_strobe](#) command.

## Generating an IDDQ Test Set

The following subsections discuss the basic IDDQ pattern generation process and provide an example of a typical IDDQ pattern generation run.

### Generating the Patterns

Prior to pattern generation, you may want to set up restrictions that the selection process must abide by when choosing the best IDDQ patterns. “[Specifying Leakage Current Checks](#)” on page 199 discusses these IDDQ restrictions. As with any other fault type, you issue the [create\\_patterns](#) command within analysis mode. This generates an internal pattern set targeting the IDDQ faults in the current list. You can turn dynamic pattern compression on with the [set\\_atpg\\_compression](#) On command, targeting multiple faults with a single pattern and resulting in a more compact test set.

### IDDQ Example

1. Invoke Tessent Shell, set the context to “patterns -scan,” read in the netlist, set up the appropriate parameters for ATPG run, pass rules checking, and then enter analysis mode.

```
...  
SETUP> set_system_mode analysis
```

2. Set the fault type to IDDQ.

```
ANALYSIS> set_fault_type iddq
```

3. Specify the number of desirable IDDQ patterns.

```
ANALYSIS> set_atpg_limits -pattern_count 128
```

4. Run ATPG, generating patterns that target the IDDQ faults in the current fault list.

Note that you could use the [set\\_iddq\\_checks](#) command prior to the ATPG run to place restrictions on the generated patterns. You can also optionally issue the [read\\_faults](#) command to add the target fault list at this point.

```
ANALYSIS> create_patterns
```

The tool then adds all faults automatically.

Note that you did not need to specify which patterns could contain IDDQ measures with [set\\_iddq\\_strobe](#), as the generated internal pattern source already contains the appropriate measure statements.

5. Save these IDDQ patterns into a file.

```
ANALYSIS> write_patterns iddq.pats
```

## Specifying Leakage Current Checks

For CMOS circuits with pull-up or pull-down resistors or tri-state buffers, the good circuit should have a nearly zero IDDQ current. The ATPG tool allows you to specify various IDDQ measurement checks to ensure that the good circuit does not raise IDDQ current during the measurement. Use the [set\\_iddq\\_checks](#) command to specify these options.

By default, the tool does not perform IDDQ checks. Both ATPG and fault simulation processes consider the checks you specify.

## Net Pair Identification with Calibre for Bridge Fault Test Patterns

You can use the ATPG tool to identify a list of net pairs that should be targeted for the bridging fault model and generated scan patterns. For complete information, refer to the “Bridge Fault Test Pattern Generation Flow” section of the *Calibre Solutions for Physical Verification* manual in your Calibre software tree or SupportNet.

The remainder of this section covers the specifics of the bridge fault model.

### Four-Way Dominant Fault Model

The ATPG tool uses the 4-Way Dominant fault model to target net pairs for bridging. The 4-Way Dominant fault model works by driving a net to a dominant value (0 or 1) and ensuring that the follower can be driven to the opposite value.

A simplified bridging fault model is adopted by defining the target net pairs as a set of stuck-at faults. Each of the net pairs targeted by the bridge fault model is classified as dominant or follower. A dominant net will force the follower net to take the same value as the dominant net in the faulty circuit when the follower has an opposite logical value than the dominant net.

Let A and B be two gates with output signals sig\_A and sig\_B. When sig\_A and sig\_B are bridged together, four faulty relationships can be defined:

- sig\_A has dominant value of 0 (sig\_A=0; sig\_B s@0)
- sig\_A has dominant value of 1 (sig\_A=1; sig\_B s@1)
- sig\_B has dominant value of 0 (sig\_B=0; sig\_A s@0)
- sig\_B has dominant value of 1 (sig\_B=1; sig\_A s@1)

By default, the ATPG tool targets each net pair using these four faulty relationships by generating patterns which drive one net to the dominant value and observing the behavior of the other (follower) net.

### Top-level Bridging ATPG

This flow generates patterns for a list of net pairs generated by another tool such as Calibre. The following input files are used in this flow:

- Gate-level Netlist
- ATPG Library
- Bridge Definition File or Calibre Server output file, here: from\_calibre.sites



The following commands are specific to bridging fault ATPG and should be issued once in analysis mode:

```
set_fault_type bridge
// FAULT TYPE SET TO 4WAY_DOM
read_fault_sites from_calibre.sites
// ALL BRIDGE NET PAIRS ARE LOADED
add_faults all
// GENERATE A LIST OF FAULTS BASED ON THE LOADED BRIDGE NET PAIRS
create_patterns
// GENERATES PATTERNS
write_patterns bridge_patterns.ascii
exit
```

The [set\\_fault\\_type](#) command with the bridge argument is the only specification needed for generation of patterns that target the 4-Way Dominant fault model. The [read\\_fault\\_sites](#) command is used to load the list of net pairs that should be targeted for the bridging fault model.

## Incremental Multi-Fault ATPG

The following flow generates patterns for a list of net pairs generated by another tool such as Calibre. After the initial ATPG for bridging, you can generate patterns for other fault models such as stuck-at. The flow fault simulates each pattern set for other fault models and generate additional patterns to target the remaining faults. The following input files are used in this flow:

- Gate-level Netlist
- ATPG Library
- Bridge Definition File or Calibre Server output file, here: *from\_calibre.sites*

The following example shows generation of patterns for bridging faults followed by stuck-at faults. The following commands should be issued once in analysis mode:

```
set_fault_type bridge
// fault type set to 4way_dom
read_fault_sites from_calibre.sites
// all bridge net pairs are loaded
create_patterns
// generate a list of faults based on the loaded bridge net pairs,
// then generate patterns
write_patterns bridge_patterns.ascii
set_fault_type stuck
add_faults all
// adds all stuck-at faults
read_patterns bridge_patterns.ascii
// load external patterns and add to internal patterns
simulate_patterns
// simulate bridge patterns for stuck-at faults
report_statistics
set_fault_protection on
// protect stuck-at faults that were detected by bridge patterns
reset_state
// remove bridge patterns that were effective in detecting stuck-at faults
create_patterns
// generate new patterns for remaining faults
write_patterns stuck_patterns.ascii
exit
```

The next example shows generation of patterns for stuck-at faults followed by bridging faults.

The following commands should be issued once in analysis mode:

```
set_fault_type stuck
create_patterns
// adds all stuck-at faults and generates patterns
write_patterns stuck_patterns.ascii
set_fault_type bridge
// fault type set to 4way_dom
read_fault_sites from_calibre.sites
// all bridge net pairs are loaded
read_patterns stuck_patterns.ascii
simulate_patterns
// simulate stuck-at patterns for bridge faults
write_faults bridge_faults.detected -class DT
// save detected bridge faults to file
create_patterns
reset_state
// remove stuck-at patterns that were effective in detecting bridge faults
read_faults bridge_faults.detected -retain
// load detected bridge faults and retain detection status
create_patterns
// generate new patterns for remaining undetected bridge faults
write_patterns patterns_bridge.ascii
exit
```

## The Bridge Parameters File

The bridge definition is a text file which you automatically generate using the Extraction Package. This section provides information for interpreting the entries in the bridge parameters file.

### File Syntax

Normally, you do not modify the generated bridge parameters file. If you do modify this file, then you must adhere to the following syntax:

- Precede each line of comment text with a pair of slashes (/).
- Do not modify keywords. They can be in upper-or lowercase.
- Use an equal sign to define a value for a keyword.
- Enclose all string values in double quotation marks (" ").
- Bridge declarations must be enclosed in braces ({ }).
- A semicolon (;) must separate each entry within the bridge declaration.

### Fault Definition File Keywords

Use the keywords described in the following table to create a bridge definition file. Keywords cannot be modified and can be in upper- or lowercase.

**Table 6-2. Bridge Definition File Keywords**

| Keyword (s)  | Usage Rules                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| VERSION      | Required. Used to specify the version of the bridge definition file and must be declared before any bridge entries. Must be a real number or integer written in non-scientific format starting with 1.1.                                                                                                                                                                                                                                                                                                                                                               |
| FAULT_TYPE   | Optional. Used to indicate what type of fault is declared by the FAULTS keyword. Must be a string value enclosed in quotation marks (" ").                                                                                                                                                                                                                                                                                                                                                                                                                             |
| BRIDGE       | Required. Used to start a bridge entry.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| NET1<br>NET2 | Required. Identifies the net/pin pair for the bridge entry. Specifies the regular or hierarchical net/pin pathname to gate(s). Use the following guidelines when using these keywords: <ul style="list-style-type: none"> <li>• Declare either net or pin value pairs.</li> <li>• Net/Pin pairs are the first two items declared in the bridge entry.</li> <li>• Net/Pin pathnames are string values and should be enclosed in quotation marks (" ").</li> <li>• A net can be used in multiple bridge entries.</li> <li>• Nets can be defined in any order.</li> </ul> |

**Table 6-2. Bridge Definition File Keywords**

| Keyword (s)               | Usage Rules                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|---------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| FAULTS                    | <p>Optional. Provides a fault classification for each of the four components of the 4-way bridge fault model. Four classifications must be specified in a comma-separated list enclosed in braces ({}). You can use any of the following 2-digit codes for each of the four components:</p> <ul style="list-style-type: none"> <li>• UC (uncontrolled)</li> <li>• UO (unobserved)</li> <li>• DS (det_simulation)</li> <li>• PU (posdet_untestable)</li> <li>• PT (posdet_testable)</li> <li>• TI (tied)</li> <li>• BL (blocked)</li> <li>• AU (ATPG_untestable)</li> <li>• NF (directs the tool to nofault this component of the bridge)</li> </ul> <p>Using fault classifications, you can filter and display the desired fault types. For more information on fault classes and codes, see <a href="#">“Fault Classes”</a> on page 55.</p> |
| NAME                      | Optional. A unique string, enclosed in quotation marks (" "), that specifies a name for the bridge.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| DISTANCE <sup>1</sup>     | Optional. Real number that specifies the distance attribute in microns (um) for the bridge entry.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| PARALLEL_RUN <sup>1</sup> | Optional. Real number that specifies the parallel_run attribute in microns (um) for the bridge entry.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| LAYER <sup>1,2</sup>      | Optional. String that specifies the layer attribute for the bridge entry. Must be enclosed in quotation marks (" ").                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| WEIGHT <sup>1,2</sup>     | Optional. Real number that specifies the weight attribute for the bridge entry.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| TYPE <sup>1,2</sup>       | <p>Optional. 3-, 4-, or 5-character code that specifies a type identification for the bridge entry. Must be enclosed in quotation marks (" "). Use one of the following codes:</p> <ul style="list-style-type: none"> <li>• S2S — Side-to-side</li> <li>• SW2S — Side-Wide-to-Side: same as S2S but at least one of the two signal lines is a wide metal line</li> <li>• S2SOW — Side-to-Side-Over-Wide: same as S2S, but the bridge is located over a wide piece of metal in a layer below</li> <li>• C2C — Corner-to-Corner</li> <li>• V2V — Via-to-Via: an S2S for vias</li> <li>• VC2VC — Via-Corner-to-Via-Corner</li> <li>• EOL — End-of-Line: the end head of a line faces another metal line</li> </ul>                                                                                                                              |

1. This keyword is not used by the software to define any specific value. However, you can use it to specify a value which can be filtered on by the [read\\_fault\\_sites](#) command.

2. You can display a histogram of the number of equivalent classes of bridges based on either their layer, weight, or type attribute using the [report\\_fault\\_sites](#) or [report\\_faults](#) command.

## File Example

The following example shows the format of the bridge definition file. The keywords used in this file are case insensitive.

```
// Any string after // is a comment and will be ignored by the tool.
VERSION 1.1// An optional item that declares the version of
// the bridge definition file. VERSION must be
// defined before any bridge entry. The version
// number starts from 1.0.
FAULT_TYPE = BRIDGE_STATIC_4WAY_DOM
// An optional header that indicates the fault type
// to be declared by keywords FAULTS in
// bridge body. Currently, the only valid fault type
// is BRIDGE_STATIC_4WAY_DOM
BRIDGE {// Define the bridge entry body and can be repeated
// as many times as necessary.
NET1 = NET_NAME; // Defines the first net name.
NET2 = NET_NAME; // Defines the second net name.
// NET1 and NET2 must be defined in each bridge
// entry and they must be declared before any other
// items defined in the bridge entry body.
FAULTS = {FAULT_CATEGORY_1, FAULT_CATEGORY_2, FAULT_CATEGORY_3,
FAULT_CATEGORY_4};
// An optional item that defines the fault
// classes for each of the four faulty
// relationships. All four fault categories must be
// declared and must be in the order shown in the
// previous section. Examples of fault classes are
// UC, UO, DS, DI, PU, // PT, TI, BL and AU. A new
// fault class NF (No Fault) can be used to exclude
// any specific faulty relationship for ATPG.
NAME = STRING; // An optional string specifying the name
// of the bridge.
PARALLEL_RUN = floating number; // An optional item specifying the
// parallel run length of the nets of the bridge.
DISTANCE = floating number; // An optional item specifying the distance
// of the nets of the bridge.
WEIGHT = floating number; // An optional item specifying a weight assigned
// to the bridge.
LAYER = LAYER_NAME; // An optional item specifying the name of the
// layer the bridge is in.
} // End of bridge body
```

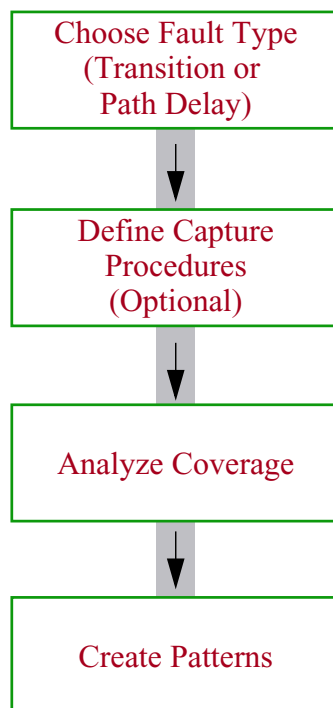
The items DISTANCE, PARALLEL\_RUN, WEIGHT, and LAYER are collectively referred to as attributes of the bridge. For ATPG purposes, these attributes are ignored, and by default not reported. You have to specify the -attribute option to according commands to see them. The unit of length used for DISTANCE and PARALLEL\_RUN is um ( $10^{-6}$ ) meters.

## Creating a Delay Test Set

Delay, or “at-speed” tests in the ATPG tool are of two types: transition delay and path delay.

[Figure 6-10](#) shows a general flow for creating a delay pattern set.

**Figure 6-10. Flow for Creating a Delay Test Set**



Your process may be different and it may involve multiple iterations through some of the steps, based on your design and coverage goals. This section describes these two test types in more detail and how you create them using the ATPG tool. The following topics are covered:

|                                                                             |     |
|-----------------------------------------------------------------------------|-----|
| Creating a Transition Delay Test Set .....                                  | 206 |
| Creating a Path Delay Test Set .....                                        | 220 |
| At-Speed Test Using Named Capture Procedures .....                          | 230 |
| Support for On-Chip Clocks (PLLs) .....                                     | 231 |
| Mux-DFF Example .....                                                       | 238 |
| Generating Test Patterns for Different Fault Models and Fault Grading ..... | 248 |

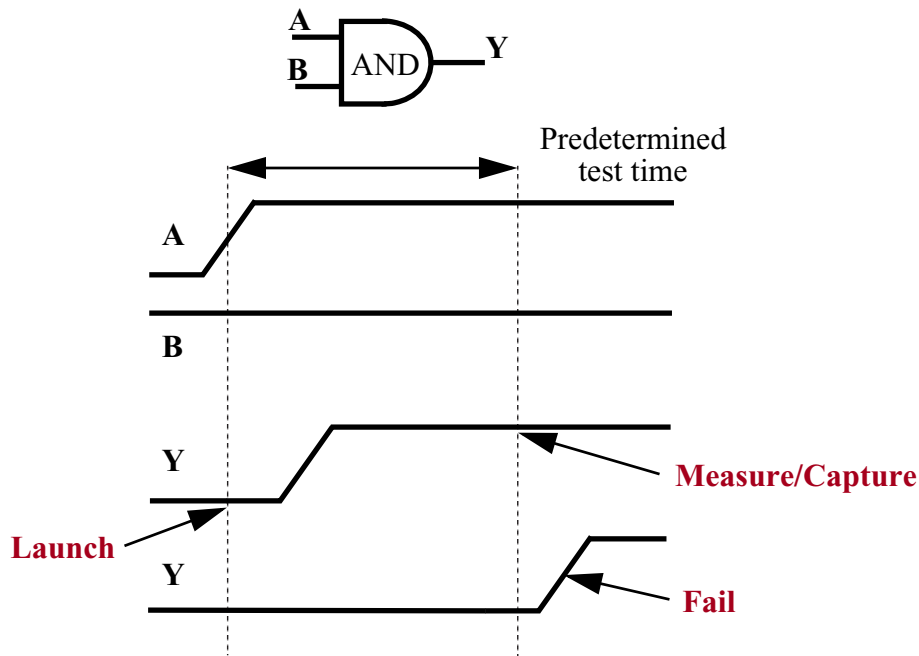
## Creating a Transition Delay Test Set

The tool can generate patterns to detect transition faults. “[At-Speed Testing and the Transition Fault Model](#)” on page 40 introduced the transition fault model. Transition faults model gross delays on gate terminals (or nodes), allowing each terminal to be tested for slow-to-rise or slow-to-fall behavior. The defects these represent may include things like partially conducting transistors or interconnections.

[Figure 6-11](#) illustrates the six potential transition faults for a simple AND gate. These are comprised of slow-to-rise and slow-to-fall transitions for each of the three terminals. Because a transition delay test checks the speed at which a device can operate, it requires a two cycle test. First, all the conditions for the test are set. In the figure, A and B are 0 and 1 respectively. Then

a change is launched on A, which should cause a change on Y within a pre-determined time. At the end of the test time, a circuit response is captured and the value on Y is measured. Y might not be stuck at 0, but if the value of Y is still 0 when the measurement is taken at the capture point, the device is considered faulty. The ATPG tool automatically chooses the launch and capture scan cells.

**Figure 6-11. Transition Delay**



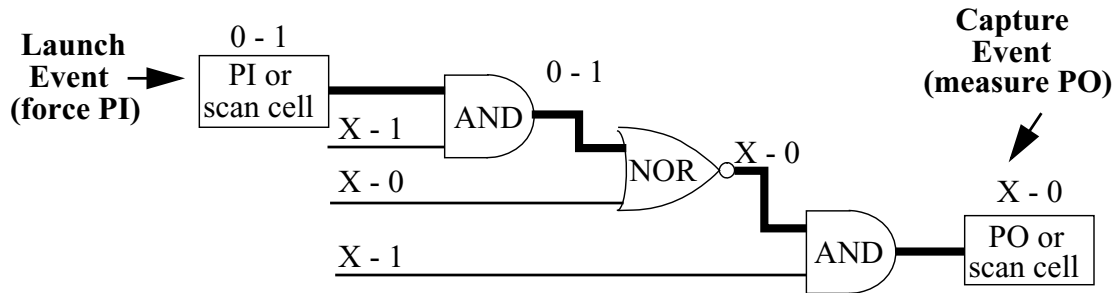
## Transition Fault Detection

To detect transition faults, the following conditions must be met:

- The corresponding stuck-at fault must be detected.
- Within a single previous cycle, the node value must be at the opposite value than the value detected in the current cycle.

Figure 6-12 depicts the launch and capture events of a small circuit during transition testing. Transition faults can be detected on any pin.

**Figure 6-12. Transition Launch and Capture Events**



To detect a transition fault, a typical pattern includes the events in [Figure 6-13](#).

**Figure 6-13. Events in a Broadside Pattern**

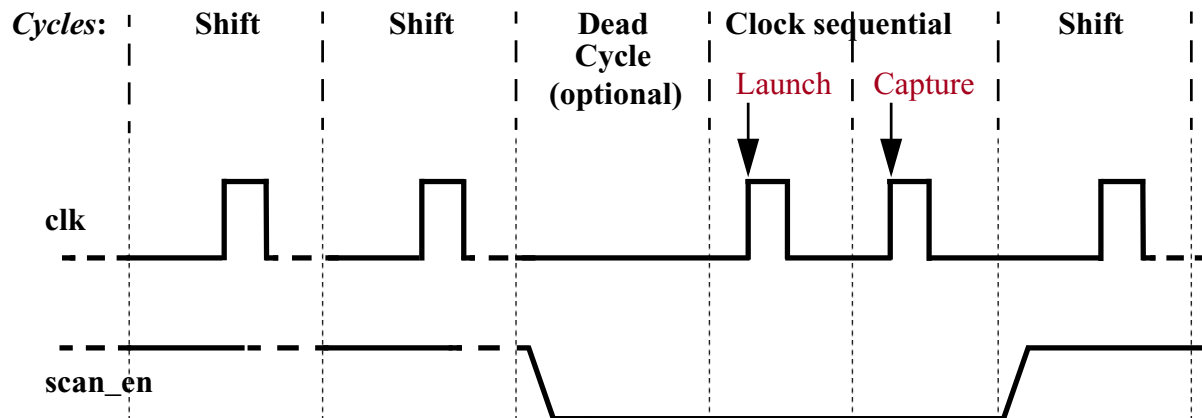
1. Load scan chains
2. Force primary inputs
3. Pulse clock
4. Force primary inputs
5. Measure primary outputs
6. Pulse clock
7. Unload scan chains

This is a clock sequential pattern, commonly referred to as a broadside pattern. It has basic timing similar to that shown in [Figure 6-14](#) and is the kind of pattern the ATPG tool attempts to create by default when the clock-sequential depth (the depth of non-scan sequential elements in the design) is two or larger. You specify this depth with the [set\\_pattern\\_type](#) command's -Sequential switch. The default setting of this switch upon invocation is 0, so you would need to change it to at least 2 to enable the tool to create broadside patterns.

Typically, this type of pattern eases restrictions on scan enable timing because of the relatively large amount of time between the last shift and the launch. After the last shift, the clock is pulsed at speed for the launch and capture cycles.



**Figure 6-14. Basic Broadside Timing**



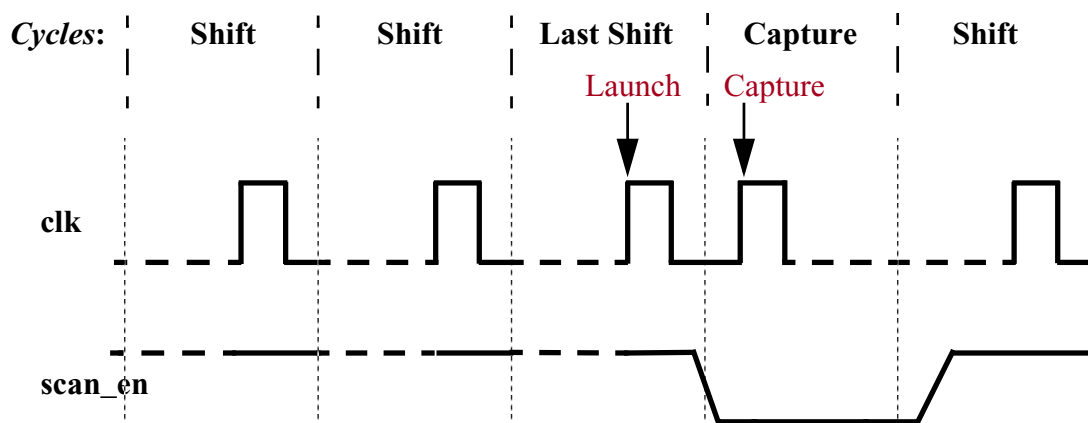
If it fails to create a broadside pattern, the tool next attempts to generate a pattern that includes the events shown in [Figure 6-15](#).

**Figure 6-15. Events in a Launch Off Shift Pattern**

1. Init\_force primary inputs
2. Load scan chains
3. Force primary inputs
4. Measure primary outputs
5. Pulse clock
6. Unload scan chains

In this type of pattern, commonly referred to as a launch off last shift or just launch off shift pattern, the transition occurs because of the last shift in the load scan chains procedure (event #2) or the forcing of the primary inputs (event #3). [Figure 6-16](#) shows the basic timing for a launch that is triggered by the last shift.

**Figure 6-16. Basic Launch Off Shift Timing**



If your design cannot support this requirement, you can direct the tool not to create launch off shift patterns by including the `-No_shift_launch` switch when specifying transition faults with the `set_fault_type` command.

---

**Note**



Launch off shift patterns require the scan enable signal for mux-scan designs to transition from shift to capture mode at speed. Therefore, the scan enable must be globally routed and timed similar to a clock. Also, because launch off shift patterns detect a certain number of faults in non-functional paths as well as in the scan path, the test coverage reported is usually higher than for broadside patterns which do not capture faults in non-functional paths.

---

The following are example commands you could use at the command line or in a dofile to generate broadside transition patterns:

```
SETUP> add_input_constraintscan_en -c0    #force for launch & capture.
SETUP> set_output_masks on                #do not observe primary outputs.
SETUP> set_transition_holdpi on           #freeze primary input values.
ANALYSIS> set_fault_type transition -no_shift_launch    #prohibit launch off last shift.
ANALYSIS> set_pattern_type -sequential 2    #sequential depth depends on design.
ANALYSIS> create_patterns
```

To create transition patterns that launch off the last shift, use a sequence of commands similar to this:

```
SETUP> set_output_masks on                #don't observe primary outputs.
...
ANALYSIS> set_fault_type transition
ANALYSIS> set_pattern_type -sequential 0    #prevent broadside patterns.
ANALYSIS> create_patterns
```

The following is a list of related commands:

- `set_abort_limit` — Specifies the abort limit for the test pattern generator.
- `set_fault_type` — Specifies the fault model for which the tool develops or selects ATPG patterns.
- `set_pattern_type` — Specifies the type of test patterns the ATPG simulation run uses.

## Basic Procedure for Generating a Transition Test Set

Use the following basic procedure for generating a transition test set:

1. Perform circuit setup tasks.
2. Constrain the scan enable pin to its inactive state. For example:  

```
SETUP> add_input_constraints scan_en -c0
```
3. Set the sequential depth to two or greater (optional):  

```
SETUP> set_pattern_type -sequential 2
```
4. Enter analysis system mode. This triggers the tool's automatic design flattening and rules checking processes.  

```
SETUP> set_system_mode analysis
```
5. Set the fault type to transition:  

```
ANALYSIS> set_fault_type transition
```
6. Add faults to the fault list:  

```
ANALYSIS> add_faults -all
```
7. Run test generation:  

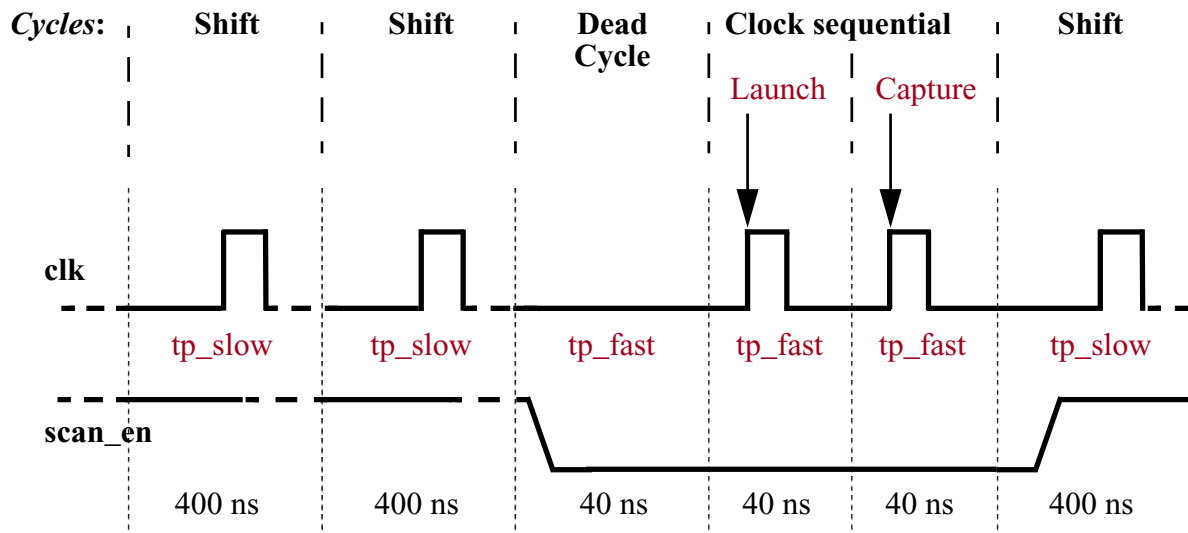
```
ANALYSIS> create_patterns
```

## Timing for Transition Delay Tests

This section describes how the timing works for transition delay tests. Basically, the tool obtains the timing information from the test procedure file. This file describes the scan circuitry operation to the tool. You can create scan circuitry manually, or use Tessent Scan to create the scan circuitry for you after it inserts scan circuitry into the design. The test procedure file contains cycle-based procedures and timing definitions that tell the ATPG tool how to operate the scan structures within a design. See “[Test Procedure File](#)” in the *Tessent Shell User's Manual*.

Within the test procedure file, timeplates are the mechanism used to define tester cycles and specify where all event edges are placed in each cycle. As shown conceptually in [Figure 6-14](#) for broadside testing, slow cycles are used for shifting (load and unload cycles) and fast cycles for the launch and capture. [Figure 6-17](#) shows the same diagram with example timing added.

**Figure 6-17. Broadside Timing Example**



This diagram now shows 400 nanosecond periods for the slow shift cycles defined in a timeplate called *tp\_slow* and 40 nanosecond periods for the fast launch and capture cycles defined in a timeplate called *tp\_fast*.

The following are example timeplates and procedures that would provide the timing shown in [Figure 6-17](#). For brevity, these excerpts do not comprise a complete test procedure. Normally, there would be other procedures as well, like setup procedures.

```
timeplate tp_slow =
  force_pi 0;
  measure_po 100;
  pulse clk 200 100;
  period 400;
end;
```

```
procedure load_unload =
  scan_group grp1;
  timeplate tp_slow;
  cycle =
    force_clk 0;
    force_scan_en 1;
  end;
  apply shift 127;
end;
```

```
procedure shift =
  timeplate tp_slow;
  cycle =
    force_sci;
    measure_sco;
    pulse clk;
  end;
end;
```

```
timeplate tp_fast =
  force_pi 0;
  measure_po 10;
  pulse clk 20 10;
  period 40;
end;
```

```
procedure capture =
  timeplate tp_fast;
  cycle =
    force_pi;
    measure_po;
    pulse_capture_clock;
  end;
end;
```

```
procedure clock_sequential =
  timeplate tp_fast;
  cycle =
    force_pi;
    pulse_capture_clock;
    pulse_read_clock;
    pulse_write_clock;
  end;
end;
```



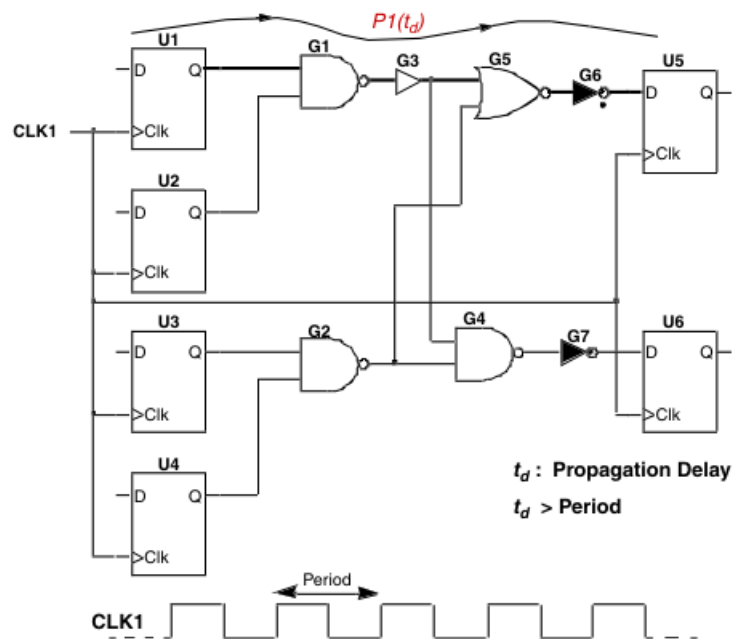
By moving the clock pulse later in the period for the load\_unload and shift cycles and earlier in the period for the capture cycle, the 40 nanosecond time period between the launch and capture clocks is achieved.

## Preventing Pattern Failures Due to Timing Exception Paths

Prior to ATPG, you can perform a timing optimization on a design using a static timing analysis (STA) tool. This process also defines timing exception paths consisting of one of the following:

- **False Path** — A path that cannot be sensitized in the functional mode of operation (the STA tool ignores these paths when determining the timing performance of a circuit).
- **Multicycle Path** — A path with a signal propagation delay of more than one clock cycle. Figure 6-19 shows path *P1* beginning at flip-flop U1, going through gates G1, G3, G5, and G6, and ending at flip-flop U5. This path has a total propagation delay longer than the clock period.

Figure 6-19. Multicycle Path Example



You should evaluate the effect of timing exception paths for any sequential pattern containing multiple at-speed capture clock pulses, either from the same clock or from different clocks. This includes the following pattern types:

- Clock sequential (broadside transition patterns and stuck-at patterns)
- RAM sequential
- Path delay

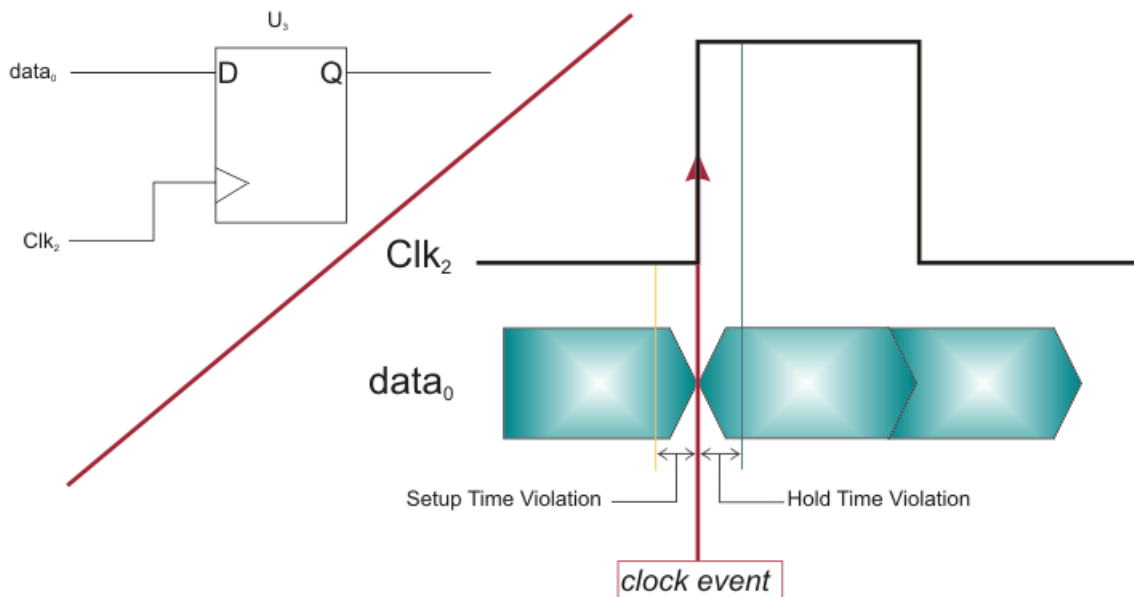
The ATPG tool automatically evaluates false and multicycle paths before creating the test patterns. When simulating patterns during ATPG, the tool identifies incorrect capture responses due to false paths and masks them (modifies them to X) in the resultant patterns. For multicycle paths, expected values are accurately simulated and stored in the pattern set.

## Types of Timing Exception Path Violations

Most designs hold data and control inputs constant for specified time periods before and after any clock events. In this context, the time period *before* the clock event is the setup time, and the time period *after* the clock event is hold time.

Figure 6-20 illustrates how setup and hold time exceptions can produce the following timing exception path violations: [Setup Time Violations](#) and [Hold Time Violations](#).

**Figure 6-20. Setup Time and Hold Time Violations**



## Setup Time Violations

A timing exception path with setup time violation does not meet the setup time requirements. This type of violation can affect test response for at-speed test patterns.

## Hold Time Violations

A timing exception path with hold time violation does not meet the hold time requirements. This type of violation can affect test response of *any* test pattern and usually occurs across different clock domains. The ATPG tool simulates hold time false paths for the following timing exception paths:

- False paths you manually specify with `add_false_paths` command using the `-hold` switch—see “[Manually Specifying False Paths for Hold Time Violation Checks](#)”.
- False paths between two clock domains.
- Multicycle paths with a path multiplier of 0 (zero).

Figure 6-21 illustrates a hold time violation occurring across different clock domains.

**Figure 6-21. Across Clock Domain Hold Time Violation**

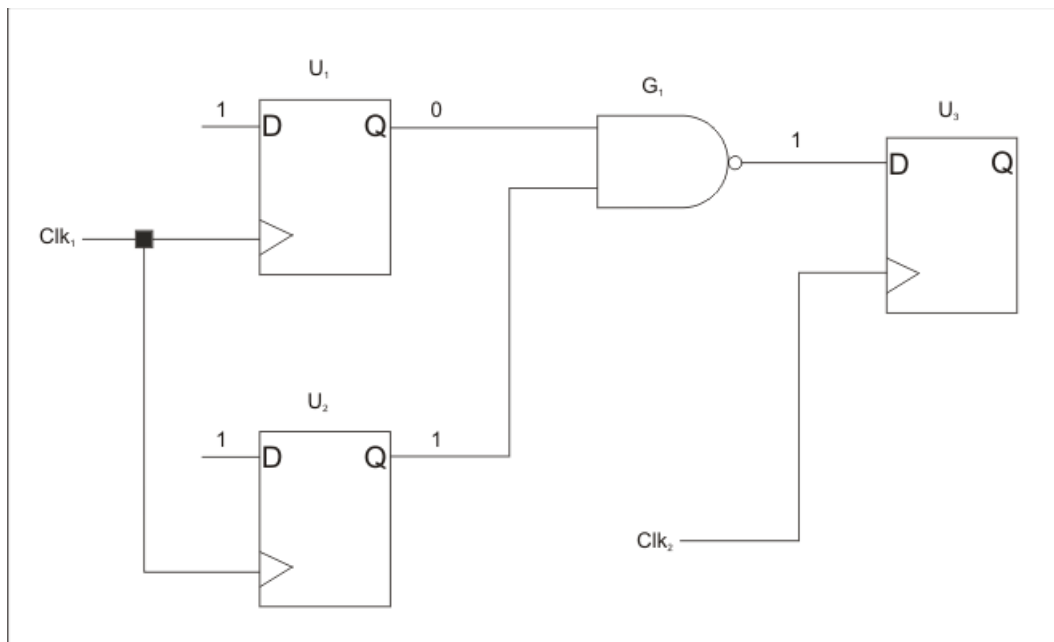


Figure 6-21 shows the false paths from the clock domain Clk1 to the clock domain Clk2. In this figure, when a test pattern has a clock sequence of simultaneously pulsing both Clk1 and Clk2, there can be a hold time violation from the flip-flop U1 or U2 to the flip-flop U3.

In Figure 6-21, pulsing clocks Clk1 and Clk2 simultaneously places the new value 1 at the D input of flip-flop U1, creating a rising transition on the flip-flop U1's Q output. This transition



sensitizes the path from the flip-flop U1 to the flip-flop U3. If the clock Clk2 arrives late at the flip-flop U3 the new value 0 is captured at the flip-flop U3 instead of the old value 1.

## Reading Timing Exception Paths from an SDC File

STA tools typically provide a command to write out the false and multicycle path information identified during the STA process into a file or script in Synopsys Design Constraint (SDC) format. For example, the Synopsys PrimeTime tool has the `write_sdc` command. You can also read in both the `-setup` and `-hold` SDC information.



**Tip:** Before using the `write_sdc` command, use the PrimeTime `transform_exceptions` command to eliminate redundant, overridden or invalid exceptions.

If you can get the information into an SDC file, you can use the `read_sdc` command to read in the false path definitions from the file.

The following is an example of the use of this command in a typical command sequence for creating broadside transition patterns:

```
<Define clocks, scan chains, constraints, and so on>
ANALYSIS> set_fault_type transition -no_shift_launch
ANALYSIS> set_pattern_type -sequential 2
ANALYSIS> read_sdc my_sdc_file
...
ANALYSIS> create_patterns
```

If you already have a pattern set for your design and want to see the effect of adding the false and multicycle path information, the command sequence is slightly different:

```
<Define clocks, scan chains, constraints, and so on>
ANALYSIS> read_sdc my_sdc_file
ANALYSIS> add_faults -all
ANALYSIS> read_patterns my_patterns.ascii
ANALYSIS> simulate_patterns
ANALYSIS> report_statistics
```

As a result of simulating the patterns using the false and multicycle path information, the patterns read in from the external pattern file will now be stored in the tool's internal pattern set, with some capture values in the internal patterns changed to "X". These changed values represent masking the tool applied to adjust for false and multicycle path effects. The Xs will increase the number of undetected faults slightly and lower the test coverage; however, the patterns will be more correct and will eliminate mismatches related to those capture values.

#### Note



You can save the patterns that include the false and/or multicycle path information as usual. For example:

```
ANALYSIS> write_patterns my_patterns_falsepaths.v -verilog
ANALYSIS> write_patterns my_patterns_falsepaths.ascii -ascii
```

---

## Ensuring the SDC File Contains Valid SDC Information

The `read_sdc` command reads an SDC file and parses it, looking for supported commands that are relevant for ATPG. ATPG tools support the following SDC commands and arguments:

- `all_clocks`
- `create_clock [-name clock_name]`
- `create_generated_clock [-name clock name]`
- `get_clocks`
- `get_generated_clocks`
- `get_pins`
- `get_ports`
- `set_case_analysis value port_or_pin_list`
- `set_clock_groups`
- `set_disable_timing [-from from_pin_name] [-to to_pin_name] cell_pin_list`
- `set_false_path [-setup] [-hold] [-from from_list] [-to to_list] [-through through_list]`
- `set_hierarchy_separator`
- `set_multicycle_path [-setup] [-hold] [-from from_list] [-to to_list] [-through through_list]`

#### Note



For complete information on these commands and arguments, refer to your SDC documentation.

---

To avoid problems extracting the timing exception paths from the SDC specifications, the best results are obtained when the SDC file is written out by the PrimeTime static timing analysis (STA) tool. Mentor Graphics highly recommends that within PrimeTime you use the `transform_exceptions` command before saving the SDC file. This command removes any redundant, invalid and overridden path exceptions. Then use the `write_sdc` command to save the updated information to a file you can use with `read_sdc`.

The following summarizes the recommended steps:

1. Read the original SDC file(s) in PrimeTime.
2. Execute the `transform_exceptions` command in PrimeTime.
3. Execute the `write_sdc` command in PrimeTime, to write out a valid SDC file.
4. In the ATPG tool, use the `read_sdc` command to read the SDC file written out in step 3.
5. Generate at-speed patterns.

## Defining False Paths Manually

Alternatively and using the `add_false_paths` command, you can *manually* specify false path definitions for both [Manually Specifying False Paths for Setup Time Violation Checks](#) and [Manually Specifying False Paths for Hold Time Violation Checks](#).

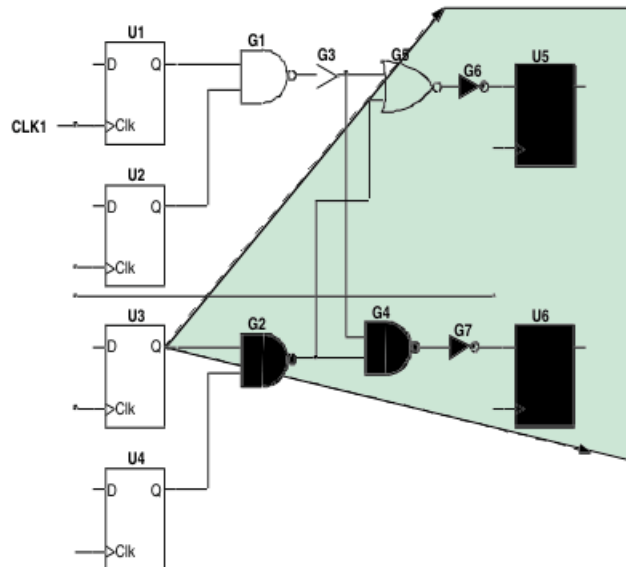
When performing this operation, you must be specific and accurate when specifying false (and multicycle) paths during the STA process, and to maintain the same accuracy when using the `add_false_paths` command.

For example, defining non-specify false path definition with the following command:

**`add_false_paths -from U3`**

would result in the propagation of the false path out through the design in an effect cone encompassing all possible paths from that node. [Figure 6-22](#) shows an illustration of this.

**Figure 6-22. Effect Cone of a Non-specific False Path Definition**



## Manually Specifying False Paths for Setup Time Violation Checks

By default, the application evaluates setup time violations for the false paths you manually specify with the [add\\_false\\_paths](#) command.

## Manually Specifying False Paths for Hold Time Violation Checks

For hold time violations, you identify the path with the [add\\_false\\_paths](#) command and also specify the -hold switch.

- [delete\\_false\\_paths](#) — Deletes the specified false path definitions.
- [report\\_false\\_paths](#) — Displays the specified false path definitions.
- [delete\\_multicycle\\_paths](#) — Deletes the specified multicycle path definitions.
- [report\\_multicycle\\_paths](#) — Displays the specified multicycle path definitions.

## Creating a Path Delay Test Set

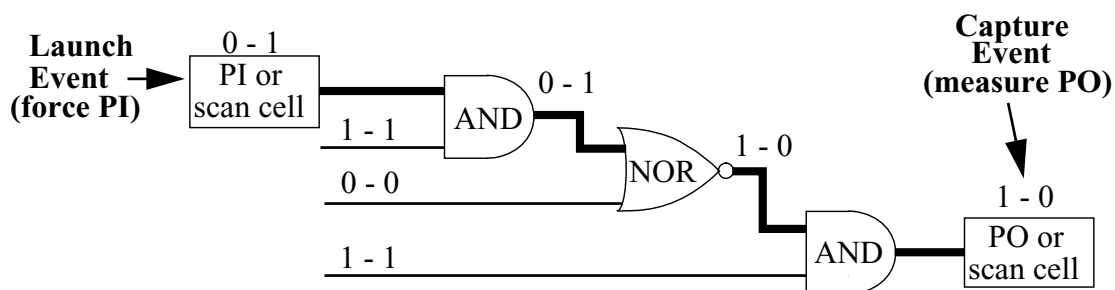
The ATPG tool can generate patterns to detect path delay faults. These patterns determine if specific user-defined paths operate correctly at-speed. “[At-Speed Testing and the Path Delay Fault Model](#)” on page 41 introduced the path delay fault model. You determine the paths you want tested (most people use a static timing analysis tool to determine these paths), and list

them in an ASCII path definition file you create. You then load the list of paths into the tool. “The Path Definition File” on page 225 describes how to create and use this file.

## Path Delay Fault Detection

Path delay testing requires a logic value transition, which implies two events need to occur to detect a fault. These events include a launch event and a capture event. Typically, both the launch and capture occur at scan cells, but they can occur at RAMs or, depending on the timing and precision of the ATE to test around a chip’s I/O, at PIs and POs. Figure 6-23 depicts the launch and capture events of a small circuit during a path delay test.

**Figure 6-23. Path Delay Launch and Capture Events**



Path delay patterns are a variant of clock-sequential patterns. A typical pattern to detect a path delay fault includes the following events:

1. Load scan chains
2. Force primary inputs
3. Pulse clock (to create a launch event for a launch point that is a state element)
4. Force primary inputs (to create a launch event for a launch point that is a primary input)
5. Measure primary outputs (to create a capture event for a capture point that is a primary output)
6. Pulse clock (to create a capture event for a capture point that is a state element)
7. Unload scan chains

The additional force\_pi/pulse\_clock cycles may occur before or after the launch or capture events. The cycles depend on the sequential depth required to set the launch conditions or sensitize the captured value to an observe point.

### Note



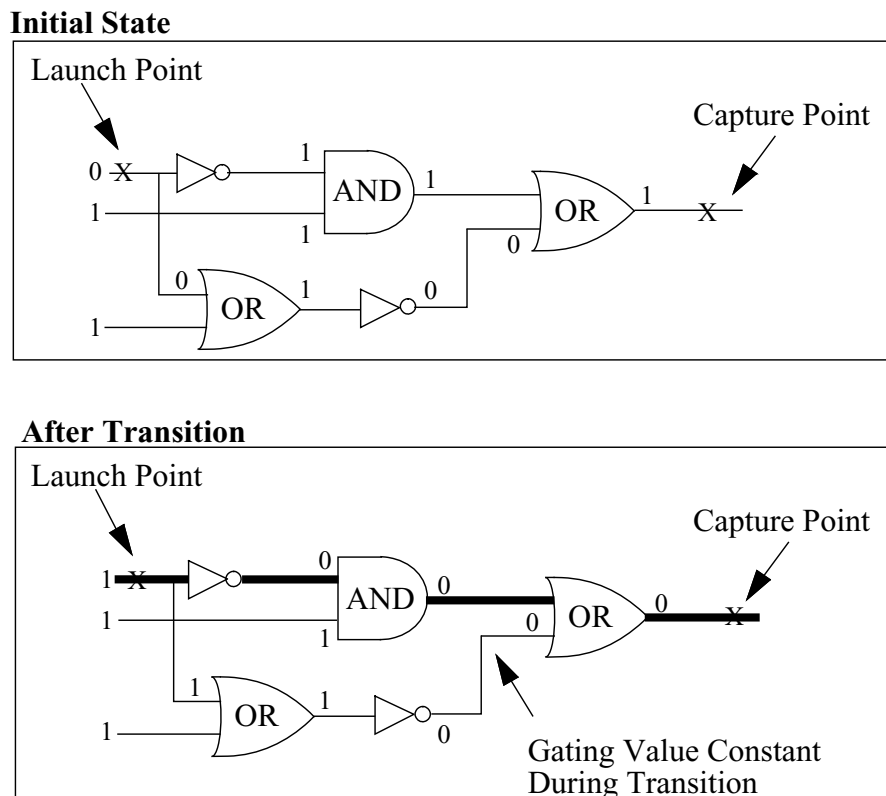
Path delay testing often requires greater depth than for stuck-at fault testing. The sequential depths that the tool calculates and reports are the minimums for stuck-at testing.

To get maximum benefit from path delay testing, the launch and capture events must have accurate timing. The timing for all other events is not critical.

The ATPG tool detects a path delay fault with either a *robust test*, a *non-robust test*, or a *functional test*. If you save a path delay pattern in ASCII format, the tool includes comments in the file that indicate which of these three types of detection the pattern uses. Robust detection occurs when the gating inputs used to sensitize the path are stable from the time of the launch event to the time of the capture event. Robust detection keeps the gating of the path constant during fault detection and thus, does not affect the path timing. Because it avoids any possible reconvergent timing effects, it is the most desirable type of detection and for that reason is the approach the ATPG tool tries first. The tool, however, cannot use robust detection on many paths because of its restrictive nature and if it is unable to create a robust test, it will automatically try to create a non-robust test. The application places faults detected by robust detection in the DR (det\_robust) fault class.

Figure 6-24 gives an example of robust detection for a rising-edge transition within a simple path. Notice that, due to the circuitry, the gating value at the second OR gate was able to retain the proper value for detection during the entire time from launch to capture events.

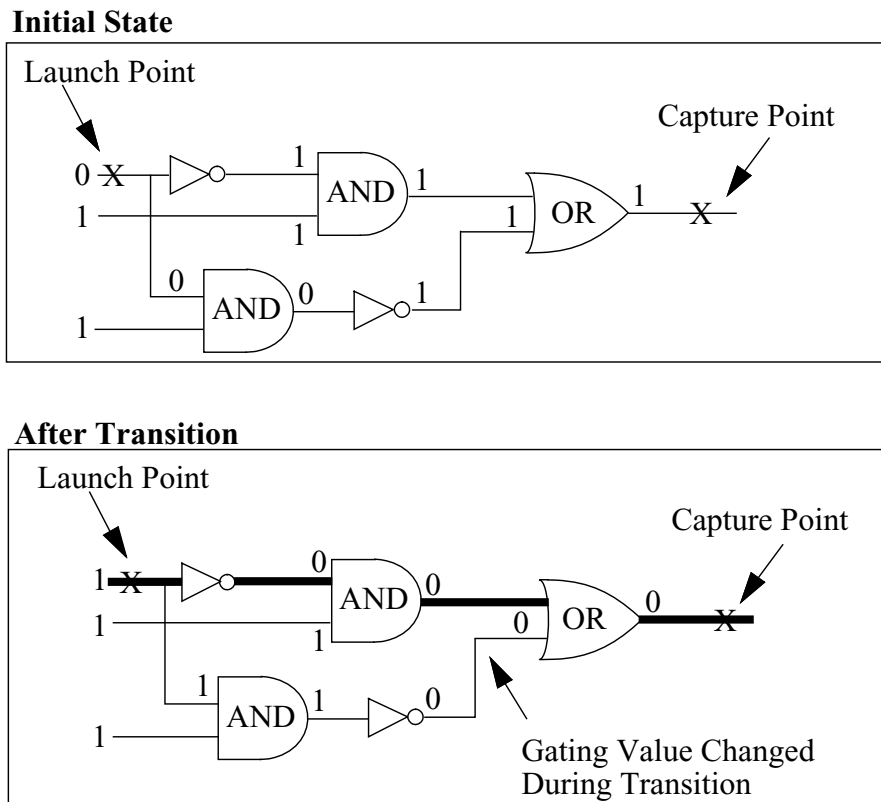
**Figure 6-24. Robust Detection Example**



Non-robust detection does not require constant values on the gating inputs used to sensitize the path. It only requires the proper gating values at the time of the capture event. The ATPG tool places faults detected by non-robust detection in the DS (det\_simulation) fault class.

Figure 6-25 gives an example of non-robust detection for a rising-edge transition within a simple path.

**Figure 6-25. Non-robust Detection Example**

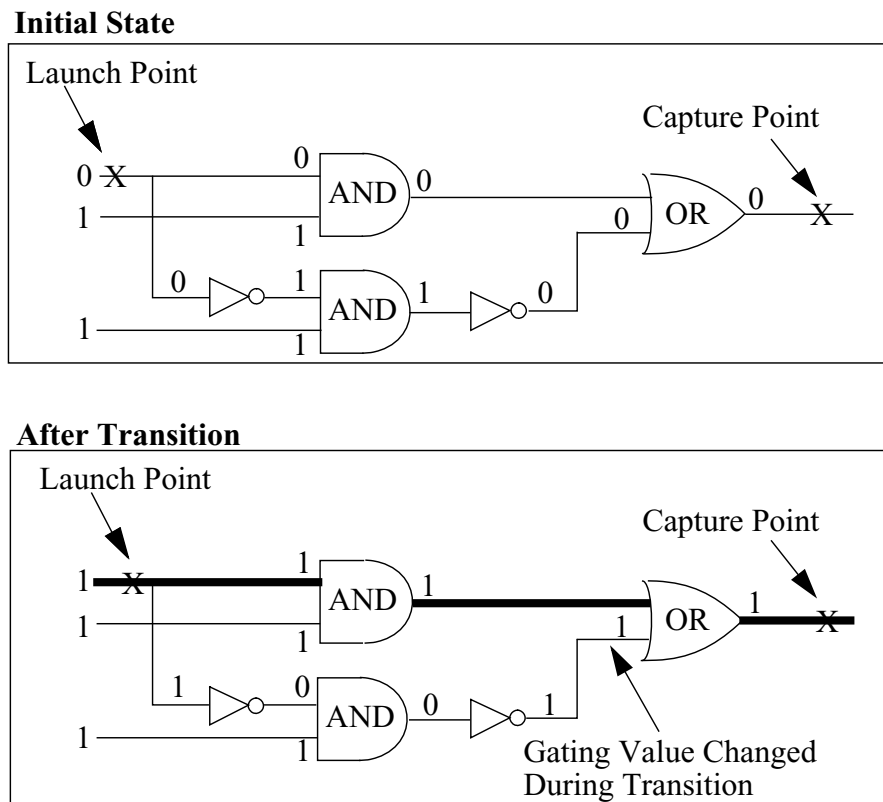


Notice that due to the circuitry, the gating value on the OR gate changed during the 0 to 1 transition placed at the launch point. Thus, the proper gating value was only at the OR gate at the capture event.

Functional detection further relaxes the requirements on the gating inputs used to sensitize the path. The gating of the path does not have to be stable as in robust detection, nor does it have to be sensitizing at the capture event, as required by non-robust detection. Functional detection requires only that the gating inputs not block propagation of a transition along the path. The tool places faults detected by functional detection in the det\_functional (DF) fault class.

Figure 6-26 gives an example of functional detection for a rising-edge transition within a simple path. Notice that, due to the circuitry, the gating (off-path) value on the OR gate is neither stable, nor sensitizing at the time of the capture event. However, the path input transition still propagates to the path output.

**Figure 6-26. Functional Detection Example**



- [add\\_ambiguous\\_paths](#) — Specifies the number of paths the tool should select when encountering an ambiguous path.
- [analyze\\_fault](#) — Analyzes a fault, including path delay faults, to determine why it was not detected.
- [delete\\_fault\\_sites](#) — Deletes paths from the internal path list.
- [read\\_fault\\_sites](#) — Loads in a file of path definitions from an external file.
- [report\\_fault\\_sites](#) — Reports information on paths in the path list.
- [report\\_statistics](#) — Displays simulation statistics, including the number of detected faults in each fault class.
- [set\\_pathdelay\\_holdpi](#) — Sets whether non-clock primary inputs can change after the first pattern force, during ATPG.
- [write\\_fault\\_sites](#) — Writes information on paths in the path list to an external file.



## The Path Definition File

In an external ASCII file, you must define all paths that you want tested in the test set. For each path, you must specify:

- **Path\_name** — A unique name you define to identify the path.
- **Path\_definition** — The topology of the path from launch to capture point as defined by an ordered list of pin pathnames. Each path must be unique.

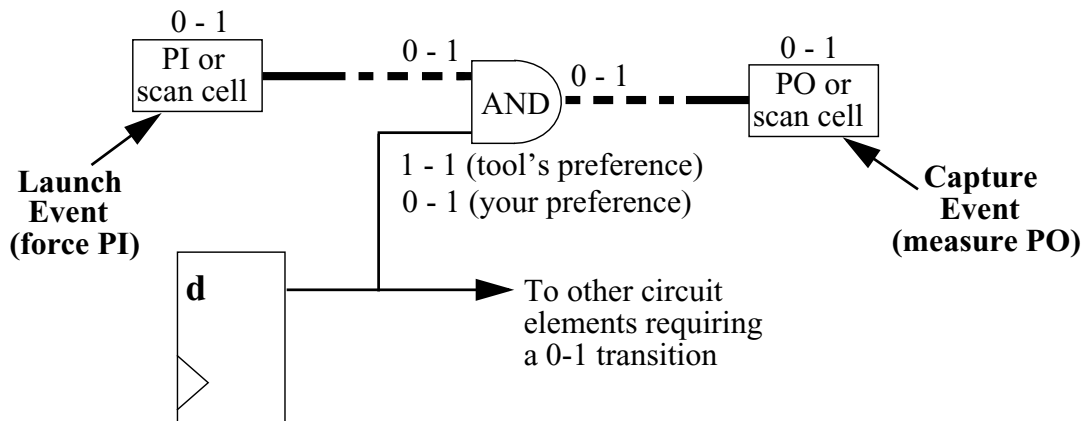
The ASCII path definition file has several syntax requirements. The tools ignore as a comment any line that begins with a double slash (//) or pound sign (#). Each statement must be on its own line. The four types of statements include:

- **Path** — A required statement that specifies the unique pathname of a path.
- **Condition** — An optional statement that specifies any conditions necessary for the launch and capture events. Each condition statement contains two arguments: a full pin pathname for either an internal or external pin, and a value for that pin. Valid pin values for condition statements are 0, 1, or Z. Condition statements must occur between the path statement and the first pin statement for the path.
- **Transition\_condition** — An optional statement that specifies additional transitions required in the test pattern. Each transition\_condition statement contains two arguments: a full pin pathname for either an internal or external pin and a direction. Transition\_condition statements must occur between the path statement and the first pin statement for the path.

The direction can be one of the following: rising, falling, same, or opposite. Rising and falling specify that a rising edge and falling edge, respectively, are required on the specified pin at the same time as launching a transition into the first pin of the path. Same specifies for the tool to create a transition in the same direction as the one on the first pin in the path definition. Opposite creates a transition in the opposite direction.

[Figure 6-27](#) shows an example where a transition\_condition statement could be advantageous.

**Figure 6-27. Example Use of Transition\_condition Statement.**



A defined path includes a 2-input AND gate with one input on the path, the other connected to the output of a scan cell. For a robust test, the AND gate's off-path or gating input needs a constant 1. The tool, in exercising its preference for a robust test, would try to create a pattern that achieved this. Suppose however that you wanted the circuit elements fed by the scan cell to receive a 0-1 transition. You could add a transition\_condition statement to the path definition, specifying a rising transition for the scan cell. The path capture point maintains a 0-1 transition, so remains testable with a non-robust test, and you also get the desired transition for the other circuit elements.

- Pin** — A required statement that identifies a pin in the path by its full pin pathname. Pin statements in a path must be ordered from launch point to capture point. A "+" or "-" after the pin pathname indicates the inversion of the pin with respect to the launch point. A "+" indicates no inversion (you want a transition identical to the launch transition on that pin), while a "-" indicates inversion (you want a transition opposite the launch transition).

#### Note



If you use "+" or "-" in any pin statement, you must include a "+" for the launch point. The polarity of the launch transition must always be "+".

You must specify a minimum of two pin statements, the first being a valid launch point (primary input or data output of a state element or RAM) and the last being a valid capture point (primary output, data or clk input of a state element, or data input of a RAM). The current pin must have a combinational connectivity path to the previous pin and the edge parity must be consistent with the path circuitry. If a statement violates either of these conditions, the tool issues an error. If the path has edge or path ambiguity, it issues a warning.

Paths can include state elements (through data or clock inputs), but you must explicitly name the data or clock pins in the path. If you do not, the tool does not recognize the path and issues a corresponding message.

- **End** — A required statement that signals the completion of data for the current path. Optionally, following the end statement, you can specify the name of the path. However, if the name does not match the pathname specified with the path statement, the tool issues an error.

The following shows the path definition syntax:

```
PATH <pathname> =  
    CONDITION <pin_pathname> <0|1|Z>;  
    TRANSITION_CONDITION <pin_pathname> <Rising|Falling|Same|Opposite>;  
    PIN <pin_pathname> [+|-];  
    PIN <pin_pathname> [+|-];  
    ...  
    PIN <pin_pathname> [+|-];  
END [pathname];
```

The following is an example of a path definition file:

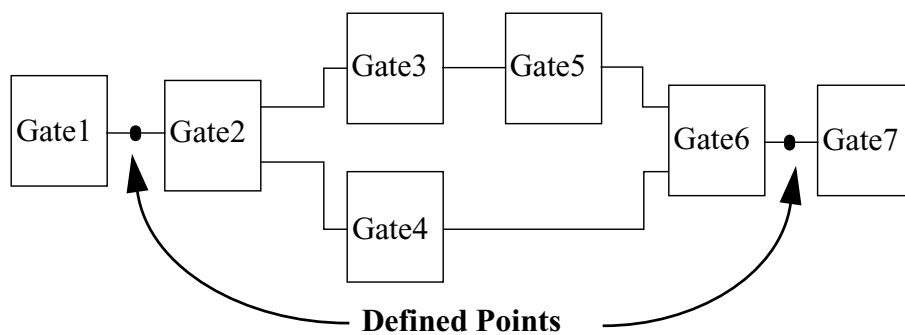
```
PATH "path0" =  
    PIN /I$6/Q + ;  
    PIN /I$35/B0 + ;  
    PIN /I$35/C0 + ;  
    PIN /I$1/I$650/IN + ;  
    PIN /I$1/I$650/OUT - ;  
    PIN /I$1/I$951/I$1/IN - ;  
    PIN /I$1/I$951/I$1/OUT + ;  
    PIN /A_EQ_B + ;  
END ;  
PATH "path1" =  
    PIN /I$6/Q + ;  
    PIN /I$35/B0 + ;  
    PIN /I$35/C0 + ;  
    PIN /I$1/I$650/IN + ;  
    PIN /I$1/I$650/OUT - ;  
    PIN /I$1/I$684/I1 - ;  
    PIN /I$1/I$684/OUT - ;  
    PIN /I$5/D - ;  
END ;  
PATH "path2" =  
    PIN /I$5/Q + ;  
    PIN /I$35/B1 + ;  
    PIN /I$35/C1 + ;  
    PIN /I$1/I$649/IN + ;  
    PIN /I$1/I$649/OUT - ;  
    PIN /I$1/I$622/I2 - ;  
    PIN /I$1/I$622/OUT - ;  
    PIN /A_EQ_B + ;  
END ;  
PATH "path3" =  
    PIN /I$5/QB + ;  
    PIN /I$6/TI + ;  
END ;
```

You use the `read_fault_sites` command to read in the path definition file. The tool loads the paths from this file into an internal path list. You can add to this list by adding paths to a new file and re-issuing the `read_fault_sites` command with the new filename.

## Path Definition Checking

The ATPG tool checks the points along the defined path for proper connectivity and to determine if the path is ambiguous. *Path ambiguity* indicates there are several different paths from one defined point to the next. [Figure 6-28](#) indicates a path definition that creates ambiguity.

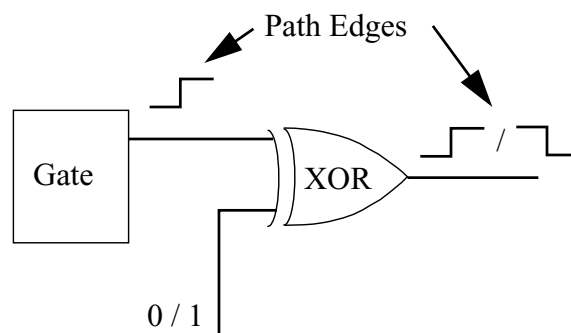
**Figure 6-28. Example of Ambiguous Path Definition**



In this example, the defined points are an input of Gate2 and an input of Gate7. Two paths exist between these points, thus creating path ambiguity. When the ATPG tool encounters this situation, it issues a warning message and selects a path, typically the first fanout of the ambiguity. If you want the tool to select more than one path, you can specify this with the `add_ambiguous_paths` command.

During path checking, the tool can also encounter *edge ambiguity*. Edge ambiguity occurs when a gate along the path has the ability to either keep or invert the path edge, depending on the value of another input of the gate. [Figure 6-29](#) shows a path with edge ambiguity due to the XOR gate in the path.

**Figure 6-29. Example of Ambiguous Path Edges**



The XOR gate in this path can act as an inverter or buffer of the input path edge, depending on the value at its other input. Thus, the edge at the output of the XOR is ambiguous. The path definition file lets you indicate edge relationships of the defined points in the path. You do this by specifying a “+” or “-” for each defined point, as was previously described in [“The Path Definition File”](#) on page 225.

## Basic Procedure for Generating a Path Delay Test Set

Use the following basic procedure to generate a path delay test set:

1. Perform circuit setup tasks.
2. Constrain the scan enable pin to its inactive state. For example:  

```
SETUP> add_input_constraints scan_en -c0
```
3. (Optional) Turn on output masking.  

```
SETUP> set_output_masks on
```
4. Add nofaults <x, y, z>
5. Set the sequential depth to two or greater:  

```
SETUP> set_pattern_type -sequential 2
```
6. Enter analysis system mode. This triggers the tool’s automatic design flattening and rules checking processes.
7. Set the fault type to path delay:  

```
ANALYSIS> set_fault_type path_delay
```
8. Write a path definition file with all the paths you want to test. [“The Path Definition File”](#) on page 225 describes this file in detail. If you want, you can do this prior to the session. You can only add faults based on the paths defined in this file.
9. Load the path definition file (assumed for the purpose of illustration to be named *path\_file\_1*):  

```
ANALYSIS> read_fault_sites path_file_1
```
10. Specify any ambiguous paths you want the tool to add to its internal path list. The following example specifies to add all ambiguous paths up to a maximum of 4.  

```
ANALYSIS> add_ambiguous_paths -all -max_paths 4
```
11. Define faults for the paths in the tool’s internal path list:  

```
ANALYSIS> add_faults -all
```

This adds a rising edge and falling edge fault to the tool’s path delay fault list for each defined path.

12. Perform an analysis on the specified paths and delete those the analysis proves are unsensitizable:

```
ANALYSIS> delete_fault_sites -unsensitizable_paths
```

13. Run test generation:

```
ANALYSIS> create_patterns
```

## Path Delay Testing Limitations

Path delay testing does not support the following:

- **RAMs Within a Specified Path** — A RAM as a launch point is supported only if the launch point is at the RAM's output. A RAM as a capture point is supported only if the capture point is at the RAM's input.
- **Paths Starting at a Combinationally Transparent Latch** — A combinational transparent latch as a capture point is supported only if the capture point is at the latch's input.
- **Path Starting and/or Ending at ROM** — You should model ROM as a read-only CRAM primitive (that is, without any `_write` operation) to enable the tool to support path delay testing starting and/or ending at ROM.

## At-Speed Test Using Named Capture Procedures

To create at-speed test patterns for designs with complicated clocking schemes, you may need to specify the actual launch and capture clocking sequences. For example, in an LSSD type design with master and slave clocks, the number and order of clock pulses might need to be organized in a specific way. You can do this using a named capture procedure in the test procedure file.

A *named capture procedure* is an optional procedure, with a unique name, used to define explicit clock cycles. Named capture procedures can be used for generating stuck-at, path delay, and broadside transition patterns, but not launch off shift transition patterns. You can create named capture procedures using the [create\\_capture\\_procedures](#) command and then write out the procedures using the [write\\_procfile](#) command. You can also manually create or edit named capture procedures using an external editor if needed. For information on manually creating and editing named capture procedures, see the “[Rules for Creating and Editing Named Capture Procedures](#)” section in the *Tessent Shell User's Manual*.

When the test procedure file contains named capture procedures, the ATPG tool generates patterns that conform to the waveforms described by those procedures. Alternatively, you can use the [set\\_capture\\_procedures](#) command to disable a subset of the named capture procedures, and only the enabled subset is used. For example, you might want to exclude named capture procedures that are unable to detect certain types of faults during test pattern generation.

You can have multiple named capture procedures within one test procedure file in addition to the default capture procedure the file typically contains. Each named capture procedure must reflect clock behavior that the clocking circuitry is actually capable of producing. When you use a named capture procedure to define a waveform, it is assumed you have expert design knowledge; the ATPG tool does not verify that the clocking circuitry is capable of delivering the waveform to the defined internal pins.

The ATPG tool uses either all named capture procedures (the default) or only those named capture procedures you enable with the [set\\_capture\\_procedures](#) command. When the test procedure file does not contain named capture procedures, or you use the “set capture procedure off -all” command, the tool uses the default capture procedure. However, usually you would not use the default procedure to generate at-speed tests. The tool does not currently support use of both named capture procedures and clock procedures in a single ATPG session.

---

**Note**

If a DRC error prevents use of a capture procedure, the run will abort.

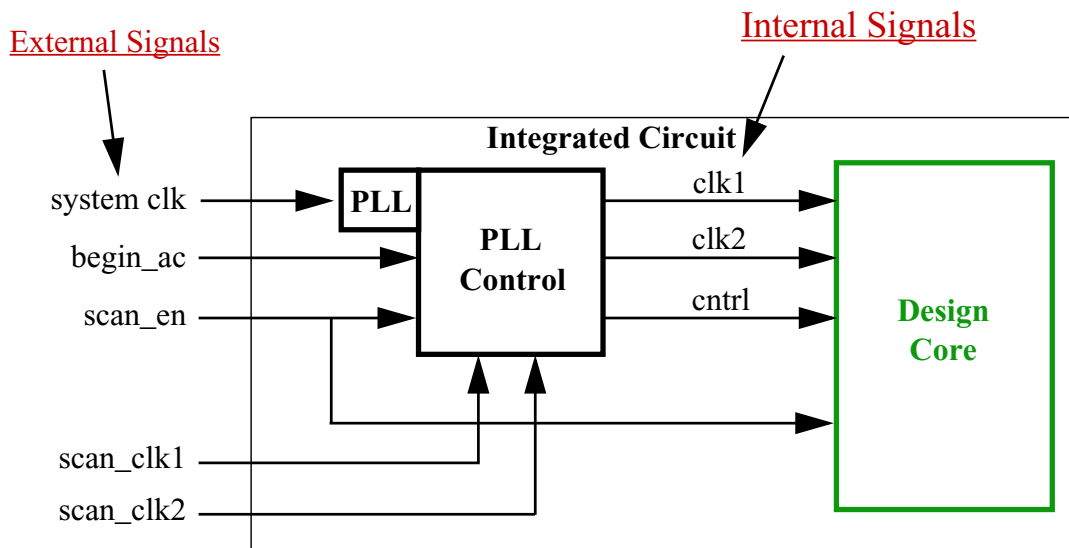
---

For more information on named capture procedures, see the “[Non-Scan Procedures](#)” section in the *Tessent Shell User’s Manual*.

## Support for On-Chip Clocks (PLLs)

One way you can use named capture procedures is for the support of on-chip or internal clocks. These are clocks generated on-chip by a phased-locked loop (PLL) or other clock generating circuitry as shown in [Figure 6-30](#). In addition, an example timing diagram for this circuit is shown in [Figure 6-31](#). A PLL can support only certain clock waveforms and named capture procedures let you specify the allowed set of clock waveforms. In this case, if there are multiple named capture procedures, the ATPG engine will use these named capture procedures instead of assuming the default capture behavior.

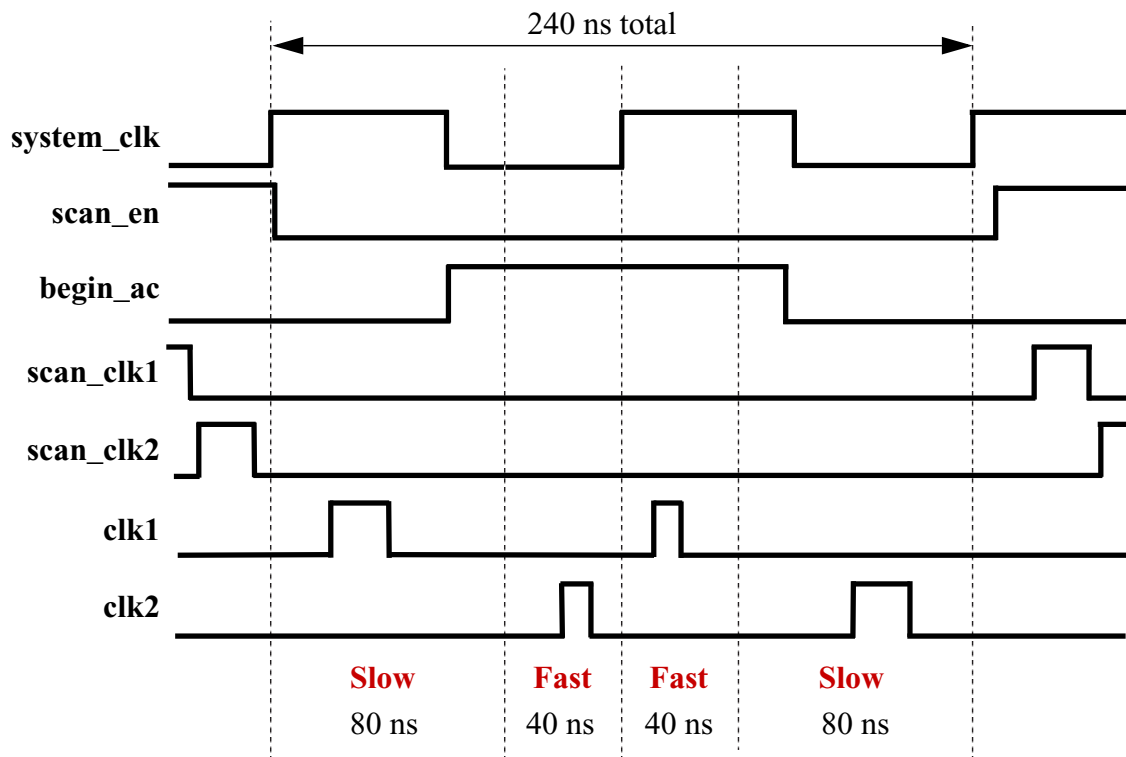
**Figure 6-30. On-chip Clock Generation**



## Defining Internal and External Modes

When manually creating or editing named capture procedures, you can use the optional keyword “mode” with two mode blocks, “internal” and “external” to describe what happens on the internal and external sides of an on-chip phase-locked loop (PLL) or other on-chip clock-generating circuitry. You use “mode internal =” and “mode external =” to define mode blocks in which you put procedures to exercise internal and external signals. You must use the internal and external modes together, and ensure no cycles are defined outside the mode definitions.



**Figure 6-31. PLL-Generated Clock and Control Signals**

The internal mode is used to describe what happens on the internal side of the on-chip PLL control logic, while the external mode is used to describe what happens on the external side of the on-chip PLL. [Figure 6-30](#) shows how this might look. The internal mode uses the internal clocks (/pll/clk1 & /pll/clk2) and signals while the external mode uses the external clocks (system\_clk) and signals (begin\_ac & scan\_en). If any external clocks or signals go to both the PLL and to other internal chip circuitry (scan\_en), you need to specify their behavior in both modes and they need to match, as shown in the following example (timing is from [Figure 6-31](#)):

```
timeplate tp_cap_clk_slow =
  force_pi 0;
  pulse /pll/clk1 20 20;
  pulse /pll/clk2 40 20;
  period 80;
end;
timeplate tp_cap_clk_fast =
  force_pi 0;
  pulse /pll/clk1 10 10;
  pulse /pll/clk2 20 10;
  period 40;
end;

timeplate tp_ext =
  force_pi 0;
  measure_po 10;
  force begin_ac 60;
  pulse system_clk 0 60;
  period 120;
```

```
end;

procedure capture clk1 =
    observe_method master;

    mode internal =
        cycle slow =
            timeplate tp_cap_clk_slow;
            force system_clk 0;
            force scan_clk1 0;
            force scan_clk2 0;
            force scan_en 0;
            force_pi;
            force /pll/clk1 0;
            force /pll/clk2 0;
            pulse /pll/clk1;
        end;
        // launch cycle
        cycle =
            timeplate tp_cap_clk_fast;
            pulse /pll/clk2;
        end;
        // capture cycle
        cycle =
            timeplate tp_cap_clk_fast;
            pulse /pll/clk1;
        end;
        cycle slow =
            timeplate tp_cap_clk_slow;
            pulse /pll/clk2;
        end;
    end;

    mode external =
        timeplate tp_ext;
        cycle =
            force system_clk 0;
            force scan_clk1 0;
            force scan_clk2 0;
            force scan_en 0;
            force_pi;
            force begin_ac 1;
            pulse system_clk;
        end;
        cycle =
            force begin_ac 0;
            pulse system_clk;
        end;
    end;
end;
```

For more information about internal and external modes, see the “[Rules for Creating and Editing Named Capture Procedures](#)” section in the *Tessent Shell User’s Manual*.

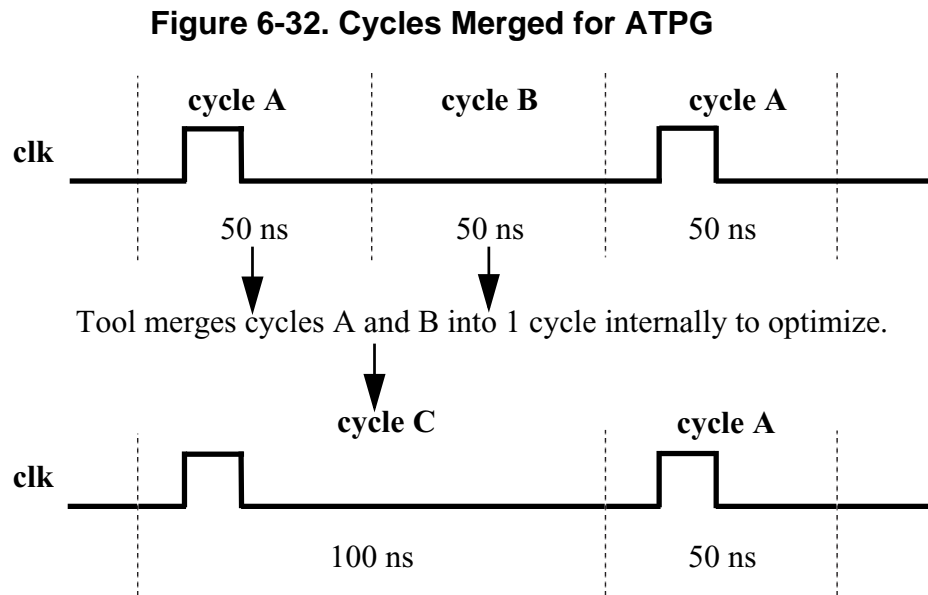
## Displaying Named Capture Procedures

When the ATPG tool uses a named capture procedure, it uses a “cyclized” translation of the internal mode. The tool may merge certain internal mode cycles in order to optimize them, and it may expand others to ensure correct simulation results. These modifications are internal only; the tool does not alter the named capture procedure in the test procedure file.

You can use the [report\\_capture\\_procedures](#) command to display the cyclized procedure information with annotations that indicate the timing of the cycles and where the at-speed sequences begin and end. If you want to view the procedures in their unaltered form in the test procedure file, use the [report\\_procedures](#) command.

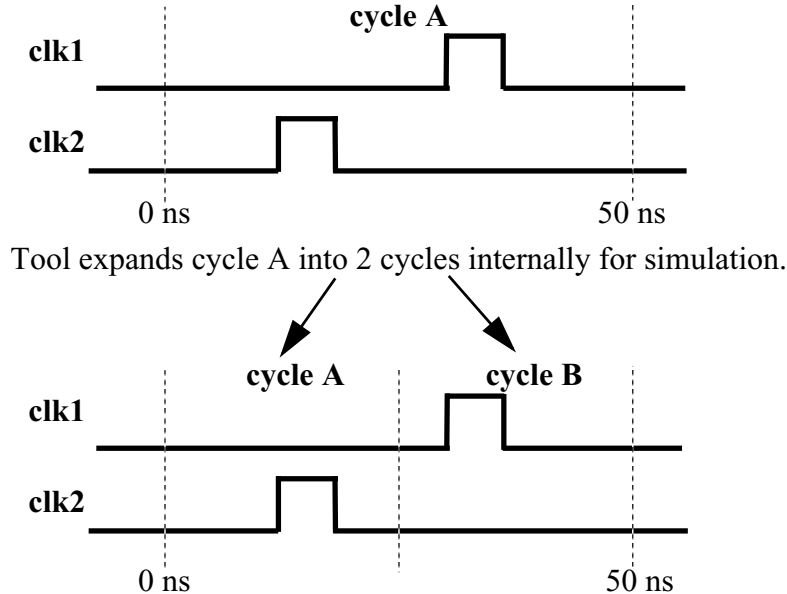
After cyclizing the internal mode information, the tool automatically adjusts the sequential depth to match the number of cycles that resulted from the cyclizing process. Patterns will automatically reflect any sequential depth adjustment the tool performs.

[Figure 6-32](#) illustrates cycle merging.



[Figure 6-33](#) illustrates cycle expansion.

**Figure 6-33. Cycles Expanded for ATPG**



## Achieving the Test Coverage Goal Using Named Capture Procedures

Use the following steps when you know the repeated clock sequences to which the test clocks will be applied and want to investigate the best sequential depth of each clock domain to achieve the test coverage goal.

1. Use the [create\\_capture\\_procedures](#) command to define the minimum clock sequences in a named capture procedure.
2. Gradually add more capture procedures with higher sequential depth until the test coverage goal is achieved or the pattern count limit is reached.

## Debugging Low Test Coverage Using Pre-Defined Named Capture Procedures

Use the following steps when you want to know which clock sequences will provide the best test coverage and then create a named capture procedure from it.

1. Start ATPG from the default capture procedure.
2. Use the [create\\_capture\\_procedures](#) command to create named capture procedures by extracting the most often applied clock sequences from the pattern set.
3. Use the [write\\_procfile](#) command to write the named capture procedure into the test procedure file.

---

**Note**

You may need to manually edit the named capture procedure in the test procedure file to achieve the functionality you want. For example, you may need to add condition statements or add external mode definitions. For information on rules to follow when editing named capture procedures, see the “[Rules for Creating and Editing Named Capture Procedures](#)” section in the *Tessent Shell User’s Manual*.

---

## Clocking During At-speed Fault Simulation

Not all clocks specified in the capture procedures are applied at-speed. During at-speed fault simulation, the tool does not activate at-speed related faults when slow clock sequences are fault simulated, even if a transition occurs in two consecutive cycles. Generally, the clock sequence defined in a capture procedure can consist of zero or more slow clock sequences, followed by zero or more at-speed clock sequences, followed by zero or more slow clock sequences.

## Internal Signals and Clocks

For clocks and signals that come out of the PLL or clock generating circuitry that are not available at the real I/O interface of the design, you use the `-Internal` switch with the [add\\_clocks](#) or [add\\_primary\\_inputs](#) commands to define the internal signals and clocks for use in ATPG.

For example, when setting up for pattern generation for the example circuit shown in [Figure 6-30](#), you would issue this command to define the internal clocks:

```
SETUP> add_clocks 0 /pll/clk1 /pll/clk2 -internal
```

The two PLL clocks would then be available to the tool’s ATPG engine for pattern generation.

## Saving Internal and External Patterns

By default, the ATPG tool uses only the primary input clocks when creating test patterns. However, if you use named capture procedures with internal mode clocks and control signals you define with the [add\\_clocks](#) `-Internal` or [add\\_primary\\_inputs](#) `-Internal` command, the tool uses those internal clocks and signals for pattern generation and simulation. To save the patterns using the same internal clocks and signals, you must use the `-Mode_internal` switch with the [write\\_patterns](#) command. The `-Mode_internal` switch is the default when saving patterns in ASCII or binary format.

---

**Note**

The `-Mode_internal` switch is also necessary if you want patterns to include internal pin events specified in scan procedures (`test_setup`, `shift`, `load_unload`).

---

To obtain pattern sets that can run on a tester, you need to write patterns that contain only the true primary inputs to the chip. These are the clocks and signals used in the external mode of any named capture procedures, not the internal mode. To accomplish this, you must use the

-Mode\_external switch with the [write\\_patterns](#) command. This switch directs the tool to map the information contained in the internal mode blocks back to the external signals and clocks that comprise the I/O of the chip. The -Mode\_external switch is the default when saving patterns in a tester format (for example, WGL) and Verilog format.

---

**Note**



The -Mode\_external switch ignores internal pin events in scan procedures (test\_setup, shift, load\_unload).

---

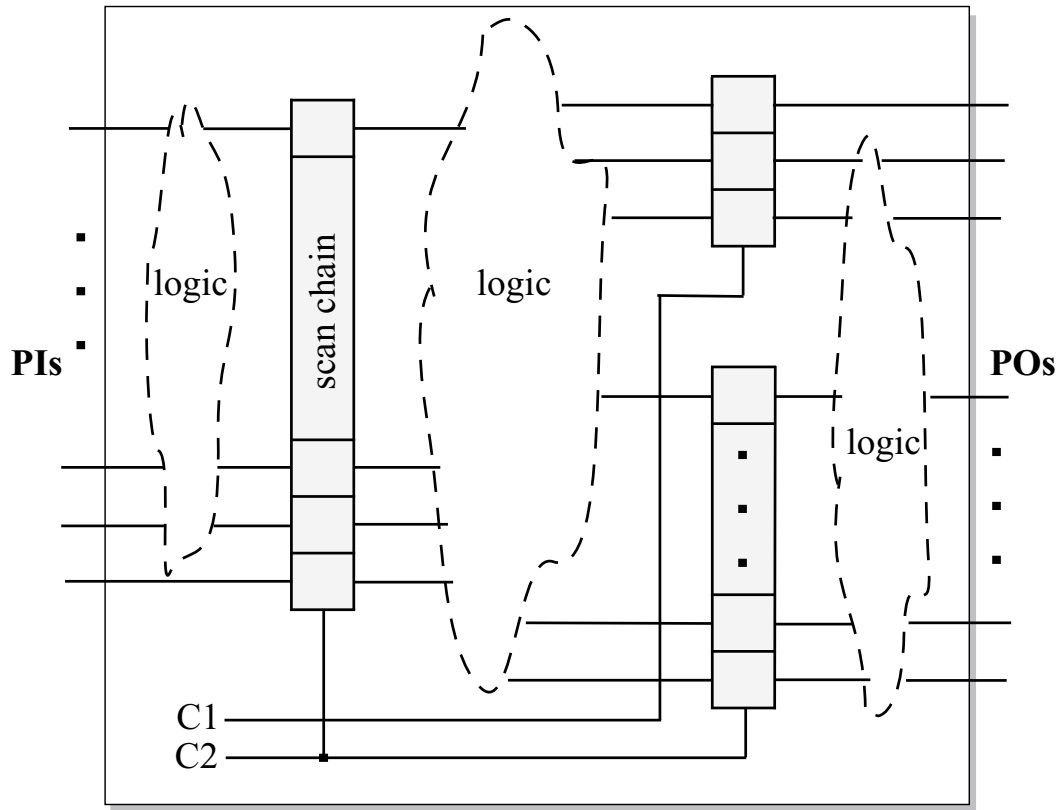
## Mux-DFF Example

In a full scan design, the vast majority of transition faults are between scan cells (or cell to cell) in the design. There are also some faults between the PI to cells and cells to the PO. Targeting these latter faults can be more complicated, mostly because running these test patterns on the tester can be challenging. For example, the tester performance or timing resolution at regular I/O pins may not be as good as that for clock pins. This section shows a mux-DFF type scan design example and covers some of the issues regarding creating transition patterns for the faults in these three areas.

[Figure 6-34](#) shows a conceptual model of an example chip design. There are two clocks in this mux-DFF design, which increases the possible number of launch and capture combinations in creating transition patterns. For example, depending on how the design is actually put together, there might be faults that require these launch and capture combinations: C1-C1, C2-C2, C1-C2, and C2-C1. The clocks may be either external or are created by some on-chip clock generator circuitry or PLL.

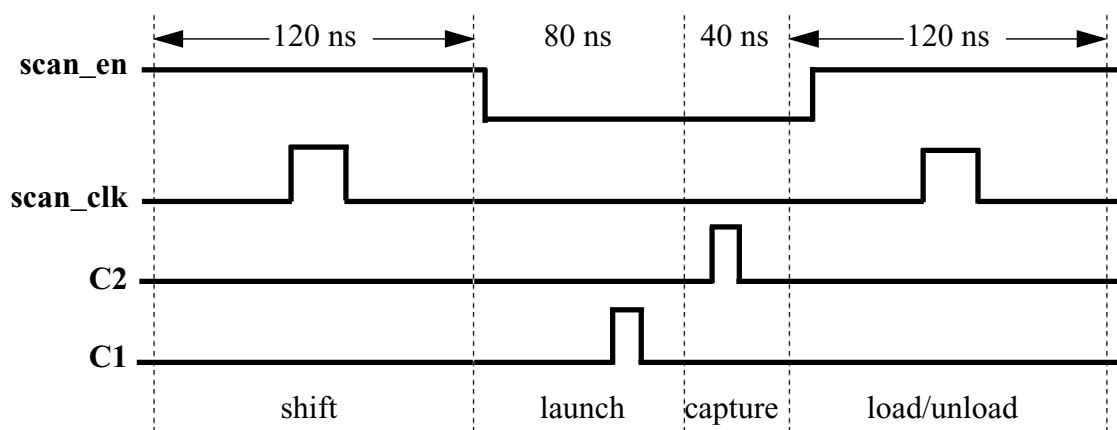
“[Timing for Transition Delay Tests](#)” on page 211 shows the basic waveforms and partial test procedure files for creating broadside and launch off shift transition patterns. For this example, named capture procedures are used to specify the timing and sequence of events. The example focuses on broadside patterns and shows only some of the possible named capture procedures that might be used in this kind of design.

**Figure 6-34. Mux-DFF Example Design**



A timing diagram for cell to cell broadside transition faults that are launched by clock C1 and captured by clock C2 is shown in [Figure 6-35](#).

**Figure 6-35. Mux-DFF Broadside Timing, Cell to Cell**



Following is the capture procedure for a matching test procedure file that uses a named capture procedure to accomplish the clocking sequence. Other clocking combinations would be handled with additional named capture procedures that pulse the clocks in the correct sequences.

```
set time scale 1.000000 ns ;
timeplate tp1 =
    force_pi 0;
    measure_po 10;
    pulse scan_clk 50 20;
    period 120;
end;
timeplate tp2 =
    force_pi 0;
    pulse c1 10 10;
    pulse c2 10 10;
    measure_po 30;
    period 40;
end;
timeplate tp3 =
    force_pi 0;
    pulse c1 50 10;
    pulse c2 10 10;
    period 80;
end;
procedure load_unload =
    timeplate tp1;
    cycle =
        force c1 0;
        force c2 0;
        force scan_en 1;
    end;
    apply shift 255;
end;
procedure shift =
    timeplate tp1;
    cycle =
        force_sci;
        measure_sco;
        pulse scan_clk;
    end;
end;
procedure capture launch_c1_cap_c2 =
    cycle =
        timeplate tp3;
        force c1 0;
        force c2 0;
        force scan_clk 0;
        force_pi; //force scan_en to 0
        pulse c1; //launch clock
    end;
    cycle =
        timeplate tp2;
        pulse c2; //capture clock
    end;
end;
```

Be aware that this is just one example and your implementation may vary depending on your design and tester. For example, if your design can turn off scan\_en quickly and have it settle before the launch clock is pulsed, you may be able to shorten the launch cycle to use a shorter

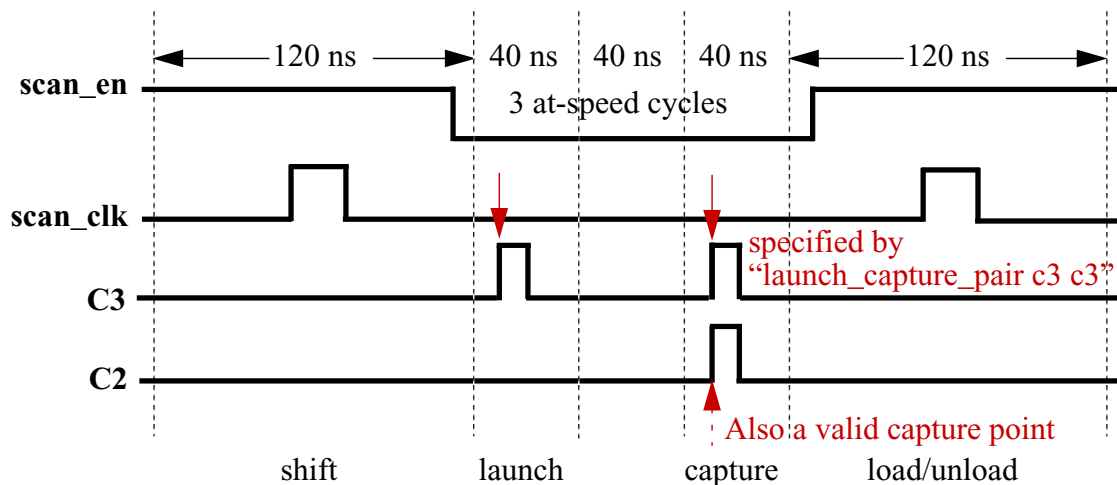


period; that is, the first cycle in the launch\_c1\_cap\_c2 capture procedure could be switched from using timeplate tp3 to using timeplate tp2.

Another way to make sure scan enable is turned off well before the launch clock is to add a cycle to the load\_unload procedure right after the “apply shift” line. This cycle would only need to include the statement, “force scan\_en 0;”.

Notice that the launch and capture clocks shown in Figure 6-35 pulse in adjacent cycles. The tool can also use clocks that pulse in non-adjacent cycles, as shown in Figure 6-36, if the intervening cycles are at-speed cycles.

**Figure 6-36. Broadside Timing, Clock Pulses in Non-adjacent cycles**



To define a pair of nonadjacent clocks for the tool to use as the launch clock and capture clock, include a “launch\_capture\_pair” statement at the beginning of the named capture procedure. Multiple “launch\_capture\_pair” statements are permitted, but the tool will use just one of the statements for a given fault. Without this statement, the tool defaults to using adjacent clocks.

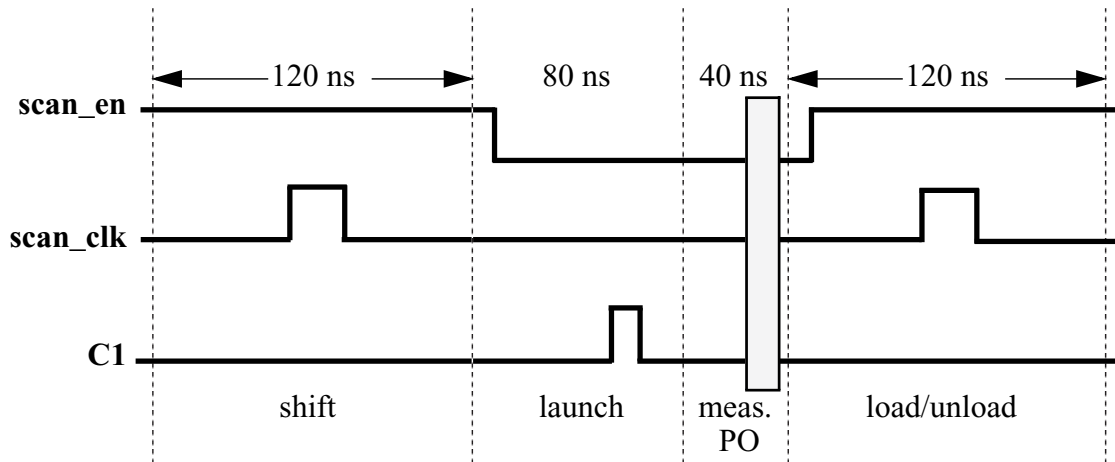
When its choice of a launch and capture clock is guided by a launch\_capture\_pair statement, the tool may use for launch, the clock specified as the launch clock in the statement or another clock that is pulsed between the launch and capture clocks specified in the statement. The capture clock, however, will be the one specified in the statement or another clock that has the *same period* as the specified capture clock.

If a named capture procedure for example pulses clocks clk1, clk2 and clk3 in that order in each of three successive at-speed cycles and the launch\_capture\_pair {clk1, clk3} is defined, the tool could use *either* clk1 or clk2 to launch and clk3 to capture. The idea of the launch and capture pair is that it allows you to specify the capture clock and the farthest launch clock from the capture clock. In this example, the {clk1, clk3} pair directs the tool to use clk3 to capture and the farthest launch clock to be clk1. The tool considers it all right for clk2 to launch since if {clk1, clk3} is at speed, {clk2, clk3} should be at speed as well.

For more information on using the “launch\_capture\_pair” statement, see the “[launch\\_capture\\_pair Statement](#)” section in the *Tessent Shell User’s Manual*.

If you want to try to create transition patterns for faults between the scan cells and the primary outputs, make sure your tester can accurately measure the PO pins with adequate resolution. In this scenario, the timing looks similar to that shown in [Figure 6-35](#) except that there is no capture clock. [Figure 6-37](#) shows the timing diagram for these cell to PO patterns.

**Figure 6-37. Mux-DFF Cell to PO Timing**



Following is the additional capture procedure that is required:

```
procedure capture launch_c1_meas_PO=
  cycle =
    timeplate tp3;
    force_pi; //force scan_en to 0
    pulse c1; //launch clock
  end;
  cycle =
    timeplate tp2;
    measure_po; //measure PO values
  end;
end;
```

#### Note



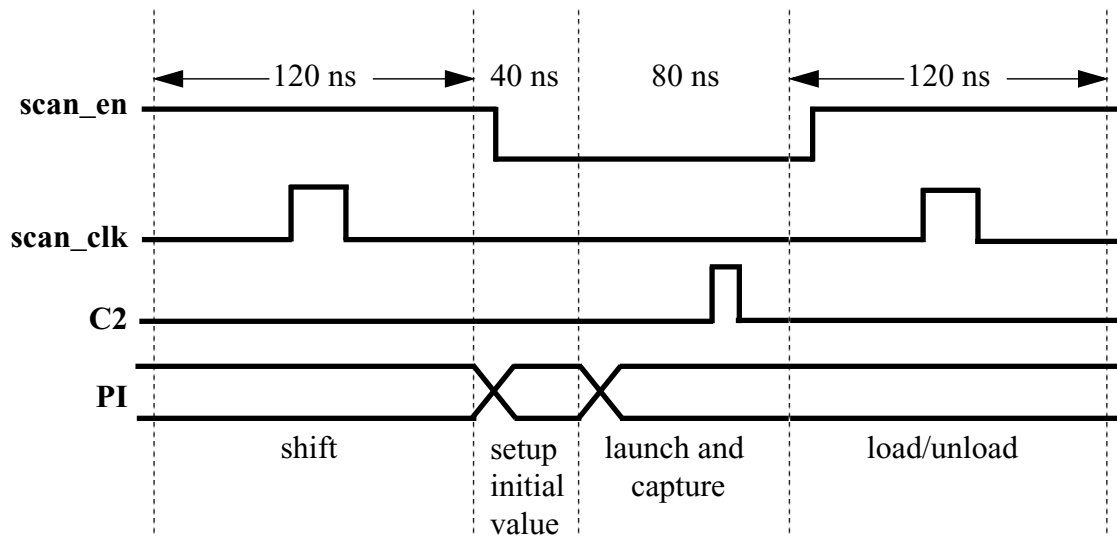
You will need a separate named capture procedure for each clock in the design that can cause a launch event.

What you specify in named capture procedures is what you get. As you can see in the two preceding named capture procedures (launch\_c1\_cap\_c2 and launch\_c1\_meas\_PO), both procedures used two cycles, with timeplate tp3 followed by timeplate tp2. The difference is that in the first case (cell to cell), the second cycle only performed a pulse of C2 while in the second case (cell to PO), the second cycle performed a measure\_po. The key point to remember is that

even though both cycles used the same timeplate, they only used a subset of what was specified in the timeplate.

To create effective transition patterns for faults between the PI and scan cells, you also may have restrictions due to tester performance and tolerance. One way to create these patterns can be found in the example timing diagram in [Figure 6-38](#). The corresponding named capture procedure is shown after the figure.

**Figure 6-38. Mux-DFF PI to Cell Timing**



```

procedure capture launch_PI_cap_C2 =
  cycle =
    timeplate tp2;
    force_pi;    //force initial values
  end;
  cycle =
    timeplate tp3;
    force_pi;    //force updated values
    pulse c2;    //capture clock
  end;
end;

```

As before, you would need other named capture procedures for capturing with other clocks in the design. This example shows the very basic PI to cell situation where you first set up the initial PI values with a force, then in the next cycle force changed values on the PI and quickly capture them into the scan cells with a capture clock.

#### Note



You do not need to perform at-speed testing for all possible faults in the design. You can eliminate testing things like the boundary scan logic, the memory BIST, and the scan shift path by using the [add\\_nofaults](#) command.

## Support for Internal Clock Control

You can use clock control definitions in your test procedure file to specify the operation of on-chip/internal clocks during capture cycles. A clock control definition is one or more blocks in the test procedure file that define internal clock operation by specifying source clocks and conditions at scan cell outputs when a clock can be pulsed. ATPG interprets these definitions and determines which clock to pulse during the capture cycle to detect the most faults.

When a clock control is defined, the clock control bits are included in chain test pattern to turn off capture clocks when the clock under control has no source clock or when the source clock defined in the clock under control is a free running clock.

By default in the ATPG tool, one capture cycle will be included in chain test patterns if there is no free running clock or free running clock drives neither observation points nor buses.

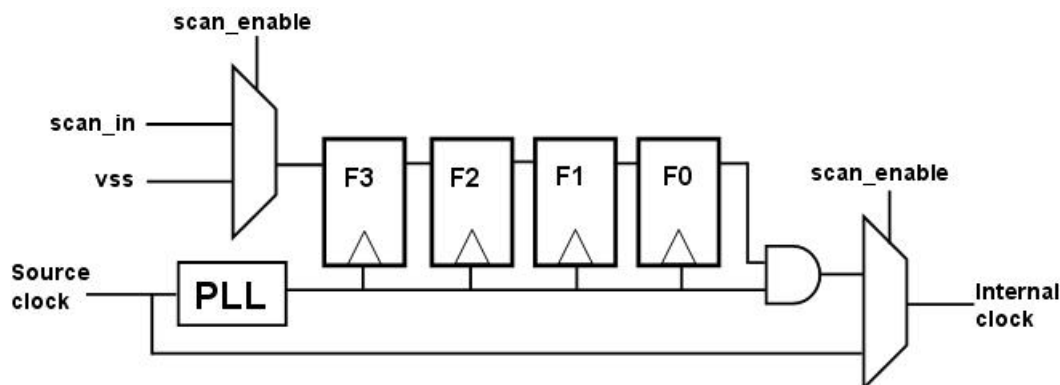
You can manually create clock control definitions or use the stil2mgc tool to generate them automatically from a STIL Procedure File (SPF) that contains a ClockStructures block.

To specify explicit launch/capture sequences or external/internal clock relationship definitions, you must use Named Capture Procedures (NCPs).

## Per-Cycle Clock Control

Per-cycle clock control allows you to define the internal clock output based on a single capture cycle using scan cell values. Per-cycle clock control is commonly used for pipeline-based clock generation where one bit controls the clock in each cycle. [Figure 6-39](#) shows a simplified per-cycle clock control model.

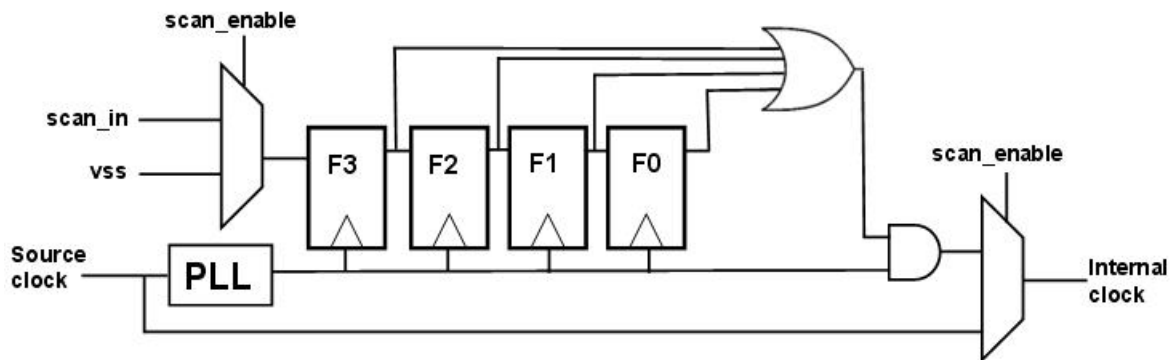
**Figure 6-39. Simplified Per-Cycle Clock Control Model**



## Sequence Clock Control

Sequence clock control allows you to define the internal clock output based on a sequence of capture cycles using scan cell values. Sequence clock control is commonly used for counter-based clock generation circuitry. [Figure 6-40](#) shows a simplified sequence clock control model.

Figure 6-40. Simplified Sequence Clock Control Model



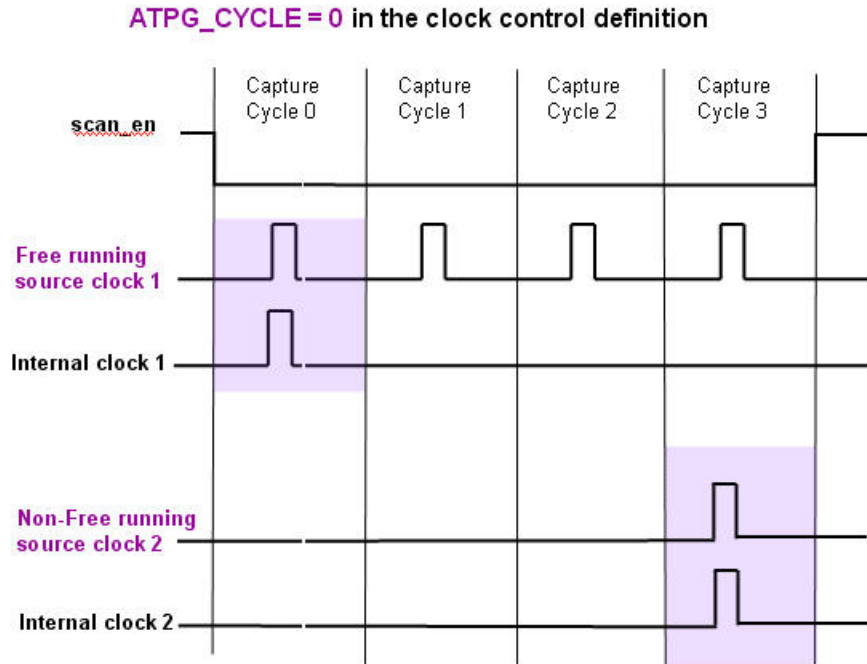
## Capture Cycle Determination

The actual capture cycle is relative to how many times the source clock pulses between scan loads as follows.

**If the source clock is free-running**, the specified capture cycle and the actual capture cycle are always the same because the source clock pulses in every ATPG cycle as shown in [Figure 6-41](#). No source clock or an-always-capture clock are considered equivalent to a free-running source clock.

**If the source clock is defined as anything but free running or always capture**, the specified capture cycle is determined by the pulse of the source clock. For example, an internal clock defined to pulse for cycle 0 may not pulse in cycle 0 but in the cycle that corresponds to the source clock pulse as shown in [Figure 6-41](#).

**Figure 6-41. Clock Control Capture Cycles**



## Applying Internal Clock Control

Use this procedure to apply internal clock control for ATPG via the test procedure file.

### Limitations

- Clock control definitions and Named Capture Procedures (NCPs) cannot both be enabled during test pattern generation. By default, clock control definitions are disabled when NCPs are enabled.

#### Note



Test patterns created with NCPs and test patterns created with the clock control definitions can be fault simulated simultaneously.

### Procedure

1. Depending on your application, either:
  - Create clock control definitions in your test procedure file.
  - OR
  - Run the stil2mgc tool on an SPF to generate a test procedure file with clock control definitions. Refer to the [stil2mgc](#) description in the *Tessent Shell Reference Manual*.

For more information, see “[Clock Control Definition](#)” in the *Tessent Shell User’s Manual*.

2. Invoke Tessent Shell, set the context to “patterns -scan,” read in the netlist, and set up the appropriate parameters for ATPG run.

3. Load the test procedure file created in step 1 and set up for ATPG. For example:

```
SETUP> add_scan_groups group1 scan_g1.procfile
SETUP> add_scan_chains chain1 group1 indata2 testout2
SETUP> add_scan_chains chain2 group1 indata4 testout4
SETUP> add_clocks 0 clk1 -internal
```

The [add\\_scan\\_groups](#) command loads the specified test procedure file from setup mode.

You can also use the [read\\_procfile](#) command load the test procedure file from analysis mode. For more information, see “[Test Pattern Formatting and Timing](#).”

4. Exit setup mode and run DRC. For example:

```
SETUP> set_system_mode analysis
```

5. Correct any DRC violations. For information on clock control definition DRCs, see “[Procedure Rules \(P Rules\)](#)” in the *Tessent Shell Reference Manual*.

Clock control definitions are enabled by default unless there are NCPs enabled in the test procedure file. If NCPs exist and are enabled, they override clock control definitions.

6. Report the clock control configurations. For example:

```
ANALYSIS> report_clock_controls

CLOCK_CONTROL "/top/core/clk1 (3457)" =
  SOURCE_CLOCK "/pll_clk (4)";
  ATPG_CYCLE 0 =
    CONDITION "/ctl_dff2/ (56)" 1;
  END;
  ATPG_CYCLE 1 =
    CONDITION "/ctl_dff1/ (55)" 1;
  END;
END;
```

Values in parenthesis are tool-assigned gate ID numbers.

7. Generate test patterns. For example:

```
ANALYSIS> create_patterns
```

8. Save test patterns. For example:

```
ANALYSIS> write_patterns ../generated/patterns_edt_p.stil -stil -replace
```

## Generating Test Patterns for Different Fault Models and Fault Grading

Use this procedure when you want to create test patterns for path delay, transition, stuck-at, and bridge fault models and fault grade the test patterns to improve fault coverage.

### Note



You can use N-detect for stuck-at and transition patterns. If you use N-detect, replace the stuck-at and/or transition patterns described in the procedure with the N-detect patterns.

---

### Prerequisites

- A test procedure file must be available. See [“Test Procedure File”](#) in the *Tessent Shell User’s Manual*.
- A path definition file must be available. See [“The Path Definition File”](#) on page 225.
- A bridge definition file must be available. See [“Net Pair Identification with Calibre for Bridge Fault Test Patterns”](#) on page 200.

### Procedure

1. Create path delay test patterns for your critical path(s) and save them to a file. Then fault grade the path delay test patterns for transition fault coverage. Following are example commands in a dofile.

```
//-----Create path delay patterns-----  
// Enable two functional pulses (launch and capture).  
set_fault_type path_delay  
read_fault_sites my_critical_paths  
report_fault_sites path0  
// Uncomment next 2 lines to display path in DFTVisualizer.  
// set_gate_level primitive  
// report fault sites path0 -display debug  
create_patterns  
// Save path delay patterns.  
write_patterns pathdelay_pat.bin.gz -binary -replace  
//-----  
//-----Grade for broadside transition fault coverage-----  
// Change the fault model (when you change the fault model, the  
// the internal pattern set database is emptied).  
set_fault_type transition -no_shift_launch  
add_faults -all  
// Read the previously saved path delay patterns into the external  
// pattern set database; include the -All_patterns switch so the  
// patterns are copied to the now empty internal pattern set  
// database when they are simulated.  
read_patterns pathdelay_pat.bin.gz  
// Simulate all the path delay patterns for transition fault  
// coverage, copying them into the internal pattern set as they  
// are simulated.  
simulate_patterns -store_patterns all
```



```
report_statistics
//-----
```

2. Create additional transition test patterns for any remaining transition faults, and add these test patterns to the original test pattern set. Then fault grade the enlarged test pattern set for stuck-at fault coverage. Following are example commands in a dofile.

```
//-----Create add'l transition fault patterns-----
// Create transition patterns that detect the remaining transition
// faults the path delay patterns did not detect during
// simulation.
create_patterns
order_patterns 3 // optimize the pattern set
// Save original path delay patterns and add'l transition patterns.
write_patterns pathdelay_trans_pat.bin.gz -binary -replace
//-----
//-----Grade for stuck-at fault coverage-----
set_fault_type stuck
add_faults -all
// Read in previously saved path delay and transition patterns and
// add them to the internal pattern set when they are simulated.
read_patterns pathdelay_trans_pat.bin.gz -all_patterns
simulate_patterns -store_patterns all
report_statistics
//-----
```

3. Create additional stuck-at test patterns for any remaining stuck-at faults and add them to the test pattern set. Then fault grade the enlarged test pattern set for bridge fault coverage. Following are example commands in a dofile.

```
//-----Create add'l (top-up) stuck-at patterns-----
create_patterns
order_patterns 3 // optimize the pattern set
// Save original path delay patterns and transition patterns, plus
// the add'l stuck-at patterns.
write_patterns pathdelay_trans_stuck_pat.bin.gz -binary -replace
//-----
//-----Grade for bridge fault coverage-----
set_fault_type bridge
read_fault_sites my_fault_definitions
add_faults -all
// Read in previously saved path delay, transition, and stuck-at
// patterns and add them to the internal pattern set
// when they are simulated.
read_patterns pathdelay_trans_stuck_pat.bin.gz
simulate_patterns -store_patterns all
report_statistics
//-----
```

4. Create additional bridge test patterns for any remaining bridge faults and add these test patterns to the test pattern set. Following are example commands in a dofile.

```
//-----Create add'l bridge patterns-----
create_patterns
order_patterns 3 // optimize the pattern set
// Save original path delay patterns, transition patterns, stuck-at
```

```
//      patterns, plus the add'l bridge patterns.  
write_patterns pathdelay_trans_stuck_bridge_pat.bin.gz -binary  
      -replace  
// Close the session and exit.  
exit
```

## Using Timing-Aware ATPG

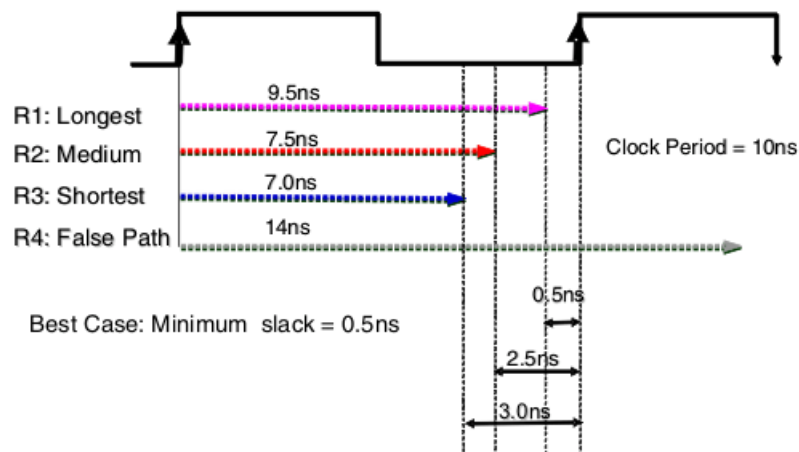
Timing-aware ATPG reads timing information from a Standard Delay Format (SDF) file and tries to generate patterns that detect transition faults using the longest detection path.

### Slack Calculation

Slack is equal to the margin between the path delay and the clock period. Slack within Small Delay Fault Model represents the smallest delay defect that can be detected.

Figure 6-42 illustrates slack calculations. Assume there are three paths that can detect a fault. The paths have a 9.5 ns, 7.5 ns and 7.0 ns delay, respectively. The clock period is 10 ns. The slacks for the paths are calculated as 0.5 ns, 2.5 ns and 3 ns, respectively. For the longest path, which has a 0.5 ns slack, the smallest delay defect can be detected is 0.5 ns. Similarly for the path with 2.5 ns, the smallest detectable delay defect is 2.5 ns. You can detect smaller delay defects by using a path with a smaller slack. Any path that is longer than the clock period is a false path.

**Figure 6-42. Timing Slack**



## Delay Test Coverage Calculation

ATPG calculates a metric called Delay Test Coverage to determine the quality of the test patterns. The delay test coverage is automatically included in the ATPG statistics report when timing-aware ATPG is enabled.

$$\text{Delay Test Coverage} = (\text{Path Delay that ATPG used}) / (\text{Longest Path Delay}) * 100\%$$

Using the same example paths from [Figure 6-42](#), if ATPG uses path R2, the delay test coverage is: 7.5 ns / 9.5 ns = 79%. If ATPG used the longest path (R1), the delay test coverage would be 100%.

Undetected faults have a delay test coverage of 0%. DI faults (Detected by Implication) have a delay test coverage of 100%. Chip-level delay test coverage is calculated by averaging the delay test coverage values for all faults.

## Timing-Aware ATPG vs. Transition ATPG

The following data was gathered from the STARC03 testcase, which STARC and Mentor Graphics used to evaluate timing-aware ATPG. [Table 6-3](#) compares transition fault ATPG and timing-aware ATPG. The testcase has the following characteristics:

- Design Size: 2.4M sim\_gate
- Number of FFs: 69,153
- CPU: 2.2Ghz
- 315 sec to read SDF file that has 10,674,239 lines

**Table 6-3. Testcase 2 Data**

| Run Parameters                               | Pattern Count | Transition TC | Delay TC | CPU Time    | Memory |
|----------------------------------------------|---------------|---------------|----------|-------------|--------|
| Before ATPG                                  | 0             | 0.00%         | 0.00%    | 0           | 2.1G   |
| Transition ATPG (1 detection)                | 3,668         | 91.23%        | 66.13%   | 4,180 sec   | 1.0G   |
| Transition ATPG (7 detections)               | 7,979         | 91.27%        | 68.07%   | 11,987 sec  | 1.0G   |
| Timing-Aware ATPG (SMFD <sup>1</sup> = 100%) | 3,508         | 91.19%        | 67.39%   | 17,373 sec  | 2.2G   |
| Timing-Aware ATPG (SMFD = 50%)               | 8,642         | 91.26%        | 76.26%   | 129,735 sec | 2.2G   |
| Timing-Aware ATPG (SMFD = 0%)                | 24,493        | 91.28%        | 77.71%   | 178,673 sec | 2.2G   |

$$1. \text{ Slack Margin for Fault Dropping} = (T_a - T_{ms}) * 100 / T_a.$$

For more information about setting run parameters, see the [set\\_atpg\\_timing](#) command.

## Timing-Aware ATPG Limitations

- For limitations regarding the SDF file, see the [read\\_sdf](#) command.
- Launch-off shift is not supported.
- The ATPG run time for timing-aware is about eight times slower than the normal transition fault ATPG. For more information, see “[Timing-Aware ATPG vs. Transition ATPG](#)”. Targeting critical faults may help.
- A large combinational loop may slow down the analysis to calculate the static slack. It also makes the actual delay analysis less accurate.
- The transition test coverage on timing-aware ATPG may be lower than the normal transition fault ATPG. Because timing-aware ATPG tries to detect a fault with a longer path, it is more likely to hit the abort limit. Use the -Retarget switch with the [create\\_patterns](#) command to improve transition test coverage.
- False and multi-cycle paths are defined by the [add\\_false\\_paths](#) or [read\\_sdc](#) commands. The slack for false and multi-cycle paths is rounded to zero, and the delay test coverage is 0%.
- Timing information in Named Capture Procedure are not included in static slack calculations.
- When saving check point, the SDF database is not stored. You must reload the SDF data, using the [read\\_sdf](#) command, when using a flattened model.
- Static Compression ([compress\\_patterns](#) command) and pattern ordering ([order\\_patterns](#) command) cannot be used when -Slack\_margin\_for\_fault\_dropping is specified.
- Clock skew is ignored. Clock skew is caused by a delay on clock paths, and timing-aware ATPG does not use the delay on clock path. Consequently, clock skew (the delay on clock path) is not accounted for in the timing-calculation.

## Understanding Inaccuracies in Timing Calculations

When performing timing-aware ATPG based on SDF, the following factors can result in inaccurate timing calculations:

- Device delay is not supported.
- Conditional delay is not supported. For a given IOPATH or INTERCONNECT pin pair, the maximal number among all conditional delays defined for this pair will be used by the tool when calculating static and actual delays.

- Negative delay is supported. However, if path delay is a negative number, the tool forces the delay value to 0 when calculating delay coverage, path delay, and slack, etc.
- Static timing is calculated in a pessimistic way. It does not consider false/multi-cycle paths.

## Running Timing-Aware ATPG

Use this procedure to create Timing-Aware test patterns with the ATPG tool.

### Prerequisites

- Because timing-aware ATPG is built on transition ATPG technology, you must set up for Transition ATPG first before starting this procedure. See [“Creating a Transition Delay Test Set.”](#)
- SDF file from static timing analysis.

### Procedure

1. Load the timing information from an SDF file using the [read\\_sdf](#) command. For example:

```
ANALYSIS> read_sdf top_worst.sdf
```

If you encounter problems reading the SDF file, see [“Errors and Warnings While Reading SDF Files”](#) on page 256.

2. Define clock information using the [set\\_atpg\\_timing](#) command. You must define the clock information for all clocks in the design, even for those not used for ATPG (not used in a named capture procedure). For example:

```
ANALYSIS> set_atpg_timing -clock clk_in 36000 18000 18000
```

```
ANALYSIS> set_atpg_timing -clock default 36000 18000 18000
```

3. Enable timing-aware ATPG using the [set\\_atpg\\_timing](#) command. For example:

```
ANALYSIS> set_atpg_timing on -slack_margin_for_fault_dropping 50%
```

If you specify a slack margin for fault dropping, the fault simulation keeps faults for pattern generation until the threshold is met. During normal transition fault simulation, faults are dropped as soon as they are detected.

4. Select timing-critical faults using the [set\\_atpg\\_timing](#) command. For example:

```
ANALYSIS> set_atpg_timing -timing_critical 90%
```

5. Run ATPG. For example:

```
ANALYSIS> create_patterns -retarget
```

6. Report delay\_fault test coverage using the [report\\_statistics](#) command. For example:

ANALYSIS> report\_statistics

## Timing-Aware Example

Figure 6-43 shows a testcase where there are 17 scan flip-flops and 10 combinational gates. Each gate has 1 ns delay and there is no delay on the scan flip-flops. A slow-to-rise fault is injected in G4/Y. The test period is 12 ns. The last scan flip-flop (U17) has an OX cell constraint so that it cannot be used as observation point.

The longest path starts at U1, moving through G1 through G10 and ending at U17. The total path delay is 10 ns. Because U17 cannot be used as observation point, timing-aware ATPG uses the path starting at U1, moving through G1 through G9 and ending at U16. The total path delay is 9 ns.

Therefore static minimum slack is 12 ns - 10 ns = 2 ns, and the best actual slack is 12 ns - 9 ns = 3 ns.

Figure 6-43. Testcase 1 Logic

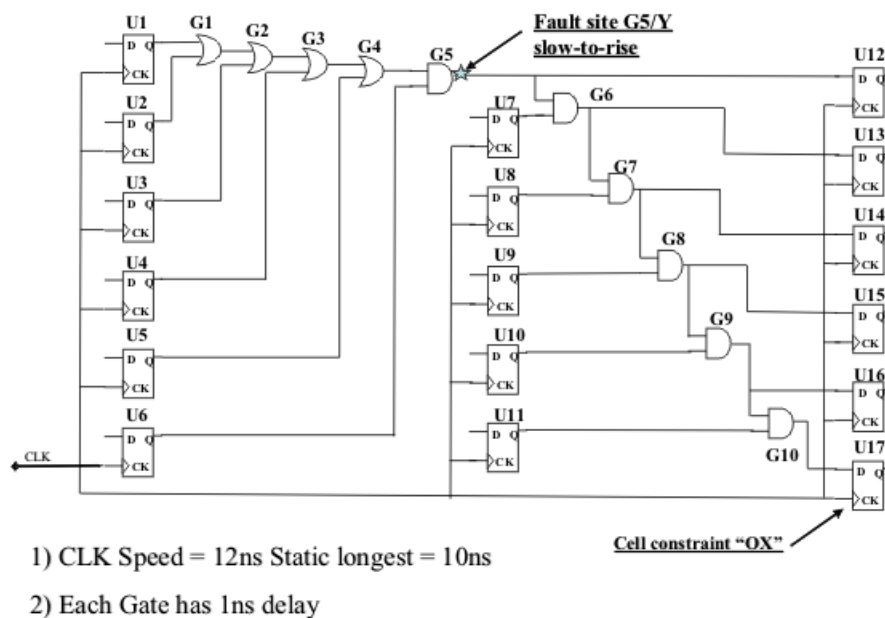


Figure 6-44 shows the dofile used for this example. As you can see in the comments, the dofile goes through the procedure outlined in “Running Timing-Aware ATPG.”

**Figure 6-44. Dofile**

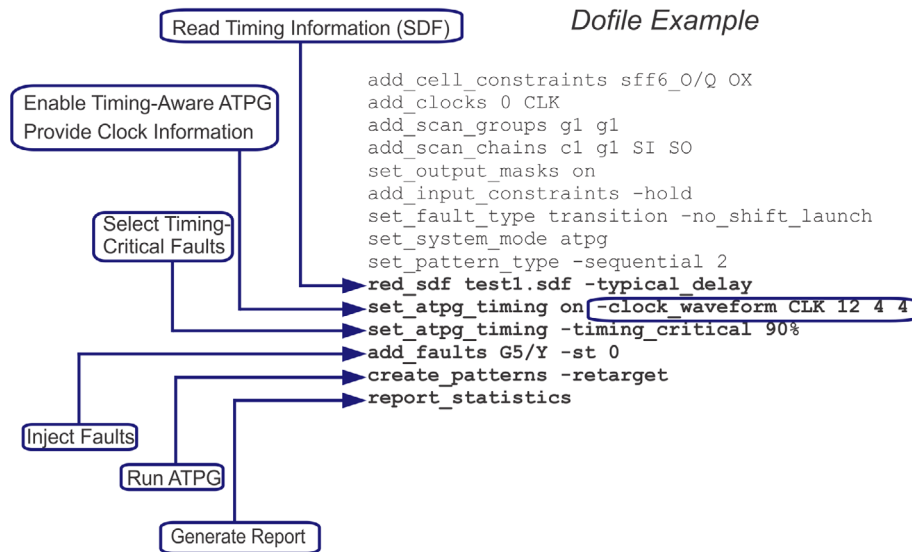


Figure 6-45 shows the fault statistics report. Timing-aware ATPG used the longest possible path, which is 9 ns. Static longest path is 10 ns. The delay test coverage is  $9 \text{ ns} / 10 \text{ ns} = 90\%$ .

**Figure 6-45. Reports**

Statistics report

| fault class                | #faults<br>(coll.) | #faults<br>(total) |
|----------------------------|--------------------|--------------------|
| FU (full)                  | 1                  | 1                  |
| DS (det_simulation)        | 1                  | 1                  |
| test_coverage              | 100.00%            | 100.00%            |
| fault_coverage             | 100.00%            | 100.00%            |
| atpg_effectiveness         | 100.00%            | 100.00%            |
| <b>Delay_test_coverage</b> | <b>90%</b>         | <b>90%</b>         |
| #test_patterns             |                    | 1                  |
| #clock_sequential_patterns |                    | 1                  |
| #simulated_patterns        |                    | 32                 |
| CPU_time (secs)            |                    | 0.1                |

## Troubleshooting Topics

The following topics describe common issues related to timing-aware ATPG:

- [Reducing run Time for Timing-Aware ATPG](#)
- [Errors and Warnings While Reading SDF Files](#)

- [Warnings During ATPG](#)
- [Actual Slack Smaller Than Tms](#)

## Reducing run Time for Timing-Aware ATPG

As shown in [Table 6-3](#) on page 251, timing-aware ATPG takes much longer than the regular transition ATPG. You can make it faster by targeting for timing-critical faults only.

You can use the [set\\_atpg\\_timing](#) command to make a fault list that includes faults with less slack time than you specified. The fault is put in the fault list if its (Longest delay)/(Test time) is more than your specified threshold.

For example, assume there are 3 faults and their longest paths are 9.5 ns, 7.5 ns and 6.0 ns respectively and the test time is 10 ns. The (Longest delay)/(Test time) is calculated 95%, 75% and 60% respectively. If you set the threshold to 80%, only the first fault is included. If you set it to 70%, the first and second faults are included.

The following series of commands inject only timing-critical faults with 70% or more.

```
ANALYSIS> add_faults -all
ANALYSIS> set_atpg_timing -timing_critical 70%
ANALYSIS> write_faults fault_list1 -timing_critical -replace
ANALYSIS> delete_faults -all
ANALYSIS> read_faults fault_list1
```

## Errors and Warnings While Reading SDF Files

Below is a list of the most common errors and warnings when reading SDF.

- Error: Near line N - The destination pin A/B/C is mapped to gate output.  
For an interconnect delay, the destination pin has to be a gate input pin. If a gate output pin is used, the tool issues the error message and ignores the delay.
- Error: Near line N - The pin A/B/C for conditional delay is mapped to gate output.  
The signals used for a condition on a conditional delay must all be gate inputs. If a gate output is used for the condition, the tool issues the error message and ignores the delay.  
For example, there is a component that has two outputs “O1” and “O2”. If a conditional delay on “O1” is defined by using “O2”, it will produce an error.
- Error: Near line N - Unable to map destination SDF pin.  
Ignore INTERCONNECT delay from pin A/B/C to pin D/E/F.  
The destination pin “D/E/F” does not have a receiver. It is likely to be a floating net, where no delay can be added.



- Error: Near line N - Unable to map source SDF pin.  
Ignore INTERCONNECT delay from pin A/B/C to pin D/E/F.  
The source pin “A/B/C” does not have a driver. It is likely to be a floating net, where no delay can be added.
- Error: Near line N - Unable to map flatten hierarchical source pin A/B/C derived from D/E/F.  
“D/E/F” is a source pin defined in SDF (either interconnect delay or IO Path delay), and “A/B/C” is a flattened (cell library based) pin corresponding to “D/E/F,” and there is no driver found.
- Error: Near line N - Unable to map flatten hierarchical destination pin A/B/C derived from D/E/F.  
“D/E/F” is a destination pin defined in SDF (either internet delay or IO Path delay), and “A/B/C” is a flattened (cell library based) pin corresponding to “D/E/F,” and there is no receiver found.
- Error: Near line N - There is no net path from pin A/B/C to pin D/E/F (fanin=1). Ignore current net delay.  
There is no connection between “A/B/C” (source) and “D/E/F” (destination). The “fanin=1” means the first input of the gate.
- Warning: Near line N - Negative delay value is declared.  
Negative delay is stored in the timing database and the negative value can be seen as the minimum delay, but the negative number is not used for delay calculation. It will be treated as a zero.
- Error: Near line N - Flatten hierarchical destination pin "A" is not included in instance "B".  
In this case, “B” is an instance name where both the source and destination pins are located (for example, an SDF file was loaded with “-instance B” switch). But a flattened hierarchical destination pin (cell based library) “A” traced from the destination pin is outside of the instance “B.”
- Error: Near line N - Flatten hierarchical source pin "A" is not included in instance "B".  
In this case, “B” is an instance name where both the source and destination pins are located (for example, an SDF file was loaded with “-instance B” switch). But a flattened hierarchical source pin (cell based library) “A” traced from the source pin is outside of the instance “B.”
- Error: Near line N - Unable to add IOPATH delay from pin A/B/C to pin D/E/F even when the pins are treated as hierarchical pin. Ignore IOPATH delay.  
There was an error when mapping the hierarchical source “A/B/C” and/or destination “D/E/F” to their flattened hierarchical pins. The delay is ignored. Most likely, one or both hierarchical pins are floating.

- Error: Near line N - Unable to add INTERCONNECT delay from pin "A/B/C" to pin "D/E/F" even when the pins are treated as hierarchical pin. Ignore INTERCONNECT delay.

There was an error when mapping the hierarchical source "A/B/C" and/or destination "D/E/F" to their flattened hierarchical pins. The delay is ignored. Most likely, one or both hierarchical pins are floating.

- Error: Near line N - The conditional delay expression is not supported. Ignore the delay.

The delay was ignored because the condition was too complex.

- Error: Near line N - Delay from pin A/B/C (fanin=1) to pin D/E/F has been defined before. Ignore current IOPATH delay.

Two or more IOPATH delays defined from SDF pin names map to the same flattened gate pin. The tool will keep the value for the first definition and ignore all the subsequent duplicates. This error with IOPATH delay is likely an SDF problem. For interconnect delay, the tool cannot handle multiple definitions. For example, the Verilog path is: /u1/Z -> net1 -> net2 -> /u2/A. If you define the first interconnect delay /u1/Z -> net1, the second net1 -> net2, and the third net2 -> /u2/A, then all the three interconnect delays are mapped to /u1/Z -> /u2/A in the gate level. This will cause an error.

- Error: Near line N - There is no combinational path from pin A/B/C (fanin=3) to pin D/E/F. Ignore current IOPATH delay.

The tool can only handle the IOPATH delay through combinational gates. Delays passing through state elements cannot be used.

## Warnings During ATPG

You may see the following message when you start ATPG:

```
Warning: Inconsistent holding PI attribute is set between test generation
and static timing analysis.
Test generation with capture procedures holds PI, but static timing
analysis allows PI change.
The inconsistent settings will impact the timing metrics reported by the
tool.
```

This is due to the conflicting setting for holding PI and masking PO between ATPG and static timing analysis.

For ATPG, holding PI and masking PO can be set by:

- Including the "add\_input\_constraints -hold" and "set\_output\_masks" commands in your dofile.
- Including the "force\_pi" and "measure\_po" statements in a named capture procedure.

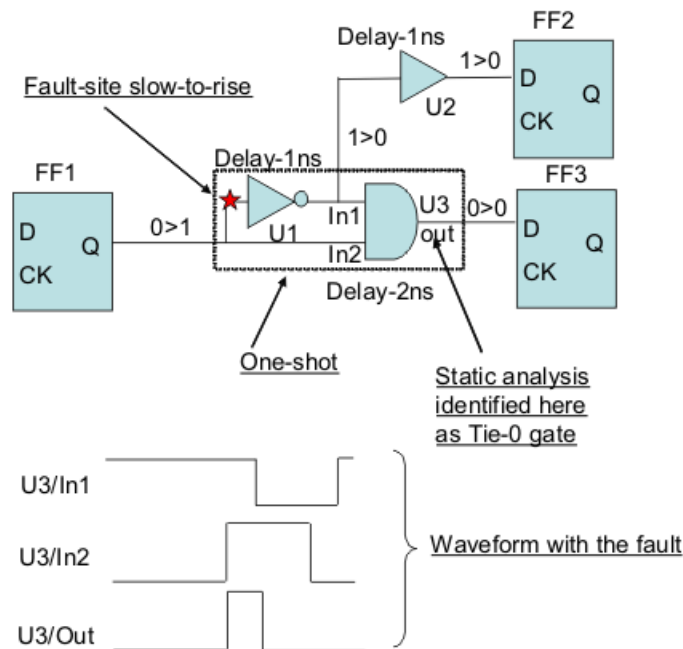
For static timing analysis, holding PI and masking PO can be set using the "-hold\_pi" and "-mask\_po" switches for the [set\\_atpg\\_timing](#) command.

The static timing database is created once, before running ATPG, with the information of holding PI and/or masking PO, whereas the setting can change for ATPG especially when using NCP (for example, one NCP has force\_pi and the other does not).

## Actual Slack Smaller Than Tms

This condition occurs when a one-shot circuit is used in the data path. In static analysis, it is identified as a “static-inactive” (STAT) path but it could be used as a fault detection path. See the circuit in [Figure 6-46](#).

**Figure 6-46. Glitch Detection Case**



A slow-to-rise fault is injected at the input of the inverter (U1). There are two paths to detect it. One is FF2 through U1 and U2 and the other is FF3 through U1 and U3. In the second path, the fault is detected as a glitch detection as shown in the waveform.

But in the static analysis (when calculating Tms), this path is blocked because U3/out is identified as a Tie-0 gate by DRC. Therefore the maximum static delay will be 2 ns (first path). If the test cycle is 10 ns, the Tms will be 8 ns. And if ATPG uses the second path to detect the fault, the actual slack will be 7 ns (10-1-2).

You will see the following message during DRC if DRC identifies a Tie-AND or Tie-OR.

```
// Learned tied gates: #TIED_ANDs=1 #TIED_ORs=1
```

Following are reports for the example shown in [Figure 6-46](#).

```
// command: report_faults -delay
// type      code      pin_pathname
//                                     static_slack actual_min_slack
// -----
//          0      DS      /U1/A
//                                     8.0000      7.0000
// command: set_gate_report delay actual_path 0
// command: report_gates /U1/A
// /U1  inv01
//      A      I  (0-1) (3.0000) /sff1/Q
//      Y      O  (1-0) (3.0000) /U2/A /U3/A1
// command: report_gates /U3/A1
// /U3  and02
//      A0      I  (0-1) (3.0000) /sff1/Q
//      A1      I  (1-0) (3.0000) /U1/Y
//      Y      O  (0-0) (-) /sff3/D

// command: set_gate_report delay static_path
// command: report_gates /U3
// /U3  and02
//      A0      I  (2.0000, 2.0000)/(0.0000, 0.0000) /sff1/Q
//      A1      I  (2.0000, 2.0000)/(2.0000, 2.0000) /U1/Y
//      Y      O  (STAT) /sff3/D
```

## Generating Patterns for a Boundary Scan Circuit

The following example shows how to create a test set for an IEEE 1149.1 (boundary scan)-based circuit. The following subsections list and explain the dofile and test procedure file.

### Dofile and Explanation

The following dofile shows the commands you could use to specify the scan data in the ATPG tool:

```
add_clocks 0 tck
add_scan_groups group1 proc_fscan
add_scan_chains chain1 group1 tdi tdo
add_input_constraints tms -c0
add_input_constraints trstz -c1
set_capture_clock TCK -atpg
```

You must define the tck signal as a clock because it captures data. There is one scan group, group1, which uses the *proc\_fscan* test procedure file (see [page 262](#)). There is one scan chain, chain1, that belongs to the scan group. The input and output of the scan chain are tdi and tdo, respectively.

The listed pin constraints only constrain the signals to the specified values during ATPG—not during the test procedures. Thus, the tool constrains tms to a 0 during ATPG (for proper pattern

generation), but not within the test procedures, where the signal transitions the TAP controller state machine for testing. The basic scan testing process is:

1. Initialize scan chain.
2. Apply PI values.
3. Measure PO values.
4. Pulse capture clock.
5. Unload scan chain.

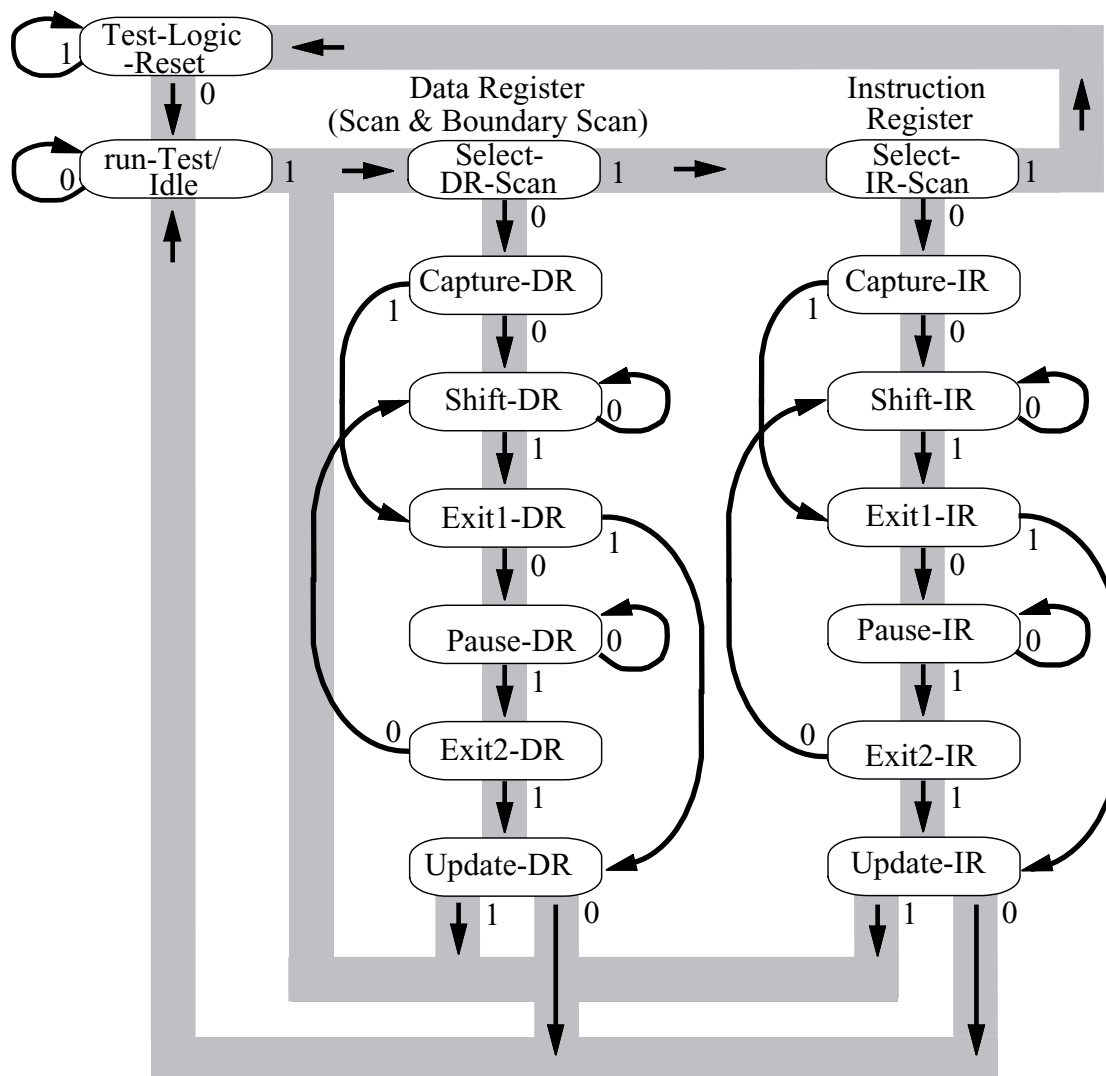
During Step 2, you must constrain `tms` to 0 so that the Tap controller's finite state machine (Figure 6-47) can go to the Shift-DR state when you pulse the capture clock (`tck`). You constrain the `trstz` signal to its off-state for the same reason. If you do not do this, the Tap controller goes to the Test-Logic-reset\_state at the end of the Capture-DR sequence.

The `set_capture_clock` TCK -ATPG command defines `tck` as the capture clock and that the capture clock must be used for each pattern (as the ATPG tool is able to create patterns where the capture clock never gets pulsed). This ensures that the Capture-DR state properly transitions to the Shift-DR state.

## TAP Controller State Machine

Figure 6-47 shows the finite state machine for the TAP controller of a IEEE 1149.1 circuit.

**Figure 6-47. State Diagram of TAP Controller Circuitry**



The TMS signal controls the state transitions. The rising edge of the TCK clock captures the TAP controller inputs. You may find this diagram useful when writing your own test procedure file or trying to understand the example test procedure file that the next subsection shows.

## Test Procedure File and Explanation

The test procedure file *proc\_fscan* follows:

```
set time scale 1 ns;
set strobe_window time 1;
timeplate tp0 =
    force_pi 100;
    measure_po 200;
    pulse TCK 300 100;
    period 500;
```

```
end;

procedure test_setup =
    timeplate tp0;

    // Apply reset procedure
    // Test cycle one

    cycle =
        force TMS 1;
        force TDI 0;
        force TRST 0;
        pulse TCK;
    end;

    // "TMS"=0 change to run-test-idle
    // Test cycle two

    cycle =
        force TMS 0;
        force TRST 1;
        pulse TCK;
    end;

    // "TMS"=1 change to select-DR
    // Test cycle three

    cycle =
        force TMS 1;
        pulse TCK;
    end;

    // "TMS"=1 change to select-IR
    // Test cycle four

    cycle =
        force TMS 1;
        pulse TCK;
    end;

    // "TMS"=0 change to capture-IR
    // Test cycle five

    cycle =
        force TMS 0;
        pulse TCK;
    end;

    // "TMS"=0 change to shift-IR
    // Test cycle six

    cycle =
        force TMS 0;
        pulse TCK;
    end;

    // load MULT_SCAN instruction "1000" in IR
    // Test cycle seven
```

```
cycle =
    force TMS 0;
    pulse TCK;
end;

// Test cycle eight

cycle =
    force TMS 0;
    pulse TCK;
end;

// Test cycle nine

cycle =
    force TMS 0;
    pulse TCK;
end;

// Last shift in exit-IR Stage
// Test cycle ten

cycle =
    force TMS 1;
    force TDI 1;
    pulse TCK;
end;

// Change to shift-dr stage for shifting in data
// "TMS" = 11100
// "TMS"=1 change to update-IR state
// Test cycle eleven

cycle =
    force TMS 1;
    force TDI 1;
    pulse TCK;
end;

// "TMS"=1 change to select-DR state
// Test cycle twelve

cycle =
    force TMS 1;
    pulse TCK;
end;

// "TMS"=0 change to capture-DR state
// Test cycle thirteen

cycle =
    force TMS 0;
    pulse TCK;
end;

// "TMS"=0 change to shift-DR state
// Test cycle fourteen
```



```

        cycle =
            force TMS 0;
            force TEST_MODE 1;
            force RESETN 1;
            pulse TCK;
        end;
    end;

    procedure shift =
        scan_group grp1;
        timeplate tp0;
        cycle =
            force_sci;
            measure_sco;
            pulse TCK;
        end;
    end;

    procedure load_unload =
        scan_group grp1;
        timeplate tp0;
        cycle =
            force TMS 0;
            force CLK 0;
        end;
        apply shift 77;

        // "TMS"=1 change to exit-1-DR state

        cycle =
            force TMS 1;
        end;
        apply shift 1;

        // "TMS"=1 change to update-DR state

        cycle =
            force TMS 1;
            pulse TCK;
        end;

        // "TMS"=1 change to select-DR-scan state

        cycle =
            force TMS 1;
            pulse TCK;
        end;

        // "TMS"=0 change to capture-DR state

        cycle =
            force TMS 0;
            pulse TCK;
        end;
    end;
end;

```

Upon completion of the `test_setup` procedure, the tap controller is in the shift-DR state in preparation for loading the scan chain(s). It is then placed back into the shift-DR state for the next scan cycle. This is achieved by the following:

- The items that result in the correct behavior are the pin constraint on `tms` of C1 and the fact that the capture clock has been specified as TCK.
- At the end of the `load_unload` procedure, the tool asserts the pin constraint on TMS, which forces `tms` to 0.
- The capture clock (TCK) occurs for the cycle and this results in the tap controller cycling from the run-test-idle to the Select-DR-Scan state.
- The `load_unload` procedure is again applied. This will start the next load/unloading the scan chain.

The first procedure in the test procedure file is **test\_setup**. This procedure begins by resetting the test circuitry by forcing `trstz` to 0. The next set of actions moves the state machine to the Shift-IR state to load the instruction register with the internal scan instruction code (1000) for the MULT\_SCAN instruction. This is accomplished by shifting in 3 bits of data (`tdi=0` for three cycles) with `tms=0`, and the 4th bit (`tdi=1` for one cycle) when `tms=1` (at the transition to the Exit1-IR state). The next move is to sequence the TAP to the Shift-DR state to prepare for internal scan testing.

The second procedure in the test procedure file is **shift**. This procedure forces the scan inputs, measures the scan outputs, and pulses the clock. Because the output data transitions on the falling edge of `tck`, the `measure_sco` command at time 0 occurs as `tck` is falling. The result is a rules violation unless you increase the period of the **shift** procedure so `tck` has adequate time to transition to 0 before repeating the shift. The **load\_unload** procedure, which is next in the file, calls the **shift** procedure.

The basic flow of the **load\_unload** procedure is to:

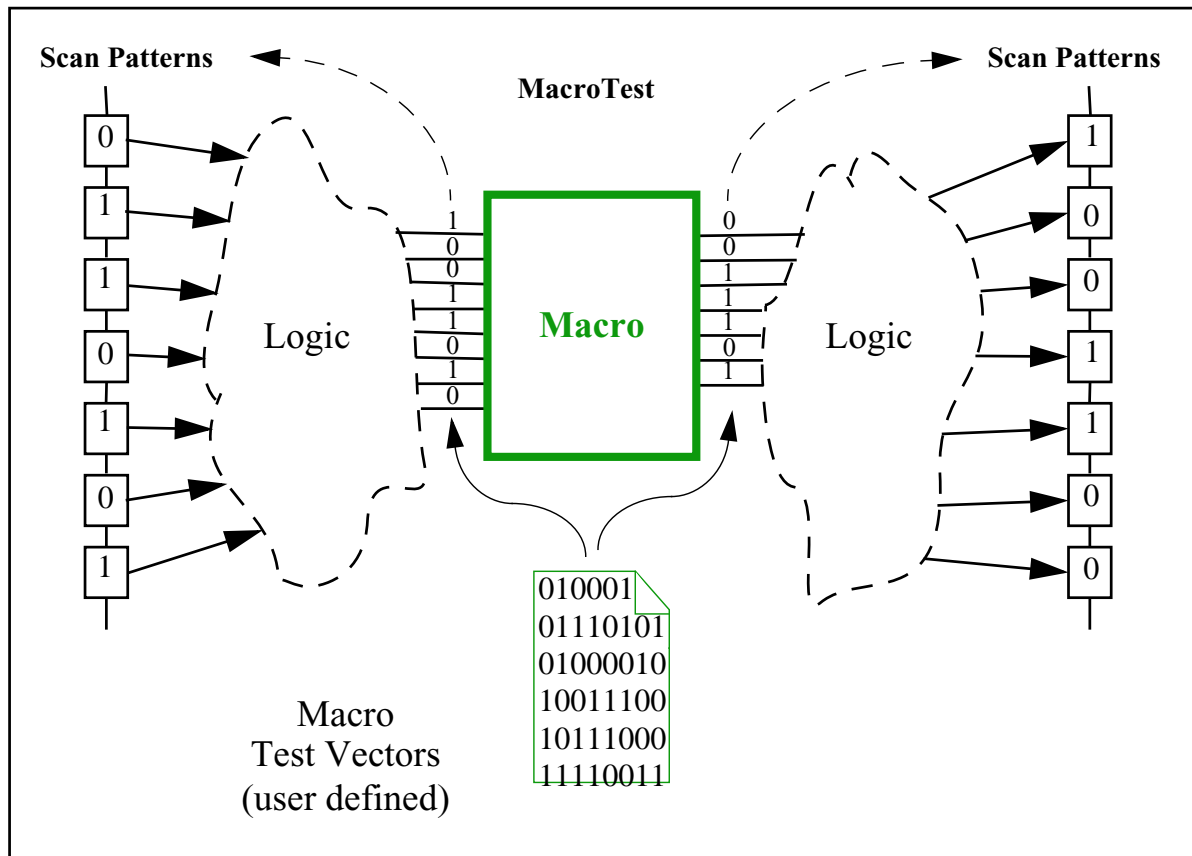
1. Force circuit stability (all clocks off, etc.).
2. Apply the **shift** procedure `n-1` times with `tms=0`
3. Apply the shift procedure one more time with `tms=1`
4. Set the TAP controller to the Capture-DR state.

The **load\_unload** procedure inactivates the reset mechanisms, because you cannot assume they hold their values from the **test\_setup** procedure. It then applies the **shift** procedure 77 times with `tms=0` and once more with `tms=1` (one shift for each of the 77 scan registers within the design). The procedure then sequences through the states to return to the Capture-DR state. You must also set `tck` to 0 to meet the requirement that all clocks be off at the end of the procedure.

## Using the MacroTest Capability

MacroTest is a utility that helps automate the testing of embedded logic and memories (macros) by automatically translating user-defined patterns for the macros into scan patterns. Because it enables you to apply your macro test vectors in the embedded environment, MacroTest improves overall IC test quality. It is particularly useful for testing small RAMs and embedded memories but can also be used for a disjoint set of internal sites or a single block of hardware represented by an instance in HDL. This is illustrated conceptually in [Figure 6-48](#).

**Figure 6-48. Conceptual View of MacroTest**



MacroTest provides the following capabilities and features:

- Supports user-selected scan observation points
- Supports synchronous memories; for example, supports positive (or negative) edge-triggered memories embedded between positive (or negative) edge-triggered scan chains
- Enables you to test multiple macros in parallel
- Analyzes single macros, and reports patterns that are logically inconsistent with the surrounding hardware
- Allows you to define macro output values that do not require observation
- Has no impact on area or performance

## The MacroTest Process Flow

To use MacroTest effectively, you need to be familiar with two commands:

- [set\\_macrotest\\_options](#) — Modifies two rules of the DRCs to allow otherwise illegal circuits to be processed by MacroTest. Black box (un-modelled) macros may require this command.
- [macrotest](#) — Runs the MacroTest utility to read functional patterns you provide and convert them into scan-based manufacturing test patterns.

The MacroTest flow requires a set of patterns and MacroTest. The patterns are a sequence of tests (inputs and expected outputs) that you develop to test the macro. For a memory, this is a sequence of writes and reads. You may need to take embedding restrictions into account as you develop your patterns. Next, you set up and run MacroTest to convert these cycle-based patterns into scan-based test patterns. The converted patterns, when applied to the chip, reproduce your input sequence at the macro's inputs through the intervening logic. The converted patterns also ensure that the macro's output sequence is as you specified in your set of patterns.

---

### Note

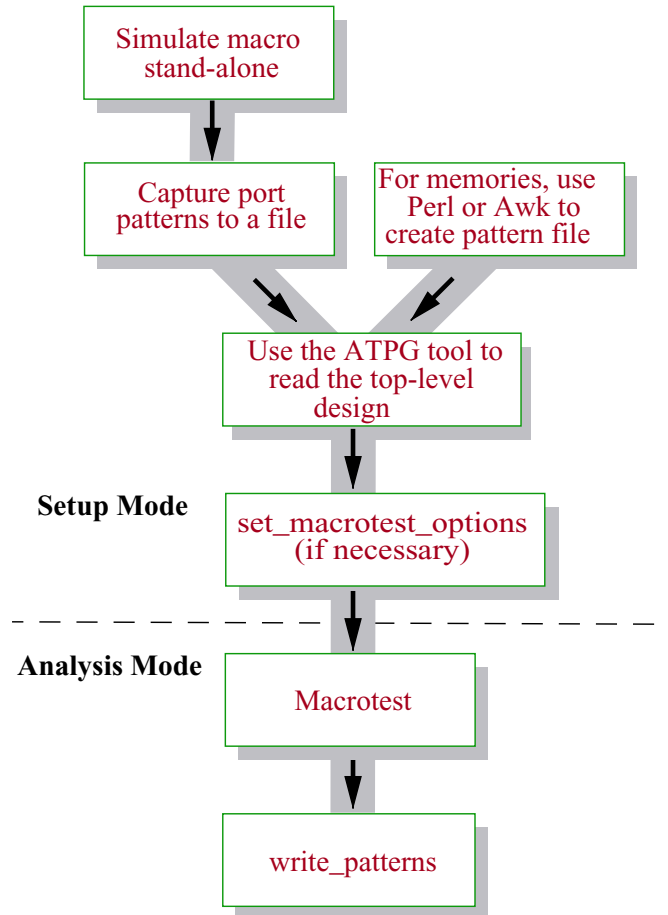


You can generate a wide range of pattern sets: From simple patterns that verify basic functionality, to complex, modified March algorithms that exercise every address location multiple times. Some embeddings (the logic surrounding the macro) do not allow arbitrary sequences, however.

---

Figure 6-49 shows the basic flow for creating scan-based test patterns with MacroTest.

**Figure 6-49. Basic Scan Pattern Creation Flow with MacroTest**



**Note**



The patterns produced by MacroTest cannot be read back into the ATPG tool. This is because the simulation and assumptions about the original macro patterns are no longer valid and the original macro patterns are not preserved in the MacroTest patterns.

Tessent Diagnosis, a Mentor Graphics test failure diagnosis tool, also cannot read a pattern set that includes MacroTest patterns. If Tessent Diagnosis is used in your manufacturing test flow, save MacroTest patterns separately from your other patterns. This will enable a test engineer to remove them from the set of patterns applied on ATE before attempting to read that set into Tessent Diagnosis for diagnosis.

When you run the macrotest command, MacroTest reads your pattern file and begins analyzing the patterns. For each pattern, the tool searches back from each of the macro's inputs to find a scan flip-flop or primary input. Likewise, the tool analyzes observation points for the macro's output ports. When it has justified and recorded all macro input values and output values, MacroTest moves on to the next pattern and repeats the process until it has converted all the patterns. The default MacroTest effort exhaustively tries to convert all patterns. If successful,

then the set of scan test patterns MacroTest creates will detect any defect inside the macro that changes any macro output from the expected value.



---

**Note**

If you add\_faults prior to running MacroTest, the ATPG tool automatically fault simulates the patterns as they are created. This is time consuming, but is retained for backward compatibility. It is advised that you generate and save macrotest patterns in a separate run from normal ATPG and faultsim and that you not issue the add\_faults command in the MacroTest run.

---

The patterns you supply to MacroTest must be consistent with the macro surroundings (embedding) to assure success. In addition, the macro must meet certain design requirements. The following sections detail these requirements, describe how and when to use MacroTest, and conclude with some examples.

## Qualifying Macros for MacroTest

If a design meets the following three criteria, then you can use MacroTest to convert a sequence of functional cycles (that describe I/O behavior at the macro boundary) into a sequence of scan patterns:

1. The design has at least one combinational observation path for each macro output pin that requires observation (usually all outputs).
2. All I/O of the RAM/macro block to be controlled or observed are unidirectional.
3. The macro/block can hold its state while the scan chain shifts, if the test patterns require that the state be held across patterns. This is the case for a March algorithm, for example.

If you write data to a RAM macro (RAM), for example, then later read the data from the RAM, typically you will need to use one scan pattern to do the write, and a different scan pattern to do the read. Each scan pattern has a load/unload that shifts the scan chain, and you must ensure that the DFT was inserted, if necessary, to allow the scan chain to be shifted without writing into the RAM. If the shift clock can also cause the RAM to write and there is no way to protect the RAM, then it is very likely that the RAM contents will be destroyed during shift; the data written in the early pattern will not be preserved for reading during the latter pattern. Only if it is truly possible to do a write followed by a read, all in one scan pattern, then you may be able to use MacroTest even with an unprotected RAM.

Because converting such a multicycle pattern is a sequential ATPG search problem, success is not guaranteed even if success is possible. Therefore, you should try to convert a few patterns before you depend on MacroTest to be able to successfully convert a given embedded macro. This is a good idea even for combinational conversions.

If you intend to convert a sequence of functional cycles to a sequence of scan patterns, you can insert the DFT to protect the RAM during shift: The RAM should have a write enable that is PI-controllable throughout test mode to prevent destroying the state of the RAM. This ensures the tool can create a state inside the macro and retain the state during the scan loading of the next functional cycle (the next scan pattern after conversion by MacroTest).

The easiest case to identify is where the ATPG tool issues a message saying it can use the RAM test mode, `RAM_SEQUENTIAL`. This message occurs because the tool can independently operate the scan chains and the RAM. The tool can operate the scan chain without changing the state of the macro as well as operate the macro without changing the state loaded into the scan chain. This allows the most flexibility for ATPG, but the most DFT also.

However, there are cases where the tool can operate the scan chain without disturbing the macro, while the opposite is not true. If the scan cells are affected or updated when the macro is operated (usually because a single clock captures values into the scan chain and is also an input into the macro), the tool cannot use `RAM_SEQUENTIAL` mode. Instead, the tool can use a sequential MacroTest pattern (multiple cycles per scan load), or it can use multiple single cycle patterns if the user's patterns keep the write enable or write clock turned off during shift.

For example, suppose a RAM has a write enable that comes from a PI in test mode. This makes it possible to retain written values in the RAM during shift. However, it also has a single edge-triggered read control signal (no separate read enable) so the RAM's outputs change any time the address lines change followed by a pulse of the read clock/strobe. The read clock is a shared clock and is also used as the scan clock to shift the scan chains (composed of MUX scan cells). In this case, it is not possible to load the scan chains without changing the read values on the output of the macro. For this example, you will need to describe a sequential read operation to MacroTest. This can be a two-cycle operation. In the first cycle, MacroTest pulses the read clock. In the second cycle, MacroTest observes and captures the macro outputs into the downstream scan cells. This works because there is no intervening scan shift to change the values on the macro's output pins. If a PI-controllable read enable existed, or if you used a non-shift clock (clocked scan and LSSD have separate shift and capture clocks), an intervening scan load could occur between the pulse of the read clock and the capture of the output data. This is possible because the macro read port does not have to be clocked while shifting the scan chain.

## When to Use MacroTest

MacroTest is primarily used to test small memories (register file, cache, FIFO, and so on). Although the ATPG tool can test the faults at the boundaries of such devices, and can propagate the fault effects through them (using the `_ram` or `_cram` primitives), it does not attempt to create a set of patterns to test them internally. This is consistent with how it treats all primitives. Because memory primitives are far more complex than a typical primitive (such as a NAND gate), you may prefer to augment ATPG tool patterns with patterns that you create to test the internals of the more complex memory primitives. Such complex primitives are usually packaged as models in the ATPG library, or as HDL modules that are given the generic name "macro".

**Note**



Although the ATPG library has specific higher level collections of models called macros, MacroTest is not limited to testing these macros. It can test library models and HDL modules as well.

---

Here, the term “macro” simply means some block of logic, or even a distributed set of lines that you want to control and observe. You must provide the input values and expected output values for the macro. Typically you are given, or must create, a set of tests. You can then simulate these tests in some time-based simulator, and use the results predicted by that simulator as the expected outputs of the macro. For memories, you can almost always create both the inputs and expected outputs without any time-based simulation. For example, you might create a test that writes a value, V, to each address. It is trivial to predict that when subsequent memory reads occur, the expected output value will be V.

MacroTest converts these functional patterns to scan patterns that can test the device after it is embedded in systems (where its inputs and outputs are not directly accessible, and so the tests cannot be directly applied and observed). For example, a single macro input enable might be the output of two enables which are ANDed outside the macro. The tests must be converted so that the inputs of the AND are values which cause the AND’s output to have the correct value at the single macro enable input (the value specified by the user as the macro input value). MacroTest converts the tests (provided in a file) and provides the inputs to the macro as specified in the file, and then observes the outputs of the macro specified in the file. If a particular macro output is specified as having an expected 0 (or 1) output, and this output is a data input to a MUX between the macro output and the scan chain, the select input of that MUX must have the appropriate value to propagate the macro’s output value to the scan chain for observation. MacroTest automatically selects the path(s) from the macro output(s) to the scan chain(s), and delivers the values necessary for observation, such as the MUX select input value in the example above.

Often, each row of a MacroTest file converts to a single 1-system cycle scan test (sometimes called a basic scan pattern in the ATPG tool). A scan chain load, PI assertion, output measure, clock pulse, and scan chain unload result for each row of the file if you specify such patterns. To specify a write with no expected known outputs, specify the values to apply at the inputs to the device and give X output values (don't care or don't measure). To specify a read with expected known outputs, specify both the inputs to apply, and the outputs that are expected (as a result of those and all prior inputs applied in the file so far). For example, an address and read enable would have specified inputs, whereas the data inputs could be X (don't care) for a memory read.

Mentor Graphics highly recommends that you not over-specify patterns. It may be impossible, due to the surrounding logic, to justify all inputs otherwise. For example, if the memory has a write clock and write enable, and is embedded in a way that the write enable is independent but the clock is shared with other memories, it is best to turn off the write using the write enable, and leave the clock X so it can be asserted or de-asserted as needed. If the clock is turned off instead of the write enable, and the clock is shared with the scan chain, it is not possible to pulse the shared clock to capture and observe the outputs during a memory read. If instead, the write



enable is shared and the memory has its own clock (not likely, but used for illustration), then it is best to turn off the write with the clock and leave the shared write enable X.

Realize that although the scan tests produced appear to be independent tests, the tool assumes that the sequence being converted has dependencies from one cycle to the next. Thus, the scan patterns have dependencies from one scan test to the next. Because this is atypical, the tool marks MacroTest patterns as such, and you must save such MacroTest patterns using the `write_patterns` command. The MacroTest patterns cannot be reordered or reduced using `compress_patterns`; reading back MacroTest patterns is not allowed for that reason. You must preserve the sequence of MacroTest patterns as a complete, ordered set, all the way to the tester, if the assumption of cycle-to-cycle dependencies in the original functional sequence is correct.

To illustrate, if you write a value to an address, and then read the value in a subsequent scan pattern, this will work as long as you preserve the original pattern sequence. If the patterns are reordered, and the read occurs before the write, the patterns will then mismatch during simulation or fail on the tester. The reason is that the reordered scan patterns try to read the data before it has been written. This is untrue of all other ATPG tool patterns. They are independent and can be reordered (for example, to allow pattern compaction to reduce test set size). Macrotest patterns are never reordered or reduced, and the number of input patterns directly determines the number of output patterns.

## Defining the Macro Boundary

The macro boundary is typically defined by its instance name with the `macrotest` command. If no instance name is given, then the macro boundary is defined by a list of hierarchical pin names (one per macro pin) given in the header of the MacroTest patterns file.

### Defining a Macro Boundary by Instance Name

The macro is a particular instance, almost always represented by a top-level model in the ATPG library. More than one instance may occur in the netlist, but each instance has a unique name that identifies it. Therefore, the instance name is all that is needed to define the macro boundary.

The definition of the instance/macro is accessed to determine the pin order as defined in the port list of the definition. MacroTest expects that pin order to be used in the file specifying the I/O (input and expected output) values for the macro (the tests). For example, the command:

```
macrotest regfile_8 file_with_tests
```

would specify for MacroTest to find the instance “regfile\_8”, look up its model definition, and record the name and position of each pin in the port list. Given that the netlist is written in Verilog, with the command:

```
regfile_definition_name regfile_8 (net1, net2, ... );
```

the portlist of regfile\_definition\_name (not the instance port list “net1, net2, ...”) is used to get the pin names, directions, and the ordering expected in the test file, *file\_with\_tests*. If the library definition is:

```
model "regfile_definition_name"  
  ("Dout_0", "Dout_1", Addr_0, "Addr_1", "Write_enable", ...)  
  ( input ("Addr_0") () ... output ("Dout_0") () ... )
```

then MacroTest knows to expect the output value Dout\_0 as the first value (character) mentioned in each row (test) of the file, *file\_with\_tests*. The output Dout\_1 should be the 2nd pin, input pin Addr\_0 should be the 3rd pin value encountered, etc. If it is inconvenient to use this ordering, the ordering can be changed at the top of the test file, *file\_with\_tests*. This can be done using the following syntax:

```
macro_inputs Addr_0 Addr_1  
macro_output Dout_1  
macro_inputs Write_enable  
...  
end
```

which would cause MacroTest to expect the value for input Addr\_0 to be the first value in each test, followed by the value for input Addr\_1, the expected output value for Dout\_1, the input value for Write\_enable, and so on.

---

#### Note



Only the pin names need be specified, because the instance name “regfile\_8” was given for the macrotest command.

---

## Defining a Macro Boundary Without Using an Instance Name

If an instance name is not given with the macrotest command, then you must provide an entire hierarchical path/pin name for each pin of the macro. This is given in the header of the MacroTest patterns file. There must be one name per data bit in the data (test values) section which follows the header. For example:

```
macro_inputs regfile_8/Addr_0regfile_8/Addr_1  
macro_outputregfile_8/Dout_1  
macro_inputsregfile_8/write_enable  
...  
end
```

The above example defines the same macro boundary as was previously defined for regfile\_8 using only pin names to illustrate the format. Because the macro is a single instance, this would

not normally be done, because the instance name is repeated for each pin. However, you can use this entire pathname form to define a distributed macro that covers pieces of different instances. This more general form of boundary definition allows a macro to be any set of pins at any level(s) of hierarchy down to the top library model. If you use names which are inside a model in the library, the pin pathname must exist in the flattened data structures. (In other words, it must be inside a model where all instances have names, and it must be a fault site, because these are the requirements for a name inside a model to be preserved in the tool).

This full path/pin name form of “macro boundary” definition is a way to treat any set of pins/wires in the design as points to be controlled, and any set of pins/wires in the design as points to be observed. For example, some pin might be defined as a macro\_input which is then given {0,1} values for some patterns, but X for others. In some sense, this “macro input” can be thought of as a programmable ATPG constraint (see [add\\_atpg\\_constraints](#)), whose value can be changed on a pattern by pattern basis. There is no requirement that inputs be connected to outputs. It would even be possible to define a distributed macro such that the “output” is really the input to an inverter, and the “input” is really the output of the same inverter. If the user specified that the input = 0, and the expected output = 1, MacroTest would ensure that the macro “input” was 0 (so the inverter output is 0, and its input is 1), and would sensitize the input of the inverter to some scan cell in a scan chain. Although this is indeed strange, it is included to emphasize the point that full path/pin forms of macro boundary definition are completely flexible and unrelated to netlist boundaries or connectivity. Any set of connected or disjoint points can be inputs and/or outputs.

## Reporting and Specifying Observation Sites

You can report the set of possible observation sites using the macrotest command switch, -Report\_observation\_candidates. This switch reports, for each macro output, the reachable scan cells and whether the scan cell is already known to be unable to capture/observe. Usually, all reachable scan cells can capture, so all are reported as possible observation sites. The report gives the full instance name of the scan cell’s memory element, and its gate id (which follows the name and is surrounded by parentheses).

Although rarely done, you can specify for one macro output at a time exactly which of those reported scan cells is to be used to observe that particular macro output pin. Any subset can be so specified. For example, if you want to force macro output pin Dout\_1 to be observed at one of its reported observation sites, such as “/top/middle/bottom/ (13125)”, then you can specify this as follows:

```
macro_output regfile_8/Dout_1
observe_at13125
```

---

### Note



There can be only one macro\_output statement on the line above the observe\_at directive. Also, you must specify only one observe\_at site, which is always associated with the single macro\_output line that precedes it. If a macro\_input line immediately precedes the observe\_at line, MacroTest will issue an error message and exit.

---

The preceding example uses the gate id (number in parentheses in the -Report output) to specify the scan cell DFF to observe at, but you can also use the instance pathname. Instances inside models may not have unique names, so the gate id is always an unambiguous way to specify exactly where to observe. If you use the full name and the name does not exactly match, the tool selects the closest match from the reported candidate observation sites. The tool also warns you that an exact match did not occur and specifies the observation site that it selected.

## Defining a Macro Boundary With Trailing Edge Inputs

MacroTest treats macros as black boxes, even if modelled, so do not assume that this information will be gathered using connectivity. Assuming nothing is known about the macro's internals, MacroTest forces the user-specified expected outputs onto the macro outputs for each pattern. This allows black-boxed macros to be used, or you to create models for normal ATPG using the `_cram` primitive, but treat the macro as a black box for internal testing. A `_cram` primitive may be adequate for passing data through a RAM, for example, but not for modelling it for internal faults. MacroTest trusts the output values you provide regardless of what would normally be calculated in the tool, allowing you to specify outputs for these and other situations.

Due to its black box treatment of even modelled RAMs/macros, MacroTest must sometimes get additional information from you. MacroTest assumes that all macro inputs capture on the leading edge of any clock that reaches them. So, for a negative pulse, MacroTest assumes that the leading (falling) edge causes the write into the macro, whereas for a positive pulse, MacroTest assumes that the leading (rising) edge causes the write. If these assumptions are not true, you must specify which data or address inputs (if such pins occur) are latched into the macro on a trailing edge.

Occasionally, a circuit uses leading DFF updates followed by trailing edge writes to the memory driven by those DFFs. For trailing edge macro inputs, you must indicate that the leading edge assumption does not hold for any input pin value that must be presented to the macro for processing on the trailing edge. For a macro which models a RAM with a trailing edge write, you must specify this fact for the write address and data inputs to the macro which are associated with the falling edge write. To specify the trailing edge input, you must use a boundary description which lists the macro's pins (you cannot use the instance name only form).

Regardless of whether you use just pin names or full path/pin names, you can replace "macro\_inputs" with "te\_macro\_inputs" to indicate that the inputs that follow must have their values available for the trailing edge of the shared clock. This allows MacroTest to ensure that the values arrive at the macro input in time for the trailing edge, and also that the values are not overwritten by any leading edge DFF or latch updates. If a leading edge DFF drives the trailing edge macro input pin, the value needed at the macro input will be obtained from the D input side of the DFF rather than its Q output. The leading edge will make  $Q=D$  at the DFF, and then that new value will propagate to the macro input and be waiting for the trailing edge to use. Without the user specification as a trailing edge input, MacroTest would obtain the needed input value from the Q output of the DFF. This is because MacroTest would assume that the leading edge of

the clock would write to the macro before the leading edge DFF could update and propagate the new value to the macro input.

It is not necessary to specify leading edge macro inputs because this is the default behavior. It is also unnecessary to indicate leading or trailing edges for macro outputs. You can control the cycle in which macro outputs are captured. This ensures that the tool correctly handles any combination of macro outputs and capturing scan cells as long as all scan cells are of the same polarity (all leading edge capture/observe or all trailing edge capture/observe).

In the rare case that a particular macro output could be captured into either a leading or a trailing edge scan cell, you must specify which you prefer by using the `-Le_observation_only` switch or `-Te_observation_only` switch with the `macrotest` command for that macro. For more information on these switches, see “[Example 3 — Using Leading Edge & Trailing Edge Observation Only](#)” and the `macrotest` description in the *Tessent Shell Reference Manual*.

An example of the TE macro input declaration follows:

```
macro_input clock
te_macro_inputs Addr_0 Addr_1 // TE write address inputs
macro_output Dout_1
...
end
```

## Defining Test Values

The test file may consist of the following:

- Comments (a line starting with “//” or #)
- Blank lines
- An optional pin reordering section (which must come before any values) that begins with “MACRO\_INputs” or “MACRO\_OUTputs” and ends with “END”
- The tests (one cycle per row of the file)

Normal (nonpulseable) input pin values include {0,1,X,Z}. Some macro inputs may be driven by PIs declared as pulseable pins (`add_clocks`, `add_read_controls`, and `add_write_controls` specify these pins in the tool). These pins can have values from {P,N} where P designates a positive pulse and N designates a negative pulse. Although you can specify a P or N on any pin, the tool issues a warning if it cannot verify that the pin connects to a pulseable primary input (PI). If the tool can pulse the control and cause a pulse at that macro pin, then the pulse will occur. If it cannot, the pulse will not occur. Users are warned if they specify the wrong polarity of pulse (an N, for example, when there is a direct, non-inverting connection to a clock PI that has been specified with an off value of 0, which means that it can only be pulsed positively). A P would need to be specified in such a case, and some macro inputs would probably have to be specified as `te_macro_inputs` since the N was probably used due to a negative edge macro. P

and N denote the actual pulse, not the triggering edge of the macro. It is the embedding that determines whether a P or N can be produced.

---

**Note**

It is the declaration of the PI pin driving the macro input, not any declaration of the macro input itself, which determines whether a pin can be pulsed in the tool.

---

Normal observable output values include {L,H}, which are analogous to {0,1}. L represents output 0, and H represents output 1. You can give X as an output value to indicate Don't Compare, and F for a Floating output (output Z). Neither a Z nor an X output value will be observed. Occasionally an output cannot be observed, but must be known in order to prevent bus contention or to allow observation of some other macro output.

If you provide a file with these characters, a check is done to ensure that an input pin gets an input value, and an output pin gets an output value. If an “L” is specified in an input pin position, for example, an error message is issued. This helps detect ordering mismatches between the port list and the test file. If you prefer to use 0 and 1 for both inputs and outputs, then use the -No\_l\_h switch with the macrotest command:

**macrotest regfile\_8 file\_with\_tests -no\_l\_h**

Assuming that the -L\_h default is used, the following might be the testfile contents for our example register file, if the default port list pin order is used.

```
// Tests for regfile_definition_name.
//
//      W
//      r
//      i
//      t
//      e
//      _
// DD AA e
// oo dd n
// uu dd a
// tt rr b
//  _  _  l
// 01 01 e

XX 00 0
XX 00 1
HH 00 0
```

The example file above has only comments and data; spaces are used to separate the data into fields for convenience. Each row must have exactly as many value characters as pins mentioned in the original port list of the definition, or the exact number of pins in the header, if pins were specified there. Pins can be left off of an instance if macro\_inputs and macro\_outputs are specified in the header, so the header names are counted and that count is used unless the instance name only form of macro boundary definition is used (no header names exist).

To specify less than all pins of an instance, omit the pins from the header when reordering the pins. The omitted pins are ignored for purposes of MacroTest. If the correct number of values do not exist on every row, an error occurs and a message is issued.

The following is an example where the address lines are exchanged, and only Dout\_0 is to be tested:

```
// Tests for regfile_definition_name testing only Dout_0
macro_output Dout_0
macro_inputs Addr_1 Addr_0 write_enable ..
...
end
//      W
//      r
//      i
//      t
//      e
//      _
// D AA e
// o dd n
// u dd a
// t rr b
// _ _ l
// 0 10 e

X 00 0
X 00 1
H 00 0
```

It is not necessary to have all macro\_inputs together. You can repeat the direction designators as necessary:

```
macro_input write_enable
macro_output Dout_0
macro_inputs Addr_1 Addr_0
macro_outputs Dout_1 ...
...
end
```

## Recommendations for Using MacroTest

When using MacroTest, Mentor Graphics recommends that you begin early in the process. This is because the environment surrounding a regfile or memory may prevent the successful delivery of the original user specified tests, and Design-for-Test hardware may have to be added to allow the tests to be delivered, or the tests may have to be changed to match the surroundings so that the conversion can occur successfully.

For example, if the write enable line outside the macro is the complement of the read enable line (perhaps due to a line which drives the read enable directly and also fans out to an inverter which drives the write enable), and you specify that both the read enable and write enable pins should be 0 for some test, then MacroTest will be unable to deliver both values. It stops and reports the line of the test file, as well as the input pins and values that cannot be delivered. If



you change the enable values in the MacroTest patterns file to always be complementary, MacroTest would then succeed. Alternatively, if you add a MUX to make the enable inputs independently controllable in test mode and keep the original MacroTest patterns unchanged, MacroTest would use the MUX to control one of the inputs to succeed at delivering the complementary values.

Once MacroTest is successful, you should simulate the resulting MacroTest patterns in a time-based simulator. This verifies that the conversion was correct, and that no timing problems exist. The tool does not simulate the internals of primitives, and therefore relies on the fact that the inputs produced the expected outputs given in the test file. This final simulation ensures that no errors exist due to modeling or simulation details that might differ from one simulator to the next. Normal ATPG tool considerations hold, and it is suggested that DRC violations be treated as they would be treated for a stuck-at fault ATPG run.

To prepare to MacroTest an empty (TieX) macro that needs to be driven by a write control (to allow pulsing of that input pin on the black box), issue the [set\\_macrotest\\_options](#) command. This command prevents a G5 DRC violation and allows you to proceed. Also, if a transparent latch (TLA) on the control side of an empty macro is unobservable due to the macro, the [set\\_macrotest\\_options](#) command prevents it from becoming a TieX, as would normally occur. Once it becomes a TieX, it is not possible for MacroTest to justify macro values back through the latch. If in doubt, when preparing to MacroTest any black box, issue the [set\\_macrotest\\_options](#) command before exiting setup mode. No errors will occur because of this, even if none of the conditions requiring the command exist.

ATPG commands and options apply within MacroTest, including cell constraints, ATPG constraints, clock restrictions (it only pulses one clock per cycle), and others. If MacroTest fails and reports that it aborted, you can use the [set\\_abort\\_limit](#) command to get MacroTest to work harder, which may allow MacroTest to succeed. Mentor Graphics recommends that you set a moderate abort limit for a normal MacroTest run, then increase the limit if MacroTest fails and issues a message saying that a higher abort limit might help.

ATPG effort should match the simulation checks for bus contention to prevent MacroTest patterns from being rejected by simulation. Therefore, if you specify [set\\_contention\\_check](#) On, you should use the -Atpg option. Normally, if you use [set\\_contention\\_check](#) Capture\_clock, you should use the -Catpg option instead. Currently, MacroTest does not support the -Catpg option, so this is not advised. Using the [set\\_decision\\_order](#) Random is strongly discouraged. It can mislead the search and diagnosis in MacroTest.

In a MacroTest run, as each row is converted to a test, that test is stored internally (similar to a normal ATPG run). You can save the patterns to write out the tests in a desired format (perhaps Verilog to allow simulation and WGL for a tester). The tool supports the same formats for MacroTest patterns as for patterns generated by a normal ATPG run. However, because MacroTest patterns cannot be reordered, and because the expected macro output values are not saved with the patterns, it is not possible to read macrotest patterns back into the ATPG tool. You should generate Macrotest patterns, then save them in all desired formats.



### Note



The macro\_output node in the netlist must not be tied to Z (floating).

## MacroTest Examples

### Example 1 — Basic 1-Cycle Patterns

#### Verilog Contents:

```
RAM mem1 (.Dout ({ Dout[7],Dout[6],Dout[5],Dout[4],Dout[3],
Dout[2], Dout[1], Dout[0]}),
.RdAddr ({ RdAddr[1] , RdAddr[0] }) ,
.RdEn ( RdEn ) ,
.Din ({ Din[7] , Din[6] , Din[5] , Din[4] , Din[3] ,
Din[2] , Din[1] , Din[0] }) ,
.WrAddr ({ WrAddr[1] , WrAddr[0] }) , .WrEn ( WrEn ));
```

#### ATPG Library Contents:

```
model RAM (Dout, RdAddr, RdEn, Din, WrAddr, WrEn) (
  input (RdAddr,WrAddr) (array = 1 : 0;)
  input (RdEn,WrEn) ()
  input (Din) (array = 7 : 0;)
  output (Dout) (
    array = 7 : 0;
    data_size = 8;
    address_size = 2;
    read_write_conflict = XW;
    primitive = _cram(,,
      _write {,,} (WrEn,,WrAddr,Din),
      _read {,,,} (,RdEn,,RdAddr,Dout)
    );
  )
)
```

### Note



Vectors are treated as expanded scalars.

Because Dout is declared as “array 7:0”, the string “Dout” in the port list is equivalent to “Dout<7> Dout<6> Dout<5> Dout<4> Dout<3> Dout<2> Dout<1> Dout<0>”. If the declaration of Dout had been Dout “array 0:7”, then the string “Dout” would be the reverse of the above expansion. Vectors are always allowed in the model definitions. Currently, vectors are not allowed in the macrotest input patterns file, so if you redefine the pin order in the header of that file, scalars must be used. Either “Dout<7>”, “Dout(7)”, or “Dout[7]” can be used to match a bit of a vector.

### Dofile Contents:

```
set_system_mode analysis
macrotest mem1 ram_patts2.pat
write_patterns results/pattern2.f -replace
```

### Test File Input (ram\_patts2.pat) Contents:

```
// model RAM (Dout, RdAddr, RdEn, Din, WrAddr, WrEn) (
//   input (RdAddr,WrAddr) (array = 1 : 0;)
//   input (RdEn,WrEn) (
//   input (Din) (array = 7 : 0;)
//
//   output (Dout) (
//     array = 7 : 0;
//     data_size = 8;
//     address_size = 2;
//     .....
// Write V1 (data vector 1) to address 0.  Data Outputs
// and Read Address are Don't Cares.
XXXXXXXX XX 0 10101010 00 P
// Read V1 from address 0.  Data Inputs and Write Address
// are Don't Cares.
HLHLHLHL 00 1 XXXXXXXX XX 0
XXXXXXXX XX 0 0x010101 01 P    // Write V2 to address 1.
LXLHLHLH 01 1 xxxxxxxx xx 0    // Read V2 from address 1.
```

### Converted Test File Output (results/pattern2.f) Contents:

```
... skipping some header information ....
SETUP =
  declare input bus "PI" = "/clk", "/Datsel",
    "/scanen_early", "/scan_in1", "/scan_en",
.... skipping some declarations ....

declare output bus "PO" = "/scan_out1";
.... skipping some declarations ....

CHAIN_TEST =
  pattern = 0;
  apply "grp1_load" 0 =
    chain "chain1" = "0011001100110011001100";
  end;
  apply "grp1_unload" 1 =
    chain "chain1" = "0011001100110011001100";
  end;
end;

SCAN_TEST =

  pattern = 0 macrotest ;
  apply "grp1_load" 0 =
    chain "chain1" = "0110101010000000000000";
  end;
  force  "PI" "001X0XXXXXXXXX" 1;
  pulse  "/scanen_early" 2;
```

```

measure "PO" "1" 3;
pulse "/clk" 4;
apply "grpl_unload" 5 =
    chain "chain1" = "XXXXXXXXXXXXXXXXXXXXXXXXX";
end;

pattern = 1 macrotest ;
apply "grpl_load" 0 =
    chain "chain1" = "1000000000000000000000";
end;
force "PI" "001X0XXXXXXXX" 1;
measure "PO" "1" 2;
pulse "/clk" 3;
apply "grpl_unload" 4=
    chain "chain1" = "XXXXXXXXXXXXXXXXX10101010";
end;
... skipping some output ...

SCAN_CELLS =
    scan_group "grpl" =
        scan_chain "chain1" =
            scan_cell = 0 MASTER FFFF "/rden_reg/ffdpb0"...
            scan_cell = 1 MASTER FFFF "/wren_reg/ffdpb0"...
            scan_cell = 2 MASTER FFFF "/datreg1/ffdpb7"...
            ... skipping some scan cells ...
            scan_cell = 20 MASTER FFFF "/doutreg1/ffdpb1"...
            scan_cell = 21 MASTER FFFF "/doutreg1/ffdpb0"...
        end;
    end;
end;

```

## Example 2— Synchronous Memories (1- & 2-Cycle Patterns)

### Verilog Contents:

For this example, the RAM is as before, except a single clock is connected to an edge-triggered read and edge-triggered write pin of the macro to be tested. It is also the clock going to the MUX scan chain. There is also a separate write enable. As a result, it is possible to write using a one-cycle pattern, and then to preserve the data written during shift by turning the write enable off in the shift procedure. However, for this example, a read must be done in two cycles—one to pulse the RAM's read enable and make the data come out of the RAM, and another to capture that data into the scan chain before shifting changes the RAM's output values. There is no independent read enable to protect the outputs during shift, so they must be captured before shifting, necessitating a 2 cycle read/observe.

### ATPG Library Contents:

```
model RAM (Dout, RdAddr, RdClk, Din, WrAddr, WrEn, WrClk) (  
    input (RdAddr,WrAddr) (array = 1 : 0;)  
    input (RdClk,WrEn, WrClk) ()  
    input (Din) (array = 7 : 0;)  
    output (Dout) (  
        array = 7 : 0;  
        data_size = 8;  
        edge_trigger = rw;  
        address_size = 2;  
        read_write_conflict = XW;  
        primitive = _cram(,,  
            _write {,,} (WrClk,WrEn,WrAddr,Din),  
            _read {,,,} (,RdClk,,RdAddr,Dout)  
        );  
    )  
)
```

Note that because the clock is shared, it is important to only specify one of the macro values for RdClk or WrClk, or to make them consistent. X means “Don’t Care” on macro inputs, so it will be used to specify one of the two values in all patterns to ensure that any external embedding can be achieved. It is easier to not over-specify MacroTest patterns, which allows using the patterns without having to discover the dependencies and change the patterns.

### Dofile Contents:

```
set_system_mode analysis  
macrotest mem1 ram_patts2.pat  
write_patterns results/pattern2.f -replace
```

### Test File Input (ram\_patts2.pat) Contents:

```
// model RAM (Dout, RdAddr, RdClk, Din, WrAddr, WrEn, WrClk) (  
//   input (RdAddr,WrAddr) (array = 1 : 0;)  
//   input (RdClk,WrEn, WrClk) ()  
//   input (Din) (array = 7 : 0;)  
//  
//   output (Dout) (  
//     array = 7 : 0;  
//     data_size = 8;  
//     edge_trigger = rw;  
//     .....  
// Write V1 (data vector 1) to address 0.  
XXXXXXXX XX X 10101010 00 1 P  
// Read V1 from address 0 -- next 2 rows (1 row per cycle).  
XXXXXXXX 00 P XXXXXXXX XX 0 X + // + indicates another cycle.  
HLHLHLHL XX X XXXXXXXX XX 0 X // Values observed this cycle.  
XXXXXXXX XX X 01010101 01 1 P // Write V2 to address 1.  
xxxxxxxx 01 P xxxxxxxx xx 0 X + // Read V2,address 1,cycle 1.  
LHLHLHLH XX X XXXXXXXX XX 0 X // Read V2,address 1,cycle 2.
```

### Converted Test File Output (results/pattern2.f) Contents:

```
... skipping some header information ....
```

```

SETUP =
    declare input bus "PI" = "/clk", "/Datsel",
        "/scanen_early", "/scan_in1", "/scan_en",
    .... skipping some declarations ....

declare output bus "PO" = "/scan_out1";
    .... skipping some declarations ....

CHAIN_TEST =
    pattern = 0;
    apply "grp1_load" 0 =
        chain "chain1" = "0011001100110011001100";
    end;
    apply "grp1_unload" 1 =
        chain "chain1" = "0011001100110011001100";
    end;
end;

SCAN_TEST =

    pattern = 0 macrotest ;
    apply "grp1_load" 0 =
        chain "chain1" = "0110101010000000000000";
    end;
    force "PI" "001X0XXXXXXXX" 1;
    pulse "/scanen_early" 2;
    measure "PO" "1" 3;
    pulse "/clk" 4;
    apply "grp1_unload" 5 =
        chain "chain1" = "XXXXXXXXXXXXXXXXXXXXXXXX";
    end;

    pattern = 1 macrotest ;
    apply "grp1_load" 0 =
        chain "chain1" = "1000000000000000000000";
    end;
    force "PI" "001X0XXXXXXXX" 1;
    pulse "/clk" 2;
    force "PI" "001X0XXXXXXXX" 3;
    measure "PO" "1" 4;
    pulse "/clk" 5;
    apply "grp1_unload" 6=
        chain "chain1" = "XXXXXXXXXXXXXXXX10101010";
    end;
    ... skipping some output ...

```

## Example 3 — Using Leading Edge & Trailing Edge Observation Only

Assume that a clock with an off value of 0 (positive pulse) is connected through buffers to a rising edge read input of a macro, and also to both rising and falling edge D flip-flops. Either of the flip-flops can capture the macro's output values for observation. If you specify that the outputs should be captured in the same cycle as the read pulse, then this will definitely occur if you invoke MacroTest with the `-Te_observation_only` switch because only the trailing edge (TE) flip-flops will be selected for observation. The rising edge of the clock triggers the macro's

read, the values propagate to the scan cells in that same cycle, and then the falling edge of the clock captures those values in the TE scan cells.

On the other hand, if you invoke MacroTest with the `-Le_observation_only` switch and indicate in the MacroTest patterns that the macro's outputs should be observed in the cycle after pulsing the read pin on the macro, the rising edge of one cycle would cause the read of the macro, and then the rising edge on the next cycle would capture into the TE scan cells.

These two command switches (`-Te_observation_only` and `-Le_observation_only`) ensure that MacroTest behaves in a manner that is compatible with the particular macro and its embedding. In typical cases, only one kind of scan cell is available for observation and the MacroTest patterns file would, of course, need to be compatible. These options are only needed if both polarities of scan cells are possible observation sites for the same macro output pin.

For additional information on the use of these switches, refer to the [macrotest](#) description in the *Tessent Shell Reference Manual*.

## Verifying Test Patterns

After testing the functionality of the circuit with a simulator, and generating the test vectors with the ATPG tool, you should run the test vectors in a timing-based simulator and compare the results with predicted behavior from the ATPG tools. This run will point out any functionality discrepancies between the two tools, and also show timing differences that may cause different results. The following subsections further discuss the verification you should perform.

## Simulating the Design with Timing

At this point in the design process, you should run a full timing verification to ensure a match between the results of golden simulation and ATPG. This verification is especially crucial for designs containing asynchronous circuitry. You should have already saved the generated test patterns with the [write\\_patterns](#) command. The tool saved the patterns in parallel unless you used the `-Serial` switch to save the patterns in series. You can reduce the size of a serial pattern file by using the `-Sample` switch; the tool then saves samples of patterns for each pattern type, rather than the entire pattern set (except MacroTest patterns, which are not sampled nor included in the sampled pattern file). This is useful when you are simulating serial patterns because the size of the sampled pattern file is reduced and thus, the time it takes to simulate the sampled patterns is also reduced.

---

### Note



Using the `-Start` and `-End` switches will limit file size as well, but the portion of internal patterns saved will not provide a very reliable indication of pattern characteristics when simulated. Sampled patterns will more closely approximate the results you would obtain from the entire pattern set.

---

If you selected -Verilog as the format in which to save the patterns, the application automatically creates a test bench that you can use in a timing-based simulator such as ModelSim to verify that the tool-generated vectors behave as predicted by the ATPG tools.

For example, assume you saved the patterns generated as follows:

```
ANALYSIS> write_patterns pat_parallel.v -verilog -replace
```

The tool writes the test patterns out in one or more pattern files and an enhanced Verilog test bench file that instantiates the top level of the design. These files contain procedures to apply the test patterns and compare expected output with simulated output.

After compiling the patterns, the scan-inserted netlist, and an appropriate simulation library, you simulate the patterns in a Verilog simulator. If there are no mismatches between the ATPG tool's expected values and the values produced by the simulator, a message reports that there is "no error between simulated and expected patterns." If any of the values do not match, a simulation mismatch has occurred and must be corrected before you can use the patterns on a tester.

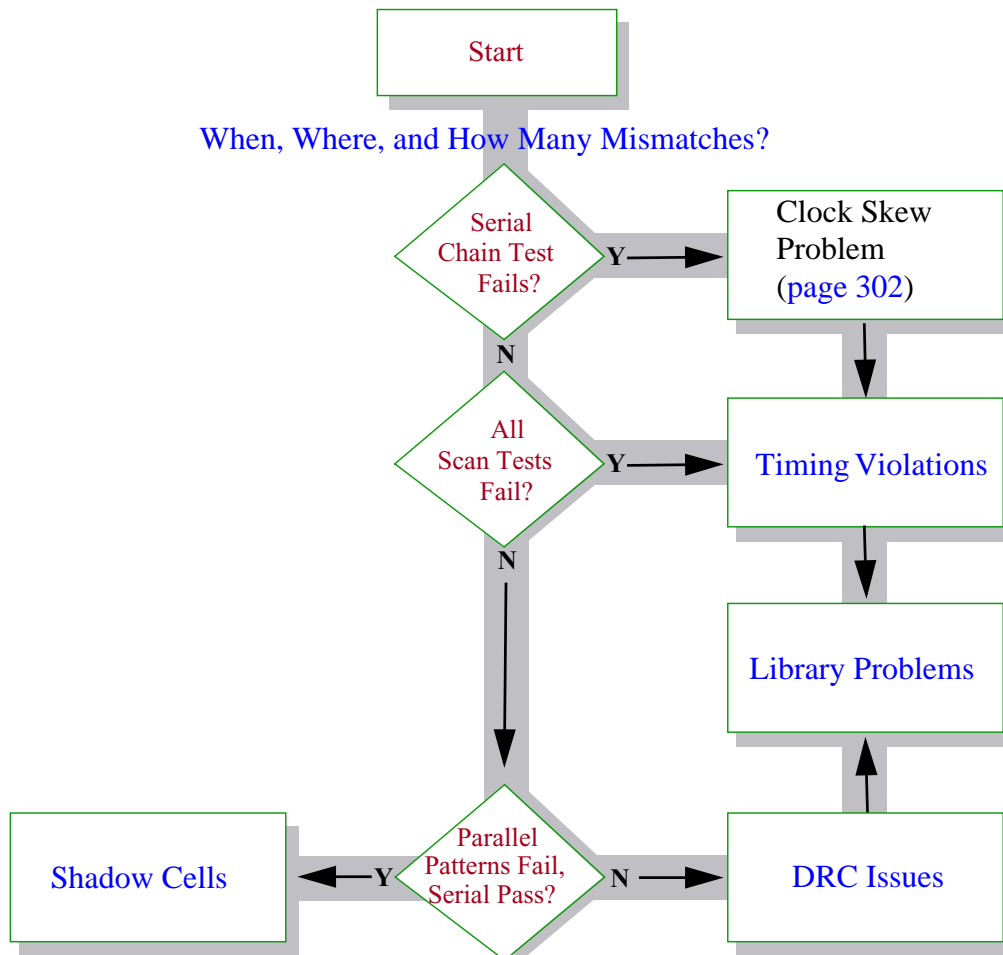
Be sure to simulate parallel patterns and at least a few serial patterns. Parallel patterns simulate relatively quickly, but do not detect problems that occur when data is shifted through the scan chains. One such problem, for example, is data shifting through two cells on one clock cycle due to clock skew. Serial patterns can detect such problems. Another reason to simulate a few serial patterns is that correct loading of shadow or copy cells depends on shift activity. Because parallel patterns lack the requisite shift activity to load shadow cells correctly, you may get simulation mismatches with parallel patterns that disappear when you use serial patterns. Therefore, always simulate at least the chain test or a few serial patterns in addition to the parallel patterns.

For a detailed description of the differences between serial and parallel patterns, refer to the first two subsections under "[Pattern Formatting Issues](#)" on page 344. See also "[Sampling to Reduce Serial Loading Simulation Time](#)" on page 346 for information on creating a subset of sampled serial patterns. Serial patterns take much longer to simulate than parallel patterns (due to the time required to serially load and unload the scan chains), so typically only a subset of serial patterns is simulated.

## Debugging Simulation Mismatches

Simulation mismatches can have any number of causes; consequently, the most challenging part of troubleshooting them is knowing where to start. Because a lot of information is available, your first step should be to determine the likeliest potential source of the mismatch. [Figure 6-50](#) is a suggested flow to help you begin this process.

**Figure 6-50. Mismatch Diagnosis Guidelines**



If you are viewing this document online, you can click on the links in the figure to see more complete descriptions of issues often at the root of particular mismatch failures. These issues are discussed in the following sections:

- [When, Where, and How Many Mismatches?](#)
- [DRC Issues](#)
- [Shadow Cells](#)
- [Library Problems](#)
- [Timing Violations](#)
- [Analyzing the Simulation Data](#)
- [Analyzing Patterns](#)
- [Checking for Clock-Skew Problems with Mux-DFF Designs](#)



## When, Where, and How Many Mismatches?

If DRC violations do not seem to be a problem, you need to take a closer look at the mismatches and check the following:

- **Are the mismatches reported on primary outputs (POs), scan cells or both?** Mismatches on scan cells can be related to capture ability and timing problems on the scan cells. For mismatches on primary outputs, the issue is more likely to be related to an incorrect value being loaded into the scan cells.
- **Are the mismatches reported on just a few or most of the patterns?** Mismatches on a few patterns indicates a problem that is unique to certain patterns, while mismatches on most patterns indicate a more generalized problem.
- **Are the mismatches observed on just a few pins/cells or most pins/cells?** Mismatches on a few pins/cells indicates a problem related to a few specific instances or one part of the logic, while mismatches on most patterns indicate that something more general is causing the problem.
- **Do both the serial and the parallel test bench fail or just one of them?** A problem in the serial test bench only, indicates that the mismatch is related to shifting of the scan chains (for example, data shifting through two cells on one clock cycle due to clock skew). The problem with shadows mentioned in the preceding section, causes the serial test bench to pass and the parallel test bench to fail.
- **Does the chain test fail?** As described above, serial pattern failure can be related to shifting of the scan chain. If this is true, the chain test (which simply shifts data from scan in to scan out without capturing functional data) also fails.
- **Do only certain pattern types fail?** If only ram sequential patterns fail, the problem is most certainly related to the RAMs (for instance incorrect modeling). If only clock\_sequential patterns fail, the problem is probably related to non-scan flip-flops and latches.

## DRC Issues

The DRC violations that are most likely to cause simulation mismatches are:

- C6
- T24

For details on these violations, refer to “[Design Rule Checking](#)” in the *Tessent Shell Reference Manual* and SupportNet KnowledgeBase TechNotes describing each of these violations. For most DRC-related violations, you should be able to see mismatches on the same flip-flops where the DRC violations occurred.

You can avoid mismatches caused by the C6 violation by enabling the [set\\_clock\\_off\\_simulation](#) command.

## Shadow Cells

Another common problem is shadow cells. Such cells do not cause DRC violations, but the tool issues the following message when going into analysis mode:

```
// 1 external shadows that use shift clocking have been identified.
```

A shadow flip-flop is a non-scan flip-flop that has the D input connected to the Q output of a scan flip-flop. Under certain circumstances, such shadow cells are not loaded correctly in the parallel test bench. If you see the above message, it indicates that you have shadow cells in your design and that they may be the cause of a reported mismatch. For more information about shadow cells and simulation mismatches, consult the online SupportNet KnowledgeBase. Refer to [“Mentor Graphics Support”](#) on page 445 for information about SupportNet.

## Library Problems

A simulation mismatch can be related to an incorrect library model; for example, if the reset input of a flip-flop is modeled as active high in the analysis model used by the tool, and as active low in the Verilog model used by the simulator. The likelihood of such problems depends on the library. If the library has been used successfully for several other designs, the mismatch probably is caused by something else. On the other hand, a newly developed, not thoroughly verified library could easily cause problems. For regular combinational and sequential elements, this causes mismatches for all patterns, while for instances such as RAMs, mismatches only occur for a few patterns (such as RAM sequential patterns).

Another library-related issue is the behavior of multi-driven nets and the fault effect of bus contention on tristate nets. The ATPG tool is conservative by default, so non-equal values on the inputs to non-tristate multi-driven nets, for example, always results in an X on the net. For additional information, see the [set\\_net\\_resolution](#) and [set\\_net\\_dominance](#) commands.

## Timing Violations

Setup and hold violations during simulation of the test bench can indicate timing-related mismatches. In some cases, you see such violations on the same scan cell that has reported mismatches; in other cases, the problem might be more complex. For instance, during loading of a scan cell, you may observe a violation as a mismatch on the cell(s) and PO(s) that the violating cell propagates to. Another common problem is clock skew. This is discussed in the section, [“Checking for Clock-Skew Problems with Mux-DFF Designs.”](#)

Another common timing related issue is that the timeplate and/or test procedure file has not expanded. By default, the test procedure and timeplate files have one “time unit” between each event. When you create test benches using the -Timingfile switch with the [write\\_patterns](#) command, the time unit expands to 1000 ns in the Verilog test benches. When you use the default -Procfile switch and a test procedure file with the [write\\_patterns](#) command, each time unit in the timeplate is translated to 1 ns. This can easily cause mismatches.

## Analyzing the Simulation Data

If you still have unresolved mismatches after performing the preceding checks, examine the simulation data thoroughly and compare the values observed in the simulator with the values expected by the tool. The process you would use is very similar in the Verilog test benches.

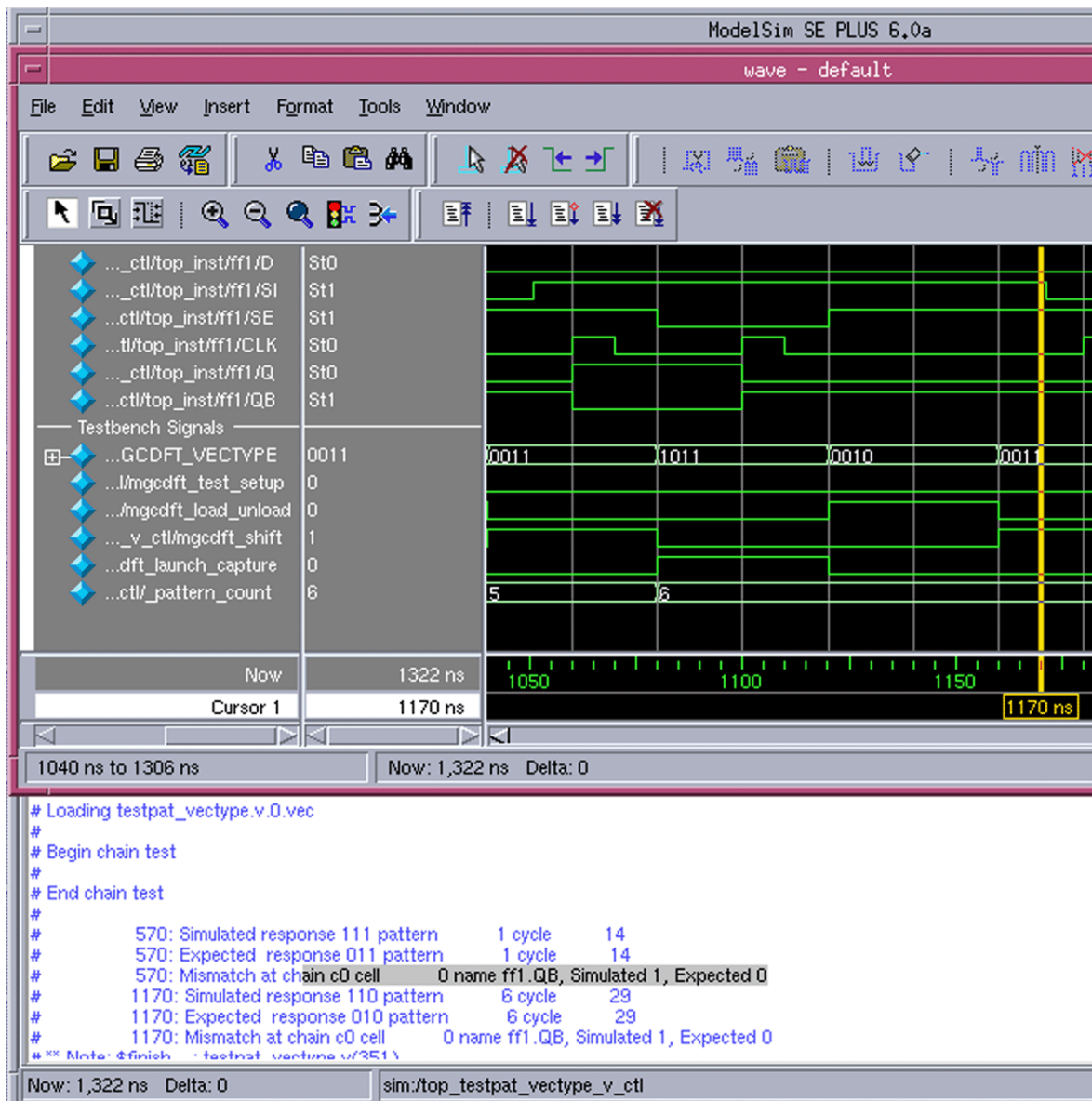
## Resolving Mismatches Using Simulation Data

When simulated values do not match the values expected by the ATPG tool, the enhanced Verilog parallel test bench reports the time, pattern number, and scan cell or primary output where each mismatch occurred. The serial test bench reports only output and time, so it is more challenging to find the scan cell where the incorrect value has been captured.

Based on the time and scan cell where the mismatch occurred, you can generate waveforms or dumps that display the values just prior to the mismatch. You can then compare these values to the values that the tool expected. With this information, you can trace back in the design (in both the ATPG tool and the simulator) to see where the mismatch originates.

When comparing Verilog simulation data to ATPG tool data, it is helpful to use the `SIM_VECTYPE_SIGNAL` keyword in the parameter file. When this keyword is used, the Verilog test bench will include additional keywords that makes it easier for you to understand the sequence of events in the test bench, and also how to compare data between the ATPG tool and the simulator. In the example simulation transcript shown below, a mismatch is reported for patterns 1 and 6. For pattern 6, the mismatch is reported at time 1170.

Figure 6-51. Simulation Transcript



Note that the waveforms include the values for the flip-flop with mismatches, as well as the signals *mgcdft\_shift*, *mgc\_dft\_launch\_capture*, and *\_pattern\_count*. Note that the *\_pattern\_count* variable will increment just prior to the capture procedure. That means that when *\_pattern\_count* variable is 6 and *mgcdft\_shift* is 1, data is shifted out for pattern 6 (and shifted in for pattern 7). By using these signals as a guide, you can see that the time of the mismatch is during the shift procedure after pattern 6. Note that the capture for pattern 6 occurs between time 1080 and 1120, when the *mgcdft\_launch\_capture* signal is high.

To see the corresponding data in the ATPG tool, you can use the [set\\_gate\\_report](#) <pattern\_index> command to see data for pattern 6. The data shown in the ATPG tool corresponds to the state just before the capture clock pulse. In this example, for pattern 6, the

ATPG tool shows data corresponding to the simulation values just prior to the clock pulse at time 1100 in the waveform. A detailed example showing this process for a Verilog test bench is contained in AppNote 3002, available on <http://supportnet.mentor.com>.

## Analyzing Simulation Mismatches

By default, the simulation VCD file, test patterns, and design data are all analyzed and the mismatch sources are identified. Once the analysis is complete, you can use DFTVisualizer to graphically display the overlapping data and pinpoint the source of each mismatch.

The ATPG simulation mismatch analysis functionality is enhanced to optimize the debugging of large (100K gates and more) designs and to support all simulators and distributed processing. Once analysis is complete, DFTVisualizer graphically displays the source of the mismatches for easy identification.

## Understanding the Simulation Mismatch Analysis Flow

In general, the simulation mismatch analysis flow includes the following stages:

- [Stage 1 — ATPG](#)
- [Stage 2 — Verilog test bench simulation](#)
- [Stage 3 — Debug test bench generation](#)
- [Stage 4 — Debug test bench simulation](#)
- [Stage 5 — Mismatch source identification](#)

You can analyze simulation mismatches using the following methods:

- [Automatically Analyzing Simulation Mismatches](#) — Using this method, the tool runs the entire flow, from [Stage 1 — ATPG](#) through [Stage 5 — Mismatch source identification](#) using a single command invocation— see [Figure 6-52](#).
- [Manually Analyzing Simulation Mismatches](#) — Using this method, you can run mismatch analysis flow in steps. For example, you correct known issues in your failure file and, instead of re-running [Stage 2 — Verilog test bench simulation](#), you can proceed to [Stage 3 — Debug test bench generation](#) using the modified failure file.

Both the automatic and manual flows follow the identical stages: the key difference is the automatic flow runs the *entire* flow while the manual flow allows you to use the flow in steps. The following sections cover the simulation mismatch analysis flow in detail.

---

### Note



This procedure does not support debugging MacroTest or chain test patterns.

---

## Stage 1 — ATPG

Generate the flattened model, test patterns, and Verilog test bench (*mentor\_default.v*). The test patterns must be saved in a format that can be read back into the ATPG tool (binary, ASCII, STIL, or WGL), and the test bench must be able to generate a failure file.

## Stage 2 — Verilog test bench simulation

Simulating the Verilog test bench generates a failure file (*mentor\_default.v.fail*). If no simulation mismatches are found, the automatic simulation mismatch analysis stops.

## Stage 3 — Debug test bench generation

Generate a test bench for just the failing patterns (*mentor\_default.v\_vcdtb.v*) by using the ATPG tool to read the flattened netlist and the read in the test patterns generated in Stage 1 and the failure file from Stage 2. The test bench is set up to output a simulation results VCD file.

## Stage 4 — Debug test bench simulation

In the same ATPG tool session, simulate the debug test bench generated in Stage 3. This simulation produces the VCD file (*mentor\_default.v\_debug.vcd*).

## Stage 5 — Mismatch source identification

Load the VCD file and trace each mismatch to its source. Then report the mismatch sources.

---

### Note



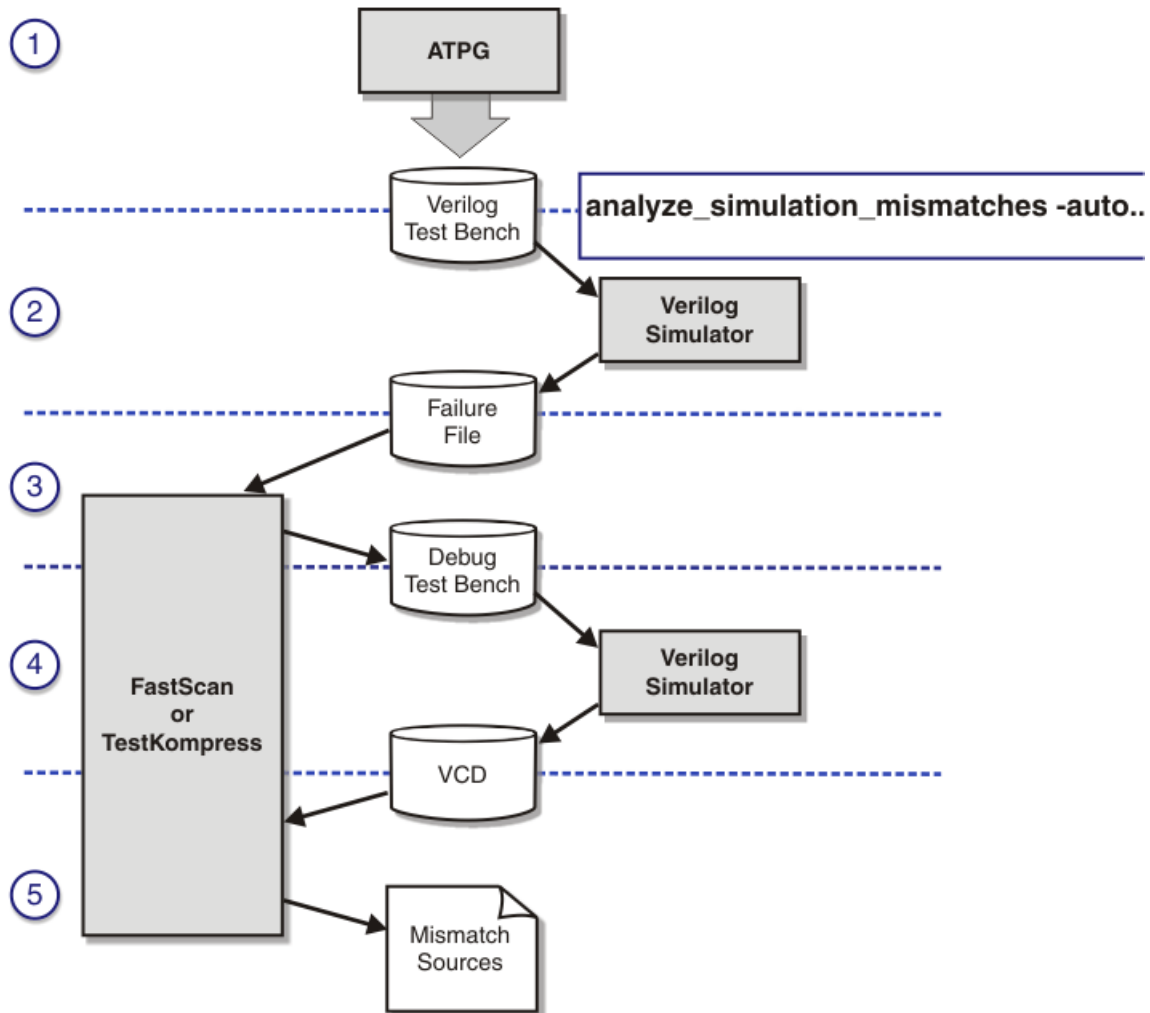
All generated files are placed in the *work\_dft\_debug* directory inside your working directory. This directory is created if it does not already exist.

---

## Automatically Analyzing Simulation Mismatches

Figure 6-52 shows the automatic simulation mismatch analysis flow. Using this procedure, you invoke the tool using the [analyze\\_simulation\\_mismatches](#) command with the -auto switch, and the tool automatically runs the all stages of the flow. This procedure supports using a third-party simulator and distributing processing to a remote server.

Figure 6-52. Automatic Simulation Mismatch Analysis Flow




## Prerequisites

- (Optional) If you are using an external third-party simulator, you must create a script to invoke, setup, and run the simulator. See [“Example 1 External Simulator Script”](#) on page 301.
- (Optional) If you want to distribute the simulation part of the analysis, you must have access to a remote server that can be accessed through rsh. For more information, see the [analyze\\_simulation\\_mismatches](#) description in the *Tessent Shell Reference Manual*.

## Procedure

---

 **Note** All generated files are placed in the *work\_dft\_debug* directory inside your working directory. This directory is created if it does not already exist.

---

1. Use the ATPG tool to read a design netlist or flat model. For example:

```
$ tesseract -shell
```

```
SETUP> set_context patterns -scan
```

```
SETUP> read_verilog data/design.v
```

2. Specify the scan data for the scan cells in the design and switch to analysis system mode. For example:

```
SETUP> add_scan_groups ...
```

```
SETUP> add_scan_chains ...
```

```
SETUP> add_clocks ...
```

```
SETUP> set_system_mode analysis
```

3. Specify the source test patterns for the design. For example:

```
ANALYSIS> read_patterns pats/testpat.bin
```

4. If you are using a third-party simulator, specify your simulator invoke script. For example:

```
ANALYSIS> set_external_simulator -simulation_script runsim
```

For more information, see [Example 1 External Simulator Script](#) and the [set\\_external\\_simulator](#) description in the *Tessent Shell Reference Manual*.

5. Run the automatic mismatch analysis. For example:

```
ANALYSIS> analyze_simulation_mismatches -auto -external_patterns
```

By default, the analysis runs on the local server. To run the simulation portion of the analysis on a remote server, use the `-host` option. For example:

```
ANALYSIS> analyze_simulation_mismatches -auto -external_patterns -host abc_test
```

For more information, see the [analyze\\_simulation\\_mismatches](#) description in the *Tessent Shell Reference Manual*.

By default, the analysis compares the specified failure file to the current test pattern source to verify that both are generated from the same version of the design. If files do not match, an error displays and the process aborts.

Once the test patterns and failure file pass the verification, a test bench is created specifically for the mismatches in the failure file and simulated. The simulation results



are compared with the test patterns and design data to determine the source of simulation mismatches listed in the failure file.

6. Open DFTVisualizer to view and further debug the mismatches. For example:

```
ANALYSIS> open_visualizer
```

7. Click **Debug Simulation Mismatches**. The Select a Mismatch ID dialog box displays.
8. Select the ID for the mismatch to debug, and click **Analyze**.

DFTVisualizer displays and highlights overlapping design, simulation, and test pattern data for the selected simulation mismatch.

---

**Note**

In the Debug window, pin names assigned to the value “.” indicate that the VCD debug test bench did not capture the VCD value for that location. It is not possible to capture VCD values for every node in the design due to very large file sizes and run time; the test bench only captures values in the combination cone behind the failure location, which in most cases provides sufficient information to explain the mismatch.

For a complete list of possible pin name values resulting from simulation mismatches, see the [report\\_mismatch\\_sources](#) description in the *Tessent Shell Reference Manual*.

---



**Tip:** Optionally, you can use the [report\\_mismatch\\_sources](#) command to replace steps 6 through 8 and automatically display the mismatches in DFTVisualizer.

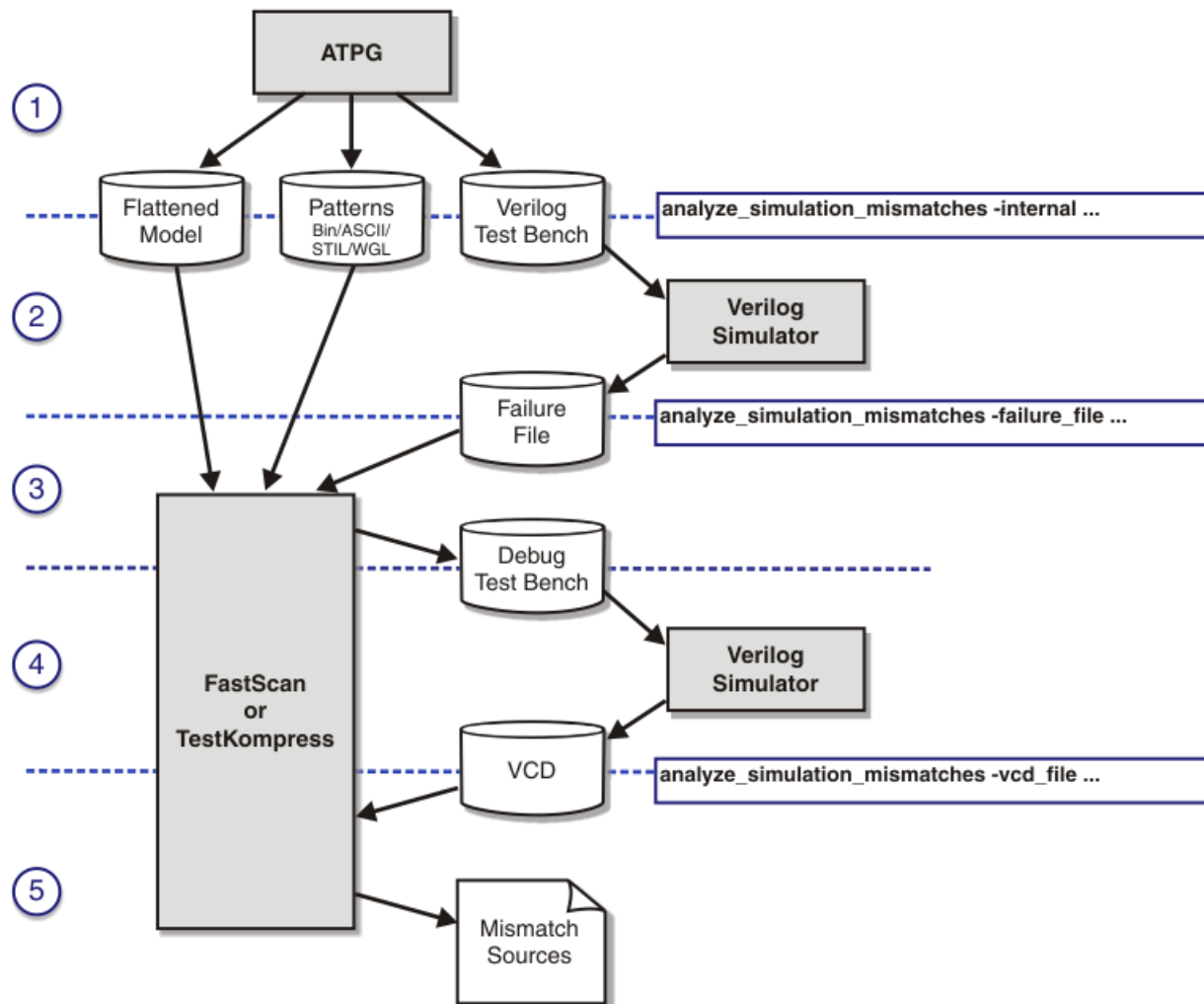
---

## Manually Analyzing Simulation Mismatches

Use this procedure to manually debug simulation mismatches. With this procedure, you manually perform the simulation mismatch analysis using the [analyze\\_simulation\\_mismatches](#) command. This procedure supports using a third-party simulator and distributing processing to a remote server.

[Figure 6-53](#) shows the manual simulation mismatch analysis flow.

Figure 6-53. Manual Simulation Mismatch Analysis Flow



## Prerequisites

- (Optional) If you are using an external third-party simulator, you must create a script to invoke, setup, and run the simulator. See “[Example 1 External Simulator Script](#)” on page 301.
- (Optional) If you want to distribute the simulation part of the analysis, you must have access to a remote server that can be accessed through rsh. For more information, see the [analyze\\_simulation\\_mismatches](#) description in the *Tessent Shell Reference Manual*.
- A design netlist or flat model and the associated test patterns are available.
- Test patterns must have been verified and mismatches exist.
- In order to generate a failure file for a manually generated test bench, you must set the “[SIM\\_DIAG\\_FILE](#)” parameter file keyword to 2 or 1(default) prior to ATPG.

- For a manual simulation, you must set the “\_write\_DIAG\_file” parameter to 1 in the Verilog test bench to generate the failure file. This is done automatically if you set the SIM\_DIAG\_FILE parameter file keyword to 2 prior to ATPG.

## Procedure

### Note



All generated files are placed in the *work\_dft\_debug* directory inside your working directory. This directory is created if it does not already exist.

---

1. Use the ATPG tool to read a design netlist or flat model. For example:

```
$ tesseract -shell
```

```
SETUP> set_context patterns -scan
```

```
SETUP> read_verilog data/design.v
```

2. Specify the scan data for the scan cells in the design and switch to analysis system mode. For example:

```
SETUP> add_scan_groups ...
```

```
SETUP> add_scan_chains ...
```

```
SETUP> add_clocks ...
```

```
SETUP> set_system_mode analysis
```

3. Specify the source test patterns for the design. For example:

```
> read_patterns pats/testpat.bin
```

4. If you are using a third-party simulator, specify your simulator invoke script. For example:

```
> set_external_simulator -simulation_script runsim
```

For more information, see [Example 1 External Simulator Script](#) and the [set\\_external\\_simulator](#) description in the *Tesseract Shell Reference Manual*.

5. Run the mismatch analysis. For example:

```
> analyze_simulation_mismatches -external_patterns
```

By default, the analysis runs on the local server. To run the simulation portion of the analysis on a remote server, use the -host option. For example:

```
ANALYSIS> analyze_simulation_mismatches -external_patterns -host abc_test
```

For more information, see the [analyze\\_simulation\\_mismatches](#) description in the *Tesseract Shell Reference Manual*.

By default, the analysis compares the specified failure file to the current test pattern source to verify that both are generated from the same version of the design. If files do not match, an error displays and the process aborts.

Once the test patterns and failure file pass the verification, a test bench is created specifically for the mismatches in the failure file and simulated. The simulation results are compared with the test patterns and design data to determine the source of simulation mismatches listed in the failure file.

6. Simulate the Verilog test bench using ModelSim or a third-party simulator to create a failure file—see the [set\\_external\\_simulator](#) command for details.

You must also create a script to set up and run an external simulator for the subsequent steps of this procedure.

7. After simulation, perform a simulation mismatch analysis using the [analyze\\_simulation\\_mismatches](#) command with the -FAilure\_file switch and argument to generate a test bench for just the failing patterns. For example:

```
ANALYSIS> analyze_simulation_mismatches -failure_file pat.fs.v.fail
```

You must use the flattened netlist and test patterns from the initial ATPG session.

This step creates the test bench for just the failing patterns (*mentor\_default.v\_vcdtb.v*). The test bench is set up to output the simulation results to a VCD file.

8. In the same ATPG tool session, simulate the test bench for the failing patterns using the [analyze\\_simulation\\_mismatches](#) command with the -TEStbench\_for\_vcd switch argument. For example:

```
ANALYSIS> analyze_simulation_mismatches -testbench_for_vcd mentor_default.v_vcdtb.v
```

This step produces a VCD file (*mentor\_default.v\_debug.vcd*).

9. After simulation, load the VCD file. For example:

```
ANALYSIS> analyze_simulation_mismatches -vcd_file mentor_default.v_debug.vcd
```

10. Report the mismatch sources using the [report\\_mismatch\\_sources](#) command.

You can also view and further debug the mismatches using DFTVisualizer.

11. Open DFTVisualizer to view and further debug the mismatches. For example:

```
ANALYSIS> open_visualizer
```

12. Click **Debug Simulation Mismatches**. The Select a Mismatch ID dialog box displays.

13. Select the ID for the mismatch to debug, and click **Analyze**.

DFTVisualizer displays and highlights overlapping design, simulation, and test pattern data for the selected simulation mismatch.

## Example 1 External Simulator Script

The following example shows a simulation script for ModelSim where the design netlist and model library have been pre-compiled into the my\_work directory. The script will be used to compile and simulate the test bench that was saved using the write\_patterns command as well as the debug test bench created by the tool. Prior to execution of the script, the \${1} variable will be replaced by the tool with the name of the test bench being compiled and \${2} will be replaced by the test bench top level being simulated.

```
#!/bin/csh -f
vlog ${1} -work my_work
vsim ${2} -c -lib my_work -do "run -all" -sdfmax \
/${2}/design_inst=~ /design/good.sdf
```

If there are simulation mismatches in the first execution of the saved test bench, the analyze\_simulation\_mismatches command creates a new version of the test bench for the analysis. To create a simulation script that modifies this test bench, such as adding fixed force statements, you must substitute the variable \$ENTITY for the test bench name.

For example: force -freeze sim:\$entity/instance/pin0 1

The script must be executable otherwise the following error is returned:

**analyze\_simulation\_mismatches -simulation\_script vsim\_scr**

```
sh: line 1: ./vsim_scr: Permission denied
// Error: Error when running script: vsim_scr
work_dft_debug/mentor_default.v circle_mentor_default_v_ctl > /dev/null
// No mismatch source is found.
```

To correct this error, use the following Linux/UNIX command in the tool before running the analyze\_simulation\_mismatches command:

```
!chmod +x vsim_scr
```

## Analyzing Patterns

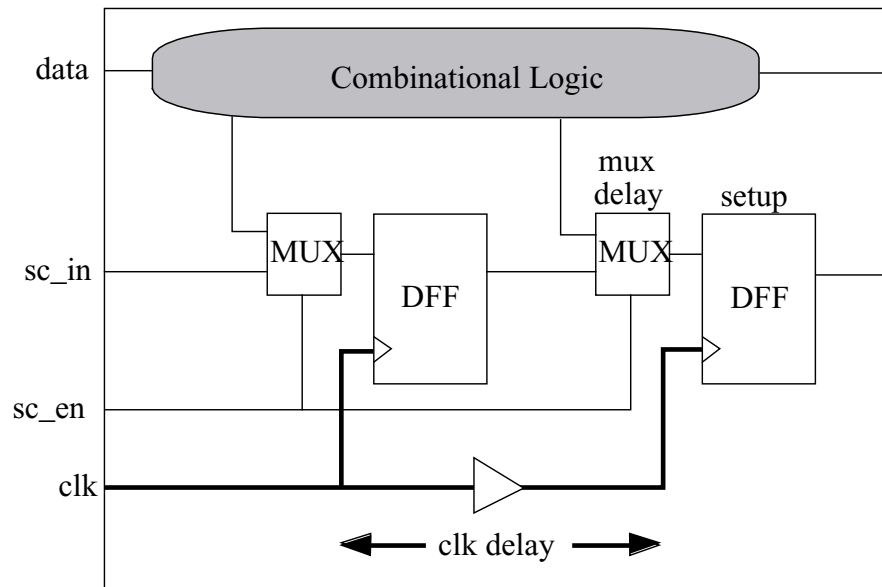
Sometimes, you can find additional information that is difficult to access in the Verilog test benches in other pattern formats. When comparing different pattern formats, it is useful to know that the pattern numbering is the same in all formats. In other words, pattern #37 in the ASCII pattern file corresponds to pattern #37 in the WGL or Verilog format.

Each of the pattern formats is described in detail in the section, “[Saving Patterns in Basic Test Data Formats](#),” beginning on [page 347](#).

## Checking for Clock-Skew Problems with Mux-DFF Designs

If you have mux-DFF scan circuitry in your design, you should be aware of, and thus test for, a common timing problem involving clock skew. Figure 6-54 depicts the possible clock-skew problem with the mux-DFF architecture.

**Figure 6-54. Clock-Skew Example**



You can run into problems if the clock delay due to routing, modeled by the buffer, is greater than the mux delay minus the flip-flop setup time. In this situation, the data does not get captured correctly from the previous cell in the scan chain and therefore, the scan chain does not shift data properly.

To detect this problem, you should run both critical timing analysis and functional simulation of the scan load/unload procedure. You can use ModelSim or another HDL simulator for the functional simulation, and a static timing analyzer such as SST Velocity for the timing analysis. Refer to the *ModelSim SE/EE User's Manual* or the *SST Velocity User's Manual* for details on performing timing verification.

# Chapter 7

## Multiprocessing for ATPG and Simulation

---

### Introduction

This chapter explains multiprocessing functionality for accelerating ATPG and simulation. Multiprocessing is a combination of distributed processing and multithreading. Multiprocessing functionality allows you to create slave processes with multiple threads to efficiently use additional processors.

### Definition of Multiprocessing Terms

This manual uses the following terminology:

- **Distribution or Distributed Processing** — The dividing up and simultaneous execution of processing tasks on multiple machines.
- **Multithreading** — The dividing up and simultaneous execution of processing tasks within one process running on one machine. This method minimizes memory use per thread by sharing the design information across all threads running on the same machine.
- **Multiprocessing** — A general term that can refer either to distribution or multithreading, or to a combination of both.
- **Manual Mode** — Starting additional processors by explicitly specifying the host(s) on which the processors run.
- **Process** — The instance of the executable running on the master or slave host, regardless of the number of threads in use.
- **Processor** — A resource that executes a thread, which is added with the `add_processors` command or by starting the tool.
- **Grid Mode** — Starting additional processors using the grid engine. You have limited control in specifying the host upon which the job is started because that is handled by the grid engine, subject to any constraints you place on the grid job submission.
- **Thread** — The smallest independent unit of a process. A process can consist of multiple threads, all of which share the same allocated memory and other resources on the same host.
- **Worker Thread** — Threads other than the main thread.

## Using Multiprocessing to Reduce Runtime

During ATPG and fault simulation, the tool can optionally use multiprocessing when executing the following commands:

- `compress_patterns` (fault simulation)
- `create_patterns` (ATPG)
- `identify_redundant_faults` (ATPG)
- `order_patterns` (fault simulation)
- `simulate_patterns` (fault simulation)

For large designs, distributing portions of these commands' processing load using multiprocessing can reduce their runtime, sometimes significantly. The reduction in runtime depends on several factors and is not directly proportional to the number of additional threads doing the processing. The particular design, the tool setups you use, and the type of patterns you generate, for example, determine the kinds of processes the tool must run and the proportion of them it can distribute to additional threads. Generally, the ATPG runtime improvement will be greater for transition patterns than for stuck-at.

Slave processors or threads can be on the machine on which the tool is running or on remote machines, wherever you can access additional processors on your network. ATPG results (coverage and patterns) using multiprocessing are the same as without multiprocessing, regardless of the number of processors.

## Multiprocessing Requirements

To enable the tool to establish and maintain communication with multiple processors on multiple host machines and run slave processing jobs on them, you need to inform the tool of the network names of available machines (manual specification) or direct the tool to use an automated job scheduler to select machines for you. The tool supports Load Sharing Function (LSF), Sun Grid Engine (SGE), or custom job schedulers. The following prerequisites must be satisfied for whichever you use:

- **Manual Specification (does not require a job scheduler)**

You can specify hosts manually, without using the SGE, LSF or a custom job scheduler. The master host must be able to create processes on the slave host via the **rsh** or **ssh** shell command.

- **rsh** — This requires that the network allow connection via **rsh**, and that your `.rhosts` file allow **rsh** access from the master host without specifying a password. This is the default.



**Note**



The `.rhosts` file on host machines must have read permission set for user. Write and execute permission can optionally be set for user, but must *not* be set for other and group.

---

**rsh** access is not required for the tool to create additional processes on the master host.

- **ssh** — This requires that the network allow connection via **ssh**. To enable use of **ssh**, issue a [set\\_multiprocessing\\_options](#) command within the tool to set the multiprocessing “remote\_shell” variable to **ssh**. Do this prior to issuing an [add\\_processors](#) command.

Master and slave machines must be correctly specified in the global DNS name server for reliable network operation, and you will need to know either the network name or IP address of each remote machine you plan to use. Consult the System Administrator at your site for additional information.

- **Job Scheduler** — You must have available at your site at least one of the following methods of network job scheduling. Whichever you use, it must allow the master process (the process started when you invoke the tool) to create slave processes on different host machines.
  - **Load Sharing Function (LSF)** — To use LSF, ensure your environment supports use of the LSF scheduler before you invoke the tool. For example, the `LSF_BINDIR` environment variable must be set appropriately, in addition to other requirements. An appropriate setup can often be performed by sourcing a configuration file supplied with the scheduler installation.
  - **Sun Grid Engine (SGE)** — To use SGE, ensure your environment supports use of the SGE scheduler before you invoke the tool. For example, the `SGE_ROOT` environment variable must be set appropriately, in addition to other requirements. An appropriate setup can often be performed by sourcing a configuration file supplied with the scheduler installation.
  - **Custom Job Scheduling** — For the tool to use a custom job scheduler, you need to inform the tool of the command used at your site to launch the custom job scheduler. You do this by issuing a [set\\_multiprocessing\\_options](#) command within the tool to set the “generic\_scheduler” variable to the appropriate site-specific command.
- **Job Scheduling Options** — You can control certain aspects of the job scheduling process with the switches you set with the [set\\_multiprocessing\\_options](#) command. For complete information about these switches, refer to the [set\\_multiprocessing\\_options](#) description in the *Tessent Shell Reference Manual*.
- **Tessent Software Tree Installation** — It must be possible to execute the Mentor Graphics Tessent executables (via fully specified paths) from any new processes.

- **Tool Versions** — All multiprocessing hosts must run the same version of the tool. This is not an issue when using a single executable for all slave hosts, but depending on the installation, may become an issue when the path to the executable points to a different physical disk location on a slave host than on the master host. The tool installation tree must be accessible via the same path on all slave hosts.

## Procedures for Multiprocessing

The following describes how to use multiprocessing functionality:

- [Overview Procedure for Using Multiprocessing](#)
- [Disabling Multithreading Functionality](#)
- [Troubleshooting SSH Environment and Passphrase Errors](#)
- [Adding Threads to the Master](#)
- [Adding Processors in Manual Mode](#)
- [Adding Processors in Grid Mode](#)
- [Adding Processors to the LSF Grid](#)
- [Deleting Processors Added in Manual Mode](#)
- [Deleting Processors Added in Grid Mode](#)

## Overview Procedure for Using Multiprocessing

The following describes the basic procedure for using multiprocessing for ATPG and fault simulation. These steps assume you have satisfied the “[Multiprocessing Requirements](#)” described in the preceding section.

1. Use the [set\\_multiprocessing\\_options](#) command to specify the values of any multiprocessing variables, such as the job scheduler you will be using or the remote shell. Manual specification requires this step only if you want the tool to use **ssh**.

Specific LSF and SGE options will depend on how SGE and LSF are configured at your site. It is a good idea to consult your System Administrator, as you may not need any options.

2. Use the [add\\_processors](#) command to define host machines and the number of processors that the tool can use for slave processes. For example:

```
SETUP> add_processors lsf:4 machineB:2
```

specifies for the tool to distribute the processing load among any four available LSF processors and two processors on the machine named “machineB.” Note that with the default settings there would be one slave with four threads from LSF, and one slave with

two threads from machineB. And without multithreading enabled, there would be four separate slaves from LSF, and two from machineB.

---

**Note**

If there are no slave processes running, the session consumes just one license. However, when you initiate slave processes using the [add\\_processors](#) command, the tool acquires additional licenses for these processes, with each additional license allowing up to four slave processors. The previous example adds six processors, so two additional licenses would be used.

---

3. Perform any other tool setups you need for ATPG or fault simulation, then issue the command to be multiprocessed. The tool displays a message indicating the design is being sent to slave processors and then let you know when the slave processors start participating.

The following is a list of related commands:

- [add\\_processors](#) — Runs multiple processors in parallel on multiple machines to reduce ATPG or fault simulation runtime.
- [delete\\_processors](#) — Removes processors previously defined using the [add\\_processors](#) command.
- [get\\_multiprocessing\\_option](#) — Returns the value of a single specified variable previously set with the [set\\_multiprocessing\\_options](#) command. This is an introspection command that returns a Tcl result.
- [report\\_multiprocessing\\_options](#) — Displays the values of all variables used when executing multiprocessing commands for ATPG or fault simulation.
- [report\\_processors](#) — Displays information about the processors used for the current multiprocessing environment.
- [set\\_multiprocessing\\_options](#) — Sets the values of one or more multiprocessing variables.

## Troubleshooting SSH Environment and Passphrase Errors

Use the information in this section to resolve problems that can occur if you attempt to use the SSH protocol when your environment has not set up the secure shell agent and password.

### Error: Cannot change from rsh to ssh in the same tool session

```
SETUP> report_processors
```

```
Error: Cannot change from rsh to ssh in the same tool session.
```

This error message indicates the `set_multiprocessing_options` command was not used to set the `remote_shell` variable to `ssh` *prior* to use of the `report_processors` or `add_processors` command.

## Error: Not running a secure shell agent (SSH\_AUTH\_SOCK does not exist, use ssh-agent)

```
SETUP> set_multiprocessing_options -remote_shell ssh
```

```
Permission denied (publickey,password,keyboard-interactive).  
// Error: Not running a secure shell agent ( SSH_AUTH_SOCK does not  
// exist, use ssh-agent ).
```

The SSH functionality checks for the presence of two environment variables, `SSH_AGENT_PID` and `SSH_AUTH_SOCK`, that describe your secure shell session. If not present, it indicates the required `ssh-agent` daemon is not running.

To fix this problem, suspend the tool session with Control-Z and run the **ssh-agent** shell program. This will start the agent and echo to the screen the required settings for the environment variables. For example:

```
ssh-agent  
setenv SSH_AUTH_SOCK /tmp/ssh-yXm13171/agent.13171;  
setenv SSH_AGENT_PID 13172;
```

---

### Note



Ensure you remove the trailing semicolons (;) if you copy and past the `ssh-agent` output from the shell environment when you resume the tool session.

---

You can then resume the suspended session and set the environment variables:

```
SETUP> setenv SSH_AUTH_SOCK /tmp/ssh-yXm13171/agent.13171  
SETUP> setenv SSH_AGENT_PID 13172
```

Then attempt to set `remote_shell` again.

## Error: ssh-agent requires passphrase (use ssh-add)

```
SETUP> set_multiprocessing_options -remote_shell ssh
```

```
Permission denied (publickey,password,keyboard-interactive).  
// Error: ssh-agent requires passphrase (use ssh-add).  
// Note: SSH protocol not engaged.
```

This error message indicates the SSH agent is running but has not been told the passphrase to allow SSH operations.

To fix this problem, suspend the tool session with Control-Z and run the **ssh-add** shell program:

```
SETUP> ^Z  
Stopped (user)
```

```
% ssh-add
Could not open a connection to your authentication agent.
% setenv SSH_AUTH_SOCK /tmp/ssh-yXm13171/agent.13171
% setenv SSH_AGENT_PID 13172
% ssh-add
Enter passphrase for /user/.ssh/id_dsa:
```

Enter a passphrase:

```
Enter passphrase for /user/.ssh/id_dsa: xxxxxxxxxxxx
Identity added: /user/.ssh/id_dsa (/user/royb/.ssh/id_dsa)
%
```

Then resume the session and attempt to set `remote_shell` again:

```
SETUP> set_multiprocessing_options -remote_shell ssh

// Note: SSH protocol is engaged.
```

## Disabling Multithreading Functionality

Multithreading functionality is enabled by default. However, if the tool reads a flat model produced by the pre-v2013.2 simulation kernel, the tool is able to use multithreading functionality only for ATPG and not simulation. In this case, the tool uses only one thread per host to simulate the patterns generated.

You can disable multithreading functionality with the following command:

```
> set_multiprocessing_options -multithreading off
```

## Adding Threads to the Master

If you add processors to the master host with multithreading enabled, the master process increases its thread count instead of adding additional slaves. The command transcript lists how many threads were started.

```
> add_processors localhost:4 ; # adds 4 threads to the master process for a total thread count of 5

// Adding 4 threads to birdeye (master)
// Master with 5 threads running.

> report_processors

// hosts      threads  arch      CPU(s)    %idle    free RAM  process size
// -----
// birdeye (master)      5  x86-64    8 x 2.9 GHz    89%    113.91 MB    232.56 MB
// master with 5 threads running.
```

Note that you can specify the master host by name or IP address.

## Adding Processors in Manual Mode

With multithreading enabled, the `add_processors` command can start only one process per slave host, and create additional processors (to master or slave) as additional threads.

The following example starts a distributed slave process on the host “odin” that runs with 4 total threads.

```
> add_processors odin:4
```

```
// Adding 4 threads to odin (new slave)
// Master with 1 thread and 1 slave with 4 threads running (5 total threads).
```

In the following example, the first command starts a distributed slave process on the host “odin” that runs with 2 threads. The second command increases the number of threads in the slave process from 2 to 4 rather than starting a new slave with 2 threads.

```
# odin is not the master host
> add_processors odin:2
```

```
// Adding 2 threads to odin (new slave)
// Master with 1 thread and 1 slave with 2 threads running (3 total threads).
```

```
> add_processors odin:2
```

```
// Adding 2 threads to odin (existing slave now using 4 threads)
// Master with 1 thread and 1 slave with 4 threads running (5 total threads).
```

Note that if you don’t specify a number after the host name, the default is 1.

If you add more threads to a host than it has CPU cores, the command adds the threads but issues a warning that the maximum number of available CPUs has been exceeded. If you specify the string “maxcpu” instead of a number, the tool fills up the process with threads until the number of threads equals the number of processors available to the host. If the host already uses the maxcpu number of threads, the tool issues a warning that no more processors were added. (Note that this is true for the master host as well as slave hosts.)

For example:

```
# Assume odin has 8 processors and is not the master host
> add_processors odin:2
```

```
// Adding 2 threads to odin (new slave)
// Master with 1 thread and 1 slave with 2 threads running (3 total threads).
```

```
> add_processors odin:maxcpu
```

```
// Adding 6 threads to odin (existing slave now using 8 threads)
// Master with 1 thread and 1 slave with 8 threads running (9 total threads).
```

The “`add_processors maxcpu`” command always adds enough threads to reach a total of “maxcpu” threads for a host (that is, the second `add_processors` command in the example above would always add enough threads to total 8 on the host).

The following is a more detailed example of adding processors in manual mode:

```
# 3 different hosts used: thor, odin, and localhost,
# each with 16 CPUs

# First show the options used (defaults)

> report_multiprocessing_options

// Multiprocessing options:
// Option                                Type    Value  Description
// -----
// generic_delete                        string   generic job scheduler delete
// generic_scheduler                     string   generic job scheduler
// license_timeout                       number   5       # mins to acquire license ( 0: infinite )
// lsf_options                           string   options for LSF job scheduler
// multithreading                        on/off  on      turn on/off multithreading flow
// processors_per_grid_request            number  -1      # processors grouped for one grid request
//   (default: 1 for SGE and GENERIC, 4 for LSF requests)
// remote_shell                          string   rsh     rsh or ssh remote_shell setting
// result_time_limit                     float    45     time limit (min) used to detect
//   non-responsive slaves.
// scheduler_timeout                     number   10     # mins for job scheduler
// sge_options                           string   options for SGE job scheduler

# add a slave process with 2 threads on odin
> add_processors odin:2

// Adding 2 threads to odin (new slave)
// Master with 1 thread and 1 slave with 2 threads running (3 total threads).

# add a slave process with 1 thread on thor
> add_processors thor

// Adding 1 thread to thor (new slave)
// Master with 1 thread and 2 slaves with 3 threads running (4 total threads).

# add 4 additional threads to the master
> add_processors localhost:4

// Adding 4 threads to masterhost (master)
// Master with 5 threads running.

> add_processors odin:maxcpu thor:maxcpu

// Adding 6 threads to odin (existing slave now using 8 threads)
// Adding 7 threads to thor (existing slave now using 8 threads)
// Master with 5 threads and 2 slaves with 16 threads running
// (21 total threads).

# do it again just to get a warning that no more threads are added
> add_processors odin:maxcpu thor:maxcpu

// Warning: Max number of processors on odin already in use (8 exist).
// No processors added.
// Warning: Max number of processors on thor already in use (8 exist).
// No processors added.

> report_processors

// hosts                                threads  arch      CPU(s)    %idle    free RAM  process size
// -----
// masterhost (master)                  5       x86-64    1 x 2.6 GHz  99%      59.28 MB  256.79 MB
// odin                                 8       x86-64    8 x 2.9 GHz  86%      6525.36 MB 154.53 MB
// thor                                 8       x86-64    8 x 2.8 GHz 100%     58782.55 MB 153.59 MB
// master with 5 threads and 2 slaves with 16 threads running.
```

## Adding Processors in Grid Mode

As in manual mode, the tool doesn't start a new slave process on a host given by the grid engine if there is already a running slave on that machine. Instead, the thread count of the existing slave (or master, if the grid request returns the master host) is increased.

Requesting one processor at a time from the grid engine is inefficient in many cases, as the tool has to request enough memory for a full slave process with every grid request. Therefore, the tool groups a number of processors together into one grid request for a host with enough processors and the memory needed by one slave with the specified number of threads.

So, if the `processors_per_grid_request` variable is set to 4, then an “`add_processors sge:8`” command results in two grid requests for 4 processors (slots) each.

The threads started are associated with the requested grid resource, and the resource is freed back to the grid only if all threads are removed.

The following is an example of adding processors in grid mode:

```
# group 4 processors per grid request
> set_multiprocessing_options -processors_per_grid_request 4
...

# add 20 processors via LSF in 5 groups of 4 processors each
> add_processors lsf:20

# this may result in a situation like this:
> report_processors
```

| // hosts               | threads | arch   | CPU(s)       | %idle | free RAM    | process size |
|------------------------|---------|--------|--------------|-------|-------------|--------------|
| // masterhost (master) | 5       | x86-64 | 16 x 2.8 GHz | 100%  | 4655.45 MB  | 189.70 MB    |
| // kraken              | 4       | x86-64 | 8 x 2.8 GHz  | 100%  | 49828.50 MB | 151.86 MB    |
| // brightly            | 8       | x86-64 | 8 x 2.9 GHz  | 100%  | 9461.83 MB  | 152.66 MB    |
| // joker111            | 4       | x86-64 | 16 x 2.8 GHz | 100%  | 41892.00 MB | 166.05 MB    |

// master with 5 threads and 3 slaves with 16 threads running.

Note that for LSF, the `processors_per_grid` is set to 4, and the requested `processors_per_grid` is automatically provided to the LSF scheduler. However, for SGE or the generic scheduler, the default is set to 1 because bundling of slot requests is site-specific. You have to configure the options specific to that grid system before using this option on non-LSF grids. For more information about how to do this, refer to the [set\\_multiprocessing\\_options](#) description in the *Tessent Shell Reference Manual*.



## Adding Processors to the LSF Grid

In addition to the general features described in [Adding Processors in Grid Mode](#), the tool provides some extra assistance for submitting jobs to the LSF grid. Whereas SGE has predefined machine types and architectures for which the tool can constrain grid requests to hosts that are supported by the tool, LSF requires individual grid installations to classify hosts using site-specific “model” and “type” designations. If you specify “type” constraints in the `lsf_options` variable (with “[set\\_multiprocessing\\_options](#) -lsf\_options”), then the tool uses those constraints. Otherwise, the tool attempts to determine which type/model combinations it can use as described below.

By default, the tool uses both LSF scheduler learning and LSF heuristics learning to determine suitable type/model combinations when submitting jobs to the grid. You can enable and disable these two features using the [set\\_multiprocessing\\_options](#) switches: `-lsf_learning` and `-lsf_heuristics`. When adding processors to the LSF grid, the sequence of events is as follows:

1. When LSF scheduler learning is enabled, the tool uses the LSF system to determine what machines and type/model combinations are available. It then runs the **lsrun** command to log onto likely machines to determine which are compatible with the tool. LSF scheduler learning cannot succeed if your LSF system is configured to prevent use of **lsrun**.

---

**i** **Tip:** If your system does not support **lsrun**, you may want to disable LSF scheduler learning to avoid the resulting overhead and warning messages.

---

2. If LSF scheduler learning is disabled or fails, the tool then uses LSF heuristics learning unless you have disabled this feature with “`-lsf_heuristics off`.” LSF heuristics learning is an attempt to automatically choose machines on the LSF grid that are compatible with the tool. It attempts to determine tool compatibility based on the names of the types and models available and successfully identify some type/model combinations that the tool can use; however, this may also result in specifying incompatible combinations or miss some compatible combinations.

---

**i** **Tip:** If you are sure that your slave requests will always result in the tool obtaining compatible hosts (for example, because you specified an LSF host queue with `-lsf_options` that contains only compatible hosts), then you may want to disable LSF heuristics learning to avoid learning potentially incorrect or incomplete results.

---

3. If LSF heuristics learning is disabled or fails, then the tool submits the job anyway without specifying any type/model restrictions.

## Deleting Processors Added in Manual Mode

The `delete_processors` command decreases the number of running threads or stops an entire slave process if the number of threads equals zero (or less) after deducting the number of processors to delete from the number of running threads.

If you don't specify a number, the command removes all threads and the slave process on the related host. If you don't specify a number or a host, the command removes all slaves and their threads, as well as all additional threads on the master.

If you issue a `delete_processors` command for the master, the tool has to make sure that the master is kept running with at least one thread. If a thread is deleted from its process, the process immediately frees all the related thread data.

## Deleting Processors Added in Grid Mode

The `delete_processors` command releases grid slots only if all threads belonging to a grid request were deleted. For example:

```
# (assume that sge_options have already been configured properly.)
> set_multiprocessing_options -processors_per_grid_request 4

# add 20 processors to SGE
> add_processors sge:20

# this may end up in a situation like this:
# each slave has at least 4 threads
> report_processors
```

| # | hosts              | threads | arch   | CPU(s)       | %idle | free RAM    | process size |
|---|--------------------|---------|--------|--------------|-------|-------------|--------------|
| # | localhost (master) | 5       | x86-64 | 16 x 2.8 GHz | 100%  | 4655.45 MB  | 189.70 MB    |
| # | kraken             | 4       | x86-64 | 8 x 2.8 GHz  | 100%  | 49828.50 MB | 151.86 MB    |
| # | brighty            | 8       | x86-64 | 8 x 2.9 GHz  | 100%  | 9461.83 MB  | 152.66 MB    |
| # | joker111           | 4       | x86-64 | 16 x 2.8 GHz | 100%  | 41892.00 MB | 166.05 MB    |

# master with 5 threads and 3 slaves with 16 threads running.

```
# delete 4 threads from brighty, which frees exactly one grid slot
# (since 4 processors were requested with each grid request)
> delete_processors brighty:4

# delete 2 threads on kraken, which does not free any grid resource
> delete_processors kraken:2

# delete everything from kraken, all grid resources (1 request) are freed
> delete_processors kraken

# What is left?
> report_processors
```

| # | hosts              | threads | arch   | CPU(s)       | %idle | free RAM    | process size |
|---|--------------------|---------|--------|--------------|-------|-------------|--------------|
| # | localhost (master) | 5       | x86-64 | 16 x 2.8 GHz | 100%  | 4655.45 MB  | 189.70 MB    |
| # | brighty            | 4       | x86-64 | 8 x 2.9 GHz  | 100%  | 9461.83 MB  | 152.66 MB    |
| # | joker111           | 4       | x86-64 | 16 x 2.8 GHz | 100%  | 41892.00 MB | 166.05 MB    |

# master with 5 threads and 2 slaves with 8 threads running.

# Chapter 8

## Scan Pattern Retargeting

---

### Overview

This manual describes scan pattern retargeting functionality in the Tessent Shell® tool.

Scan pattern retargeting improves efficiency and productivity by enabling you to generate core-level test patterns and *retarget* them for reuse at the top level. Patterns for multiple cores can be merged and applied simultaneously at the chip level. This capability can be used for cores that include any configuration the tool supports for ATPG; this includes multiple EDT blocks and/or uncompressed chains, pipeline stages, low power, and cores with varying shift lengths.

The scan pattern retargeting capability enables you to:

- Design and develop cores in isolation from the rest of the design.
- Generate and verify test patterns of the cores at the core level.
- Generate test patterns for identical cores just once.
- Divide and conquer a large design by testing groups of cores.
- Have pipeline stages and/or inversion between the core boundary and chip level.
- Broadcast stimuli to multiple instances of the same core.
- Merge patterns generated for multiple cores and apply them simultaneously.
- Automatically trace core-level scan pins to the chip level to extract connections, pipelining, and inversion outside the cores.
- Perform reverse mapping of silicon failures from the chip level to the core level to allow diagnosis to be performed at the core level.

---

#### Caution



You should verify the chip-level patterns through simulation since the tool's DRCs may not detect every setup error or timing issue.

---

If you would like to use Tessent Diagnosis to diagnose your retargeted patterns, see section “[Reverse Mapping Top-Level Failures to the Core](#)” in the *Tessent Diagnosis User’s Manual*.

## Tools and Licensing

Scan pattern retargeting requires a TestKompress license. Reverse mapping of chip-level failures, as well as diagnosis of those failures, requires a Tessent Diagnosis license.

Scan pattern retargeting and the generation of core-level retargetable patterns must be done within the Tessent Shell tool. Instructions for invoking Tessent Shell are provided in the following section.

For complete information on using Tessent Shell, see the [\*Tessent Shell User's Manual\*](#).

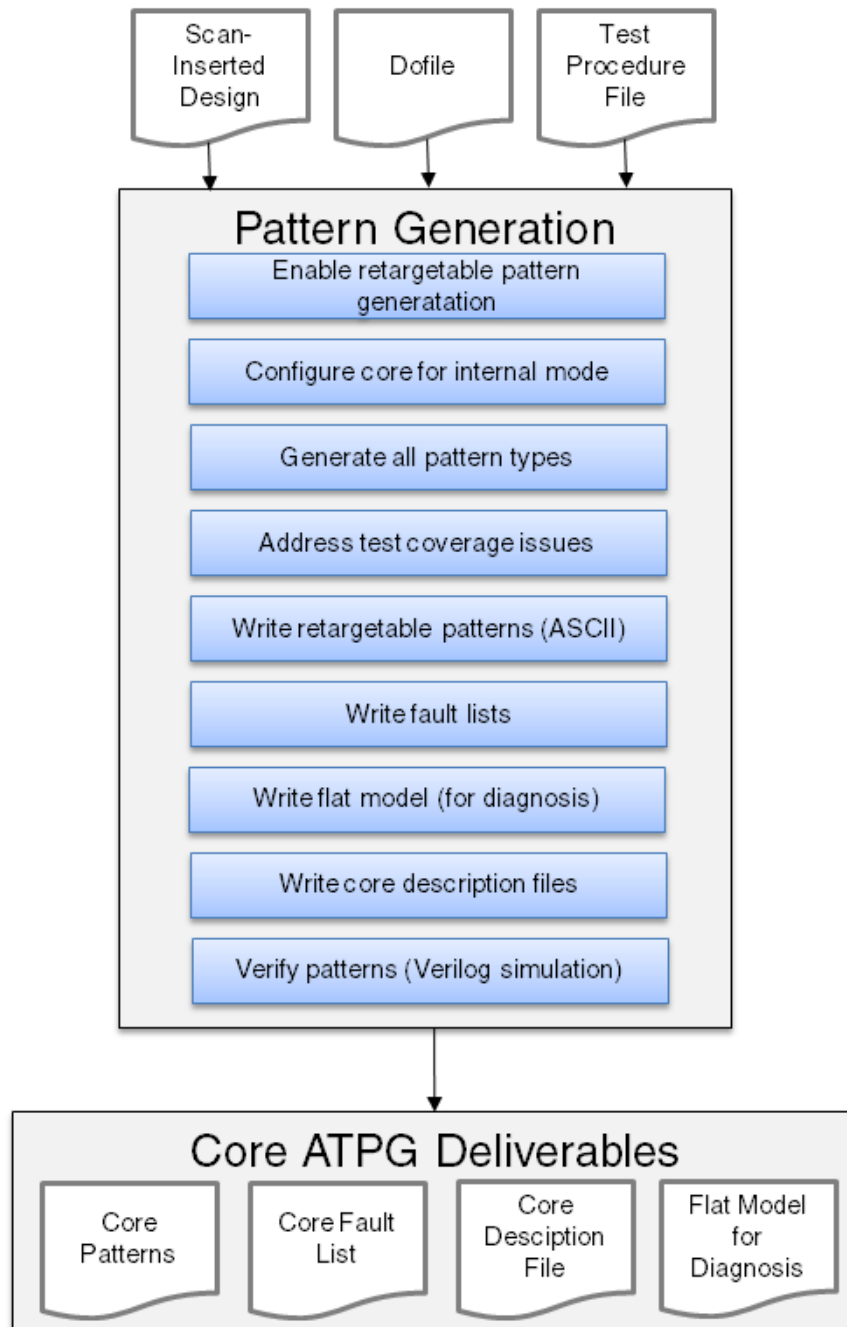
# Scan Pattern Retargeting Process

An example of the scan pattern retargeting process is presented in “[Retargeting Example](#)” on page 332.

The scan pattern retargeting process is:

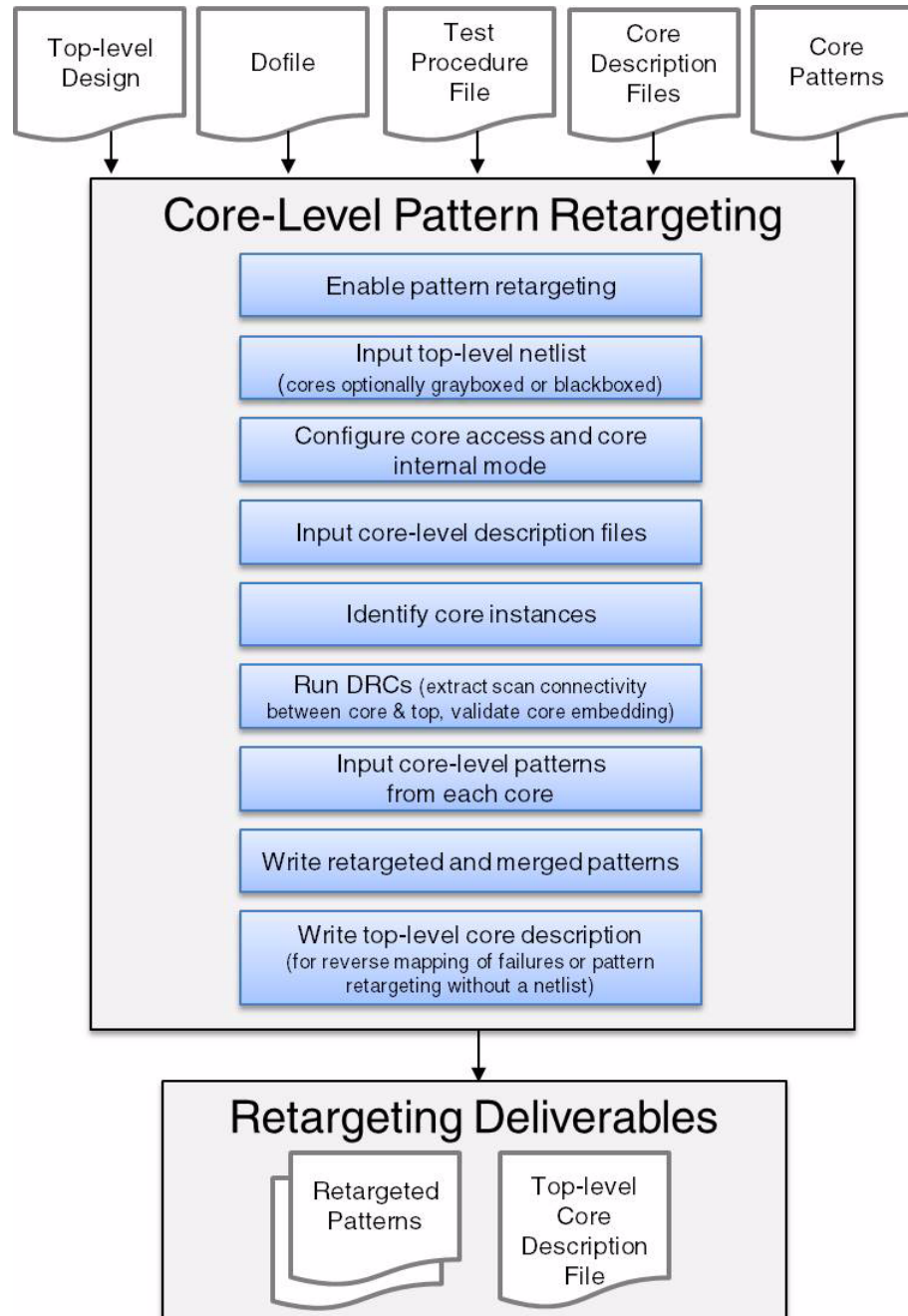
1. Generate patterns for all cores. This process is illustrated in Figure 8-1. For more information, see “[Generating Patterns at the Core Level](#)” on page 328.”

**Figure 8-1. Core-level Pattern Generation Process**



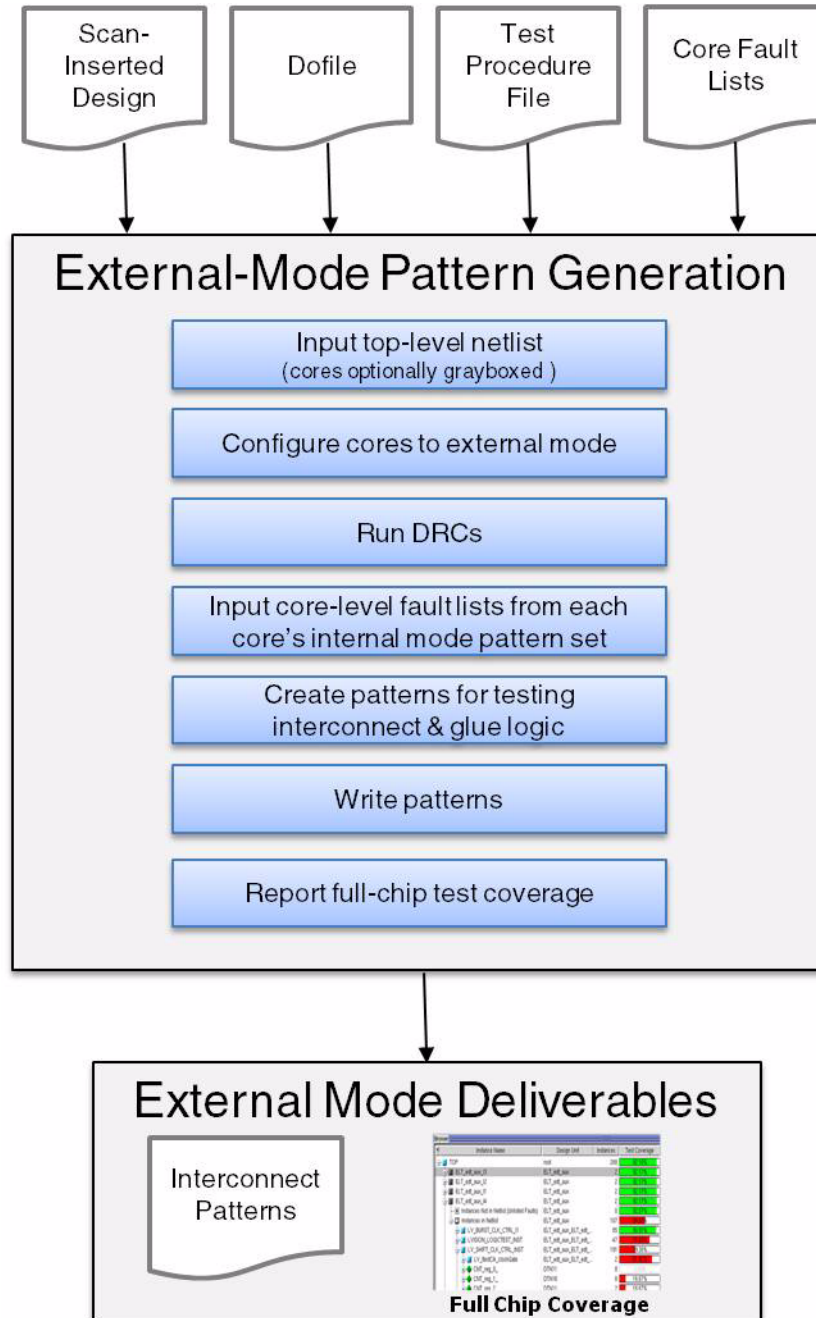
2. Retarget core-level test patterns to the chip level. This process is illustrated in Figure 8-2. For more information, see [“Retargeting Patterns in Internal Mode”](#) on page 329.”

**Figure 8-2. Scan Pattern Retargeting - Internal Mode**



3. Generate patterns for the top-level chip logic only. This process is illustrated in Figure 8-3. For more information, see “Generating Patterns in External Mode” on page 331.

**Figure 8-3. Pattern Generation - External Mode**



## Core-Level Pattern Generation

In order to generate retargetable patterns for a core, the core must already have wrapper chains inserted into it and it must be configured to its internal test mode. Otherwise, you can lose significant coverage. To enable core-level generation of retargetable patterns, you must execute “[set\\_current\\_mode](#) [mode\_name] -type internal” in setup mode.

When later retargeting core-level patterns to the top level, the tool maps only scan data from the core level to the top level. The tool does not map core-level PIs and POs to the top level. Nor does the tool map core-level capture cycle events from the core level to the top level. Therefore, primary inputs must be constrained during capture, outputs must be masked, and clocking must meet certain requirements, as explained next.

During core-level pattern generation, the tool automatically adds X constraints on most input ports and adds output masks on all output ports so that generated patterns can be mapped to the chip level for retargeting. Free-running clocks, always-capture clocks, and constrained pins are not forced to X. The tool also omits internal primary inputs (added with “[add\\_clocks](#) -internal” or “[add\\_primary\\_inputs](#) -internal”) since those cut points are for modeling and are usually controlled by Named Capture Procedures (NCPs) or clock control definitions.

You can exclude additional primary inputs from X constraints; this is necessary if the pins are controlled by an NCP that models chip-level constraints such as a clock controller located outside of the core. All inputs must be constrained or controlled (e.g., using all enabled NCPs).

---

### Note



Even `scan_enable`, like any other input, must either be constrained to 0 or forced using an NCP.

---

If a pin is not a free-running or always-capture clock, is not constrained, and is excluded from isolation constraints (i.e., the tool did not constrain it to X), the `create_patterns` command checks that the pin is controlled. If the pin is not controlled, the tool issues an R7 DRC violation.

Be careful when you apply the `is_excluded_from_isolation_constraints` attribute to a port, and never ignore an R7 DRC violation. Doing this can result in ATPG controlling the unconstrained port to any arbitrary value which can vary from pattern to pattern, even though pattern retargeting only retargets the scan data and does NOT retarget values on primary inputs from the core level to the top.

Core-level test patterns are saved in ASCII format when using the [write\\_patterns](#) command. Only patterns saved in ASCII format can be retargeted. You can also use the `write_patterns` command, as in the normal ATPG flow, to generate simulation test benches or write the patterns in other formats.

You can use existing commands to write out core-level fault list and fault detection information.



### Caution



When generating the EDT IP, if output channel pipeline stages will be added later, you must specify “set\_edt\_pins -change\_edge\_at\_compactor\_output trailing\_edge” to ensure that the compactor output changes consistently on the trailing edge of the EDT clock. Output channel pipeline stages should then start with leading-edge sequential elements.

---

## Clocking Architecture

The scan pattern retargeting functionality imposes the following restrictions on clock architecture:

- The tool does not perform per-pattern capture cycle mapping. The capture clock conditions at the core boundary must be identical for every pattern in the pattern set.
- The recommended methodology is to have a programmable clock chopper inside each core that is programmed by scan cell values. The scan cells used to program the clock chopper must be inside the core. This provides ATPG with the flexibility to generate the required clocking sequences yet ensures that the clock conditions at the boundary of the cores are the same (free-running clocks, or clocks that are pulsed only during setup and/or shift but constrained off during capture).
- If the clock controller is outside the core, it must have been programmed statically during test\_setup to deliver the same clock sequence during the capture phase of every pattern. During core-level ATPG, the clocking at the core boundary must be enforced by defining an NCP. In addition, any unconstrained pins on the core boundary must have the is\_excluded\_from\_isolation\_constraints attribute set; otherwise, the tool will constrain them to X.
- No support is provided if the clocking on the boundary of the core is different for each pattern, such as when the capture clocks are programmed using a scan chain outside the core, or when the capture clocks are driven directly by chip-level pins.

## The Tessent Core Description File

The pattern generation step produces a Tessent core description (TCD) file. The TCD file contains the information the tool needs to map the core-level patterns to the next level of hierarchy. This includes information such as description of the EDT hardware and scan chains.

## Scan Pattern Retargeting of Test Patterns

To retarget core-level test patterns, Tessent Shell maps the scan path pins from the cores to the top level, adjusts the core-level patterns based on the additional retargeting pipeline stages learned between the core and chip level (to ensure that all patterns from different cores have the same shift length via padding), and maps them to the chip-level pins they connect to.

Note that retargeting of core-level patterns does *not* require you to provide a full netlist for every core. Alternatively, you can specify a core as either a graybox or blackbox in the following ways:

- If you have generated a graybox and want to handle the core as a graybox model, use the `read_verilog` command to read the Verilog model that was written out from the graybox generation step. Although the wrapper scan chains in the graybox model are not needed for the retargeting mode, feedthroughs are preserved in the graybox model. If feedthroughs are used for connecting cores to the chip level and blackboxing the cores would break that path from the core boundary to the top, it is recommended to use the graybox model.
- If you have the full core netlist in a separate file from the top-level module(s) and just want to read it in and preserve the boundary but discard the contents, use the `read_verilog <design> -blackbox` command. This method can only be used if there are no feedthroughs or control logic within the core that are needed for the current mode of operation.
- If you want to read the full design and blackbox specific core instances during design flattening, use the `add_black_box` command. This method can only be used if there are no feedthroughs or control logic within the core that are needed for the current mode of operation.

For more information, see [“Scan Pattern Retargeting Without a Netlist”](#) on page 324.

### Chip-Level Test Procedures

You must provide the chip-level `test_setup` procedure needed for the retargeting step. You must also either provide the chip-level `load_unload` and `shift` procedures, or allow the tool to create them automatically by mapping and merging the core-level `load_unload` and `shift` procedures as described in [“Test Procedure Retargeting”](#) on page 323.

### External\_capture Procedure

You specify the capture cycle in an `external_capture` procedure, using the `set_external_capture_options -capture_procedure` switch. The `-capture_procedure` switch specifies the `external_capture` procedure to be used. The tool uses this external capture procedure for all capture cycles between each scan load, even when the pattern is a multi-load pattern. An example `external_capture` procedure is shown here:

```
procedure external_capture ext_procedure_1 =  
    timeplate tmp_1;  
    cycle =
```

```
    force_pi;  
    pulse clk_trigger 1;  
    pulse reference_clock;  
end;  
cycle =  
    pulse reference_clock;  
end;  
end;
```

The `external_capture` procedure has the same syntax as an NCP, except it does not have internal and external modes. In comparison to an NCP, the `external_capture` procedure has the following restrictions:

- Can only have one `force_pi` statement.
- Cannot have any `measure_po` statements.
- Cannot have any `load_cycles` as it is used between each scan load of a pattern.
- Cannot contain any events on internal signals.

Pin constraints defined at the top level are used in this generic capture sequence. No pattern-specific capture information is mapped from the patterns.

## Test Procedure Retargeting

When core-level patterns are retargeted to the chip-level, chip-level `load_unload` and shift test procedures are needed. If chip-level `load_unload` and shift test procedures do not exist, the tool automatically generates them based on the information in the core-level test procedure files. This retargeting step merges the core test procedures if there is more than one core, and maps them to the top-level test procedures.

You can disable this default behavior by setting the [set\\_procedure\\_retargeting](#) command to off. If test procedure retargeting is disabled and the chip-level `load_unload` and shift test procedures are missing, the tool generates an error.

### Constraint of the Top-Level Port During the Load\_Unload Test Procedure

You can set a constant value for a top-level port during the simulation of the `load_unload` test procedure, when needed, to enable tracing of signals from the core level to the chip level. You can do this by setting the value of the `constraint_value_during_load_unload` attribute using the `set_attribute_value` command.

If, for example, `scan_enable` must be 1 to trace signals needed for `load_unload` and shift procedure retargeting, you would need to define this `load_unload` constraint in the tool for test

procedure retargeting to work; especially, because this value will likely conflict with the 0 input constraint typically added to scan\_enable for capture.

For more information see the [set\\_attribute\\_value](#) command and the [Port](#) data model attributes in the *Tessent Shell Reference Manual*.

## Scan Pattern Retargeting Without a Netlist

When retargeting patterns for the first time, you are not required to provide a full netlist for every core; instead, you can specify a core as either a graybox or blackbox. If you add a core as a blackbox, the tool retrieves all of the information necessary for retargeting its patterns from the core-level TCD file.

You can use the same approach when retargeting patterns for the top level. In the absence of a netlist, the tool may use the previously-extracted top-level TCD file to retarget patterns. The top-level core description includes all of the information needed to perform retargeting, including the core-level descriptions, connectivity of the scan pins, and top-level scan procedures used and validated during extraction. The top-level TCD file removes the need to read in the netlist for pattern retargeting. Only the top-level core description and core-level patterns need to be read into the tool to perform pattern retargeting.

This enables efficient retargeting of new core-level patterns if the design and procedures have not changed since DRC and core connectivity extraction were performed.

An example of this procedure is shown in [“Retargeting of Patterns at the Chip Level Without a Netlist”](#) on page 335.

## Retargeting Support For Patterns Generated at the Top Level

Scan patterns are typically retargeted with respect to lower-level cores instantiated in the design; in this case, the core is a lower-level core. However, in some cases, you can test part of the top-level logic in parallel with lower-level cores by treating this partial view of the top level as a core instance; in this case, the core is actually the top level.

During pattern retargeting, you use the [add\\_core\\_instances](#) -current\_design command and option to add the top-level logic as a core instance. This option allows you to add the core instance with respect to the current design; this core instance then becomes the top level. The -current\_design option also allows patterns for the current design to be merged with patterns of other core instances in the hierarchy as shown in [“Example 2”](#) on page 326.

A core can have multiple modes. A *mode* of a core is a specific configuration or view of that design with a specific set of procedures and scan definitions. Two modes of a design can be mutually exclusive (such as the internal and external test of a wrapper core), or independent of each other (such as when we generate patterns for non-overlapping parts of a core and then merge those patterns as in [Example 1](#)).

**Example 1**

The following example demonstrates the use of the `-current_design` switch to merge two modes of a core; in this case, the “core” is the top level. In the example, patterns are generated separately for two sub-blocks of a design from the top-level ports. Sub-Block1 is tested as mode M1 of the top level, and Sub-Block2 is tested as mode M2 of the top level. The result, shown in [Figure 8-4](#), is the merging of the two pattern sets originally generated for each individual mode into a single pattern set applied at the top level.

The first dofile reads in the top level design and writes out the patterns and TCD files for mode M1 targeting Sub-Block1:

```
// Invoke tool to generate patterns for each mode of top_level
set_context pattern -scan
read_verilog top_level.v           // Read netlist of top_level

// Mode M1 consists of the Sub-Block1 design; the user applies constraints to access Sub-Block1
// and can optionally blackbox Sub-Block2 to reduce memory consumption
set_current_mode M1 -type internal
...
set_system_mode analysis
...
create_patterns
write_patterns top_levelM1.ascii
write_core_description top_levelM1.tcd
```

The second dofile reads in the design and writes out the patterns and TCD files for mode M2 targeting Sub-Block2:

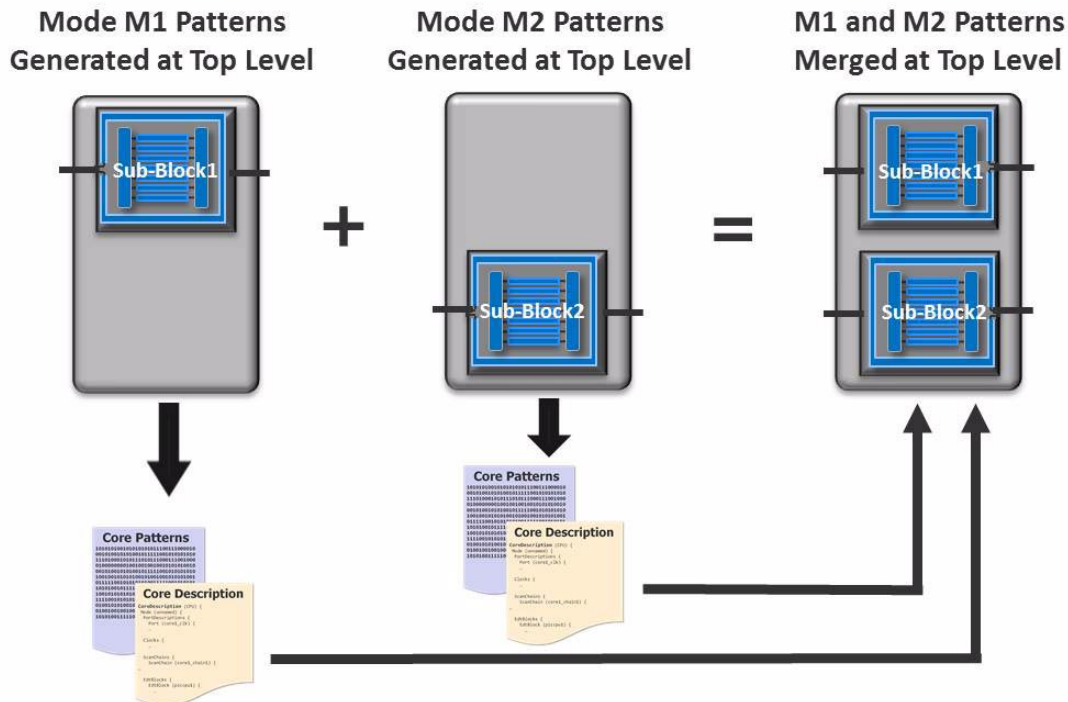
```
// Mode M2 consists of the Sub-Block2 design; the user applies constraints to access Sub-Block2
// and can optionally blackbox Sub-Block1 to reduce memory consumption
set_current_mode M2 -type internal
...
set_system_mode analysis
...
create_patterns
write_patterns top_levelM2.ascii
write_core_description top_levelM2.tcd
```

The third dofile merges the two separately generated top-level pattern sets (modes M1 and M2) into a single pattern set:

```
// Invoke the tool again to retarget patterns to top level
set_context patterns -scan_retargeting
...
read_core_description top_levelM1.tcd
read_core_description top_levelM2.tcd
add_core_instance -core top_level -mode M1 -current_design
add_core_instance -core top_level -mode M2 -current_design
set_system_mode analysis

// Mode extracted from ASCII file
read_patterns top_levelM1.ascii
read_patterns top_levelM2.ascii
write_patterns ...
```

**Figure 8-4. Merging Patterns for Multiple Views of a Core at the Top Level**



### Example 2

The following example demonstrates the use of the `add_core_instances -current_design` switch to merge mode M1 of the current design and mode internal of CoreB. In the example, one pattern set is generated at the top level (mode M1 of the top level) to test Sub-Block1, and a second pattern set is generated at the core level of CoreB. The result, as shown in [Figure 8-5](#), is the integration of the two pattern sets, originally generated for mode M1 of the top level and mode internal of CoreB, into the top level.

The first dofile reads in the top level design and writes out the patterns and TCD files for mode M1 targeting Sub-Block1.

```
// Invoke tool to generate patterns for mode M1 of the top level
set_context pattern -scan
read_verilog top_level.v                                // Read netlist of top_level

// Mode M1 consists of the Sub-Block1 design; the user applies constraints to access
// Sub-Block1 and can optionally blackbox any other blocks not targeted to reduce memory
// consumption
set_current_mode M1 -type internal
...
set_system_mode analysis
...
create_patterns
write_patterns top_levelM1.ascii
write_core_description top_levelM1.tcd
```

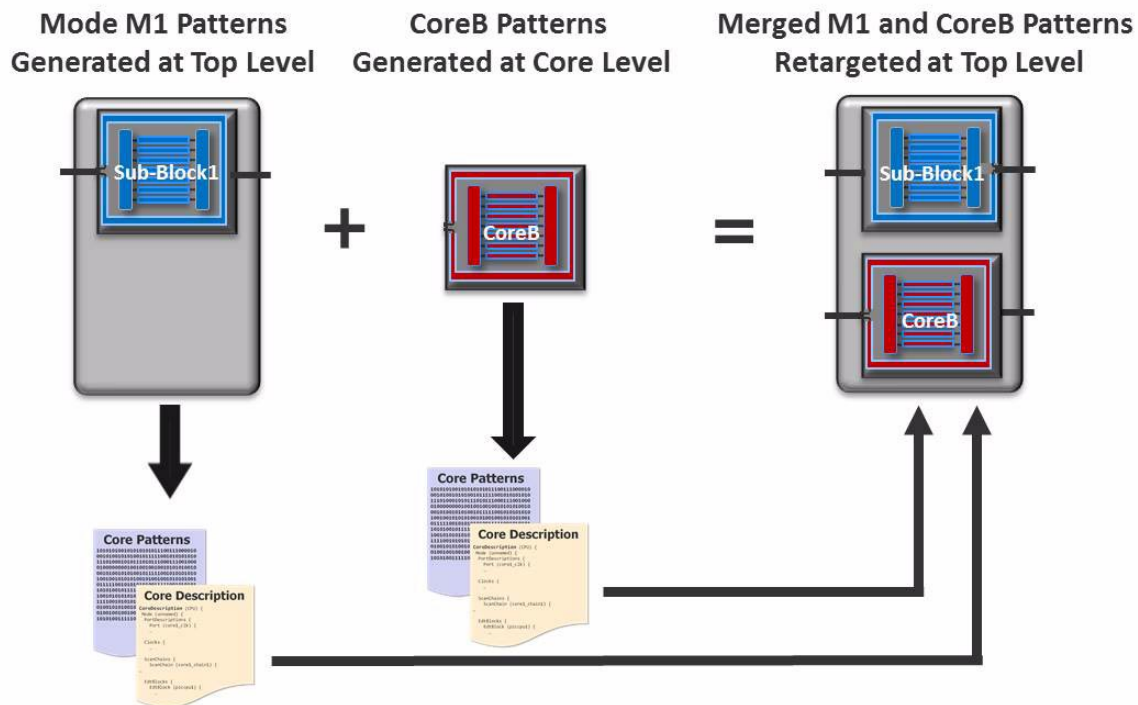
The second dofile reads in the CoreB design and writes out the patterns and TCD files.

```
set_system_mode setup
delete_design
read_verilog CoreB.v
set_current_mode internal -type internal
...
set_system_mode analysis
create_patterns
write_patterns CoreB.ascii
write_core_description CoreB.tcd
```

The third dofile merges the top level pattern set (mode M1) with the retargeted CoreB patterns.

```
// Invoke tool to retarget patterns to top level
set_context patterns -scan_retargeting
...
read_core_description top_levelM1.tcd
read_core_description CoreB.tcd
add_core_instance -core top_level -mode M1 -current_design
add_core_instance -core CoreB -mode internal -instances CoreB_i
...
set_system_mode analysis
read_patterns top_levelM1.ascii
read_patterns CoreB.ascii
write_patterns ....
```

Figure 8-5. Merging Patterns for Top Level and Core at the Top Level



# Generating Patterns at the Core Level

This procedure describes how to generate core-level test patterns.

## Prerequisites

- Core-level netlist
- Cell library

## Procedure

1. Enable ATPG using the “[set\\_context patterns -scan](#)” command.
2. Read the core-level netlist and cell library.
3. Configure the core for internal mode.
4. Generate all pattern types.
5. Address all test coverage issues.
6. Verify the test patterns (test benches, STA).
7. Write the retargetable core test patterns using the [write\\_patterns](#) command.
8. Write the core fault lists using the [write\\_faults](#) command.
9. Write the TCD file using the [write\\_core\\_description](#) command.
10. Write the flat model for diagnosis using the [write\\_flat\\_model](#) command.



# Retargeting Patterns in Internal Mode

This procedure describes how to retarget core-level test patterns to the chip level.

## Prerequisites

- TCD file for each of the core types.
- Retargetable patterns resulting from core-level pattern generation.
- Top-level netlist (with cores optionally blackboxed or grayboxed).
- Top-level test procedures.

## Procedure

1. Enable the retargeting of patterns using the “[set\\_context](#) patterns -scan\_retarting” command.

### Note



A full netlist for cores whose patterns are being retargeted is *not* required. Retargeting only requires a graybox or blackbox model of the cores to both retarget the patterns and to generate chip-level serial and parallel simulation test benches.

---

2. Read the TCD file for each of the core types.
3. Bind each core description to the core instances it represents in the design using the [add\\_core\\_instances](#) command.
4. Read a dofile and test procedure file to configure core access and to configure the cores whose patterns are being retargeted into their internal mode.
5. Run DRCs when “[set\\_system\\_mode](#) analysis” is invoked, including extracting scan connections from the core instances to the top level and validating the embedding of the cores.
6. Read in the retargetable patterns for each core type using the [read\\_patterns](#) command.
7. Write the retargeted patterns using the [write\\_patterns](#) command. The tool automatically performs pin mapping (between core and chip-level) and pattern merging as it writes the chip-level patterns.
8. Write the top-level TCD file. This file includes the core-level core description information as well as scan connectivity information from the cores to the top. This information is later used for reverse mapping of silicon failures back to the core level for diagnosis, or for pattern retargeting without a netlist.

#### Note



The top-level TCD file also enables generation of top-level patterns without a netlist. For more information, see [“Retargeting of Patterns at the Chip Level Without a Netlist.”](#)

---

Chip-level patterns cannot be read back into Tessent Shell since the internal clocking information is lost. However this is not needed since diagnosis is done at the core-level. The generated simulation test bench can be simulated for validation. In addition, the design objective of this functionality is to eliminate the need to load the full design into the tool.

## Retargeting Patterns Without a Top-Level Netlist

You can retarget core-level patterns at the top level without a netlist by using the information contained in the top-level TCD file.

### Prerequisites

- Top-level TCD file.
- Retargetable patterns resulting from core-level pattern generation.

### Procedure

1. Enable the retargeting of patterns using the `“set_context patterns -scan_retarting”` command.
2. Read the top-level TCD file using the `read_core_descriptions` command. In the absence of a netlist, the tool recreates what is needed for the design using this file.
3. Change to analysis mode using `“set_system_mode analysis”`. This command automatically sets the current design, adds the cores, sets pin constraints, and provides any other information needed for the design.
4. Read in the retargetable patterns for each core type using the `read_patterns` command.
5. Report core instances using the `report_core_instances` command.
6. Retarget and merge patterns as they are written out with respect to the chip-level boundary using the `write_patterns` command.

## Generating Patterns in External Mode

This procedure describes how to generate patterns in external mode.

### Prerequisites

- Complete top-level netlist or top-level netlist with cores grayboxed.
- Core-level fault lists.
- Top-level dofile.
- Top-level test procedure file.

### Procedure

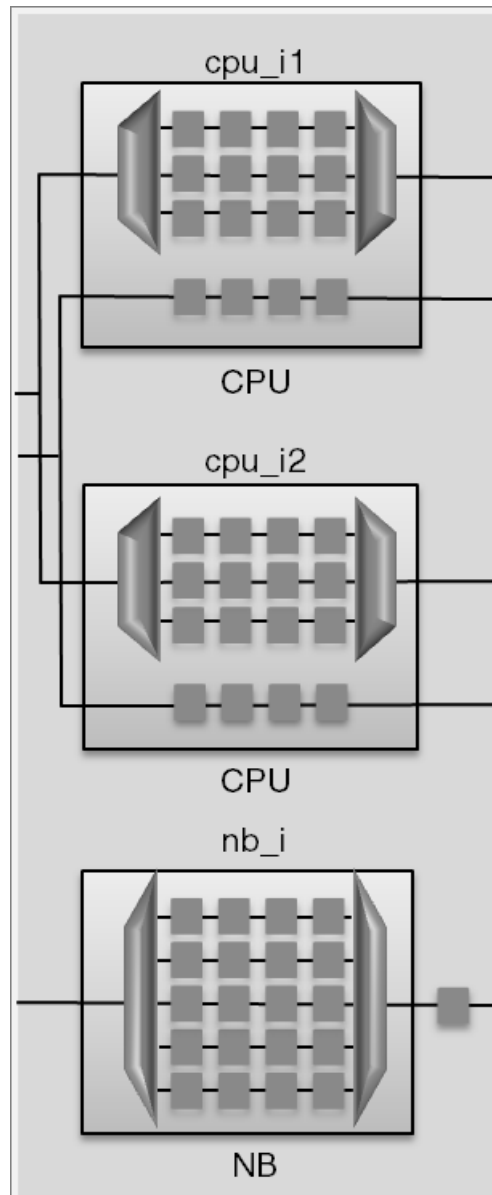
1. Enable the generation of patterns using the “[set\\_context](#) patterns -scan” command.
2. Read a complete top-level netlist, or one where each core is replaced by a light-weight graybox model, and the cell library.
3. Read the core-level fault lists from each core’s internal mode pattern set. If the netlist is not complete due to the use of the graybox models, add the -graybox switch to the “[read\\_faults](#) *fault\_file* -instance *instances* -merge” command.
4. Read a dofile and test procedure file to configure cores to external mode.
5. Run DRCs when “[set\\_system\\_mode](#) analysis” is invoked.
6. Create patterns for testing interconnect and top-level glue logic.
7. Write the generated patterns using the [write\\_patterns](#) command.
8. Report the full chip test coverage.

## Retargeting Example

This example demonstrates the retargeting use model based on the design shown in [Figure 8-6](#).

The design has three cores of two types: two identical instances of the CPU core and one instance of the NB core. TOP is the chip level of the design. In the example, the input is broadcast to identical core instances.

**Figure 8-6. Retargeting of Core-level Patterns**



## Core-level Pattern Generation for CPU Cores

Core-level pattern generation for the CPU core is shown here:

```
# Set the context for ATPG
set_context patterns -scan

# Read cell library (ATPG library file)
read_cell_library atpglib

# Read the core netlist
read_verilog CPU.v

# Specify to generate patterns that are retargetable for the internal mode
# of the core. Because the mode name was not specified, the default is the
# same as the mode type "internal".
set_current_mode -type internal

# Specify pins the tool should not constrain to X because they will be
# explicitly controlled by a Named Capture Procedure that models a
# chip-level clock controller driving core-level clocks
set_attribute_value {clk1 clk2} \
    -name is_excluded_from_isolation_constraints

# Add commands to set up design for ATPG. To avoid coverage loss due to
# auto isolation constraints on scan_enable signals, constrain scan_enable
# signals to a non-X state
...
add_input_constraint input_scan_enable -C1
add_input_constraint output_scan_enable -C0
add_input_constraint core_scan_enable -C0

# Change to analysis mode
set_system_mode analysis

# Generate TCD file
write_core_description CPU.tcd -replace

# Specify the fault model type and generate patterns
set_fault_type stuck
create_patterns

# Write ASCII patterns for subsequent retargeting; must be ASCII format
write_patterns CPU_stuck.retpat.gz -replace

# Write fault list
write_faults CPU_stuck.faults.gz -replace

# Write flat model for diagnosis
write_flat_model CPU_stuck.flat_model.gz -replace
```

## Core-level Pattern Generation for NB Core

Core-level pattern generation for the NB core is the same as for the CPU core:

```
# Set the context for ATPG
set_context patterns -scan

# Read cell library (ATPG library file)
read_cell_library atpglib

# Read core netlist
read_verilog NB.v

# Specify to generate patterns that are retargetable for the internal mode
# of the core. Because the mode name was not specified, the default is the
# same as the mode type "internal".
set_current_mode -type internal
...
```

## Retargeting of Patterns at the Chip Level

The steps for scan pattern retargeting are shown here. The tool performs design rule checking as it changes to analysis mode.

```
# Specify that this is the scan pattern retargeting phase
set_context patterns -scan_retargeting

# Read cell library (ATPG library file)
read_cell_library atpglib

# Read all netlists and the current design. You can replace the core
# netlists with graybox models, or blackbox them to reduce memory usage
# and run time. You can blackbox them using the read_verilog -blackbox,
# or add_black_box commands.
read_verilog TOP.v
read_verilog CPU.v -blackbox
read_verilog NB.v -blackbox
set_current_design TOP

# Specify the top-level test procedure file
set_procfile_name TOP.testproc

# Read all TCD files
read_core_description CPU.tcd
read_core_description NB.tcd

# Bind core descriptions to cores
add_core_instances -core CPU -modules CPU
add_core_instances -core NB -instances /nb_i

# Change to analysis mode
set_system_mode analysis

# Read core (retargetable) patterns
read_patterns CPU_stuck.retpat.gz
read_patterns NB_stuck.retpat.gz
```

```
# Specify generic capture window
set_external_capture_options -capture_procedure \
    <external_capture_procedure_name>

# Report core instances, then retarget and merge patterns as they are
# written out with respect to the chip-level boundary
report_core_instances
write_patterns TOP_stuck.stil -stil -replace

# Write top-level core description. Used for reverse mapping of silicon
# failures back to the core level for diagnosis, or for pattern
# retargeting without a netlist
write_core_description TOP_stuck.tcd -replace
```

## Retargeting of Patterns at the Chip Level Without a Netlist

The steps for scan pattern retargeting *without a netlist* are shown here. The tool recreates what is needed for the design from the top-level TCD file. The tool performs design rule checking as it changes to analysis mode.

```
# Specify that this is the scan pattern retargeting phase
set_context patterns -scan_retargeting

# Read the top-level TCD file
read_core_description TOP_stuck.tcd

# Changing to analysis mode automatically sets the current design, adds
# the cores, sets pin constraints and any other necessary information for
# the design
set_system_mode analysis

# Read core (retargetable) patterns
read_patterns CPU_stuck.retpat.gz
read_patterns NB_stuck.retpat.gz

# Report core instances (created automatically), then retarget and merge
# patterns as they are written out with respect to the chip-level boundary
report_core_instances
write_patterns TOP_stuck.stil -stil -replace
```

## Limitations

The limitations of the scan pattern retargeting functionality are listed here.

- DRCs exist to validate that the design setup at the top level is consistent with the setup that was used for core-level ATPG. But this validation is not complete. For example, the capture cycle clocking is not validated. Consequently, you should perform chip-level serial simulation of a small number of scan patterns to verify the setup is correct.
- Core-level ATPG has limited flat model support. You can save a flat model in analysis mode for performing diagnosis or rerunning ATPG with that same configuration, but you must perform the initial design setup (setup mode) and design rule checking on a Verilog design and not a flat model. Note, you cannot access the `is_excluded_from_isolation_constraints` port attribute when running on a flat model.
- During the retargeting phase, flat models are not supported. A Verilog design must be read in. A flat model cannot be read in or written out because it lacks information necessary for retargeting.
- Scan pattern retargeting does not support multiple scan groups.
- When generating retargetable core-level patterns, native launch-off-shift is not supported. Native launch-off-shift is one of two methods for generating launch-off-shift patterns.

Native launch-off-shift — With this method, the transition tested is triggered by the last shift applied by the shift procedure. The capture occurs when ATPG generates a single cycle test. This method is not supported. When generating retargetable patterns for the transition fault type, launch-off-shift will be automatically disabled as if you had invoked the “`set_fault_type transition`” command with the “`-no_shift_launch`” option.

Pseudo launch-off-shift — This method, which is supported, is modeled within the capture cycles of ATPG. The patterns typically include two cycles. During the first capture cycle, the design is kept in shift mode. During the second cycle, the scan enable is de-asserted and the capture is performed. This method is more commonly used because it allows the tool to perform shift and capture at-speed using PLL clocks.

- The `shadow_control`, `shadow_observe`, and `master_observe` procedures are not supported. You should not use these procedures during core-level ATPG or in the top-level test procedure file.



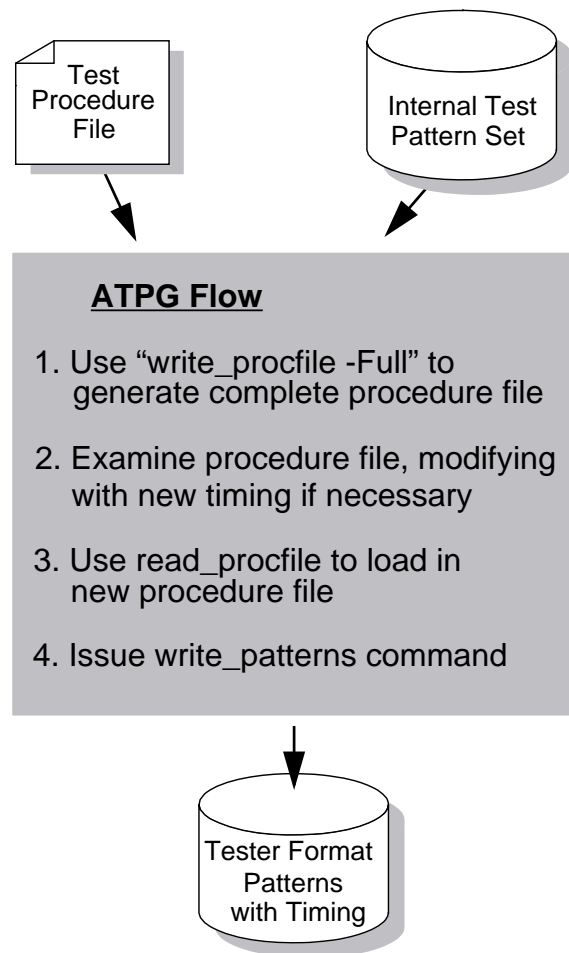
# Chapter 9

## Test Pattern Formatting and Timing

---

Figure 9-1 shows a basic process flow for defining test pattern timing.

**Figure 9-1. Defining Basic Timing Process Flow**



## Test Pattern Timing Overview

Test procedure files contain both scan and non-scan procedures. All timing for all pattern information, both scan and non-scan, is defined in this procedure file.

While the ATPG process itself does not require test procedure files to contain real timing information, automatic test equipment (ATE) and some simulators do require this information. Therefore, you must modify the test procedure files you use for ATPG to include real timing information. [“General Timing Issues”](#) on page 339 discusses how you add timing information to existing test procedures.

After creating real timing for the test procedures, you are ready to save the patterns. You use the `write_patterns` command with the proper format to create a test pattern set with timing information. For more information, refer to [“Saving Timing Patterns”](#) on page 344.

Test procedures contain groups of statements that define scan-related events. See [“Test Procedure File”](#) on the *Tessent Shell User's Manual*.

## Timing Terminology

The following list defines some timing-related terms:

- **Non-return Timing** — Primary inputs that change, at most, once during a test cycle.
- **Offset** — The timeframe in a test cycle in which pin values change.
- **Period** — The duration of pin timing—one or more test cycles.
- **Return Timing** — Primary inputs, typically clocks, that pulse high or low during every test cycle. Return timing indicates that the pin starts at one logic level, changes, and returns to the original logic level before the cycle ends.
- **Suppressible Return Timing** — Primary inputs that can exhibit return timing during a test cycle, although not necessarily.

## General Timing Issues

ATEs require test data in a cycle-based format. The patterns you apply to such equipment must specify the waveforms of each input, output, or bidirectional pin, for each test cycle.

Within a test cycle, a device under test must abide by the following restrictions:

- At most, each non-clock input pin changes once in a test cycle. However, different input pins can change at different times.
- Each clock input pin is at its off-state at both the start and end of a test cycle.
- At most, each clock input pin changes twice in a test cycle. However, different clock pins can change at different times.
- Each output pin has only one expected value during a test cycle. However, the equipment can measure different output pin values at different times.
- A bidirectional pin acts as either an input or an output, but not both, during a single test cycle.

To avoid adverse timing problems, the following timing requirements satisfy some ATE timing constraints:

- **Unused Outputs** — By default, test procedures without measure events (all procedures except **shift**) strobe unused outputs at a time of  $\text{cycle}/2$ , and end the strobe at  $3*\text{cycle}/4$ . The **shift** procedure strobcs unused outputs at the same time as the scan output pin.
- **Unused Inputs** — By default, all unused input pins in a test procedure have a force offset of 0.

- **Unused Clock Pins** — By default, unused clock pins in a test procedure have an offset of cycle/4 and a width of cycle/2, where cycle is the duration of each cycle in the test procedure.
- **Pattern Loading and Unloading** — During the **load\_unload** procedure, when one pattern loads, the result from the previous pattern unloads. When the tool loads the first pattern, the unload values are X. After the tool loads the last pattern, it loads a pattern of X's so it can simultaneously unload the values resulting from the final pattern.
- **Events Between Loading and Unloading (“patterns -scan” context only)** — If other events occur between the current unloading and the next loading, in order to load and unload the scan chain simultaneously, The tool performs the events in the following order:
  - a. **Observe Procedure Only** — The tool performs the observe procedure before loading and unloading.
  - b. **Initial Force Only** — The tool performs the initial force before loading and unloading.
  - c. **Both Observe Procedure and Initial Force** — The tool performs the observe procedures followed by the initial force before loading and unloading.

## Generating a Procedure File

Figure 9-1 illustrates the basic process flow for defining test pattern timing is as follows:

1. Use the [write\\_procfile](#) -Full command and switch to generate a complete procedure file.
2. Examine the procedure file, modify timeplates with new timing if necessary.
3. Use the [read\\_procfile](#) command to load in the revised procedure file.
4. Issue the [write\\_patterns](#) command.

The “[Test Procedure File](#)” section of the *Tessent Shell User’s Manual* gives an in depth description of how to create a procedure file.

There are three ways to load existing procedure file information into the tool:

- During SETUP mode, use the [add\\_scan\\_groups](#) *procedure\_filename* command. Any timing information in these procedure files will be used when [write\\_patterns](#) is issued if no other timing information or procedure information is loaded.
- Use the [read\\_procfile](#) command. This is only valid when not in SETUP mode. Using this command loads a new procedure file that will overwrite or merge with the procedure and timing data already loaded. This new data is now in effect for all subsequent [write\\_patterns](#) commands.

- If you specify a new procedure file on the [write\\_patterns](#) command line, the timing information in that procedure file will be used for that write\_patterns command only, and then the previous information will be restored.

## Defining and Modifying Timeplates

This section gives an overview of the test procedure file timeplate syntax, to facilitate Step 2 in the process flow listed previously. For a more detailed overview of timeplates, see the “[Timeplate Definition](#)” section of the *Tessent Shell User’s Manual*.

After you have used the [write\\_procfile](#) -Full command and switch to generate a procedure file, you can examine the procedure file, modifying timeplates with new timing if necessary. Any timing changes to the existing TimePlates, *cannot* change the event order of the timeplate used for scan procedures. The times may change, but the event order must be maintained.

In the following example, there are two events happening at time 20, and both are listed as event 4. These may be skewed, but they may not interfere with any other event. The events must stay in the order listed in the comments:

```
force_pi          0; // event 1
bidi_force_pi     12; // event 3
measure_po        31; // event 7
bidi_measure_po   32; // event 8
force_InPin       9; // event 2
measure_OutPin    35; // event 9
pulse_Clk1        20 5; // event 4 & 5 respectively
pulse_Clk2        20 10; // event 4 & 6 respectively
period 50;        // no events but all events
                  // have to happen in period
```

Test procedure files have the following format:

```
[set_statement ...]
[alias_definition]
timeplate_definition [timeplate_definition]
procedure_definition [procedure_definition]
```

The timeplate definition describes a single tester cycle and specifies where in that cycle all event edges are placed. You must define all timeplates before they are referenced. A procedure file must have at least one timeplate definition. The timeplate definition has the following format:

```
timeplate timeplate_name =
    timeplate_statement
    [timeplate_statement ...]
period time;
end;
```

The following list contains available timeplate\_statement statements. The timeplate definition should contain at least the force\_pi and measure\_po statements.

#### Note



You are not required to include pulse statements for the clocks. But if you do not “pulse” a clock, the tool uses two cycles to pulse it, resulting in larger patterns.

---

```
timeplate_statement:  
  offstate pin_name off_state;  
  force_pi time;  
  bidi_force_pi time;  
  measure_po time;  
  bidi_measure_po time;  
  force pin_name time;  
  measure pin_name time;  
  pulse pin_name time width;
```

- **timeplate\_name** — A string that specifies the name of the timeplate.
- **offstate pin\_name off\_state** — A literal and double string that specifies the inactive, off-state value (0 or 1) for a specific named pin that is not defined as a clock pin by the [add\\_clocks](#) command. This statement must occur before all other timeplate\_statement statements. This statement is only needed for a pin that is not defined as a clock pin by the add\_clocks command but will be pulsed within this timeplate.
- **force\_pi time** — A literal and string pair that specifies the force time for all primary inputs.
- **bidi\_force\_pi time** — A literal and string pair that specifies the force time for all bidirectional pins. This statement allows the bidirectional pins to be forced after applying the tri-state control signal, so the system avoids bus contention. This statement overrides “force\_pi” and “measure\_po”.
- **measure\_po time** — A literal and string pair that specifies the time at which the tool measures (or strobos) the primary outputs.
- **bidi\_measure\_po time** — A literal and string pair that specifies the time at which the tool measures (or strobos) the bidirectional pins. This statement overrides “force\_pi” and “measure\_po”.
- **force pin\_name time** — A literal and double string that specifies the force time for a specific named pin.

#### Note



This force time overrides the force time specified in force\_pi for this specific pin.

---

- **measure pin\_name time** — A literal and double string that specifies the measure time for a specific named pin.

### Note



This measure time overrides the measure time specified in `measure_po` for this specific pin.

- **pulse *pin\_name time width*** — A literal and triple string that specifies the pulse timing for a specific named clock pin. The *time* value specifies the leading edge of the clock pulse and the *width* value specifies the width of the clock pulse. This statement can only reference pins that have been declared as clocks by the [add\\_clocks](#) command or pins that have an offstate specified by the “offstate” statement. The sum of the time and width must be less than the period.
- **period *time*** — A literal and string pair that defines the period of a tester cycle. This statement ensures that the cycle contains sufficient time, after the last force event, for the circuit to stabilize. The time you specify should be greater than or equal to the final event time.

## Example 1

```
timeplate tp1 =
  force_pi 0;
  pulse T 30 30;
  pulse R 30 30;
  measure_po 90;
  period 100;
end;
```

## Example 2

The following example shows a shift procedure that pulses `b_clk` with an off-state value of 0. The timeplate `tp_shift` defines the off-state for pin `b_clk`. The `b_clk` pin is not declared as a clock in the ATPG tool.

```
timeplate tp_shift =
  offstate b_clk 0;
  force_pi 0;
  measure_po 10;
  pulse clk 50 30;
  pulse b_clk 140 50;
  period 200;
end;

procedure shift =
  timeplate tp_shift;
  cycle =
    force_sci;
    measure_sco;
    pulse clk;
    pulse b_clk;
  end;
end;
```

## Saving Timing Patterns

You can write the patterns generated during the ATPG process both for timing simulation and use on the ATE. Once you create the proper timing information in a test procedure file, the tool uses an internal test pattern data formatter to generate the patterns in the following formats:

- Text format (ASCII)
- Binary format
- Wave Generation Language (WGL)
- Standard Test Interface Language (STIL)
- Verilog
- Texas Instruments Test Description Language (TDL 91)
- Fujitsu Test data Description Language (FTDL-E)
- Mitsubishi Test Description Language (MITDL)
- Toshiba Standard Tester interface Language 2 (TSTL2)

## Features of the Formatter

The main features of the test pattern data formatter include:

- Generating basic test pattern data formats. text, Verilog, and WGL (ASCII and binary).
- Generating ASIC Vendor test data formats: TDL 91, FTDL-E, MITDL, and TSTL2.
- Supporting parallel load of scan cells (in Verilog format).
- Reading in external input patterns and output responses, and directly translating to one of the formats.
- Reading in external input patterns, performing good or faulty machine simulation to generate output responses, and then translating to any of the formats.
- Writing out just a subset of patterns in any test data format.
- Facilitating failure analysis by having the test data files cross-reference information between tester cycle numbers and pattern numbers.
- Supporting differential scan input pins for each simulation data format.

## Pattern Formatting Issues

The following subsections describe issues you should understand regarding the test pattern formatter and pattern saving process.



## Serial Versus Parallel Scan Chain Loading

When you simulate test patterns, most of the time is spent loading and unloading the scan chains, as opposed to actually simulating the circuit response to a test pattern. You can use either serial or parallel loading, and each affects the total simulation time differently.

The primary advantage of simulating serial loading is that it emulates how patterns are loaded on the tester. You thus obtain a very realistic indication of circuit operation. The disadvantage is that for each pattern, you must clock the scan chain registers at least as many times as you have scan cells in the longest chain. For large designs, simulating serial loading takes an extremely long time to process a full set of patterns.

The primary advantage of simulating parallel loading of the scan chains is it greatly reduces simulation time compared to serial loading. You can directly (in parallel) load the simulation model with the necessary test pattern values because you have access, in the simulator, to internal nodes in the design. Parallel loading makes it practical for you to perform timing simulations for the entire pattern set in a reasonable time using popular simulators like ModelSim that utilize the Verilog format.

## Parallel Scan Chain Loading

You accomplish parallel loading through the scan input and scan output pins of scan *sub-chains* (a chain of one or more scan cells, modeled as a single library model) because these pins are unique to both the timing simulator model and the Tessent Shell internal model. For example, you can parallel load the scan chain by using Verilog force statements to change the value of the scan input pin of each sub-chain.

After the parallel load, you apply the **shift** procedure a few times (depending on the number of scan cells in the longest subchain, but usually only once) to load the scan-in value into the sub-chains. Simulating the **shift** procedure only a few times can dramatically improve timing simulation performance. You can then observe the scan-out value at the scan output pin of each sub-chain.

Parallel loading ensures that all memory elements in the scan sub-chains achieve the same states as when serially loaded. Also, this technique is independent of the scan design style or type of scan cells the design uses. Moreover, when writing patterns using parallel loading, you do not have to specify the mapping of the memory elements in a sub-chain between the timing simulator and Tessent Shell. This method does not constrain library model development for scan cells.

#### Note



When your design contains at least one stable-high scan cell, the **shift** procedure period must exceed the shift clock off time. If the **shift** procedure period is less than or equal to the shift clock off time, you may encounter timing violations during simulation. The test pattern formatter checks for this condition and issues an appropriate error message when it encounters a violation.

---

For example, the test pattern timing checker would issue an error message when reading in the following **shift** procedure and its corresponding timeplate:

```
timeplate gen_tpl =
    force_pi 0;
    measure_po 100;
    pulse CLK 200 100;
    period 300; // Period same as shift clock off time
end;

procedure shift =
    scan_group grp1;
    timeplate gen_tpl;
    cycle =
        force_sci;
        measure_sco;
        pulse CLK; // Force shift clock on and off
    end;
end;
```

The error message would state:

```
// Error: There is at least one stable high scan cell in the design. The
shift procedure period must be greater than the shift clock off time to
avoid simulation timing violations.
```

The following modified timeplate would pass timing rules checks:

```
timeplate gen_tpl =
    force_pi 0;
    measure_po 100;
    pulse CLK 200 100;
    period 400; // Period greater than shift clock off time
end;
```

## Sampling to Reduce Serial Loading Simulation Time

When you use the [write\\_patterns](#) command, you can specify to save a sample of the full pattern set by using the **-Sample** switch. This reduces the number of patterns in the pattern file(s), reducing simulation time accordingly. In addition, the **-Sample** switch allows you to control how many patterns of each type are included in the sample. By varying the number of sample patterns, you can fine-tune the trade-off between file size and simulation time for serial patterns.

**Note**

Using the -Start and -End switches limits file size as well, but the portion of internal patterns saved does not provide a very reliable indication of pattern characteristics when simulated. Sampled patterns more closely approximate the results you would obtain from the entire pattern set.

After performing initial verification with parallel loading, you can use a sampled pattern set for simulating series loading until you are satisfied test coverage is reasonably close to desired specification. Then, perform a series loading simulation with the unsampled pattern set only once, as your last verification step.

**Note**

The [set\\_pattern\\_filtering](#) command serves a similar purpose to the -Sample switch of the write\_patterns command. The set\_pattern\_filtering command creates a temporary set of sampled patterns within the tool.

## Test Pattern Data Support for IDDQ

For best results, you should measure current after each non-scan cycle if doing so catches additional IDDQ faults. However, you can only measure current at specific places in the test pattern sequence, typically at the end of the test cycle boundary. To identify when IDDQ current measurement can occur, the pattern file adds the following command at the appropriate places:

```
measure IDDQ ALL;
```

Several test pattern data formats support IDDQ testing. There are special IDDQ measurement constructs in TDL 91 (Texas Instruments), MITDL (Mitsubishi), TSTL2 (Toshiba), and FTDL-E (Fujitsu). The tools add these constructs to the test data files. All other formats (WGL and Verilog) represent these statements as comments.

## Saving Patterns in Basic Test Data Formats

The [write\\_patterns](#) command saves the patterns in the basic test data formats including text, binary, Verilog, and WGL (ASCII and binary). You can use these formats for timing simulation.

### Text Format

This is the default format that the tool generates when you run the write\_patterns command. The tool can read back in this format in addition to WGL, STIL, and binary format.

This format contains test pattern data in a text-based parallel format, along with pattern boundary specifications. The main pattern block calls the appropriate test procedures, while the header contains test coverage statistics and the necessary environment variable settings. This

format also contains each of the scan test procedures, as well as information about each scan memory element in the design.

To create a basic text format file, enter the following at the application command line:

```
ANALYSIS> write_patterns filename -ascii
```

The formatter writes the complete test data to the file named *filename*.

For more information on the `write_patterns` command and its options, refer to the [write\\_patterns](#) description in the *Tessent Shell Reference Manual*.

---

**Note**

This pattern format does not contain explicit timing information. For more information about this test pattern format, refer to [“Test Pattern File Formats”](#) on page 355.

---

## Comparing the Text Format with Other Test Data Formats

The text format describes the contents of the test set in a human readable form. In many cases, you may find it useful to compare the contents of a simulation or test data format with that of the text format for debugging purposes. This section provides detailed information necessary for this task.

Often, the first cycle in a test set must perform certain tasks. The first test cycle in all test data formats turns off the clocks at all clock pins, drives Z on all bidirectional pins, drives an X on all other input pins, and disables measurement at any primary output pins.

The test pattern set can contain two main parts: the *chain test* block, to detect faults in the scan chain, and the *scan test* or *cycle test* block, to detect other system faults.

### The Chain Test Block

The chain test applies the **test\_setup** procedure, followed by the **load\_unload** procedure for loading scan chains, and the **load\_unload** procedure again for unloading scan chains. Each **load\_unload** procedure in turn calls the **shift** procedure. This operation typically loads a repeating pattern of “0011” into the chains. However, if scan chains with less than four cells exist, then the operation loads and unloads a repeating “01” pattern followed by a repeating “10” pattern. Also, when multiple scan chains in a group share a common scan input pin, the chain test process separately loads and unloads each of the scan chains with the repeating pattern to test them in sequence.

The test procedure file applies each event in a test procedure at the specified time. Each test procedure corresponds to one or more test cycles. Each test procedure can have a test cycle with a different timing definition. By default, all events use a timescale of 1 ns.

**Note**

If you specify a capture clock with the `set_capture_clock` command, the test pattern formatter does not produce the chain test block. For example, the formatter does not produce a chain test block for IEEE 1149.1 devices in which you specify a capture clock during tool setup.

## The Scan Test Block

The scan test block in the pattern set starts with an application of the **test\_setup** procedure. The scan test block contains several test patterns, each of which typically applies the **load\_unload** procedure, forces the primary inputs, measures the primary outputs, and pulses a capture clock. The **load\_unload** procedure translates to one or more test cycles. The force, measure, and clock pulse events in the pattern translate to the ATPG-generated capture cycle.

Each event has a sequence number within the test cycle. The sequence number's default time scale is 1 ns.

Unloading of the scan chains for the current pattern occurs concurrently with the loading of scan chains for the next pattern. Therefore the last pattern in the test set contains an extra application of the **load\_unload** sequence.

More complex scan styles (for example, like LSSD) use **master\_observe** and **skewed\_load** procedures in the pattern. For designs with sequential controllers, like boundary scan designs, each test procedure may have several test cycles in it to operate the sequential scan controller. Some pattern types (for example, RAM sequential and clock sequential types) are more complex than the basic patterns. RAM sequential patterns involve multiple loads of the scan chains and multiple applications of the RAM write clock. Clock sequential patterns involve multiple capture cycles after loading the scan chains. Another special type of pattern is the `clock_po` pattern. In these patterns, clocks may be held active throughout the test cycle and without applying capture clocks.

If the test data format supports only a single timing definition, the tool cannot save both `clock_po` and non-`clock_po` patterns in one pattern set. This is so because the tester cannot reproduce one clock waveform that meets the requirements of both types of patterns. Each pattern type (combinational, `clock_po`, `ram_sequential`, and `clock_sequential`) can have a separate timing definition.

## General Considerations

During a test procedure, you may leave many pins unspecified. Unspecified primary input pins retain their previous state.

#### Note



If you run ATPG after setting pin constraints, you should also ensure that you set these pins to their constrained states at the end of the **test\_setup** procedure. The **add\_input\_constraints** command constrains pins for the non-scan cycles, not the test procedures. If you do not properly constrain the pins within the **test\_setup** procedure, the tool does it for you, internally adding the extra force events after the **test\_setup** procedure. This increases the period of the **test\_setup** procedure by one time unit. This increased period can conflict with the test cycle period, potentially forcing you to re-run ATPG with the modified test procedure file.

---

All test data formats contain comment lines that indicate the beginning of each test block and each test pattern. You can use these comments to correlate the test data in the text format with other test data formats.

These comment lines also contain the *cycle count* and the *loop count*, which help correlate tester pattern data with the original test pattern data. The cycle count represents the number of test cycles, with the shift sequence counted as one cycle. The loop count represents the number of all test cycles, including the shift cycles. The cycle count is useful if the tester has a separate memory buffer for scan patterns, otherwise the loop count is more relevant.

#### Note



The cycle count and loop count contain information for all test cycles—including the test cycles corresponding to test procedures. You can use this information to correlate tester failures to a pattern for fault diagnosis.

---

## Binary

This format contains test pattern data in a binary parallel format, which is the only format (other than text format) that the tool can read. A file generated in this format contains the same information as text format, but uses a condensed form. You should use this format for archival purposes or when storing intermediate results for very large designs.

To create a binary format file, enter the following command:

```
ANALYSIS> write_patterns filename -binary
```

The tool writes the complete test data to the file named *filename*.

For more information about the **write\_patterns** command and its options, refer to the [write\\_patterns](#) description in the *Tessent Shell Reference Manual*.

## Verilog

This format contains test pattern data and timing information in a text-based format readable by both the Verilog and Verifault simulators. This format also supports both serial and parallel

loading of scan cells. The Verilog format supports all Tessent Shell timing definitions, because Verilog stimulus is a sequence of timed events.

To generate a basic Verilog format test pattern file, use the following arguments with the `write_patterns` command:

**`write_patterns filename [-Parallel | -Serial] -Verilog`**

The Verilog pattern file contains procedures to apply the test patterns, compare expected output with simulated output, and print out a report containing information about failing comparisons. The tools write all patterns and comparison functions into one main file (*filename*), while writing the primary output names in another file (*filename.po.name*). If you choose parallel loading, they also write the names of the scan output pins of each scan sub-chain of each scan chain in separate files (for example, *filename.chain1.name*). This allows the tools to report output pins that have discrepancies between the expected and simulated outputs. You can enhance the Verilog testbench with Standard Delay Format (SDF) back annotation.

For more information on the `write_patterns` command and its options, refer to the [write\\_patterns](#) description in the *Tessent Shell Reference Manual*.

## Wave Generation Language (ASCII)

The Wave Generation Language (WGL) format contains test pattern data and timing information in a structured text-based format. You can translate this format into a variety of simulation and tester environments, but you must first read it into the Waveform database and use the appropriate translator. This format supports both serial and parallel loading of scan cells.

Some test data flows verify patterns by translating WGL to stimulus and response files for use by the chip foundry's golden simulator. Sometimes this translation process uses its own parallel loading scheme, called memory-to-memory mapping, for scan simulation. In this scheme, each scan memory element in the ATPG model must have the same name as the corresponding memory element in the simulation model. Due to the limitations of this parallel loading scheme, you should ensure the following:

1) there is only one scan cell for each DFT library model (also called a scan subchain), 2) the hierarchical scan cell names in the netlist and DFT library match those of the golden simulator (because the scan cell names in the ATPG model appear in the scan section of the parallel WGL output), and 3) the scan-in and scan-out pin names of all scan cells are the same.

To generate a basic WGL format test pattern file, use the following arguments with the `write_patterns` command:

**`write_patterns filename [-Parallel | -Serial] -Wgl`**

For more information about the `write_patterns` command and its options, refer to the [write\\_patterns](#) description in the *Tessent Shell Reference Manual*.

For more information on the WGL format, contact Integrated Measurement Systems, Inc.

## Standard Test Interface Language (STIL)

To generate a STIL format test pattern file, use the following arguments with the `write_patterns` command:

**`write_patterns filename [-Parallel | -Serial] -STIL`**

For more information about the `write_patterns` command and its options, refer to the [write\\_patterns](#) description in the *Tessent Shell Reference Manual*.

## Saving in ASIC Vendor Data Formats

The ASIC vendor test data formats include Texas Instruments TDL 91, Fujitsu FTDL-E, Mitsubishi MITDL, and Toshiba TSTL2. The ASIC vendor's chip testers use these formats.

All the ASIC vendor data formats are text-based and load data into scan cells in a parallel manner. Also, ASIC vendors usually impose several restrictions on pattern timing. Most ASIC vendor pattern formats support only a single timing definition. Refer to your ASIC vendor for test pattern formatting and other requirements.

The following subsections briefly describe the ASIC vendor pattern formats.

### TI TDL 91

This format contains test pattern data in a text-based format.

The tool supports features of TDL 91 version 3.0 and of TDL 91 version 6.0. The version 3.0 format supports multiple scan chains, but allows only a single timing definition for all test cycles. Thus, all test cycles must use the timing of the main capture cycle. TI's ASIC division imposes the additional restriction that comparison should always be done at the end of a tester cycle.

To generate a basic TI TDL 91 format test pattern file, use the following arguments with the `write_patterns` command:

**`write_patterns filename -Tltdl`**

The formatter writes the complete test data to the file *filename*. It also writes the chain test to another file (*filename.chain*) for separate use during the TI ASIC flow.

For more information about the `write_patterns` command and its options, refer to the [write\\_patterns](#) description in the *Tessent Shell Reference Manual*.

### Fujitsu FTDL-E

This format contains test pattern data in a text-based format. The FTDL-E format splits test data into patterns that measure 1 or 0 values, and patterns that measure Z values. The test patterns divide into test blocks that each contain 64K tester cycles.



To generate a basic FTDL-E format test pattern file, use the following arguments with the `write_patterns` command:

**`write_patterns filename -Fjtdl`**

The formatter writes the complete test data to the file named *filename.fjtdl.func*. If the test pattern set contains IDDQ measurements, the formatter creates a separate DC parametric test block in a file named *filename.fjtdl.dc*.

For more information about the `write_patterns` command and its options, refer to the [write\\_patterns](#) description in the *Tessent Shell Reference Manual*.

## Mitsubishi TDL

This format contains test pattern data in a text-based format. To generate a basic Mitsubishi Test Description Language (TDL) format test pattern file, use the following arguments with the `write_patterns` command:

**`write_patterns filename -Mltdl`**

The formatter represents all scan data in a parallel format. It writes the test data into two files: the program file (*filename.td0*), which contains all pin definitions, timing definitions, and scan chain definitions; and the test data file (*filename.td1*), which contains the actual test vector data in a parallel format.

For more information about the `write_patterns` command and its options, refer to the [write\\_patterns](#) description in the *Tessent Shell Reference Manual*.

## Toshiba TSTL2

This format contains only test pattern data in a text-based format. The test pattern data files contain timing information. This format supports multiple scan chains, but allows only a single timing definition for all test cycles. TSTL2 represents all scan data in a parallel format.

To generate a basic Toshiba TSTL2 format test pattern file, use the following arguments with the `write_patterns` command:

**`write_patterns filename -TSTl2`**

The formatter writes the complete test data to the file named *filename*. For more information about the `write_patterns` command and its options, refer to the [write\\_patterns](#) description in the *Tessent Shell Reference Manual*.



# Chapter 10

## Test Pattern File Formats

---

### ASCII File Format

The ASCII file that describes the scan test patterns is divided into five sections, which are named `header_data`, `setup_data`, `functional_chain_test`, `scan_test`, and `scan_cell`. Each section (except the `header_data` section) begins with a `section_name` statement and ends with an end statement. Also in this file, any line starting with a double slash (`//`) is a comment line. The format of the data contained in each section is described as follows.

#### Header\_Data

The `header_data` section contains the general information, or comments, associated with the test patterns. This is an optional section that requires a double slash (`//`) at the beginning of each line in this section. The data printed may be in the following format:

```
// model_build_version - the version of the model build program that was used to create
the scan model.

// design_name - the design name of the circuit to be tested.

// date - the date in which the scan model creation was performed.

// statistics - the test coverage, the number of faults for each fault class, and the total
number of test patterns.

// settings - the description of the environment in which the ATPG is performed.

// messages - any warning messages about bus contention, pins held, equivalent pins,
clock rules, and so on are noted.
```

#### Setup\_Data

The `setup_data` section contains the definition of the scan structure and general test procedures that will be referenced in the description of the test patterns.

---

**Note**

Additional formats are added to the `setup_data` section for BIST patterns. For information on these formats, see [“BIST Pattern File Format”](#) on page 363”.

---

The data printed will be in the following format:

```
SETUP =  
    <setup information>  
END;
```

The setup information will include the following:

declare input bus “PI” = <ordered list of primary inputs>;

This defines the list of primary inputs that are contained in the circuit. Each primary input will be enclosed in double quotes and be separated by commas. For bidirectional pins, they will be placed in both the input and output bus.

declare output bus “PO” = <ordered list of primary outputs>;

This defines the list of primary outputs that are contained in the circuit. Each primary output will be enclosed in double quotes and be separated by commas.

```
CLOCK "clock_name1" =  
    OFF_STATE = <off_state_value>;  
    PULSE_WIDTH = <pulse_width_value>;  
END;  
CLOCK "clock_name2" =  
    OFF_STATE = <off_state_value>;  
    PULSE_WIDTH = <pulse_width_value>;  
END;
```

This defines the list of clocks that are contained in the circuit. The clock data will include the clock name enclosed in double quotes, the off-state value, and the pulse width value. For edge-triggered scan cells, the off-state is the value that places the initial state of the capturing transition at the clock input of the scan cell.

```
WRITE_CONTROL "primary_input_name" =  
    OFF_STATE = <off_state_value>;  
    PULSE_WIDTH = <pulse_width_value>;  
END;
```

This defines the list of write control lines that are contained in the circuit. The write control line will include the primary input name enclosed in double quotes, the off-state value, and the pulse width value. If there are multiple write control lines, they must be pulsed at the same time.

```
PROCEDURE TEST_SETUP "test_setup" =  
    FORCE "primary_input_name1" <value> <time>;  
    FORCE "primary_input_name2" <value> <time>;  
    ....  
    ....  
END;
```

This is an optional procedure that can be used to set nonscan memory elements to a constant state for both ATPG and the load/unload process. It is applied once at the beginning of the test pattern set. This procedure may only include force commands.

```
SCAN_GROUP "scan_group_name1" =  
    <scan_group_information>  
END;  
SCAN_GROUP "scan_group_name2" =  
    <scan_group_information>  
END;  
....  
....
```

This defines each scan chain group that is contained in the circuit. A scan chain group is a set of scan chains that are loaded and unloaded in parallel. The scan group name will be enclosed in double quotes and each scan group will have its own independent scan group section. Within a scan group section, there is information associated with that scan group, such as scan chain definitions and procedures.

```
SCAN_CHAIN "scan_chain_name1" =  
    SCAN_IN = "scan_in_pin";  
    SCAN_OUT = "scan_out_pin";  
    LENGTH = <length_of_scan_chain>;  
END;  
SCAN_CHAIN "scan_chain_name2" =  
    SCAN_IN = "scan_in_pin";  
    SCAN_OUT = "scan_out_pin";  
    LENGTH = <length_of_scan_chain>;  
END;  
....  
....
```

The scan chain definition defines the data associated with a scan chain in the circuit. If there are multiple scan chains within one scan group, each scan chain will have its own independent scan chain definition. The scan chain name will be enclosed in double quotes. The scan-in pin will be the name of the primary input scan-in pin enclosed in double quotes. The scan-out pin will be the name of the primary output scan-out pin enclosed in double quotes. The length of the scan chain will be the number of scan cells in the scan chain.

```
PROCEDURE <procedure_type> "scan_group_procedure_name" =  
    <list of events>  
END;
```

The type of procedures may include shift procedure, load and unload procedure, shadow-control procedure, master-observe procedure, shadow-observe procedure, and skew-load procedure. The list of events may be any combination of the following commands:

```
FORCE "primary_input_pin" <value> <time>;
```

This command is used to force a value (0,1, X, or Z) on a selected primary input pin at a given time. The time values must not be lower than previous time values for that procedure. The time for each procedure begins again at time 0. The primary input pin will be enclosed in double quotes.

```
APPLY "scan_group_procedure_name" <#times> <time>;
```

This command indicates the selected procedure name is to be applied the selected number of times beginning at the selected time. The scan group procedure name will be enclosed in double quotes. This command may only be used inside the load and unload procedures.

```
FORCE_SCI "scan_chain_name" <time>;
```

This command indicates the time in the shift procedure that values are to be placed on the scan chain inputs. The scan chain name will be enclosed in double quotes.

```
MEASURE_SCO "scan_chain_name" <time>;
```

This command indicates the time in the shift procedure that values are to be measured on the scan chain outputs. The scan chain name will be enclosed in double quotes.

## Functional\_Chain\_Test

The functional\_chain\_test section contains a definition of a functional scan chain test for all scan chains in the circuit to be tested. For each scan chain group, the scan chain test will include a load of alternating double zeros and double ones (00110011...) followed by an unload of those values for all scan chains of the group. The format is as follows:

```
CHAIN_TEST =  
  APPLY "test_setup" <value> <time>;  
  PATTERN = <number>;  
  APPLY "scan_group_load_name" <time> =  
    CHAIN "scan_chain_name1" = "values....";  
    CHAIN "scan_chain_name2" = "values....";  
    ....  
    ....  
  END;  
  APPLY "scan_group_unload_name" <time> =  
    CHAIN "scan_chain_name1" = "values....";  
    CHAIN "scan_chain_name2" = "values....";  
    ....  
    ....  
  END;  
END;
```

The optional "test\_setup" line is applied at the beginning of the functional chain test pattern if there is a test\_setup procedure in the Setup\_Data section. The number for the pattern is a zero-based pattern number where a functional scan chain test for all scan chains in the circuit is to be tested. The scan group load and unload name and the scan chain name will be enclosed in double quotes. The values to load and unload the scan chain will be enclosed in double quotes.

During the loading of the scan chains, each value of the corresponding scan chain will be placed at its scan chain input pin. The shift procedure will shift the value through the scan chain and continue shifting the next value until all values for all the scan chains have been loaded. Since the number of shifts is determined by the length of the longest scan chain, X's (don't care) are placed at the beginning of the shorter scan chains. This will ensure that all the values of the scan chains will be loaded properly.

During the unloading of the scan chains, each value of the corresponding scan chain will be measured at its scan chain output pin. The shift procedure will shift the value out of the scan chain and continue shifting the next value until all values for all the scan chains have been unloaded. Again, since the number of shifts is determined by the length of the longest scan chain, X's (don't measure) are placed at the end of the shorter scan chains. This will ensure that all the values of the scan chains will be unloaded properly.

Here is an example of a functional scan chain test:

```
CHAIN_TEST =
  APPLY "test_setup" 1 0;
  PATTERN = 0;
  APPLY "gl_load" 0 =
    CHAIN "c2" = "XXXXXXXXXX0011001100110011001100";
    CHAIN "c1" = "XXXXXXXXXXXXXXXX001100110011001100";
    CHAIN "c0" = "0011001100110011001100110011001";
  END;
  APPLY "gl_unload" 1 =
    CHAIN "c2" = "00110011001100110011001100XXXXXXXXXX";
    CHAIN "c1" = "00110011001100110011001100XXXXXXXXXXXX";
    CHAIN "c0" = "0011001100110011001100110011001";
  END;
END;
```

## Scan\_Test

The scan\_test section contains the definition of the scan test patterns that were created by Tessent Shell. A scan pattern will normally include the following:

```
SCAN_TEST =
  PATTERN = <number>;
  FORCE "PI" "primary_input_values" <time>;
  APPLY "scan_group_load_name" <time> =
    CHAIN "scan_chain_name1" = "values....";
    CHAIN "scan_chain_name2" = "values....";
    ....
  END;
  FORCE "PI" "primary_input_values" <time>;
  MEASURE "PO" "primary_output_values" <time>;
  PULSE "capture_clock_name1" <time>;
  PULSE "capture_clock_name2" <time>;
  APPLY "scan_group_unload_name" <time> =
    CHAIN "scan_chain_name1" = "values....";
    CHAIN "scan_chain_name2" = "values....";
    ....
  END;
  ....
  ....
  ....
END;
```

The number of the pattern represents the pattern number in which the scan chain is loaded, values are placed and measured, any capture clock is pulsed, and the scan chain is unloaded. The pattern number is zero-based and must start with zero. An additional force statement will be applied at the beginning of each test pattern, if transition faults are used. The scan group load and unload names and the scan chain names will be enclosed by double quotes. All the time values for a pattern must not be lower than the previous time values in that pattern. The values to load and unload the scan chain will be enclosed in double quotes. Refer to the [“Functional\\_Chain\\_Test”](#) section on how the loading and unloading of the scan chain operates.

The primary input values will be in the order of a one-to-one correspondence with the primary inputs defined in the setup section. The primary output values will also be in the order of a one-to-one correspondence with the primary outputs defined in the setup section.

If there is a test\_setup procedure in the Setup\_Data section, the first event, which is applying the test\_setup procedure, must occur before the first pattern is applied:

```
APPLY "test_setup" <value> <time>;
```

If there are any write control lines, they will be pulsed after the values have been applied at the primary inputs:

```
PULSE "write_control_input_name" <time>;
```

If there are capture clocks, then they will be pulsed at the same selected time, after the values have been measured at the primary outputs. Any scan clock may be used to capture the data into the scan cells that become observed.

Scan patterns will reference the appropriate test procedures to define how to control and observe the scan cells. If the contents of a master is to be placed into the output of its scan cell where it may be observed by applying the unload operation, the master\_observe procedure must be applied before the unloading of the scan chains:

```
APPLY "scan_group_master_observe_name" <value> <time>;
```

If the contents of a shadow is to be placed into the output of its scan cell where it may be observed by applying the unload operation, the shadow\_observe procedure must be applied before the unloading of the scan chains:

```
APPLY "scan_group_shadow_observe_name" <value> <time>;
```

If the master and slave of a scan cell are to be at different values for detection, the skew\_load procedure must be applied after the scan chains are loaded:

```
APPLY "scan_group_skew_load_name" <value> <time>;
```

Each scan pattern will have the property that it is independent of all other scan patterns. The normal scan pattern will contain the following events:

1. Load values into the scan chains.



2. Force values on all non-clock primary inputs.
3. Measure all primary outputs not connected to scan clocks.
4. Exercise a capture clock. (optional)
5. Apply observe procedure (if necessary)
6. Unload values from scan chains.

When unloading the last pattern, the tool loads the last pattern a second time to completely shift the contents of the last capture cycle so that the output matches the calculated value. For more information, see [“Handling of Last Patterns”](#) in the *Tessent TestKompress User’s Manual*.

Although the load and unload operations are given separately, it is highly recommended that the load be performed simultaneously with the unload of the preceding pattern when applying the patterns at the tester.

For observation of primary outputs connected to clocks, there will be an additional kind of scan pattern that contains the following events:

1. Load values into the scan chains.
2. Force values on all primary inputs including clocks.
3. Measure all primary outputs that are connected to scan clocks.

## Scan\_Cell

The scan\_cell section contains the definition of the scan cells used in the circuit. The scan cell data will be in the following format:

```
SCAN_CELLS =
  SCAN_GROUP "group_name1" =
    SCAN_CHAIN "chain_name1" =
      SCAN_CELL = <cellid> <type> <sciinv> <scoinv>
                  <relsciinv> <relscoinv> <instance_name>
                  <model_name> <input_pin> <output_pin>;
      ....
    END;
  SCAN_CHAIN "chain_name2" =
    SCAN_CELL = <cellid> <type> <sciinv> <scoinv>
                <relsciinv> <relscoinv> <instance_name>
                <model_name> <input_pin> <output_pin>;
    ....
  END;
  ....
END;
```

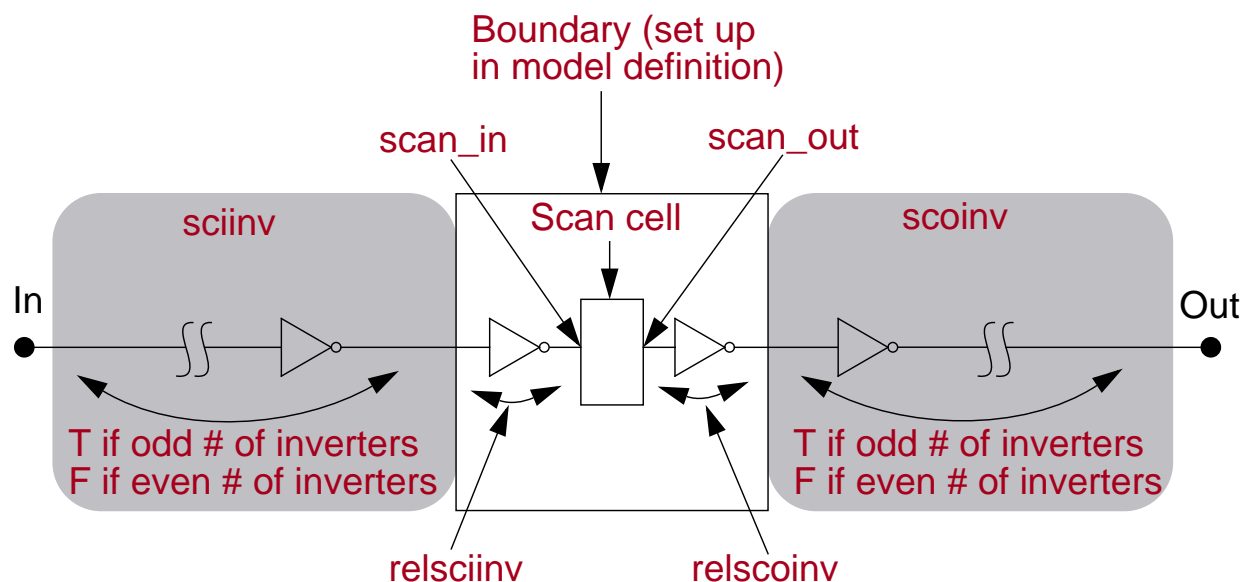
The fields for the scan cell memory elements are the following:

- **cellid** - A number that identifies the position of the scan cell in the scan chain. The number 0 indicates the scan cell closest to the scan-out pin.
- **type** - The type of scan memory element. The type may be MASTER, SLAVE, SHADOW, OBS\_SHADOW, COPY, or EXTRA.
- **sciinv** - Inversion of the library input pin of the scan cell relative to the scan chain input pin. The value may be T (inversion) or F (no inversion).
- **scoinv** - Inversion of the library output pin of the scan cell relative to the scan chain output pin. The value may be T (inversion) or F (no inversion).
- **relsciinv** - Inversion of the memory element relative to the library input pin of the scan cell. The value may be T (inversion) or F (no inversion).
- **relscoinv** - Inversion of the memory element relative to the library output pin of the scan cell. The value may be T (inversion) or F (no inversion).
- **instance\_name** - The top level boundary instance name of the memory element in the scan cell.
- **model\_name** - The internal instance pathname of the memory element in the scan cell (if used - blank otherwise).
- **input\_pin** - The library input pin of the scan cell (if it exists, blank otherwise).
- **output\_pin** - The library output pin of the scan cell (if it exists, blank otherwise).

## Example Circuit

Figure 10-1 illustrates the scan cell elements in a typical scan circuit.

**Figure 10-1. Example Scan Circuit**



# BIST Pattern File Format

All of the test pattern file formats described in the “[ASCII File Format](#)” section directly apply to BIST patterns also. However, an additional set of BIST-specific formats are added to the ASCII pattern file format. The BIST-specific formats are described in this section.

## Setup\_Data

The `setup_data` section for BIST contains a subsection that defines the BIST pattern-specific configuration. This subsection includes a file version identifier and definitions for signature registers (`prpg_register` and `misr_register`) in the BIST pattern.

A BIST-specific statement is used to identify that the following configuration is BIST pattern-specific. The BIST-specific statement has the following format:

```
declare bist pattern specific configuration =
```

A version tag is used to indicate the BIST pattern version. The version tag has the following format:

```
BIST_ASCII_PATTERN_FILE_SUBVERSION
```

Here is an example of a BIST pattern-specific statement combined with a version tag statement:

```
declare bist pattern specific configuration =  
BIST_ASCII_PATTERN_FILE_SUBVERSION = 1;
```

### Note



The “`bist pattern specific configuration`” statement disappears if a LFSM value is not included.

A `pattern_internal_view` switch that indicates whether the BIST pattern internal view is written to the pattern file; its possible values are “ON” or “OFF”. If the switch is set to on, as shown in the following example, BIST patterns are included in the pattern file. For information about pattern format types, refer to the [LBIST Architect Reference Manual](#).

```
pattern_internal_view = "on"
```

The `prpg_register` and `misr_register` identifiers define signature registers such as PRPG and MISR in the BIST pattern. The signature register statements use the following format:

```
signature_register <name of signature register> =  
  length = <length of the register>;  
  type = <type of the register>;  
  init_value = <initial state of the register>;  
end;
```

Here are some examples of signature register statements:

```
prpg_register "decomp1" =
    length = 22;
    type = PRPG;
    init_value = "0000000000000000000000";
end;

prpg_register "decomp2" =
    length = 12;
    type = PRPG;
    init_value = "000000000000";
end;

prpg_register "prpg1" =
    length = 16;
    type = PRPG;
    init_value = "0000000000000000";
end;

prpg_register "prpg2" =
    length = 16;
    type = PRPG;
    init_value = "0000000000000000";
end;

misr_register "misr1" =
    length = 32;
    type = MISR;
    init_value = "1111111111111111111111111111";
end;

misr_register "misr2" =
    length = 24;
    type = MISR;
    init_value = "111111111111111111111111";
end;
```

## Scan\_Test

Each BIST pattern includes one *lfsm\_snapshot* statement. Within the pattern statement, the keyword *pre\_load*, *pre\_unload*, or *post\_unload* is followed by a register name to indicate when the snapshot for the register is to be taken. The keywords are applied as follows:

- *pre\_load* — Used for PRPG type registers.
- *pre\_unload* and *post\_unload* — Used for MISR type registers.

The snapshot statement is composed so that the simulation of each BIST pattern is independent from all others:

- Given the *pre\_load* value of PRPG registers, the loading data for each BIST pattern can be computed.
- Given the *pre\_unload* value of MISR registers, the MISR registers after unload can be computed and thus verified.

Here is an example of a pattern containing a `lfsm_snapshot` statement:

```
pattern = 0;
lfsm_snapshot =
  pre_load "decomp1" = "000000000000000000000000";
  pre_load "decomp2" = "00000000000000";
  pre_load "prpg1" = "1111111111111111";
  pre_load "prpg2" = "1111111111111111";
  pre_unload "misr1" = "11111111111111111111111111111111";
  pre_unload "misr2" = "1111111111111111111111111111";
  post_unload "misr1" = "11111111110001111110111111011111";
  post_unload "misr2" = "1111111110000000001111111111";
end;

apply "grp1_load" 0 =
...
end;

force "PI" "...." 1;
measure "PO" "...." 2;

apply "grp1_unload" 3 =
...
end;
```



# Chapter 11

## Power-Aware DRC and ATPG

---

This chapter describes the power-aware DRC and ATPG flow for use with the ATPG tool, and contains the following sections:

- [Power-Aware Overview](#)
  - [Assumptions and Limitations](#)
  - [Multiple Power Mode Test Flow](#)
  - [Power-Aware ATPG for Traditional Fault Models](#)
- [CPF and UPF Parser](#)
- [Power-Aware ATPG Procedure](#)
- [Power-Aware Flow Examples](#)

## Power-Aware Overview

The electronics industry has adopted low-power features in major aspects of the design continuum. In response, EDA vendors and major semiconductor companies have defined the commonly-used power data standard formats to describe the power requirements: UPF and CPF. Tessent Shell supports the following versions of the UPF and CPF formats:

- IEEE 1801 standard / UPF 2.0
- Common Power Format (CPF) 1.0 and 1.1

You load this power data directly into the tool to collect the power information. Once loaded, the tools perform the necessary DRC's to ensure that the DFT logic is inserted properly with respect to the design's power domains and, if the design passes the rule checks, perform ATPG with the given power mode configuration. For information about the power-aware DRC (V) rules, refer to the [Tessent Shell Reference Manual](#).

The tool's low-power functionality provides you with a method to do the following:

- Provide DRC's to trace the active power mode and ensure that the scan operation works under the current power configuration.
- Provide capability to generate test for the traditional fault models while aware of the power mode configuration.

## Assumptions and Limitations

The power-aware functionality comes with the following assumptions and limitations:

- Circuit hierarchy is preserved in the CPF or UPF file and is the same as the netlist. Note that for modular EDT design, the module hierarchy that includes EDT logic needs to be preserved, otherwise the tool may not be able to associate an EDT block with a power domain. Consequently, some power DRC rules related to EDT logic may not be performed.
- The power-aware DRC rules and reporting are based on the loaded power data (UPF or CPF). The DRC rules do not include testing that crosses different power modes. Additionally, the DRC rules do not cover the testing of the shutoff power domain such as the retention cells testing.

## Multiple Power Mode Test Flow

For a design with multiple test modes, the scan chain configuration may be different for each power mode. You should perform the DRC and ATPG separately for each power mode using the following steps:

1. Configure the power mode to be tested using the test\_setup procedure.
2. Load the CPF file. The tool automatically identifies the power mode that the system is currently configured to and report it to user.

In addition, new V rules are defined and are checked when you switch to a to non-SETUP mode. (See “[Power-Aware Rules \(V Rules\)](#) in the *Tessent Shell Reference Manual*.) This ensure that the loaded UPF/CPF file contains consistent power information and the inserted DFT logic (for example, scan insertion) has considered the design power domains properly.

---

### Note



For the purpose of DRC, you must load to load a UPF/CPF file in the SETUP mode. When loading a UPF/CPF file in a non-SETUP mode, the tool does not perform V DRC until you switch the tool from the SETUP mode to a non-SETUP mode.

---

## Pattern Generation

ATPG generates patterns only for the current power mode. If there are different power modes which enable the identical power domains (just in different voltage configuration), then the pattern set can be reused by loading the pattern file and performing fault grading. Reuse of the pattern set is at your discretion.

Despite the pattern reuse, you should still write one test\_setup procedure for every power mode to be tested, and perform DRC check for each power mode to ensure the scan chains can operate properly in the power mode. In addition, a new pattern set should be stored to reflect the



updated test\_setup for the corresponding power mode. Finally, when writing out patterns after ATPG, the tool should also save the current power mode information (as a comment) to the pattern file for the user's reference.

## Power-Aware ATPG for Traditional Fault Models

During the traditional fault model test stage, the circuit is configured into a static power mode by the test procedure at the beginning of the capture cycle and stays at the same power mode for the entire capture cycle. The ATPG engine explicitly prevents the power control logic from changing the active power mode.

A low-power DRC is performed before ATPG to check if the active power mode can be disturbed and the analysis result is used by ATPG to determine if any additional ATPG effort is needed to keep the static power mode. This is similar to E10 rule for the bus contention check which is used by ATPG to enable the extra justification to prevent the bus contention. If the circuit contains the power mode that powers on all power domains (called ALL\_ON state), you can use this state for the traditional fault models. An ALL\_ON state allows the circuit to be tested for logic faults in one test set run. In addition, this state also allows the tool to perform DRC for entire circuit in this run.

## Power Partitioning

If the circuit needs to partition with partial power domains powered in a given time, then you must perform multiple ATPG runs; specifically, each run with its procedure file and the scan configuration. You must ensure that every power domain is covered by at least one run. Additionally, the chip-level test coverage can be computed manually from each separate run.

In the case of multiple power partition flow, the always on power domain, the faults may be targeted multiple times. To reduce the creation of patterns for same faults in the always-on domains, you can use the following command:

**add\_faults -power\_domains always\_on -delete**

This explicitly removes the faults in always\_on domain if the faults have been targeted by other run.

The other way to avoid creation of duplicated patterns is to load the previous generated fault list (using [read\\_faults](#) -merge) so that the detected faults by previous runs will not be re-targeted again.

## CPF and UPF Parser

Table 11-1 lists the power data commands relevant to the power-aware DRC and ATPG process. The other CPF and UPF commands are parsed by the tool but discarded.

**Table 11-1. Power Data Commands Directly Related to ATPG and DRC**

| Power Features                                                       | IEEE 1801 / UPF 2.0                                                                                                                                                                  | CPF 1.0 / CPF 1.1                                                                                                       |
|----------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| Power Domains                                                        | create_power_domain<br>add_domain_elements<br>create_composite_domain<br>merge_power_domain                                                                                          | create_power_domain<br>update_power_domain                                                                              |
| Power Modes<br>(Power States)                                        | add_power_state <sup>1</sup><br>add_pst_state<br>create_pst                                                                                                                          | assert_illegal_domain_configurations<br>create_assertion_control <sup>2</sup><br>create_power_mode<br>update_power_mode |
| State Transitions                                                    | describe_state_transition <sup>3</sup>                                                                                                                                               | create_mode_transition                                                                                                  |
| Power Network<br>(to derive power-on<br>domains and active<br>state) | add_port_state<br>create_power_switch<br>map_power_switch <sup>4</sup><br>set_power_switch<br>create_supply_port<br>create_supply_net<br>connect_supply_net<br>set_domain_supply_net |                                                                                                                         |
| Retention Cells                                                      | map_retention_cell<br>set_retention<br>set_retention_control<br>set_retention_elements <sup>5</sup>                                                                                  | create_state_retention_rule<br>update_state_retention_rule<br>define_state_retention_cell <sup>6</sup>                  |
| Isolation Cells                                                      | map_isolation_cell<br>set_isolation<br>set_isolation_control<br>use_interface_cell <sup>7</sup>                                                                                      | create_isolation_rule<br>update_isolation_rule<br>define_isolation_cell <sup>8</sup>                                    |
| Level Shifters                                                       | map_level_shifter_cell<br>set_level_shifter<br>use_interface_cell <sup>9</sup>                                                                                                       | create_level_shifter_rule<br>update_level_shifter_rule<br>define_level_shifter_cell <sup>10</sup>                       |

**Table 11-1. Power Data Commands Directly Related to ATPG and DRC**

| Power Features | IEEE 1801 / UPF 2.0                                                                         | CPF 1.0 / CPF 1.1                                                                                                                                                                                         |
|----------------|---------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Design Scope   | set_design_top<br>set_scope<br>upf_version<br>load_upf<br>load_upf_protected<br>Tcl support | Include<br>get_parameter<br>set_design<br>set_instance<br>set_cpf_version<br>set_time_unit<br>set_power_mode_control_group<br>set_array_naming_style<br>set_macro_model<br>end_macro_model<br>Tcl support |

1. add\_power\_state, describe\_state\_transition and map\_power\_switch are not used currently for DRC and ATPG purpose.
2. The CPF commands for defining power cells are not used by ATPG to identify the power cells that exist in the design. These commands may be needed during test synthesis to automatically insert power cells.
3. add\_power\_state, describe\_state\_transition and map\_power\_switch are not used currently for DRC and ATPG purpose.
4. add\_power\_state, describe\_state\_transition and map\_power\_switch are not used currently for DRC and ATPG purposes.
5. The UPF commands set\_retention\_elements and use\_interface\_cell are not used to identify the location of power cells. These commands may be needed during test synthesis to automatically insert power cells.
6. The CPF commands for defining power cells are used for DRC but are not used by ATPG to identify the power cells that exist in the design. These commands may be needed during test synthesis to automatically insert power cells.
7. The UPF commands set\_retention\_elements and use\_interface\_cell are not used to identify the location of power cells. These commands may be needed during test synthesis to automatically insert power cells.
8. The CPF commands for defining power cells are used for DRC but are not used by ATPG to identify the power cells that exist in the design. These commands may be needed during test synthesis to automatically insert power cells.
9. The UPF commands set\_retention\_elements and use\_interface\_cell are not used to identify the location of power cells. These commands may be needed during test synthesis to automatically insert power cells.
10. The CPF commands for defining power cells are used for DRC but are not used by ATPG to identify the power cells that exist in the design. These commands may be needed during test synthesis to automatically insert power cells.

## Power-Aware ATPG Procedure

When using the power-aware ATPG flow, Mentor Graphics recommends separating the test for the logic faults from the test for the low-power features. When testing the logic faults, the traditional fault models (for example, the stuck-at fault model and the transition fault model) are applied and the power switch logic need not to be presented.

The power data in UPF or CPF format should be applied to the tool so the low-power rules (see “[Power-Aware Rules \(V Rules\)](#)”) can be checked and the circuit can be simulated according to the active power mode.

In general, you perform the following steps with the power-aware ATPG flow, either directly from the command line or scripted in a dofile:

1. Invoke the Tessent Shell, set the context the “patterns -scan,” and read in the low-power design. For example:

```
SETUP> read_verilog low-power_design.v
```

2. Add scan chains and other configuration information as appropriate.
3. Load the power data using the [read\\_cpf](#) or [read\\_upf](#) command. For example:

```
SETUP> read_upf test.upf
```

4. Set the mode to analysis. For example:

```
SETUP> set_system_mode analysis
```

At this point, the tool performs the DRC checks. See “[Power-Aware Rules \(V Rules\)](#) in the *Tessent Shell Reference Manual*.

5. Add faults using the applicable power-aware switch to the [add\\_faults](#) command. For example:

```
SETUP> add_faults -on_domains
```

6. Optionally write the faults to a fault list for multiple power-aware test methodologies—see “[Multiple Power Mode Test Flow](#).” For example:

```
SETUP> write_faults on_domain_fault_list.txt
```

7. Create the patterns using the `create_patterns` command.

The power-aware ATPG flow supports reporting mechanisms tailored to the power information. See the following commands for more information:

- [report\\_faults](#)
- [report\\_power\\_data](#)

# Power-Aware Flow Examples

## Example 1

Table 11-2 shows one example design with four power domains and four power modes.

**Table 11-2. Example Design With Four Power domains and Power Modes**

|             |                  | Power Domains |        |       |                    |
|-------------|------------------|---------------|--------|-------|--------------------|
| Power Modes | CPU              | MEM1          | CTL    | Radio | test_setup file    |
| active      | Active           | Active        | ON     | Tx,Rx | test_setup_active  |
| standby     | Idle             | Sleep         | ON     | Rx    | test_setup_standby |
| idle        | Sleep            | Sleep         | ON     | Rx    | test_setup_idle    |
| sleep       | Sleep            | Sleep         | OFF    | Off   | N/A                |
| scan chains | chain1<br>chain2 | chain3        | chain4 | none  |                    |

The design contains four scan chains arrayed as follows:

- **Chain1 and Chain 2** — Located in the CPU power domain.
- **Chain3** — Located in MEM1 power domain.
- **Chain4** — Located in the CTL power domain.

To test the design requires multiple test\_setup procedures, one for each power mode except for the one turning off all power domains. To reduce the maintenance overhead of test procedure files, test\_setup procedures are written in the following separate files:

- test\_setup\_active
- test\_setup\_standby
- test\_setup\_idle

The main test procedure file, which contains the rest of procedures, uses an include statement to include the test\_setup for the power mode to be tested. For example, using the following include statement:

```
# include test_setup_active"
```

will test the “active” power mode.

The recommended flow is to test the power mode with all power domains active first, if such power mode exists (power mode “active” in Table 11-2). This allows the tool to view all scan chains and check the “Power-Aware Rules (V Rules)” rules crossing all power domains. Note

that when testing different power modes, the scan chains need to be defined accordingly to prevent DRC violations. For example when testing “standby” mode where “MEM1” power domain is off, chain3 should not be defined, otherwise it is a [V8](#) DRC violation.

See “[Power-Aware Rules \(V Rules\)](#)” in the *Tessent Shell Reference Manual* for a complete discussion.

## Example 2

In this example, the design contains the following:

- Three power domains {D1, D2, D3}
- Two power mode
  - S1: (D1=ON, D2=ON, D3=OFF)
  - S2: (D1=OFF, D2=ON, D3=ON)

Additionally, assume that the design holds PS1 property (specifically, during the shift cycles and capture cycles), and the design can be kept in the same power mode.

To test the design, you must perform the following *multiple* tool runs:

- **run 1** — ATPG for power mode S1
  - a. Configure the design to power mode S1 by the end of test\_setup.
  - b. Add scan chains only in power domain D1 and D2.
  - c. Add faults only in the power ON domains (D1 and D2) using the [add\\_faults](#) –on\_domains command.
  - d. Create patterns and write patterns.
  - e. Write faults to a file named *flist\_S1.txt* using the write\_faults command.
  - f. Write isolation faults to a file named *flist\_S1\_iso.txt* using write\_faults –isolation to save for later use.
  - g. Write level-shifter faults to a file named *flist\_S1\_ls.txt* using write\_faults –level\_shifter to save for later use.
- **run 2** — ATPG for power mode S2
  - a. Configure the design to power mode S2 by the end of test\_setup.
  - b. Add scan chains only in power domain D2 and D3.
  - c. Add faults only in the power ON domains (D2 and D3) using [add\\_faults](#) –all command.

- d. Load the previously-saved S1 fault list using the following command:  
**ANALYSIS> read\_faults flist\_S1.txt –merge –Power\_check on**  
This updates the fault status of faults in D2 but discard faults in D3 as they are not power on faults. See the [read\\_faults](#) command for more information.
- e. Create patterns and write patterns.
- f. Write faults to a file named *flist\_S2.txt* using the `write_faults` command.
- g. Write isolation faults to a file named *flist\_S2\_iso.txt* using `write_faults –isolation` to save for later use.
- h. Write level-shifter faults to a file named *flist\_S2\_ls.txt* using `write_faults –level_shifter` to save for later use.

After run 2, you can calculate the entire fault coverage by loading multiple fault lists into the tool. You do this by using the [read\\_faults –Power\\_check OFF](#) command and switch as shown in the following steps:

- **run 3** — Overall test coverage report
  - a. Load the previously-saved fault list for D1:  
**ANALYSIS> read\_faults flist\_S1.txt –merge –Power\_check off**
  - b. Load the previously-saved fault list for D2:  
**ANALYSIS> read\_faults flist\_S2.txt –merge –Power\_check off**
  - c. Issue the `report_statistics` command to report the overall test coverage.
  - d. Write the faults to a file named *faults flist\_all.txt* using the `write_faults` command.

The final tool run reports isolation fault test coverage ATPG and level-shifter fault test coverage and saves these in separate fault lists.

1. Report the isolation fault test coverage:  
**ANALYSIS> report\_statistics -isolation**
2. Report the level-shifter fault test coverage:  
**ANALYSIS> report\_statistics –level\_shifter**
3. Write all isolation faults to a file:  
**ANALYSIS> write\_faults flist\_all\_iso.txt –isolation**
4. Write all level-shifter faults to a file:  
**ANALYSIS> write\_faults flist\_all\_ls.txt –level\_shifter**





# Chapter 12

## Testing Low-Power Designs

---

This chapter describes the low-power scan insertion and ATPG flow for use with Tessent Scan and the ATPG tool, and contains the following sections:

|                                             |            |
|---------------------------------------------|------------|
| <b>Low-Power Testing Overview</b> .....     | <b>378</b> |
| Assumptions and Limitations .....           | 378        |
| Low-Power CPF/UPF Parameters .....          | 378        |
| <b>Test Insertion</b> .....                 | <b>380</b> |
| Low-Power Test Flow .....                   | 380        |
| Scan Insertion with Tessent Scan .....      | 381        |
| <b>Power-Aware Design Rule Checks</b> ..... | <b>386</b> |
| Low-Power DRCs .....                        | 386        |
| Low-Power DRC Troubleshooting .....         | 386        |
| <b>Power State-Aware ATPG</b> .....         | <b>388</b> |
| Power Domain Testing .....                  | 388        |
| Low-Power Cell Testing .....                | 390        |

## Low-Power Testing Overview

Tessent Scan supports the following operations:

- Insertion of dedicated wrapper cells in the presence of isolation cells.
- Assigning dedicated wrapper cells to power domains based on the logic it is driving and the priority of the power domains.

The low-power design flow includes the following steps:

1. Specify low-power data specifications in the CPF/UPF file. See “[Low-Power CPF/UPF Parameters](#)”.
2. Insert scan cells and EDT logic in the design. See “[Test Insertion](#)”.
3. Validate low-power data, scan, and EDT logic. See “[Power-Aware Design Rule Checks](#)”.
4. Generate power domain-aware test patterns. See “[Power State-Aware ATPG](#)”.
5. Test low-power design components.

## Assumptions and Limitations

The power-aware functionality comes with the following assumptions and limitations:

- Tessent Scan does not write or update CPF/UPF files. Tessent Scan assumes that everything that belongs to a power domain is explicitly listed in the UPF/CPF file. Because of this assumption, you must make the following changes to the UPF/CPF file after the scan insertion step and before ATPG:
  - Add any inserted input wrapper cells to the correct power domain in the CPF/UPF file.
  - Explicitly define all isolation cells and their control signals, and level shifters in the CPF/UPF file. Tessent Scan checks for their presence but does not add them. For more information, see “[Scan Insertion with Tessent Scan](#).”

## Low-Power CPF/UPF Parameters

Tessent Scan uses CPF/UPF files to identify the power domains in the design and to constrain scan insertion within the power domain boundaries. Tessent Scan supports power structure and configuration data from the following formats:

- CPF 1.0 and 1.1
- UPF 2.0 (IEEE1801)

If you are a member of Si2, you can access complete information about CPF standards at the following URL:

[https://www.si2.org/openeda.si2.org/project/showfiles.php?group\\_id=51](https://www.si2.org/openeda.si2.org/project/showfiles.php?group_id=51)

You can purchase complete information on UPF standards at the following URL:

[http://www.techstreet.com/standards/ieee/1801\\_2009?product\\_id=1744966](http://www.techstreet.com/standards/ieee/1801_2009?product_id=1744966)

In the CPF/UPF file, you must specify power domains and information related to the power domains, power states (modes) of the design, and isolation cells and control signals. You can use the commands listed in [Table 12-1](#) to specify this information.

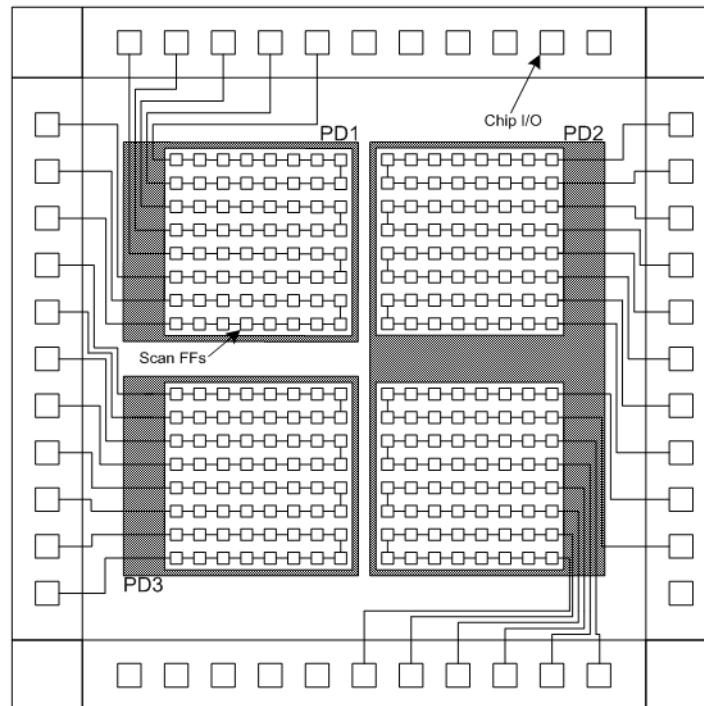
**Table 12-1. Power Data Commands Directly Related to ATPG**

| Power Features                | IEEE 1801 / UPF 2.0                                                                         | CPF 1.0 / CPF 1.1                                                                                          |
|-------------------------------|---------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| Power Domains                 | create_power_domain<br>add_domain_elements<br>create_composite_domain<br>merge_power_domain | create_power_domain<br>update_power_domain                                                                 |
| Power Modes<br>(Power States) | add_power_state<br>add_pst_state<br>create_pst                                              | assert_illegal_domain_configurations<br>create_assertion_control<br>create_power_mode<br>update_power_mode |
| Isolation Cells               | set_isolation_command<br>set_isolation_control<br>map_isolation_cell                        | create_isolation_rule<br>define_isolation_cell<br>update_isolation_cell                                    |
| Level Shifters                | map_level_shifter_cell                                                                      | create_level_shifter_rule<br>update_level_shifter_rule                                                     |

## Test Insertion

Test insertion includes scan substitution and stitching. You can insert scan chains on a power domain basis. Alternately, you can group test logic based on power states. [Figure 12-1](#) show an example of chains inserted on a power domain basis. Note, no scan chains span the power domains because the tool does not insert scan across power domains.

**Figure 12-1. Scan Chain Insertion by Power Domain**



## Low-Power Test Flow

This section outlines the steps for first loading low-power data and then using the loaded power domain information to guide the process of scan partitioning and scan insertion.

1. Load the design data and libraries.
2. Read the CPF or UPF file:

**`read_cpf filename`**

**`read_upf filename`**

3. Set up appropriate design constraints.
4. Enter analysis mode:

**`SETUP> set_system_mode analysis`**

5. If wrapper cells are enabled, scan chain partitioning will be affected. Refer to [“Managing Wrapper Cells with Power Domains”](#) for more information.

- Report the scan partitions created based on the power domain information in the CPF/UPF file:

```
> report_scan_partitions -all
```

| ScanPartitionName      | TotalNumCells/ScannableCells | Members               |
|------------------------|------------------------------|-----------------------|
| PD_interleaver         | 16693/16693                  | i0 [instance]         |
| PD_mem_ctrl            | 12/12                        | mc0 [instance]        |
| default_scan_partition | 9/9                          | <all_remaining_cells> |

- Specify scan chains based on maximum scan chain length or scan chain number using the `set_power_domain` command:

```
set_power_domain {{Power_domain_name... | -All [-EDT]}
[-Number_of_chains integer | -Max_length integer]}
```

For example:

```
> set_power_domain PD_interleaver -number 65
> set_power_domain PD_mem_ctrl -max_length 6
```

Any scan flip-flop not specified by the `set_power_domain` command is added to the default top-level power domain.

- Insert the test structures into the design netlist using the `insert_test_logic` command.
- Report the design statistics using the `report_statistics` command.
- Write out the netlist and ATPG setup.

## Scan Insertion with Tessent Scan

|                                                                |     |
|----------------------------------------------------------------|-----|
| Managing Scan Enable Signals for Multiple Power Domains .....  | 383 |
| Managing Subchains and Wrapper Chains with Power Domains ..... | 383 |
| Managing Wrapper Cells with Power Domains .....                | 384 |

You can use Tessent Scan to do both scan cell substitution and scan stitching. By loading the low-power CPF or UPF file for the design before beginning the scan insertion and stitching process, the power domain information can be used to guide the scan partitioning process. Tessent Scan does not insert scan across power domains.

You can explicitly specify the number and length of scan chains in each power domain of the design using the `set_power_domain` command. If you do not use `set_power_domain`, Tessent Scan adopts the default settings based on the following command/options settings:

- `insert_test_logic`: `-MAx_length` and `-NNumber` arguments
- `add_scan_partition`: `-NNumber` and `-MAx_length` arguments

- set\_wrapper\_chains: *-INPUT\_NUMber*, *-INPUT\_MAX\_length*, *-OUTPUT\_NUMber*, and *-OUTPUT\_MAX\_length* arguments

---

**Note**



Because Tessent Scan does not function with a concept of power mode or power state, chain balancing does not occur during low-power scan insertion.

---

During scan insertion, the tool does not generate a design that is clean in terms of power domains. Tessent Scan creates a design that is functionally correct in the context of power domains but it does not insert isolation cells or level shifters when routing global signals across power domains. You will need to add these objects themselves using a synthesis or physical implementation tool and then manually add them to the CPF/UPF file.

For more information on isolation cells and level shifters, see “[Low-Power Cell Testing](#)”.

## Managing Scan Enable Signals for Multiple Power Domains

The low-power test flow allows you to insert multiple scan enable signals in your design. This means you can insert a scan enable signal for each power domain in the design; you can also insert multiple scan enable signals for a single power domain. The behavior of multiple scan enable signals is the same regardless of whether your design has no power domains defined or has multiple power domains defined.

You assign one scan enable signal to a power domain by using the `set_scan_enable` command and specifying the power domain with the *partition\_name* argument as shown in the example here:

```
set_scan_enable scan_en_pin_path pd1 pd2
```

For more information on assigning a scan enable signal to a power domain, refer to the [set\\_scan\\_enable](#) description in the *Tessent Shell Reference Manual*.

To assign more than one scan enable signal to a single power domain, you use the `set_scan_enable_sharing` command and specify the *-scan\_partition* switch which assigns a unique scan enable signal to each of the specified power domains. By default, the types of the added scan enable signals are different for the core chains, input wrapper chains, and output wrapper chains

```
set_scan_enable_sharing -prefix base_name -scan_partition
```

## Managing Subchains and Wrapper Chains with Power Domains

By default, the `set_scan_enable` command assigns scan enable signals to core scan chains. You can use the *-wrapper\_chain* argument to specify the *type* of scan chain the scan enable signals are to be assigned to.

For more information, refer to the [set\\_scan\\_enable](#) description in the *Tessent Shell Reference Manual*.

## Managing Wrapper Cells with Power Domains

Newly-inserted dedicated wrapper cells are stitched into wrapper chains based on the power domain affiliation of the wrapper cells (dedicated and shared) within the wrapper chains. The tool identifies the power domain a dedicated wrapper cell is affiliated with during wrapper cell identification; the power domain is chosen based on which power domain the PI or PO being registered is connected to. If a PI is connected to more than one power domain, the dedicated wrapper cell is associated with the first encountered power domain.

Figure 12-2 shows the stitching of the input wrapper cells for each power domain. The newly added dedicated wrapper cells are stitched with the shared wrapper cells in the same power domain.

**Figure 12-2. Input Wrapper Cells and Power Domains**

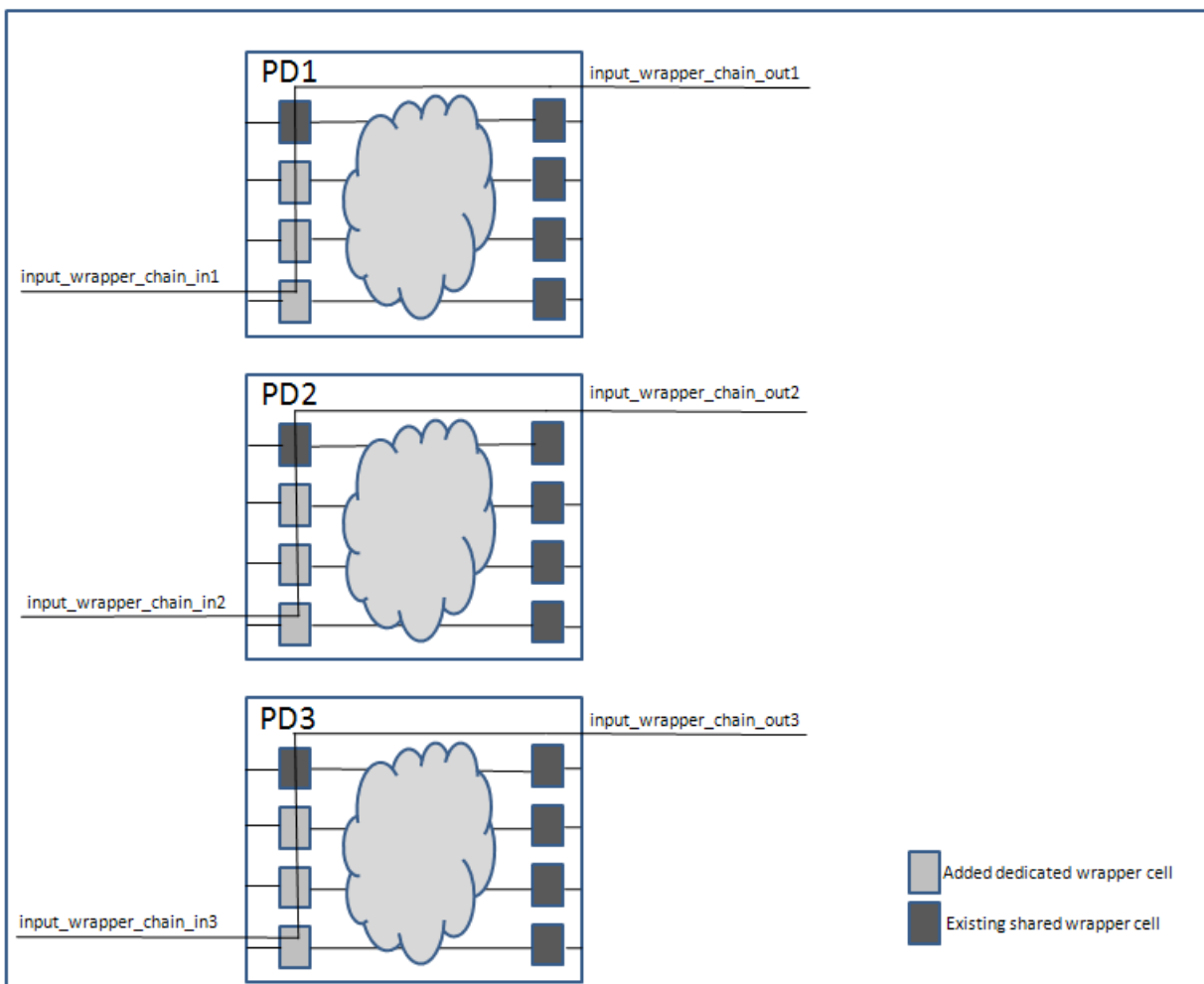
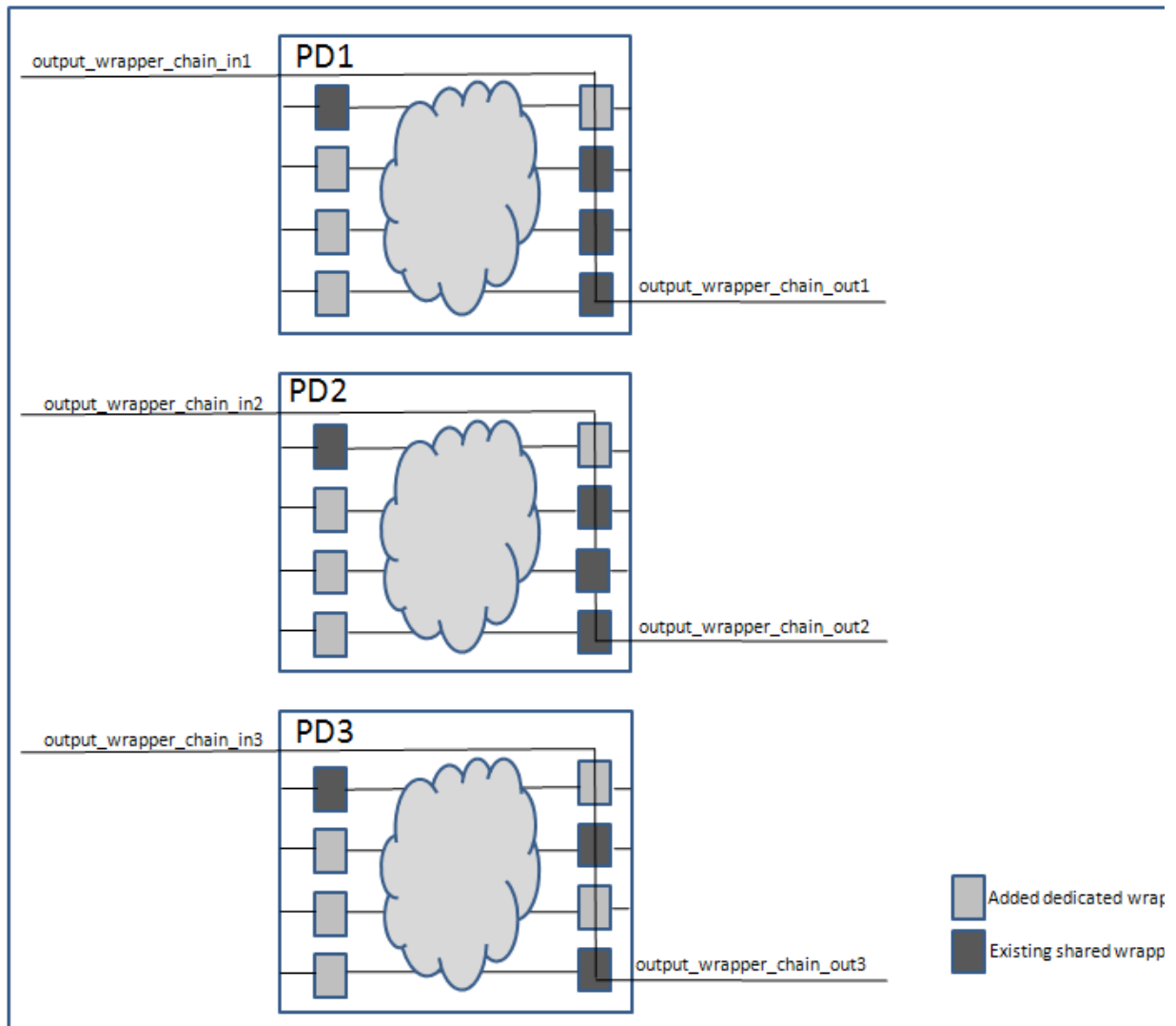




Figure 12-2 shows the stitching of output wrapper cells for each power domain. The newly added dedicated wrapper cells are stitched with the shared wrapper cells in the same power domain.

**Figure 12-3. Output Wrapper Cells and Power Domains**



## Power-Aware Design Rule Checks

Tessent Scan performs power-aware design rule checking to validate power data and power-aware reporting to facilitate troubleshooting design rule violations.

### Low-Power DRCs

In addition to checking design logic and the cell library, power-aware design rules (DRCs V1-V21) check the design for violations and correctness of power data provided through CPF/UPF files.

---

#### Note



DFTVisualizer does not have any direct support for power domain-related DRC failures. However, you can use standard design tracing features in DFTVisualizer to pinpoint issues.

---

- DRCs V1 to V7 — Checked after reading a power data file.
- DRCs V8 to V21 — Checked when you switch from setup mode to a non-setup mode.

For information on specific V DRCs, see “[Power-Aware Rules \(V Rules\)](#)” in the *Tessent Shell Reference Manual*.

### Low-Power DRC Troubleshooting

You can use the [report\\_scan\\_partitions](#) command to generate a scan cell report that includes power domain information as shown in the following example. Tessent Scan treats power domains as scan partitions during scan chain stitching.

The `report_scan_partitions` command extracts all updated information from the CPF/UPF files and reports each power domain (scan partition) and the pathnames of all of the sequential instances in that power domain; newly added dedicated wrapper cells and lockup cells are indicated by the string “(new cell)” following their name.

```
DFT> report_scan_partitions -all -expand
```

```
-----  
ScanPartitionName  Members  
-----  
PD2                I6  
                    I10/lckup3 (new cell)  
                    I10/lckup2 (new cell)  
                    I10/lckup1 (new cell)  
                    I10/outreg2 (new cell)  
                    I10/outreg1 (new cell)  
                    I10/sf_0001_1  
                    I10/sf_0002_1  
                    I10/sf_0003_1  
PD3                I2/I2/lckup3 (new cell)
```

```
I2/I2/lckup2 (new cell)
I2/I2/lckup1 (new cell)
I2/I2/outreg1 (new cell)
I2/I2/sf_0001_1
I2/I2/sf_0002_1
I2/I2/sf_0003_1
I2/I2/I1_2
DEF      lckup2 (new cell)
          lckup1 (new cell)
          inreg2 (new cell)
          inreg1 (new cell)
          outreg1 (new cell)
-----
```

## Reporting Power Domains for Scan Cells

You can use the `report_scan_cells` command to generate a scan cell report that includes power domain information as shown here:

```
ANALYSIS> report_scan_cells -power_domain
```

For more information, refer to the [report\\_scan\\_cells](#) description in the *Tessent Shell Reference Manual*.

## Reporting Power Domains for Gates

You can use the `set_gate_report` command to add power domain information to the results of the `report_gates` report as shown in the example here:

```
ANALYSIS> set_gate_report -power on
```

```
ANALYSIS> report_gates 154421
```

```
// /mc0/present_state_reg_3_/inst1122_4/mlc_dff (154421) DFF
// "S"      I(ON)  153682-
// "R"      I(ON)  153681-
// "C0"     I(ON)  26078-
// "D0"     I(ON)  43013-
// "OUT"    O(ON)  1115-
// MASTER   cell_id=0 chain=chain66 group=grp1 invert_data=FFFF
// in power domain PD_mem_ctrl
```

For more information, refer to the [set\\_gate\\_report](#) description in the *Tessent Shell Reference Manual*.

## Tracing Power Domain Violations

You can use DFTVisualizer design tracing capabilities to pinpoint failures and issues reported during design rule checking. For more information on pinpointing DRC violations in DFTVisualizer, see “[Analysis a DRC Violation](#)” in the *Tessent Shell Reference Manual*.

## Power State-Aware ATPG

The recommended method for ATPG is to have all power domains on if possible. With this method, you can manage power considerations due to switching activity during test by sequencing the test of design blocks and using low-power ATPG. This is the simplest approach and maximizes fault coverage. If you cannot use this method due to static power limitations, you should select a minimum set of power states to achieve the required fault coverage for all the blocks in the design as well as inter-block connectivity.

## Power Domain Testing

You can run ATPG in multiple power modes and power states by setting the appropriate power mode/state conditions as specified in the CPF or UPF. To demonstrate, assume a design has the following two power modes:

- PM1 — All domains on
- PM2 — *PD\_mem\_ctrl* powered down when *mc\_pwr* is “0”

By setting a pin constraint on *mc\_pwr* as shown here, ATPG recognizes which power mode is currently set and runs appropriately:

```
SETUP> add_input_constraints mc_pwr -c1
SETUP> set_system_mode analysis
...
// -----
// Begin circuit learning analyses.
// -----
...
SETUP> create_patterns
No faults in fault list. Adding all faults...
...
// Enabling power-aware ATPG: initial power mode in capture cycle = PM2
...
```

If the power mode can change during the capture cycle, the tool will issue V12 violations. When V12 handling is a warning (default), the tool automatically adds ATPG constraints to fix the power mode during the capture cycles:

```
...
// 1 ATPG constraint is added to fix power mode during capture.
// The active power mode at the beginning of capture cycle is PM2.
// Power-aware ATPG is enabled.
...
```

When power-aware ATPG is enabled, the tool performs ATPG and fault simulation according to the power mode: the regular gates in a power-off domain will be X and the corresponding faults are AU. The fault grouping will classify the fault as AU due to power off if faults in a power domain are added:

ANALYSIS> **create\_patterns**

ANALYSIS> **report\_statistics**

```
-----
FU (full)                                354
-----
DS (det_simulation)      52 (14.69%)
DI (det_implication)     40 (11.30%)
UU (unused)              26 ( 7.34%)
RE (redundant)           134 (37.85%)
AU (atpg_untestable)     102 (28.81%)
-----
Untested Faults
-----
AU (atpg_untestable)
  POFF (power_off)       102 (28.81%)
-----
Coverage
-----
test_coverage            47.42%
fault_coverage           25.99%
atpg_effectiveness       100.00%
-----
```

Instead of adding all faults, you can choose to add, delete, report, and write faults on power-on domains or any user-specified power domains:

```
add_/delete_/report_/write_faults
[[-ON_domains] | [-OFF_domains] |
[ -POWER_domains {domain_name ...}]]
[ -ISolation_cells] [ -LEvel_shifters ] [ -REtention_cells ]
```

## Low-Power Cell Testing

The following sections describe issues related to test coverage.

### Level Shifters

In most cases, you can handle level shifters as standard buffers and do not need any special handling to achieve full test coverage. However, if one or more power domains in the design can operate at more than one supply voltage, you should run ATPG for all permutations of the supply voltage for both input and output sides of any given level shifter to ensure full coverage.

### Isolation Cells

You can test isolation cells in one of two modes:

- Normal transmission mode — In this mode, when an isolation cell is disabled and stuck-at “0” or “1”, the fault can be detected on the input and output of the cell. A stuck-at fault (isolation on) for the isolation enable pin is also detected because this failure forces the cell into isolation and prevents the transmission of data.
- Isolation mode — In this mode, input side stuck-at faults are not detectable if the isolation cell is a clamp-style cell (output held to “0” or “1” when isolation on). If the isolation cell is a latch-style cell, input side stuck-at faults are detectable by latching the value of the driving domain before enabling isolation.

## Introduction

This chapter describes MTFI (Mentor Tessent Fault Information) features, which are available for use in the ATPG tool and Tessent LogicBIST. MTFI is a common and extendable file format for storing fault status information. This chapter describes MTFI syntax beginning on [page 392](#), as well as the major MTFI features:

- “Support of Fault Classes and Sub-Classes” on [page 395](#)
- “Support of Stuck and Transition Fault Information” on [page 397](#)
- “Support of N-Detect Values” on [page 397](#)
- “Support of Different Fault Types in the Same File” on [page 399](#)
- “Support for Hierarchical Fault Accounting” on [page 399](#)

[Table 13-1](#) summarizes the commands that support MTFI.

**Table 13-1. MTFI Command Summary**

| Command                       | Description                                                                |
|-------------------------------|----------------------------------------------------------------------------|
| <a href="#">read_faults</a>   | Updates the current fault list with the faults listed in a specified file. |
| <a href="#">report_faults</a> | Displays data from the current fault list.                                 |
| <a href="#">write_faults</a>  | Writes data from the current fault list to a specified file.               |

## MTFI Syntax

This section describes the basic syntax of MTFI files.

Here is a simple MTFI example that defines two Stuck faults:

```
FaultInformation {
  version : 1;
  FaultType (Stuck) {
    FaultList {
      Format : Identifier, Class, Location;
      Instance ("/i1") {
        0, DS, "F1";
        1, DS, "F2";
      }
    }
  }
}
```

The basic syntax elements of MTFI are as follows:

- **Comments**  
The “//” identifies the start of the comment until the end of line.
- **FaultInformation { ... }**  
Keyword that specifies the top-level block and file type.
- **version : *integer***  
Keyword that specifies the syntax version of the MTFI file.
- **FaultType (*type*) { ... }**  
Keyword that specifies the fault type of the data: Stuck, Iddq, Toggle, Transition, Path\_delay, Bridge, and Udfm. The keywords are identical to those available for the [set\\_fault\\_type](#) command described in the *Tessent Shell Reference Manual*.
- **FaultList { ... }**  
Keyword that defines a data block that stores per-fault data.
- **UnlistedFaultsData { ... }**  
Keyword that defines a data block that stores the coverage information for specific graybox instances to allow hierarchical fault accounting. For more information, refer to [“Support for Hierarchical Fault Accounting”](#) on page 399.
- **FaultCollapsing { true | false }**  
Keyword that specifies the fault collapsing status in the fault list. A value of “true” means that the faults in the list are collapsed; a value of “false” means that the faults in the list are uncollapsed. You can use this statement only inside the FaultList data block.



- **DetectionLimit** : *integer*

Keyword that specifies the detection drop limit for a DS fault.

- **Format**

Keyword that specifies the sequence of values stored in a specific data block (FaultList or UnlistedFaultsData). The following keywords are available for use in the Format statement:

- **Class**

Required keyword that specifies the fault category, and optionally the sub category (for example, DS, UC, AU.BB). For more information, refer to [“Support of Fault Classes and Sub-Classes”](#) on page 395.

- **Location**

Keyword that specifies the pin pathname. You can use this keyword only in the FaultList data block. In the case of a stuck-at fault, the keyword is required.

- **Identifier**

Required keyword that specifies the fault identifier. In the case of stuck-at or transition faults, the value can be 0 or 1. No other fault types are supported.

- **Detections**

Optional keyword that specifies the number of detections for the fault types stuck-at and transition. For more information, refer to [“Support of N-Detect Values”](#) on page 397.

- **CollapsedFaultsCount**

Required keyword that specifies the number of collapsed faults. You can use this keyword only in the UnlistedFaultsData data block.

- **UncollapsedFaultsCount**

Required keyword that specifies the number of uncollapsed faults. You can use this keyword only in the UnlistedFaultsData data block.

- **Instance** ( *“pathname”* ) { ... }

Required keyword that specifies the instance pathname to all of the pins in the data block. You must enclose the pathname in parentheses and double quotes. The pathname can be an empty string.

## MTFI File Example

The following is an example of a typical MTFI file:

```
//  
// Tessent FastScan v9.6  
//  
// Design = test.v  
// Created = Tue Dec 20 20:08:46 2011  
//  
// Statistics:  
//   Test Coverage   = 50.00%  
//   Total Faults    = 6  
//   UC (uncontrolled) = 2  
//   DS (det_simulation) = 1  
//   DI (det_implication) = 1  
//   AU (atpg_untestable) = 2  
//  
FaultInformation {  
  version : 1;  
  FaultType ( Stuck ){  
    FaultList {  
      FaultCollapsing : false;  
      Format : Identifier, Class, Location;  
      Instance ( "" ) {  
        1, DS, "/i1/IN0";  
        1, AU, "/i1/IN1";  
        1, EQ, "/i2/Y";  
        0, UC, "/i5/Z";  
        1, DI, "/i5/Z";  
        0, AU, "/i4/IN1";  
        1, UC, "/i4/IN1";  
      }  
    }  
  }  
}
```

## MTFI Features

The following text explains the major features of the MTFI format:

- [“Support of Fault Classes and Sub-Classes”](#) on page 395
- [“Support of Stuck and Transition Fault Information”](#) on page 397
- [“Support of N-Detect Values”](#) on page 397
- [“Support of Different Fault Types in the Same File”](#) on page 399
- [“Support for Hierarchical Fault Accounting”](#) on page 399

## Support of Fault Classes and Sub-Classes

The following example shows how MTFI supports fault classes and sub-classes using the Format statement's Class keyword:

```
FaultInformation {
  version : 1;
  FaultType ( Stuck ){
    FaultList {
      FaultCollapsing : false;
      Format : Identifier, Class, Location;
      Instance ( "" ) {
        1, DS, "/i1/IN0";
        1, AU, "/i1/IN1";
        1, EQ, "/i2/Y";
        0, UC, "/i5/Z";
        1, DI, "/i5/Z";
        0, AU, "/i4/IN1";
        1, UC, "/i4/IN1";
      }
    }
  }
}
```

In the following example, one AU fault is identified as being AU due to black box, and one DI fault is specified as being due to LBIST. So for these two faults, the MTFI file reports both class and sub-class values.

```
FaultInformation {
  version : 1;
  FaultType ( Stuck ) {
    FaultList {
      FaultCollapsing : false;
      Format : Identifier, Class, Location;
      Instance ( "" ) {
        1, DS,      "/i1/IN0";
        1, AU,      "/i1/IN1";
        1, EQ,      "/i2/Y";
        0, UC,      "/i5/Z";
        1, DI,      "/i5/Z";
        1, DI.LBIST, "/i5/Z";
        0, AU.BB,   "/i4/IN1";
        1, UC,      "/i4/IN1";
      }
    }
  }
}
```

In the preceding example, the sub-class information is part of the class value and separated by the dot. Any of the AU, UC, UO and DI fault classes can be further divided into different sub-classes. For all the available AU fault sub-classes, refer to the [set\\_relevant\\_coverage](#) command description in the *Tessent Shell Reference Manual*. The tool supports the following UC and UO

fault sub-classes: ATPG\_Abort (AAB), Unsuccessful (UNS), EDT Abort (EAB). The tool supports the pre-defined fault sub-class EDT for DI faults.

## User-Defined Fault Sub-Classes

Besides the predefined fault sub-classes described previously, the tool also supports a user-defined sub-class for AU and DI faults. The user-defined sub-class can be any string based on the following character set:

- A-Z
- a-z
- 0-9
- -
- \_

It is your responsibility to ensure that the name of the user-defined sub-class differs from any of the predefined sub-classes. Otherwise, the faults may be accounted for incorrectly.

The statistics report displays the breakdown of DI faults when there are some DI faults in specific subcategories. The following example shows the statistics report with seven DI faults declared in different subcategories; in this example, the sub-class analysis for DI.EDT is disabled:

### > report\_statistics

| Statistics Report<br>Stuck-at Faults |                    |
|--------------------------------------|--------------------|
| Fault Classes                        | #faults<br>(total) |
| FU (full)                            | 61                 |
| DS (det_simulation)                  | 4 ( 6.56%)         |
| DI (det_implication)                 | 7 (11.48%)         |
| UU (unused)                          | 13 (21.31%)        |
| BL (blocked)                         | 1 ( 1.64%)         |
| AU (atpg_untestable)                 | 36 (59.02%)        |
| DI Faults                            |                    |
| DI (det_imp)                         |                    |
| DUMMY                                | 5 ( 8.20%)         |
| DUMMY_alias                          | 1 ( 1.64%)         |
| dummy                                | 1 ( 1.64%)         |
| Untested Faults                      |                    |
| AU (atpg_untestable)                 |                    |
| TC* (tied_cells)                     | 27 (44.26%)        |
| my_TC                                | 4 ( 6.56%)         |

|                                                          |            |
|----------------------------------------------------------|------------|
| mya_TC                                                   | 5 ( 8.20%) |
| *Use "report_statistics -detailed_analysis" for details. |            |
| -----                                                    |            |
| Coverage                                                 |            |
| -----                                                    |            |
| test_coverage                                            | 23.40%     |
| fault_coverage                                           | 18.03%     |
| atpg_effectiveness                                       | 100.00%    |
| -----                                                    |            |
| #test_patterns                                           | 0          |
| #simulated_patterns                                      | 0          |
| CPU_time (secs)                                          | 18.0       |
| -----                                                    |            |

## Support of Stuck and Transition Fault Information

MTFI is able to represent stuck fault information in a fashion similar to the classic format, as shown in the following example:

```
FaultInformation {
  version : 1;
  FaultType ( Stuck ) {
    FaultList {
      FaultCollapsing : false;
      Format : Identifier, Class, Location;
      Instance ( " " ) {
        1, UC, "/in1";
        0, DS, "/in1";
        0, DS, "/i1/IN0";
        1, DS, "/i1/OUT";
        1, DS, "/i1/IN0";
        1, AU, "/i1/IN1";
        1, EQ, "/i2/Y";
        0, UC, "/i5/Z";
        1, DS, "/out";
        1, DI, "/i5/Z";
      }
    }
  }
}
```

## Support of N-Detect Values

The information about how often a fault has been detected can be handled for a single fault in the FaultList block, as well as for a group of faults in the UnlistedFaultsData block.

In the FaultList block, you can add the value to the list using the Format statement's Detections keyword. The value must be a positive integer that specifies the number of detections of this fault. All classes other than DS must have a value of 0.

In the following example, the detection drop limit is 4. For any DS faults that are detected four times or more, the tool reports the detection value as “4”.

```
FaultInformation {
  version : 1;
  FaultType ( Stuck ) {
    FaultList {
      FaultCollapsing : false;
      DetectionLimit : 4;
      Format : Identifier, Class, Detections, Location;
      Instance ( "" ) {
        1, DS, 3, "/i1/OUT";
        1, DS, 4, "/i1/IN0";
        1, AU.BB, 0, "/i1/IN1";
        1, EQ, 0, "/i2/Y";
        0, UC, 0, "/i5/Z";
        1, DS, 2, "/out";
        1, DI, 0, "/i5/Z";
      }
    }
  }
}
```

In general, the same rules are valid for the UnlistedFaultsData block, as shown in the following example:

```
FaultInformation {
  version : 1;
  FaultType ( Stuck ) {
    UnlistedFaultsData {
      DetectionLimit : 4;
      Format : Class,Detections,CollapsedFaultCount,
        UncollapsedFaultCount;
      Instance ( "/CoreD/i1" ) {
        UC,      0, 1252, 2079;
        UC.EAB,  0,  452, 543;
        DS,      1, 69873, 87232;
        DS,      2, 12873, 21432;
        DS,      3,  9752, 11974;
        DS,      4, 4487, 6293;
        AU.BB,   0,  8708, 10046;
        AU,      0,  2374, 3782;
      }
    }
  }
}
```

In the case of the UnlistedFaultsData, the preceding example shows the fault count after the number of detections. Typically, there is one line per detection until reaching the current detection limit. So in the preceding example, the number of faults have reached the detection limit of 4.

Also note that in the previous example, the line

```
UC, 0, 1252, 2079
```

shows the collapsed and uncollapsed fault count for the UC faults that are in the unclassified sub-class (that is, those that do not fall into any of the predefined or user-defined sub-classes). Similarly, the line

```
AU, 0, 2374, 3782
```

shows the fault counts for the unclassified AU faults.

While loading the MTFI file, the n-detection number stored in the file is capped at the detection limit currently set in the tool. This applies to both the detection number in FaultList data block and that in UnlistedFaultsData block. Depending on the switch, the n-detection data in the external MTFI file can be appended to the internal detection number of the corresponding fault, or replace the detection number of the corresponding fault in the internal fault list.

## Support of Different Fault Types in the Same File

When reading an MTFI file using the [read\\_faults](#) command, the current fault type (set using [set\\_fault\\_type](#)) must match the specified fault type within the MTFI file. In order to share fault information between the fault types Stuck and Transition, MTFI allows you to manually specify both fault types in the same file, as shown in the following example:

```
FaultInformation {  
  version : 1;  
  FaultType ( Stuck, Transition ) {  
    FaultList {  
      FaultCollapsing : false;  
      Format : Identifier, Class, Location;  
      Instance ( "" ) {  
        1, AU.PC, "/i1/OUT";  
        1, AU.PC, "/i1/IN0";  
        1, AU.BB, "/i1/IN1";  
        1, AU.BB, "/i2/Y";  
        0, AU.TC, "/i5/Z";  
        1, AU.TC, "/out";  
        1, AU.TC, "/i5/Z";  
      }  
    }  
  }  
}
```

MTFI does not support combinations other than Stuck and Transition. And note that none of the tools that support MTFI can output MTFI files that contain multiple fault types.

## Support for Hierarchical Fault Accounting

MTFI allows you to efficiently handle fault information for hierarchical designs, where individual small cores are tested and their fault statistics saved in individual MTFI files. You can then instantiate those small cores in a larger core using “graybox” versions of the small cores plus the fault information from their MTFI files. Using this method, the test patterns for the larger core need not deal with the internal details of the small cores, only with their graybox versions (which contain only the subset of the logic needed for testing at the next higher level).

Consider the example in [Figure 13-1](#). Core\_A and Core\_B are stand-alone designs with their own individual test patterns and fault lists. After running ATPG on Core\_A and Core\_B, you store fault information in MTFI files using the following commands:

```
ANALYSIS> write_faults Core_A.mtfi  
ANALYSIS> write_faults Core_B.mtfi
```

Note that the logic in Core\_A and Core\_B is fully observable and controllable, so there are no unlisted faults in the Core\_A.mtfi or Core\_B.mtfi files.

If there are multiple fault lists for a given core, each reflecting the coverage achieved by a different test mode, and the intent is to merge these results into a single coverage for the core, you must perform the merge during a core-level ATPG run. This is because when you use the “[read\\_faults -graybox](#)” command at the next level of hierarchy, the command supports only a single fault list per core. Note that you can only merge fault lists of a single fault type. You cannot, for example, merge fault lists for stuck and transition at the core level.

Taking the example of Core\_A previously described, you would first merge the fault lists of any other previously-run test modes before writing out the final fault list for the core. The following sequence of commands is an example of this:

```
ANALYSIS> read_faults Core_A_mode1.mtfi -merge  
ANALYSIS> read_faults Core_A_mode2.mtfi -merge  
ANALYSIS> write_faults Core_A.mtfi
```

Next you create Core\_C by instantiating graybox versions of Core\_A and Core\_B, and then you use the fault information you wrote previously using the following commands:

```
ANALYSIS> read_faults Core_A.mtfi -Instance Core_C/Core_A -Graybox  
ANALYSIS> read_faults Core_B.mtfi -Instance Core_C/Core_B -Graybox
```

The -Graybox switch directs the tool to map any faults it can to existing nets in Core\_C and to essentially discard the remaining faults by classifying them as unlisted. The tool retains the fault statistics for the unlisted faults.

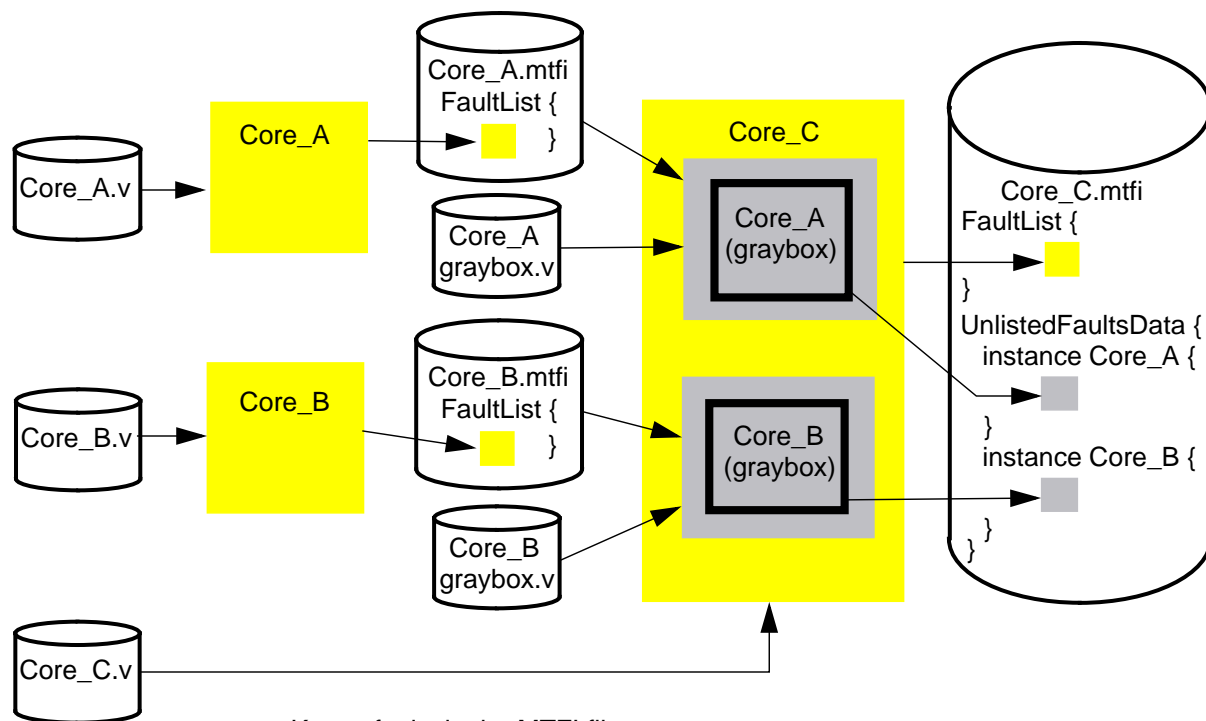
After you’ve run ATPG on Core\_C, you can write out the fault information to an MTFI file:

```
ANALYSIS> write_faults Core_C.mtfi
```

The command writes out data for both the listed and unlisted faults that were created when you issued the read\_faults command. You could then instantiate a graybox version of Core\_C in a larger core, Core\_D, and load the fault information from the Core\_C.mtfi file. That way, Core\_A, Core\_B, and Core\_C will have already been fully tested and would not be tested again, but the fault statistics from all three cores would be available for viewing and analysis with the listed faults in Core\_D.



Figure 13-1. Using MTFI with Hierarchical Designs



Key to faults in the MTFI files:

- = Faults within the core that are classified as listed (faults whose sites are present in the netlist).
- = Faults within the core that are classified as unlisted (faults whose sites are not present in the netlist).

For more information about the [read\\_faults](#) and [write\\_faults](#) commands, refer to their descriptions in the *Tessent Shell Reference Manual*.



# Chapter 14

## Graybox Overview

---

### Introduction

Graybox functionality streamlines the process of scan insertion and ATPG processing in a hierarchical design by allowing you to perform scan and ATPG operations on a sub-module, and then allowing you to use a simplified, graybox representation of that sub-module when performing scan and ATPG operations at the next higher level of hierarchy. Because the graybox representation of a sub-module contains only a minimal amount of interconnect circuitry, the use of grayboxes in a large, hierarchical design can dramatically reduce the amount of memory and tool runtime required to perform scan insertion, optimize timing, analyze faults, and create test patterns.

#### Note



Currently, only Mux-DFF scan architecture is supported with the graybox functionality.

Table 14-1 summarizes the commands that support graybox functionality, which is available in the ATPG tool.

**Table 14-1. Graybox Command Summary**

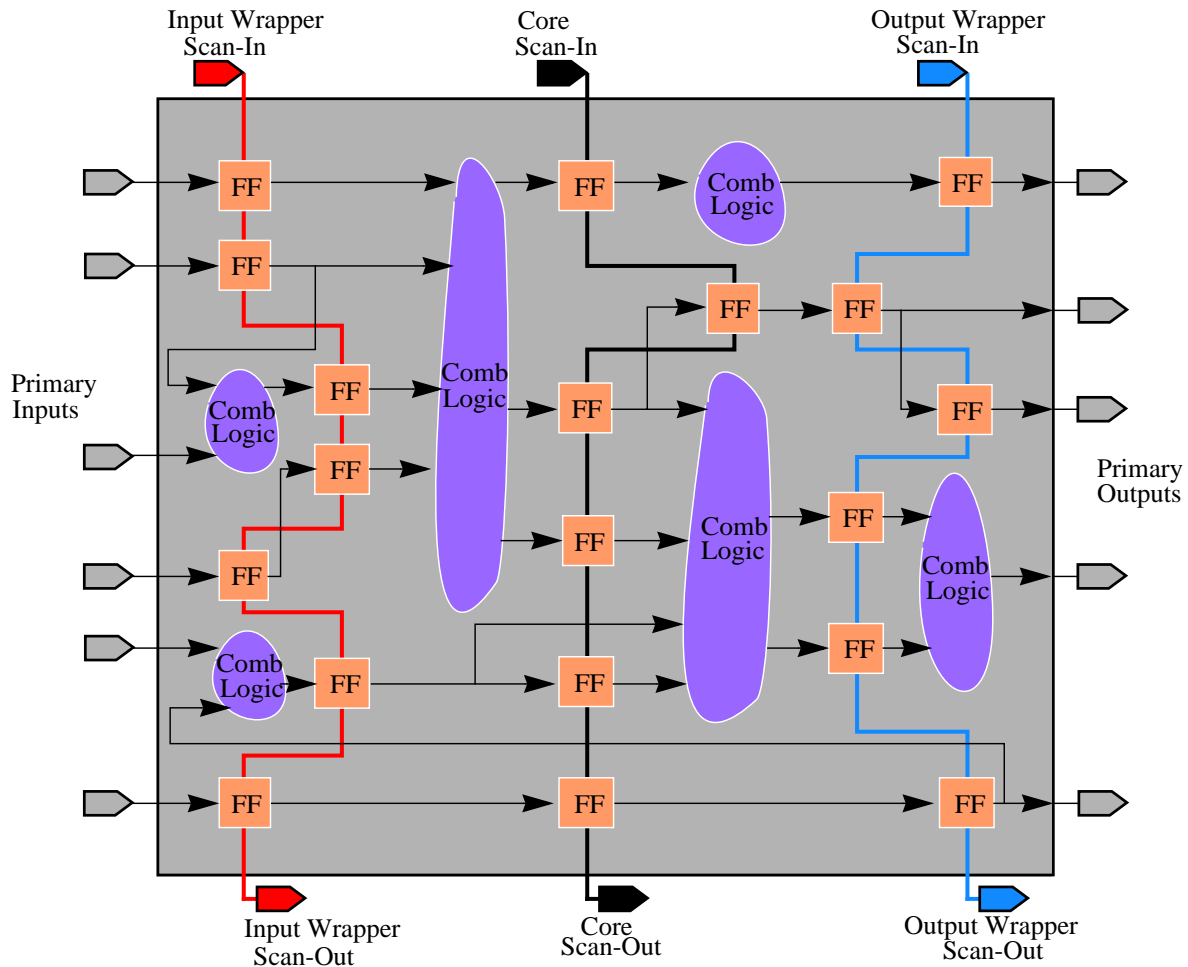
| Command                                   | Description                                                                                                                      |
|-------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">analyze_graybox</a>           | Identifies the instances and nets to be included in the graybox netlist.                                                         |
| <a href="#">set_attribute_value</a>       | Assigns an attribute and value to specified design objects. For a list of graybox attributes, refer to this command description. |
| <a href="#">report_graybox_statistics</a> | Reports the statistics gathered by graybox analysis.                                                                             |
| <a href="#">write_design</a>              | Writes the current design in Verilog netlist format to the specified file. Optionally writes a graybox netlist.                  |

### What is a Graybox?

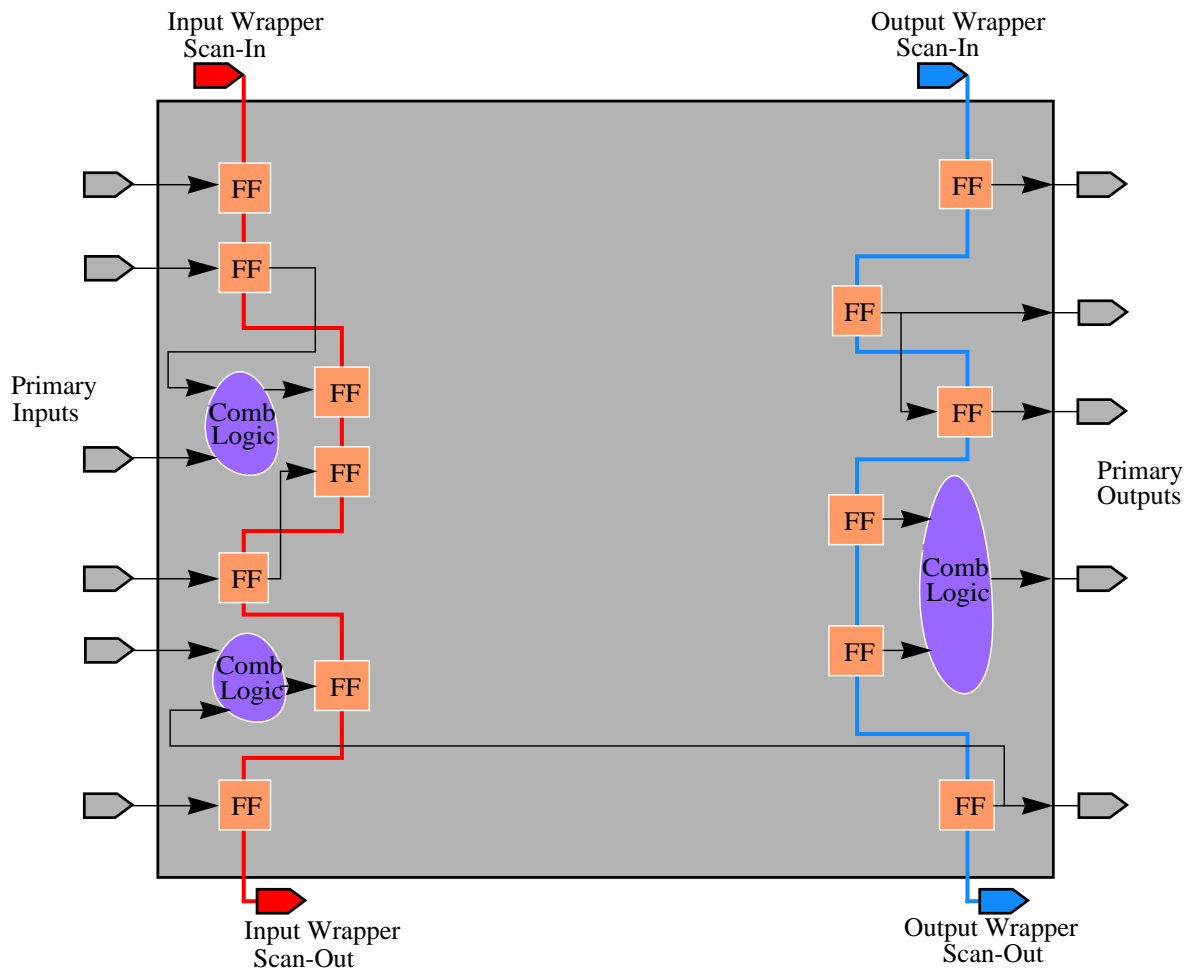
A graybox is a simplified representation of a sub-module that contains only the minimum amount of interconnect circuitry (primary inputs/outputs, wrapper chains, and the glue logic outside of the wrapper chains) required to process the grayboxed sub-module at the next higher level of hierarchy. To understand a graybox representation of a sub-module, first consider the full netlist representation shown in [Figure 14-1](#). This figure shows the input and output wrapper chains, the core scan chains, and the combinational logic that exists both inside and outside the wrapper chains.

After performing scan insertion, fault accounting, and pattern creation for this sub-module, you create a graybox representation of the sub-module, as shown in [Figure 14-2](#).

**Figure 14-1. Full Hierarchical Block Netlist**



[Figure 14-2](#) is a graybox representation of the sub-module shown in [Figure 14-1](#). Note that the graybox contains only the primary inputs/outputs, wrapper chains, and combinational logic that exists outside of the wrapper chains (that is, any combinational logic between a primary input or output and the nearest connected flip-flop).

**Figure 14-2. Graybox Version of Block Netlist**

## Graybox Process Overview

The following is a description of the overall process of generating a graybox netlist. Graybox functionality is available when the tool is in the analysis system mode and the design is in external mode. External mode means that the input wrapper chains are used in regular test modes (both capture and shift), whereas the output wrapper chains are used only in a non-capture mode (shift, hold or rotate). Wrapper chains inserted by Tessent Scan are configured in external mode by constraining the output wrapper chain `scan_enable` signal to active during both shift and capture phases.

The dofile used for graybox netlist generation does the following:

1. Defines clock pins used in external mode (using the `add_clocks` command).
2. Constrains test control pins that place the circuit in external mode (using the `add_input_constraints` command).

3. Defines wrapper chains (using the `add_scan_chains` command).
4. Places the circuit in external mode using a test procedure file. This test procedure file should do the following:
  - Define a test-setup procedure for the external mode to force primary inputs that enable signal paths to wrapper cells.
  - Define a shift and load-unload procedure to force wrapper chain scan enable signals and toggle the shift clocks for the external mode.

Other types of scan and clock procedures (such as master-observe or shadow-observe) and non-scan procedures (such as capture) might also be required to ensure that the circuit operates correctly in external mode.

---

**Note**

Test-setup procedures are not allowed if they pulse the clocks to initialize non-scan cells to constant values in order to sensitize the control signals of external mode. In other words, the only allowable control signals that place the circuit in external mode are the primary inputs to the block (core).

---

5. Identifies graybox logic using the `analyze_graybox` command. The command also displays a summary to indicate the combinational and sequential logic gates identified by the analysis. The tool marks the identified graybox instances by setting their “in\_graybox” attribute. You can also include additional instances in the graybox netlist (or exclude specific instances from the graybox netlist) by turning on/off this attribute using the `set_attribute_value` command.

The graybox analysis performs the identification by tracing backward from all primary output pins and wrapper chains. However, the scan-out pins of the core chains are excluded from the backward tracing. Since core chains are not defined with the `add_scan_chains` command, you accomplish this by setting the `ignore_for_graybox` attribute of the scan-out pins using the `set_attribute_value` command.

6. The “`write_design -graybox`” command writes out all the instances marked with the `in_graybox` attribute. The tool uniquifies all modules that are included in the graybox netlist (except the top module). The interface (port declarations) of a uniquified module is preserved. The uniquification is required because the partial inclusion of the logic inside a module into the graybox netlist could cause conflicts between the different instances of the module, as these instances could be interacting differently with the wrapper chains.

## Example dofile for Creating a Graybox Netlist

The following dofile example shows how to create a graybox netlist:

```
# Define clock pins used for external mode
add_clocks 0 NX2
```

```
add_clocks 0 NX1

# Set up for external mode
# Hold output wrapper chain scan enable active
add_input_constraints sen_out -C1

# Define wrapper chains
add_scan_groups grp1 external_mode.testproc
add_scan_chains wrapper_chain1 grp1 scan_in1 scan_out1
add_scan_chains wrapper_chain2 grp1 scan_in2 scan_out2

# Ignore core chains scan_out pins for graybox analysis to exclude the
# logic intended for internal test mode
set_attribute_value scan_out3 -name ignore_for_graybox -value true
set_attribute_value scan_out4 -name ignore_for_graybox -value true

set_system_mode analysis

# Identify graybox logic
analyze_graybox -collect_reporting_data
report_graybox_statistics -top 10

# NOTE: At this point, you can use the set_attribute_value command with
# the in_graybox attribute to include/exclude specific instances into/from
# graybox netlist.

# Write graybox netlist
write_design -graybox -output_file graybox.v -replace
```

## Generating Graybox Netlist for EDT Logic Inserted Blocks

A graybox netlist can also be generated for a block with EDT logic. Since the EDT logic drives all the scan chains in the block, the wrapper chains of the block can be accessed during extest mode through dedicated wrapper-extest ports. These ports are inserted to the block by Tessent Scan when you insert the wrapper chains using the “[set\\_wrapper\\_chains](#) -wrapper\_extest\_ports” command.

The following is an example dofile. In order to generate the graybox netlist for the block, the wrapper-extest ports are activated by constraining the global signal wrapper\_extest to a logic 1. Also, to exclude the EDT logic and subsequently the core chains from the graybox netlist, the dofile marks the EDT channel outputs with the ignore\_for\_graybox attribute before performing the graybox analysis.

```
# Define clocks
add_clocks 0 clk1
add_clocks 0 clk2
add_clocks 0 clk3

# Set up for external mode
# Enable access to wrapper chain extest ports and hold output wrapper
# chain scan enable active
add_input_constraints wrapper_extest -C1
add_input_constraints scan_en_out -C1
```

```
# Define wrapper chains
add_scan_groups grp1 cpu_block1_extest.testproc
add_scan_chains chain1 grp1 scan_in1 scan_out2
add_scan_chains chain2 grp1 scan_in3 scan_out4
add_scan_chains chain3 grp1 scan_in5 scan_out6
add_scan_chains chain4 grp1 scan_in7 scan_out8
add_scan_chains chain5 grp1 scan_in9 scan_out10

# Ignore EDT channel outputs to exclude EDT logic and core chains from
# the graybox netlist.
set_attribute_value edt_channels_out1 -name ignore_for_graybox -value true
set_attribute_value edt_channels_out2 -name ignore_for_graybox -value true
set_attribute_value edt_channels_out3 -name ignore_for_graybox -value true

set_system_mode analysis

# Identify graybox logic
analyze_graybox -collect_reporting_data
report_graybox_statistics -top 10

# Write graybox netlist
write_design -graybox -output_file graybox.v
```

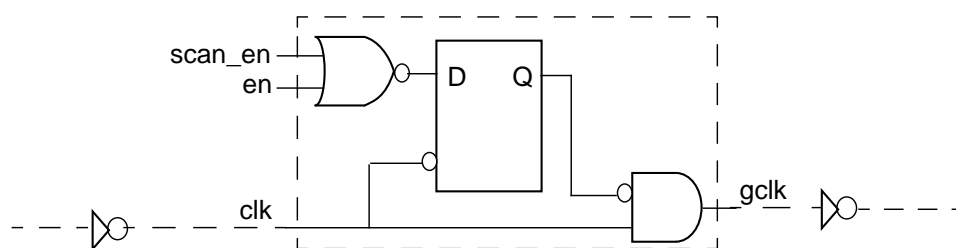


The use of clock gaters to reduce power consumption is becoming a widely adopted design practice. Although effective for reducing power, clock gaters create new challenges for ATPG tools because of the additional complication introduced in clock paths. Among the challenges, the most frequently encountered in the tool is the C1 DRC violation. The C1 rule is fairly strict, requiring clock ports of scan cells to be at their off states when all clock primary inputs (PIs) are at their off states. Not all designs abide by this rule when there are clock gaters in clock paths.

## Basic Clock Gater Cell

There is one common type of clock gater cell. [Figure A-1](#) shows the structure of the cell.

**Figure A-1. Basic Clock Gater Cell**

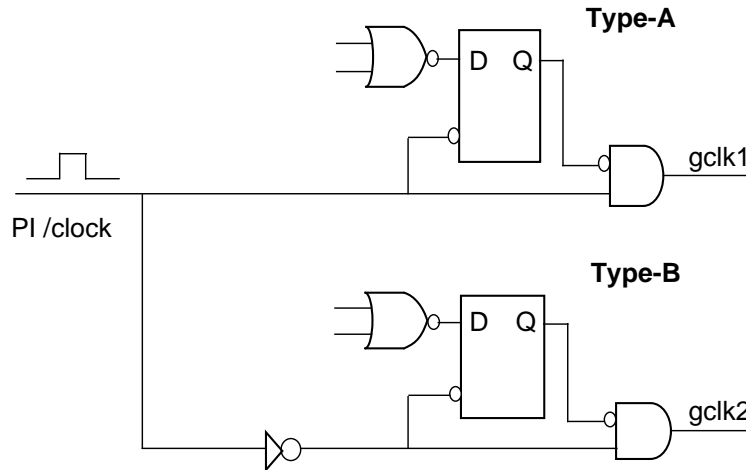


The cell (inside the dashed box) has three inputs: scan\_en, en, and clk. The output of the cell is the gated clock, gclk. The scan\_en input is for test mode and is usually driven by a scan enable signal to allow shifting. The en input is for functional mode and is usually driven by internal control logic. Sometimes test\_en is used to drive the scan\_en input to avoid any trouble caused by the clock gater. However, this ends up keeping the clock gater always on and results in loss of coverage in the en fanin cone. The latch in the clock gater eliminates potential glitches. Depending on the embedding, there could be inversions of the clk signal both in front of clk and after gclk, which in the figure are shown as inverters on the dashed lines representing the clk input and gclk output.

## Two Types of Embedding

The key factor affecting whether the tool produces DRC violations is whether the clk signal gets inverted in front of the clk input to the cell. To better understand the impact of such an inversion, it is helpful to understand two common ways designers embed the basic clock gater cell (see [Figure A-1](#)) in a design. [Figure A-2](#) shows the two possibilities, referred to as Type-A and Type-B.

**Figure A-2. Two Types of Embedding for the Basic Clock Gater**



In the figure, assume the PI /clock drives two clock gaters and that the off state of /clock is 0. The behavior of the two types of embeddings is as follows:

- **Type-A** — When PI /clock is at its off state, the latch in the clock gater is transparent, and the AND gate in the clock gater gets a controlling value at its input. As a result, the output gclk1 is controlled to a deterministic off state.
- **Type-B** — When /clock is at its off state, the latch in the clock gater is not transparent, and the AND gate in the clock gater gets a *non*-controlling value at its input. As a result, the output gclk2 is not controlled to a deterministic off state. Its off state value will depend on the output of the latch.

---

**Note**



The preceding classification has nothing to do with whether there is inversion after gclk. It only depends on whether the off state of /clock can both make the latch transparent and control the AND gate.

---

## Ideal Case (Type-A)

The tool fully supports the Type-A embedding for the following reasons:

- Since the latch is transparent, when scan\_en is asserted at the beginning of load\_unload, the clock gater is immediately turned on. Therefore, subsequent shifting is reliable and there will be no scan chain tracing DRC violations.
- Since gclk is controlled to a deterministic off state, there are no C1 DRC violations for downstream scan cells driven by it.
- The tool can pick up full fault coverage in the fanin cone logic of the en and scan\_en inputs of the clock gater. This coverage is often too significant to be sacrificed.

**Type-A embedding is the preferred design practice and is strongly recommended.**

---

**i** **Tip:** Type-A embedding does not necessarily mean the downstream scan cells have to be either all leading edge (LE) flip-flops or all trailing edge (TE) flip-flops. A designer can have both of them by inserting inversions after gclk.

---

## Potential DRC Violator (Type-B)

Type-B embeddings have the following undesirable characteristics:

- When PI /clock is at its off state, the latch in the clock gater is not transparent. This means a change on its D input at the beginning of a cycle will not immediately be reflected at its Q output. In other words, the action of turning the clock gater on or off lags its instruction.
- The clock off state of gclk is not deterministic. It depends on what value the latch in the clock gater captures in the previous cycle. Therefore, the tool will issue a C1 DRC violation.

The kinds of problems that can arise due to these undesirable characteristics are described in the following sections:

|                                                                       |     |
|-----------------------------------------------------------------------|-----|
| Avoiding Type-B Scan Chain Tracing Failures (T3 Violations) . . . . . | 411 |
| Preventing Scan Cell Data Captures When Clocks are Off . . . . .      | 412 |
| Accounting for Type-B Clock Gaters Driving LE Flops . . . . .         | 413 |


## Avoiding Type-B Scan Chain Tracing Failures (T3 Violations)

If the scan flip-flops downstream from the Type-B clock gater are LE triggered, the result will be scan chain tracing failures. This is because the first shift edge in the load\_unload is missed, as illustrated in [Figure A-3](#) and [Figure A-4](#).

[Figure A-3](#) shows an example circuit where a Type-B clock gater drives both TE and LE scan flip-flops.

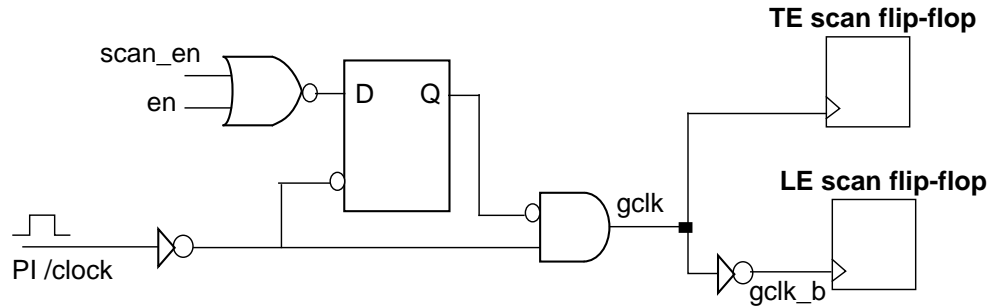
---

**Note**

 Assuming the off state of the PI /clock is 0, an inverter is needed after gclk to make the downstream flip-flop LE. The inverted gclk is indicated as gclk\_b.

---

**Figure A-3. Type-B Clock Gater Causes Tracing Failure**



**Figure A-4. Sample EDT Test Procedure Waveforms**

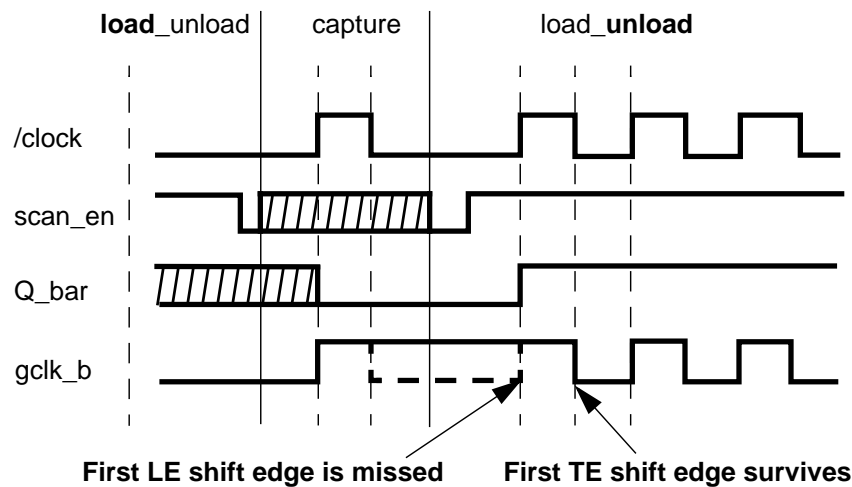


Figure A-4 shows the timing diagram of several signals during capture and the beginning of load\_unload. Suppose the latch in the Type-B clock gater captures a 1 in the last capture cycle (shown as Q\_bar going low in the capture window). You can see that the first leading edge is suppressed because the latch holds its state after capture and into load\_unload. Although scan\_en is high at the beginning of load\_unload, the output of the latch will not change until the first shift clock arrives. This lag between the “instruction” and the “action” causes the downstream LE scan flip-flops to miss their first shift edge. However, TE-triggered scan flip-flops are still able to trigger on the first trailing edge.

**i Tip:** To work around this problem, you can use test\_en instead of scan\_en to feed the clock gater. But be aware that this workaround loses fault coverage in the fanin cone of the clock gater’s en input.

## Preventing Scan Cell Data Captures When Clocks are Off

T3 violations rarely occur in designs with Type-B clock gaters that drive LE scan flip-flops, typically because the downstream scan flip-flops in those designs are all made TE by synthesis

tools. The tools handle Type-B clock gaters that drive TE scan flip-flops just fine, and change any downstream latches, LE nonscan flip-flops, RAMs, and POs to TIEX. This prevents C1 DRC violations, coverage loss in the cone of the clock gater, and potential simulation mismatches when you verify patterns in a timing based simulator.

## Accounting for Type-B Clock Gaters Driving LE Flops

When you have a Type-B clock gater driving LE flops in the chain, you must do the following in sequence:

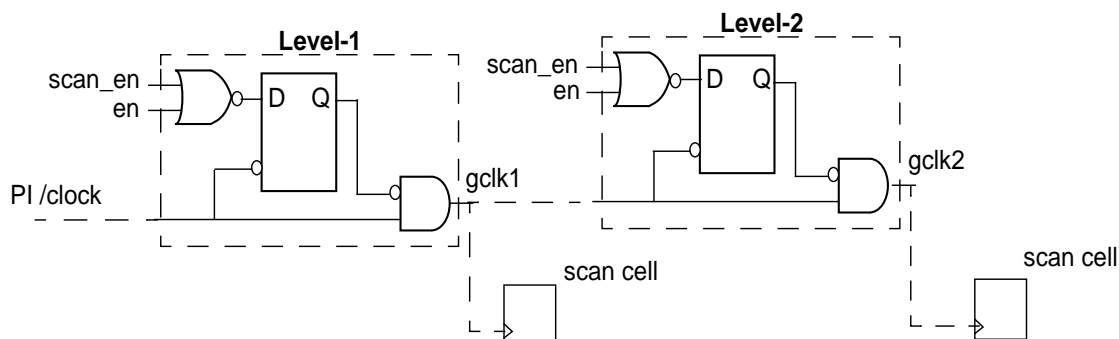
1. Initialize the clock gaters in one cycle of test\_setup by driving scan\_enable high and pulsing the clocks.
2. Issue the `set_stability_check` command using the following arguments:  
**set\_stability\_check on -sim\_static\_atpg\_constraints on**
3. Add ATPG constraints of “1” on the functional enable of the Type-B clock gaters using the `add_atpg_constraints` command using the following arguments:

**add\_atpg\_constraints 1 functional\_enable\_pin -static**

## Cascaded Clock Gaters

One level of clock gating is sometimes not enough to implement a complex power controlling scheme. [Figure A-5](#) shows a two-level clock gating scheme using the clock gater cell. The dashed lines in the figure indicate places where inversions may exist. PI /clock is first gated by a level-1 gater and then the output gclk1 is further gated by a level-2 gater, generating the final clock gclk2.

**Figure A-5. Two-level Clock Gating**



## Understanding a Level-2 Clock Gater

In a two-level clock gating scheme, if the level-1 clock gater is of Type-A, then the clock off value of gclk1, being deterministic, has no problem propagating to the level-2 clock gater. The latter can be understood based on the definition in “[Basic Clock Gater Cell](#)” on page 409.

However, when the level-1 gater is of Type-B, the tool cannot apply the basic definition to the level-2 gater directly, since the clock off value of gclk1 is undetermined. In this case, the type for the level-2 clock gater is defined by assuming the level-1 clock gater has been turned on. Therefore, even if the level-1 clock gater is of Type-B, the tool can still classify the level-2 clock gater.

## Example Combinations of Cascaded Clock Gaters

Following are the four possible combinations of two-level cascading clock gaters and the support the tool provides for them:

- **Type-A Level-1 + Type-A Level-2** — This is the preferred combination, is fully supported by the tool, and is strongly recommended.
- **Type-A Level-1 + Type-B Level-2** — This can be reduced to a single level type-B case. All earlier discussion of type-B clock gaters applies. See [“Potential DRC Violator \(Type-B\)”](#) on page 411 for more information.
- **Type-B Level-1 + Type-A Level-2** — Avoid this combination. Scan chain tracing will be broken even if the downstream scan flip-flops of the level-2 gater are TE. The problem occurs when both level-1 and level-2 clock gaters capture a 1 in the last cycle of the capture window (meaning both clock gaters are turned off). Then the downstream scan flip-flops will miss both the first LE shift edge and the first TE shift edge in the subsequent load\_unload. This is a design problem, not a tool limitation.
- **Type-B Level-1 + Type-B Level-2** — Supported only if all the downstream scan flip-flops are TE.

## Summary

[Table A-1](#) summarizes the support that the tool provides for each clock gater configuration.

**Table A-1. Clock Gater Summary**

| Gater Configuration             | LE Scan Flip-flops Downstream                     | TE Flip-flops Downstream         | Latches, RAM, ROM, PO Downstream |
|---------------------------------|---------------------------------------------------|----------------------------------|----------------------------------|
| Type-A                          | Supported, strongly recommended.                  | Supported, strongly recommended. | Supported.                       |
| Type-B                          | T3 DRC violations. A design issue. Not supported. | Supported.                       | Changed to TIEX.                 |
| Type-A Level-1 + Type-A Level-2 | Supported.                                        | Supported.                       | Supported.                       |

**Table A-1. Clock Gater Summary**

| <b>Gater Configuration</b>         | <b>LE Scan Flip-flops<br/>Downstream</b> | <b>TE Flip-flops<br/>Downstream</b>  | <b>Latches, RAM,<br/>ROM, PO<br/>Downstream</b> |
|------------------------------------|------------------------------------------|--------------------------------------|-------------------------------------------------|
| Type-A Level-1 +<br>Type-B Level-2 | Same as Type-B<br>alone.                 | Same as Type-B<br>alone.             | Same as Type-B<br>alone.                        |
| Type-B Level-1 +<br>Type-A Level-2 | T3 DRC violations.<br>Not supported.     | T3 DRC violations.<br>Not supported. | Changed to TIEX.                                |
| Type-B Level-1 +<br>Type-B Level-2 | T3 DRC violations.<br>Not supported.     | Supported.                           | Changed to TIEX.                                |





# Appendix B

## Debugging State Stability

---

### Understanding State Stability

*State stability* refers to state elements that are stable, hold their values across shift cycles and patterns, and their values can be used for DRC (for example, scan chain tracing). Debugging state stability is useful when you set up a state (for example, in a TAP controller register, in the test\_setup procedure) that you use for sensitizing the shift path. Unless the state value is preserved for all shift cycles and all patterns, then scan chain tracing can fail. If you set up the correct value in the test\_setup procedure but the value is changed due to the application of shift or the capture cycle, the value cannot be depended on. For such cases, you can debug state stability by identifying what caused a stable state element to unexpectedly change states.

---

#### Note



The information in this appendix uses the [set\\_stability\\_check](#) command set to On (the default for this command), except as noted in Example 3 and Example 9.

---

### Displaying the State Stability Data

Using the [set\\_gate\\_report](#) command with the Drc\_pattern option, you can display simulation data for different procedures using the [report\\_gates](#) command or DFTVisualizer. The following examples show how to use the set\_gate\_report command:

```
set_gate_report drc_pattern test_setup
```

```
set_gate_report drc_pattern load_unload
```

```
set_gate_report drc_pattern shift
```

For “drc\_pattern load\_unload” and “drc\_pattern shift”, the superimposed values for all applications of the procedure are reported. This means a pin that is 1 for only the first application of shift, but 0 or X for the second application of shift shows up as X.

When you set the Drc\_pattern option of the set\_gate\_report command to State\_stability as shown in the following example:

```
set_gate_report drc_pattern state_stability
```

the state stability report also includes the load\_unload and shift data for the first application of these procedures.

When you debug state stability, you normally compare the state stability values (the values during the first application of the procedures) with the superimposed (stable) values from “drc\_pattern load\_unload” and “drc\_pattern shift.”

## State Stability Data Format

State stability data is only available after DRC and only if there is a test\_setup procedure. The format of the state stability data displayed for a pin is shown in the following example:

```

                (ts) (ld) (shift) (ld) (shift) (cell_con) (cap) (stbl)
//  CLK  I      ( 0) ( 0) (010~0) (00) (010)  ( 0 )  (0X0)  ( 0 )

```

The first row lists the column labels in parentheses. The second row displays the pin name and five or more groups of data in parentheses. Each group has one or more bits of data corresponding to the number of events in each group.

Following is a description of the data columns in the state stability report:

- **ts** — Last event in the test\_setup procedure. This is always one bit.
- **ld** — Any event in the load\_unload procedure that is either before and/or after the apply shift statement. When load\_unload is simulated, all primary input pins that are not explicitly forced in the load\_unload procedure or constrained with an add\_input\_constraints command are set to X. Constrained pins are forced to their constrained values at the end of test\_setup prior to load\_unload. This is because even if your test\_setup procedure gives those pins different values coming out of test\_setup, in the final patterns saved, an event is added to test\_setup to force those pins to their constrained values. (You can see that event in the test\_setup procedure if you issue write\_procf file after completing DRC.) Consequently, for the first pattern, a pin has its constrained value going into load\_unload. For all subsequent patterns, load\_unload follows the capture cycle, during which the tool would have enforced the constrained value. Therefore, for all patterns, a constrained pin is forced to its constrained value going into load\_unload and hence this value can be used for stability analysis. This is also true for the state stability analysis where the first application of load\_unload is simulated.

The number of bits in this group depends on the number of events in the procedure prior to the apply shift statement. If there are no events prior to the apply shift statement, this group still exists (with one bit).

- **shift** — Apply shift statement (main shift or independent shift). If the load\_unload procedure has multiple apply shift statements, then multiple (shift) groups are displayed. If this is also the main shift application (that is, the number of shifts applied is greater than one), the group includes a tilde (~). The values before the tilde correspond to the very first application of the shift procedure, with the number of bits corresponding to the

number of primary input events in the shift procedure. The group without a tilde (~) is the independent shift.

The last bit (after the tilde) corresponds to the stable state after application of the last shift in the main shift statement. By default, the precise number of shifts is not simulated. The stable state shown corresponds to an infinite number of shift cycles; that is, in some cases, when for instance the depth of non-scan circuitry is deeper than the scan chain length, sequential circuitry that should be 1 or 0 after the first load\_unload may show up as X. For more information, see the description of the “set\_stability\_check\_all\_shift” command in “[Example 4 — Single Post Shift](#)” on page 429.

After the third group, there could be additional groups if there are the following:

- Multiple apply shift statements
- Events in the load\_unload procedure after the apply shift statement
- **shdw\_con** — Shadow\_control procedure. This column is not shown in the examples. Look at the test procedure file to determine the meaning of this group.
- **cell\_con** — Cell constraints. If there are cell constraints, an extra simulation event is added to show the value. This is always one bit.
- **cap** — Capture procedure. This is the simulation of the first capture cycle. Notice that this is not the simulation of pattern 0 or any specific pattern. Therefore, normally the capture clock going to 0X0 (or 1X1 for active low clocks) is displayed, indicating that the clock may or may not pulse for any given pattern. If the set\_capture\_clock -Atpg command is used to force a specific capture clock to be used exactly once for every pattern, the capture cycle simulation is less pessimistic and “010” or “101” is reported.
- **stbl** — Final stable values after several iterations of simulating load\_unload and capture procedures. If the value of a gate is always the same at the end of the test\_setup and capture procedures, then its stable value is the same; otherwise, the stable value is X.

---

**Note**

The skew\_load, master\_observe, or shadow\_observe procedures are not simulated as part of state stability analysis, so their simulation data is not displayed in a state stability report. To view the simulation data for any of these procedures, use the [set\\_gate\\_report](#) Drc\_pattern command with the specific *procedure\_name* of interest.

---

## State Stability Examples

This section contains examples of state stability reporting that show different behaviors in state stability analysis. The examples are based on the design shown in [Figure B-1](#). The design has the following three clocks:

- **clk1** — The only scan clock
- **clk2** — Clocks a particular non-scan flip-flop

- **reset** — Resets four non-scan flip-flops connected as a register

The design has five pins (A, B, C, D, and E) that are exercised in the procedures to specifically show state stability analysis. The circuit has the following characteristics:

- Pin D has a C0 pin constraint. There are no other constrained pins.
- Non-scan flip-flop ff00 is clocked by clk1 and driven by pin A. The flip-flop is initialized in test\_setup, but loses state when it is pulsed after test\_setup.
- Non-scan flip-flop ff10 is clocked by clk1 and driven by pin D. Notice that this one gets initialized and “sticks” (is converted to TIE0) due to the pin constraint.
- Non-scan flip-flop ff20 is clocked by clk2 and driven by pin A.
- Non-scan flip-flop ff32 is the fourth flip-flop in a serial shift register. The first flip-flop in the register is ff30 and driven by pin A. The flip-flops are all clocked by clk1 and reset by the reset signal. ff32 is reset in test\_setup, maintains state through the first application of shift, but then loses stability.
- There is one scan chain with three scan cells: sff1, sff2, and sff3.

With the exception of Example 3, the only difference between the eleven examples is the test procedure file. The timeplate and procedures from this file that were used for the first (basic) example are shown following the design. The procedures used in subsequent examples differ from the basic example as follows:

[Example 1 — Basic Example](#)

[Example 2 — Multiple Cycles in Load\\_unload Prior to Shift](#)

[Example 3 — Drc\\_pattern Reporting for Pulse Generators](#) (the same procedures as Example 2, but uses a pulse generator to clock one of the flip-flops)

[Example 4 — Single Post Shift](#)

[Example 5 — Single Post Shift with Cycles Between Main and Post Shift](#)

[Example 6 — Cycles After Apply Shift Statement in Load\\_unload](#)

[Example 7 — No Statements in Load\\_unload Prior to Apply Shift](#)

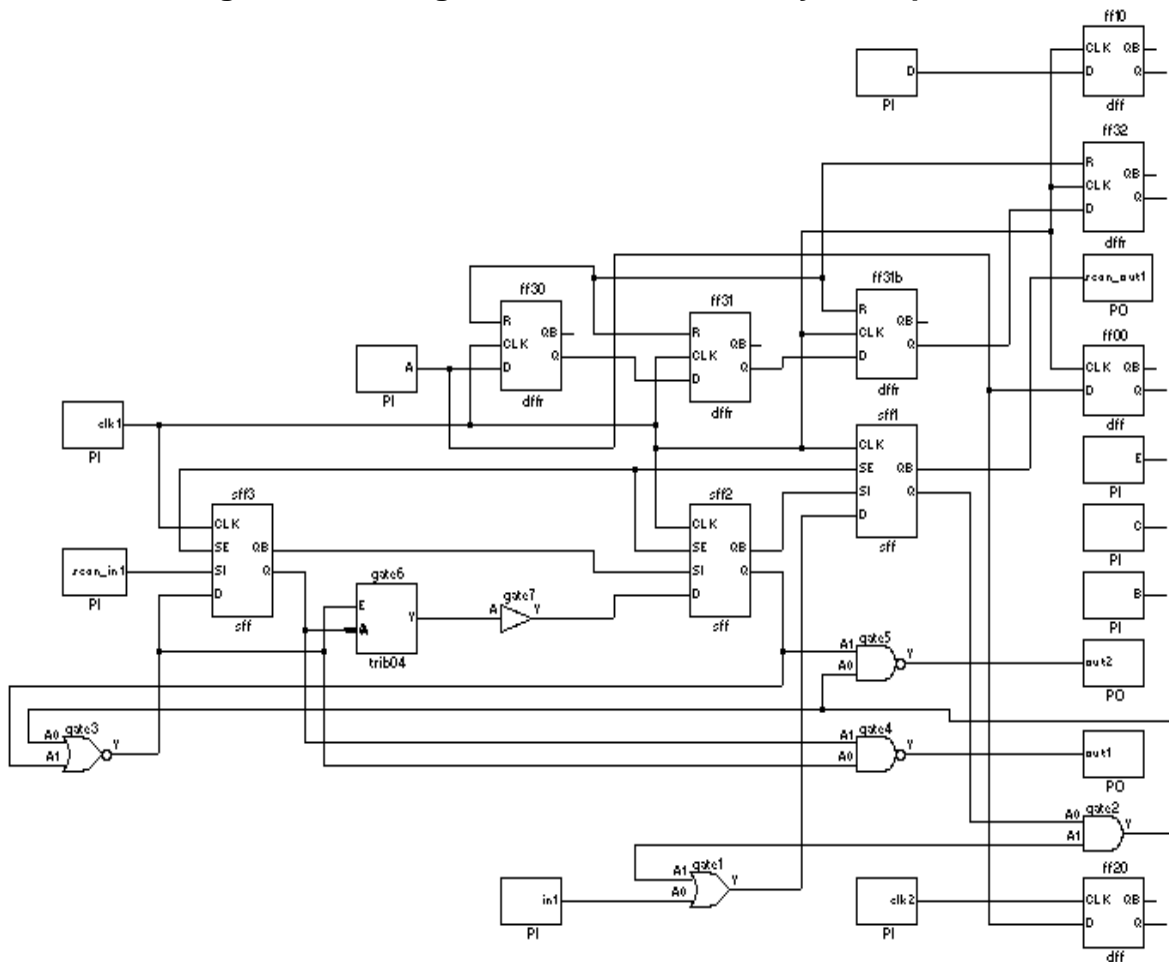
[Example 8 — Basic with Specified Capture Clock](#)

[Example 9 — Setting Stability Check to Off and All\\_shift](#)

[Example 10 — Pin Constraints, Test\\_setup, and State Stability](#)

[Example 11 — Single Pre Shift](#)

Figure B-1. Design Used in State Stability Examples



```

timeplate gen_tp1 =
    force_pi 0;
    measure_po 10;
    pulse clk1 20 10;
    pulse clk2 20 10;
    pulse reset 20 10;
    period 40;
end;

procedure capture =
    timeplate gen_tp1;
    cycle =
        force_pi;
        measure_po;
        pulse_capture_clock;
    end;
end;

procedure shift =
    scan_group grp1;
    timeplate gen_tp1;

```

```
        cycle =
            force_sci;
            measure_sco;
            pulse clk1;
            force C 0;
        end;
    end;

    procedure test_setup =
        scan_group grp1;
        timeplate gen_tpl;
        // First cycle, one PI event (force)
        cycle =
            force clk1 0;
            force clk2 0;
            force reset 0;
            force A 0;
            force B 0;
            force C 0;
            force D 0;
            force E 0;
        end;
        // Second cycle, two PI events (pulse on, pulse off)
        cycle =
            pulse reset;
            pulse clk2;
        end;
        // Third cycle, three PI events (force, pulse on, and pulse off)
        cycle =
            force A 1;
            pulse clk1;
        end;
    end;

    procedure load_unload =
        scan_group grp1;
        timeplate gen_tpl;
        // First cycle, one PI event (force)
        cycle =
            force clk1 0;
            force clk2 0;
            force reset 0;
            force scan_en 1;
            force B 1;
            force C 1;
        end;
        apply shift 3;
    end;
```

## Example 1 — Basic Example

In the basic example, the tool is invoked on the design and the following commands are entered:

```
add_clocks 0 clk1 clk2 reset
add_scan_groups grp1 scan1.testproc
add_scan_chains c0 grp1 scan_in1 scan_out1
add_input_constraints D -c0
set_gate_report drc_pattern state_stability
set_system_mode analysis
report_gates A B C D E

...
//          (ts) (ld) (shift) (cap) (stbl)
//      A    O    ( 1) ( 1) (111~1) (XXX) ( X )
//      B    O    ( 0) ( 1) (111~1) (XXX) ( X )
//      C    O    ( 0) ( 1) (000~0) (XXX) ( X )
//      D    O    ( 0) ( 0) (000~0) (000) ( 0 )
//      E    O    ( 0) ( 0) (000~0) (XXX) ( X )
```

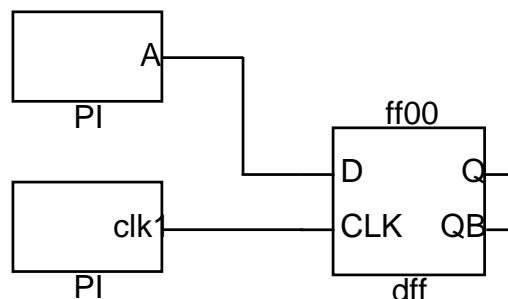
The results of this operation are:

- Pins A through E are explicitly forced in test\_setup.
- Pins A, B, C, and E are forced in load\_unload.
- Pin D is constrained (but not forced in load\_unload) and is the only one of the pins with an explicit value during capture.

A typical initialization problem is illustrated in ff00. [Figure B-2](#) shows the relevant circuitry and statements in the test\_setup procedure that seemingly initialize this flip-flop.

**Figure B-2. Typical Initialization Problem**

```
procedure test_setup =
...
// Third cycle...
cycle =
    force A 1;
    pulse clk1;
end;
...
```



In this case, you might expect the output to always be 1 after initialization because the third cycle of the test\_setup procedure forced a 1 on input A and pulsed clk1. For comparison purposes, following is the reported state\_stability data together with the data reported for load\_unload and shift. Notice especially the Q output:

|    |       |     | state_stability                |  | load_unload  |  | shift |
|----|-------|-----|--------------------------------|--|--------------|--|-------|
| // | /ff00 | dff |                                |  |              |  |       |
| // |       |     | (ts) (ld) (shift) (cap) (stbl) |  | (ld) (shift) |  |       |
| // | CLK   | I   | ( 0) ( 0) (010~0) (0X0) ( 0 )  |  | ( 0) ( 010)  |  | (010) |
| // | D     | I   | ( 1) ( X) (XXX~X) (XXX) ( X )  |  | ( X) ( XXX)  |  | (XXX) |
| // | Q     | O   | ( 1) ( 1) (1XX~X) (XXX) ( X )  |  | ( 1) ( XXX)  |  | (XXX) |
| // | QB    | O   | ( 0) ( 0) (0XX~X) (XXX) ( X )  |  | ( 0) ( XXX)  |  | (XXX) |

You can see from the state stability display that, after test\_setup, the output of Q is set to 1. In the first application of load\_unload it is still 1, but it goes to X during the first shift. Compare this to what is shown for “drc\_pattern load\_unload” and “drc\_pattern shift”.

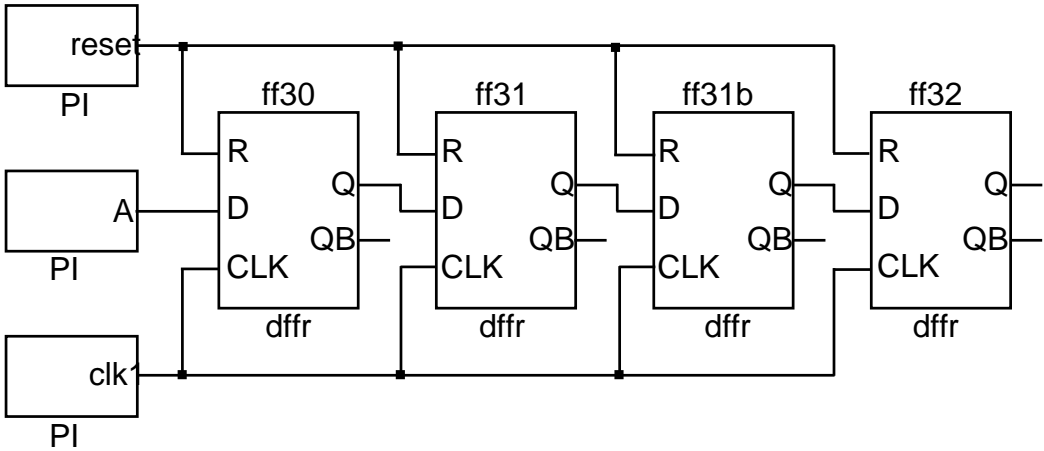
A stable initialization value can be better achieved by doing for ff00 something similar to what occurs for ff10, where the D input is connected to the constrained D pin:

|    |       |     | state_stability                |  | load_unload  |  | shift |
|----|-------|-----|--------------------------------|--|--------------|--|-------|
| // | /ff00 | dff |                                |  |              |  |       |
| // |       |     | (ts) (ld) (shift) (cap) (stbl) |  | (ld) (shift) |  |       |
| // | CLK   | I   | ( 0) ( 0) (010~0) (0X0) ( 0 )  |  | ( 0) ( 010)  |  | (010) |
| // | D     | I   | ( 1) ( X) (000~0) (000) ( X )  |  | ( X) ( 000)  |  | (000) |
| // | Q     | O   | ( 1) ( 1) (000~0) (000) ( X )  |  | ( 0) ( 000)  |  | (000) |
| // | QB    | O   | ( 0) ( 0) (111~1) (111) ( X )  |  | ( 1) ( 111)  |  | (111) |

Another interesting observation can be made for ff32. This flip-flop is at the end of a 4-bit shift register where all the flip-flops are reset during test\_setup as shown in [Figure B-3](#).



Figure B-3. Three-bit Shift Register (Excerpted from Figure D-1)



```
procedure test_setup =
...
// Second cycle...
cycle =
    pulse reset;
...
end;
...
```

Notice how Q in this case is stable for the first application of load\_unload and shift, but the stable state after the last shift (after ~) is X. This is due to an optimization the tool does by default for the state\_stability check. (Compare this output to example [Example 9 — Setting Stability Check to Off and All\\_shift.](#))

|    | state_stability | load_unload                | shift            |
|----|-----------------|----------------------------|------------------|
| // | //ff32 dffr     |                            |                  |
| // |                 | (ts)(ld)(shift)(cap)(stbl) | (ld)(shift)      |
| // | R I             | ( 0)( 0)(000~0)(0X0)( 0 )  | ( 0)( 000) (000) |
| // | CLK I           | ( 0)( 0)(010~0)(0X0)( 0 )  | ( 0)( 010) (010) |
| // | D I             | ( 0)( 0)(000~0)(XXX)( X )  | ( 0)( 000) (XXX) |
| // | Q O             | ( 0)( 0)(000~1)(XXX)( X )  | ( 0)( 000) (XXX) |
| // | QB O            | ( 1)( 1)(111~1)(XXX)( X )  | ( 1)( 111) (XXX) |

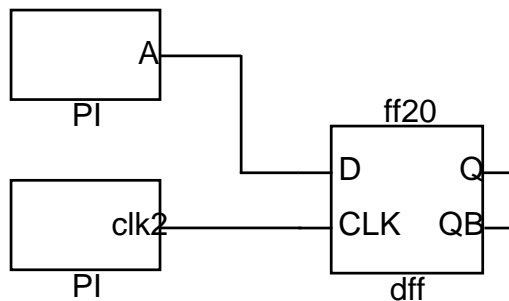
Non-scan flip-flop ff20 is clocked by clk2, which is not a shift clock. This flip-flop is also initialized during test\_setup as shown in [Figure B-4.](#)

**Figure B-4. Initialization with a Non-Shift Clock**

```

procedure test_setup =
  ...
  // First cycle...
  cycle =
    ...;
    force A 0;
    ...
  end;
  // Second cycle...
  cycle =
    ...
    pulse clk2;
  end;
  ...

```



The Q output is disturbed during capture, not during shift, because this element is not exercised during shift:

|    | state_stability                     | load_unload  | shift |
|----|-------------------------------------|--------------|-------|
| // | /ff20 dff                           |              |       |
| // | (ts) (ld) (shift) (cap) (stbl)      | (ld) (shift) |       |
| // | CLK I ( 0) ( 0) (000~0) (0X0) ( 0 ) | ( 0) ( 000)  | (000) |
| // | D I ( 1) ( X) (XXX~X) (XXX) ( X )   | ( X) ( XXX)  | (XXX) |
| // | Q O ( 0) ( 0) (000~0) (0XX) ( X )   | ( 0) ( 000)  | (XXX) |
| // | QB O ( 1) ( 1) (111~1) (1XX) ( X )  | ( 1) ( 111)  | (XXX) |

Notice that the load\_unload and shift data for ff32 and ff20 is almost identical (except for the clock data), but that the state\_stability data enables you to see that they become unstable in very different ways.

## Example 2 — Multiple Cycles in Load\_unload Prior to Shift

The main difference between this example and the basic example is that in the load\_unload procedure there are multiple cycles prior to the apply shift statement (statements that are new in this example are highlighted in bold):

```
procedure load_unload =
  scan_group grp1 ;
  timeplate gen_tpl ;
  // First cycle, one PI event (force)
  cycle =
    force clk1 0 ;
    force clk2 0 ;
    force reset 0 ;
    force scan_en 1 ;
    force B 1;
    force C 1;
    force E 1;
  end ;
  // Second cycle, three PI events (force, pulse on, pulse off)
  cycle =
    force E 0;
    pulse clk2;
  end ;
  // Third cycle, two PI events (pulse on, pulse off)
  cycle =
    pulse clk2;
  end ;
  apply shift 3;
end;
```

As a result, multiple events are displayed in the second group of state stability data. Notice there are now three cycles. The following gate report excerpts show six bits of data (in bold), corresponding to the total number of events. The first bit is from the first cycle (one event), the next three bits are from the second cycle (three events), and the last two bits are from the third cycle, which has two events.

```
// /E primary_input
//      (ts)( ld )(shift)(cap)(stbl)
//      E      O ( 0)(100000)(000~0)(XXX)( X)
// /A primary_input
//      (ts)( ld )(shift)(cap)(stbl)
//      A      O ( 1)(xxxxxx)(XXX~X)(XXX)( X)
// /clk2 primary_input
//      (ts)( ld )(shift)(cap)(stbl)
//      clk2   O ( 0)(001010)(000~0)(0X0)( 0)

// /ff20 dff
//      (ts)( ld )(shift)(cap)(stbl)
//      CLK    I ( 0)(001010)(000~0)(0X0)( 0)
//      D      I ( 1)(xxxxxx)(XXX~X)(XXX)( X)
//      Q      O ( 0)(00xxxx)(XXX~X)(XXX)( X)
//      QB     O ( 1)(11xxxx)(XXX~X)(XXX)( X)
```

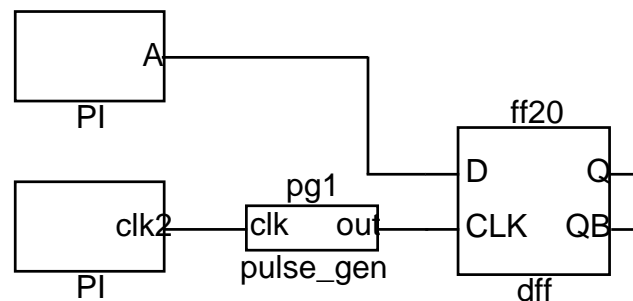
Notice how A goes to X for the load\_unload simulation. This is because it is not explicitly forced in the load\_unload procedure (or constrained with an [add\\_input\\_constraints](#) command).

## Example 3 — Drc\_pattern Reporting for Pulse Generators

If a circuit contains a pulse generator (PG) with user-defined timing, the tool performs additional simulation steps for the PG's output changes. When a rising edge event occurs at its input, a PG outputs a 1 after a certain delay, which the tool simulates as an additional event. After another delay, the PG's output signal returns to 0, which is also simulated as a separate event. Both output events are added to reports within a pair of brackets ([ ]) following the input event.

Suppose the non-scan flip-flop ff20 of the preceding example is clocked by a PG as shown in [Figure B-5](#). Excerpts below the figure show how the PG events would appear in gate reports.

**Figure B-5. Clocking ff20 with a Pulse Generator**



```
// /clk2 primary_input
//      (ts) (      ld      ) (shift) ( cap ) (stbl)
//      clk2  O  ( 0) (001[11]01[11]0) (000~0) (0X[X]0) ( 0)  /pg1/clk

// /pg1 pulse_gen
//      (ts) (      ld      ) (shift) ( cap ) (stbl)
//      clk   I  ( 0) (001[11]01[11]0) (000~0) (0X[X]0) ( 0)  /clk2
//      out   O  ( 0) (000[10]00[10]0) (000~0) (00[X]0) ( 0)  /ff20/CLK

// /ff20 dff
//      (ts) (      ld      ) (shift) ( cap ) (stbl)
//      CLK   I  ( 0) (000[10]00[10]0) (000~0) (00[X]0) ( 0)  /pg1/out
//      D     I  ( 1) (XXX[XX]XX[XX]X) (XXX~X) (XX[X]X) ( X)  /A
//      Q     O  ( 0) (000[XX]XX[XX]X) (XXX~X) (XX[X]X) ( X)
//      QB    O  ( 1) (111[XX]XX[XX]X) (XXX~X) (XX[X]X) ( X)
```

The rising edge events on clk2 initiate pg1's two output pulses (highlighted in bold). Notice the pulses are not shown simultaneous with the input changes that caused them. This is an exception to the typical display of output changes simultaneous with such input changes, as shown for ff20. Notice also how the active clock edge at ff20's CLK input is one event later than clk2's active edge and is seen to be a PG signal due to the brackets.

For an introduction to pulse generators, see [“Pulse Generators”](#) on page 108. For detailed information about the tool's pulse generator primitive, see [“Pulse Generators with User Defined Timing”](#) in the *Tessent Cell Library Manual*.

## Example 4 — Single Post Shift

This example has multiple cycles in the load\_unload procedure. What is new is the single post shift (highlighted in bold).

```
procedure load_unload =
  scan_group grp1 ;
  timeplate gen_tpl ;
  // First cycle, one PI event (force)
  cycle =
    force clk1 0 ;
    force clk2 0 ;
    force reset 0 ;
    force scan_en 1 ;
    force B 1;
    force C 1;
  end ;
  // Second cycle, three PI events (force, pulse on, pulse off)
  cycle =
    force E 0;
    pulse clk2;
  end;
  apply shift 2;
  apply shift 1;
end;
```

In this case, the state stability data has an additional group (shown in bold) between the main shift and the capture cycle. This corresponds to the first application of the post shift:

```
// /ff32  dffr
//          (ts) ( ld ) (shift) (shift) (cap) (stbl)
//      R      I ( 0) (0000) (000~0) ( 000 ) (0X0) (   0)  /reset
//      CLK     I ( 0) (0000) (010~0) ( 010 ) (0X0) (   0)  /clk1
//      D       I ( 0) (0000) (000~X) ( xxx ) (XXX) (   X)  /ff31b/Q
//      Q       O ( 0) (0000) (000~X) ( xxx ) (XXX) (   X)
//      QB      O ( 1) (1111) (111~X) ( xxx ) (XXX) (   X)
```

You can see that ff32 is really stable during the first application of shift. If you use the [set\\_stability\\_check](#) All\_shift command in this case, the output is slightly different:

**set\_stability\_check all\_shift**

**set\_system\_mode setup**

**set\_system\_mode analysis**

**report\_gates ff32**

```
// /ff32  dffr
//          (ts) ( ld ) (shift) (shift) (cap) (stbl)
//      R      I ( 0) (0000) (000~0) ( 000 ) (0X0) (   0)  /reset
//      CLK     I ( 0) (0000) (010~0) ( 010 ) (0X0) (   0)  /clk1
//      D       I ( 0) (0000) (000~1) ( 1xx ) (XXX) (   X)  /ff31b/Q
//      Q       O ( 0) (0000) (000~0) ( 011 ) (1XX) (   X)
//      QB      O ( 1) (1111) (111~1) ( 100 ) (0XX) (   X)
```

Notice how ff32 is now 0 throughout the main shift application, but is set to 1 during the post shift. This is due to how A is set to 1 in test\_setup and this pulse is clocked through.

## Example 5 — Single Post Shift with Cycles Between Main and Post Shift

In this example, a post shift exists but there is an additional cycle (shown in bold font) between the main shift and the post shift. This causes yet another group of data to be displayed when you report the state stability data.

```
procedure load_unload =
  scan_group grp1 ;
  timeplate gen_tpl ;
  // First cycle, one PI event (force)
  cycle =
    force clk1 0 ;
    force clk2 0 ;
    force reset 0 ;
    force scan_en 1 ;
    force B 1;
    force C 1;
  end ;
  // Second cycle, three PI events (force, pulse on, pulse off)
  cycle =
    force A 1;
    pulse clk2;
  end ;
  apply shift 3;
  // Third cycle, three PI events (force, pulse on, pulse off)
  cycle =
    force C 1;
    force A 0;
    pulse clk2;
  end ;
  apply shift 1;
end;

procedure shift =
  scan_group grp1 ;
  timeplate gen_tpl ;
  cycle =
    force_sci ;
    measure_sco ;
    pulse clk1 ;
    force C 0;
  end;
end;
```

The fourth data group (highlighted in bold) represents the cycle between the two applications of shift (for the first application of the load\_unload procedure). The fifth data group represents the application of the post shift, and the last (sixth) group represents capture. Notice how pins A and C vary state due to the values forced on them in the load\_unload and shift procedures.

```
// /A primary_input
```

```
//          (ts) ( ld ) (shift) ( ld ) (shift) (cap) (stbl)
//      A      O  ( 1) (X111) (111~1) (000) ( 000 ) (XXX) (  X)
//  /B primary_input
//          (ts) ( ld ) (shift) ( ld ) (shift) (cap) (stbl)
//      B      O  ( 0) (1111) (111~1) (111) ( 111 ) (XXX) (  X)
//  /C primary_input
//          (ts) ( ld ) (shift) ( ld ) (shift) (cap) (stbl)
//      C      O  ( 0) (1111) (000~0) (111) ( 000 ) (XXX) (  X)
```

Notice also that clk2 in this case is pulsed during load\_unload only, not in test\_setup. Here is the modified test\_setup procedure (with the clk2 pulse removed from the second cycle):

```
procedure test_setup =
  scan_group grp1 ;
  timeplate gen_tpl ;
  // First cycle, one event (force)
  cycle =
    force clk1 0 ;
    force clk2 0 ;
    force reset 0;
    force A 0;
    force B 0;
    force C 0;
    force D 0;
    force E 0;
  end ;
  // Second cycle, two events (pulse on, pulse off)
  cycle =
    pulse reset ;
  end;
  // Third cycle, three events (force, pulse on, pulse off)
  cycle =
    force A 1;
    pulse clk1 ;
  end;
end;
```

This results in the following behavior for ff20, which is clocked by clk2:

```
//  /ff20 dff
//          (ts) ( ld ) (shift) ( ld ) (shift) (cap) (stbl)
//      CLK  I  ( 0) (0010) (000~0) (010) ( 000 ) (0X0) (  0)  /clk2
//      D    I  ( 1) (X111) (111~1) (000) ( 000 ) (XXX) (  X)  /A
//      Q    O  ( X) (XX11) (111~1) (100) ( 000 ) (0XX) (  X)
//      QB   O  ( X) (XX00) (000~0) (011) ( 111 ) (1XX) (  X)
```

Notice how the state of this flip-flop is disturbed during capture.

## Example 6 — Cycles After Apply Shift Statement in Load\_unload

This example reuses the load\_unload procedure of the basic example (with only one apply shift statement), but adds three new cycles (shown in bold font) after the application of shift.

```
procedure load_unload =
  scan_group grp1 ;
  timeplate gen_tpl ;
  // First cycle, one PI event (force)
  cycle =
    force clk1 0 ;
    force clk2 0 ;
    force reset 0 ;
    force scan_en 1 ;
    force B 1;
    force C 1;
  end ;
  apply shift 3;
  // Second cycle, one PI event (force)
  cycle =
    force C 1;
  end ;
  // Third cycle, three PI events (force, pulse on, pulse off)
  cycle =
    force C 0;
    pulse clk2;
  end ;
  // Fourth cycle, one PI event (force)
  cycle =
    force C 1;
  end ;
end;
procedure shift =
  scan_group grp1 ;
  timeplate gen_tpl ;
  cycle =
    force_sci ;
    measure_sco ;
    pulse clk1 ;
    force C 0;
  end;
end;
```

In this case, the second data group in the state stability report represents the one event in the cycle before the apply shift statement. The fourth data group represents the events in the three cycles after the apply shift statement (but still for the first application of load\_unload). The first bit is the one event in the second cycle, the next three bits represent the third cycle, and the last bit represents the fourth cycle:

```
// /C primary_input
//      (ts)(ld)(shift)( ld )(cap)(stbl)
//      C      0      ( 0 )( 1 )(000~0)(10001)(XXX)( X)
// /clk2 primary_input
//      (ts)(ld)(shift)( ld )(cap)(stbl)
//      clk2 0      ( 0 )( 0 )(000~0)(00100)(0X0)( 0) /ff20/CLK
```



## Example 7 — No Statements in Load\_unload Prior to Apply Shift

In this example, there are no statements in the load\_unload procedure prior to the apply shift statement and scan\_en is forced in the shift procedure instead of in a separate cycle in load\_unload. Other procedures are as in the basic example. Here are the modified load\_unload and shift procedures:

```
procedure load_unload =
    scan_group grp1 ;
    timeplate gen_tp1 ;
    apply shift 3;
end;

procedure shift =
    scan_group grp1 ;
    timeplate gen_tp1 ;
    cycle =
        force_scan_en 1 ;
        force sci ;
        measure_sco ;
        pulse clk1 ;
        force C 0;
    end;
end;
```

In this case, the report still shows an event for load\_unload in the second data group. This makes it easier to see differences between the end of test\_setup and the entry into the first load\_unload. Such differences can occur because during load\_unload the tool sets to X any primary input pins that are not constrained with an [add\\_input\\_constraints](#) command or explicitly forced in the load\_unload procedure. This is the case for pin A in this example:

```
// /A primary_input
//          (ts) (ld) (shift) (cap) (stbl)
//    A      O  ( 1) ( X) (XXX~X) (XXX) (  X)  /ff30/D  /ff20/D  /ff00/D
// /ff20  dff
//          (ts) (ld) (shift) (cap) (stbl)
//    CLK    I  ( 0) ( 0) (000~0) (0X0) (  0)  /clk2
//    D      I  ( 1) ( X) (XXX~X) (XXX) (  X)  /A
//    Q      O  ( 0) ( 0) (000~0) (0XX) (  X)
//    QB     O  ( 1) ( 1) (111~1) (1XX) (  X)
```

## Example 8 — Basic with Specified Capture Clock

If you specify a capture clock using the [set\\_capture\\_clock](#) command with the -Atpg switch, the format looks slightly different: the specified clock shows up as “010” (or “101”) during capture instead of “0X0” (or “1X1”). If the `set_capture_clock clk1 -Atpg` command is used, the state stability display for a scan cell clocked by `clk1` in the example design looks like the following:

```
// /sff1 sff
//          (ts) (ld) (shift) (cap) (stbl)
//      SE      I  ( X) ( 1) (111~1) (XXX) ( X) /scan_en
//      D       I  ( X) ( X) (XXX~X) (XXX) ( X) /gate1/Y
//      SI      I  ( X) ( X) (XXX~X) (XXX) ( X) /sff2/QB
//      CLK     I  ( 0) ( 0) (010~0) (010) ( 0) /clk1
//      Q       O  ( X) ( X) (XXX~X) (XXX) ( X) /gate2/A0
//      QB      O  ( X) ( X) (XXX~X) (XXX) ( X) /scan_out1
```

## Example 9 — Setting Stability Check to Off and All\_shift

The preceding examples, except as noted in example 3, used the default On setting of the `set_stability_check` command. This command has two other settings: All\_shift and Off.

For All\_shift, the tool simulates the exact number of shifts. This means for situations where particular events take place during shift, the tool will simulate these exactly rather than simulating the stable state after an “infinite” number of shifts. The following state\_stability displays show the difference between using On and using All\_shift for ff32 and the procedures of the basic example (values of interest are highlighted in bold):

set\_stability\_check On:

```
// /ff32  dffr
//          (ts) (ld) (shift) (cap) (stbl)
//      R      I  ( 0) ( 0) (000~0) (0X0) (  0) /reset
//      CLK     I  ( 0) ( 0) (010~0) (0X0) (  0) /clk1
//      D      I  ( 0) ( 0) (000~X) (XXX) (  X) /ff31b/Q
//      Q      O  ( 0) ( 0) (000~X) (XXX) (  X)
//      QB     O  ( 1) ( 1) (111~X) (XXX) (  X)
```

set\_stability\_check All\_shift:

```
// /ff32  dffr
//          (ts) (ld) (shift) (cap) (stbl)
//      R      I  ( 0) ( 0) (000~0) (0X0) (  0) /reset
//      CLK     I  ( 0) ( 0) (010~0) (0X0) (  0) /clk1
//      D      I  ( 0) ( 0) (000~X) (XXX) (  X) /ff31b/Q
//      Q      O  ( 0) ( 0) (000~1) (1XX) (  X)
//      QB     O  ( 1) ( 1) (111~0) (0XX) (  X)
```

In the All\_shift case, notice how the stable state after shift differs from the On case. After exactly three applications of the shift procedure, the state is 1, but after “infinite” applications of the shift procedure, it is X.

When stability checking is set to off, the tool reports only dashes:

```
// /ff32  dffr
//      R      I  (-) /reset
//      CLK     I  (-) /clk1
//      D      I  (-) /ff31/Q
//      Q      O  (-)
//      QB     O  (-)
```

## Example 10 — Pin Constraints, Test\_setup, and State Stability

In this example, the test\_setup procedure is modified so that pin D, which is constrained to C0 by an `add_input_constraints` command, is not forced in the first cycle, but *is* forced to 1 in the second cycle. It is never forced to its constrained value 0.

It is a good practice to always force the constrained pins to their constrained state at the end of the test\_setup procedure. If you do not do that, the tool will add an additional cycle to the test\_setup procedure when you write out the patterns. This recommended practice is not followed in this example:

```
procedure test_setup =
  scan_group grp1
  timeplate gen_tp1 ;
  // First cycle, three PI events (force, pulse on, pulse off)
  cycle =
    force clk1 0 ;
    force clk2 0 ;
    force reset 0 ;
    force A 0 ;
    force B 0 ;
    force C 0 ;
    force E 0 ;
    pulse clk1 ;
  end ;
  // Second cycle, three PI events (force, pulse on, pulse off)
  cycle =
    force D 1 ;
    pulse clk1 ;
  end ;
end;
```

Notice what happens during test\_setup:

**set\_gate\_report drc\_pattern test\_setup**

**report\_gates D ff10**

```
// /D primary_input
// D 0 /ff10/D
//
// Cycle: 0 1
// -----
// 23 467
// Time: 000 000
// -----
// D XXX 111
//
// /ff10 dff
// CLK I /clk1
// D I /D
// Q 0
// QB 0
//
// Cycle: 0 1
```

```
//      -----
//      23 467
//      Time: 000 000
//      -----
//      CLK      010 010
//      D        XXX 111
//      Q        XXX X11
//      QB       XXX X00
```

Then, for state stability, notice how D changes from 1 to 0 between test\_setup and the first application of load\_unload. This is because of the pin constraint to 0 on this pin:

**set\_gate\_report drc\_pattern state\_stability**

**report\_gates D ff10**

```
// /D primary_input
//      (ts)(ld)(shift)(cap)(stbl)
//      D      O  ( 1)( 0)(000~0)(000)(  X) /ff10/D
// /ff10 dff
//      (ts)(ld)(shift)(cap)(stbl)
//      CLK    I  ( 0)( 0)(010~0)(0X0)(  0) /clk1
//      D      I  ( 1)( 0)(000~0)(000)(  X) /D
//      Q      O  ( 1)( 1)(100~X)(XXX)(  X)
//      QB     O  ( 0)( 0)(011~X)(XXX)(  X)
```

## Example 11 — Single Pre Shift

In the load\_unload procedure for this example, a single pre shift is followed by one cycle and then the main shift. Also, pins C and E vary in the different cycles of the load\_unload and shift operations. The differences from the basic versions of these procedures are highlighted in bold.

```

procedure load_unload =
  scan_group grp1 ;
  timeplate gen_tpl ;
  // First cycle, one event (force)
  cycle =
    force clk1 0 ;
    force clk2 0 ;
    force reset 0 ;
    force scan_en 1 ;
    force B 1;
    force C 1;
    force E 1;
  end ;
  apply shift 1;      // Pre shift
  // Second cycle, three PI events (force, pulse on, pulse off)
  cycle =
    force C 0;
    force E 0;
    pulse clk2;
  end;
  apply shift 2;      // Main shift
end;

procedure shift =
  scan_group grp1 ;
  timeplate gen_tpl ;
  cycle =
    force_sci ;
    measure_sco ;
    pulse clk1 ;
    force C 0;
  end;
end;

```

The third group in the state\_stability display (highlighted in bold) is the pre-shift. The fourth group is the cycle between the shift applications, and the fifth group is the main shift.

**set\_gate\_report drc\_pattern state\_stability**

**report\_gates C E clk1 clk2**

```

// /C   primary_input
//           (ts)(ld)(shift)( ld)(shift)(cap)(stbl)
//      C    O  ( 0)( 1)( 000 )(000)(000~0)(XXX)(  X)
// /E   primary_input
//           (ts)(ld)(shift)( ld)(shift)(cap)(stbl)
//      E    O  ( 0)( 1)( 111 )(000)(000~0)(XXX)(  X)
// /clk1  primary_input
//           (ts)(ld)(shift)( ld)(shift)(cap)(stbl)

```

```
//      clk1  O  ( 0)( 0)( 010 )(000)(010~0)(0X0)( 0)/ff32/CLK
// /ff31b/CLK...
// /clk2  primary_input
//      (ts)(ld)(shift)( ld)(shift)(cap)
//      clk2  O  ( 0)( 0)( 000 )(010)(000~0)(0X0)/ff20/CLK
```





## Appendix C

# Running Tessent Shell as a Batch Job

---

You can use Tessent Shell in either an interactive or non-interactive manner. You conduct a tool session interactively by entering the commands manually, or the session can be completely scripted and driven using a dofile. This non-interactive mode of operation allows the entire session to be conducted without user interaction. This method of using Tessent Shell can be further expanded to allow the session to be scheduled and run as a true batch or cron job. This appendix focuses on the features of Tessent Shell that support its use in a batch environment.

## Commands and Variables for the dofile

The following shell script invokes Tessent Shell and then runs a dofile that sets the context to “patterns -scan,” reads a design, and reads a cell library. The dofile also specifies an option to exit from the dofile upon encountering an error:

```
set_dofile_abort exit
```

The exit option sets the exit code to a non-zero value if an error occurs during execution of the dofile. This allows a shell script that launches a Tessent Shell session to control process flow based on the success or failure of a tool operation. Note the line check for the exit status following the line that invokes Tessent Shell.

```
#!/bin/csh -b
##
## Add the pathname of the <Tessent_Tree_Path>/bin directory to the PATH
## environment variable so you can invoke the tool without typing the full
## pathname
##
setenv PATH <Tessent_Tree_Path>/bin:${PATH}
##
setenv DESIGN `pwd`
##
##
tessent -shell -dofile ${DESIGN}/tshell.do \
    -license_wait 30 -log ${DESIGN}/`date +%m_%d_%y_%H:%M:%S`
setenv proc_status $status
if (" $proc_status" == 0 ) then
    echo "Session was successful"
    echo " The exit code is: " $proc_status
else
    echo "Session failed"
    echo " The exit code is: " $proc_status
endif
echo $proc_status " is the exit code value."
```

You can use environment variables in a Tessent Shell dofile. For example, the shell script sets the DESIGN environment variable to the current working directory. When a batch job is

created, the process may not inherit the same environment that existed in the shell environment. To assure that the process has access to the files referenced in the dofile, the DESIGN environment variable is used. A segment of a Tessent Shell dofile displaying the use of an environment variable follows:

```
# The shell script that launches this dofile sets the DESIGN environment
# variable to the current working directory.
add_scan_groups g1 ${DESIGN}/procfile
#
add_scan_chains c1 g1 scan_in CO
...
#
write_faults ${DESIGN}/fault_list -all -replace
```

You can also use a startup file to alias common commands. To set up the predefined alias commands, use the file *.tessent\_startup* (located by default in your home directory). For example:

```
alias save_my_pat write_patterns $1/pats.v -$2 -replace
```

The following dofile segment displays the use of the alias defined in the *.tessent\_startup* file:

```
# The following alias is defined in the .tessent_startup file
#
save_my_pat $DESIGN verilog
```

Another important consideration is to exit in a graceful manner from the dofile. This is required to assure that Tessent Shell exits instead of waiting for additional command line input.

```
# The following command terminates the Tessent Shell dofile.
#
exit -force
```

## Command Line Options

Several Tessent Shell command line options are useful when running Tessent Shell as a batch job. One of these options is the *-LICENSE\_wait* option, which sets a limit for retrying license acquisition after you set a context. The default is no time limit for a license.

If Tessent Shell is unable to obtain a license after the specified number of retries, the tool exits. An example of the Tessent Shell invocation line with this option follows:

```
% tessent -shell-dofile tshell.do -license_wait 30 \
-log ${DESIGN}/`date +%m_%d_%y_%H:%M:%S`
```

Another item of interest is the logfile name created using the UNIX “date” command for each Tessent Shell run. The logfile is based on the month, day, year, hour, minute, and second that the batch job was launched. An example of the logfile name that would be created follows:

```
log_file_05_30_12_08:42:37
```

## Scheduling a Batch Job for Execution Later

You can schedule a batch job using the UNIX “at” or cron” command. For more information about using either of these commands, use the UNIX “man” command.



There are several ways to get help when setting up and using Tessent® software tools. Depending on your need, help is available from documentation, online command help, and Mentor Graphics Support.

## Documentation

A comprehensive set of reference manuals, user guides, and release notes is available in two formats:

- HTML for searching and viewing online
- PDF for searching, viewing online, and printing

The documentation is available from each software tool and online at:

<http://supportnet.mentor.com>

For more information on setting up and using Tessent documentation, see the “[Using Tessent Documentation](#)” chapter in the *Managing Mentor Graphics Tessent Software* manual.

## Mentor Graphics Support

Mentor Graphics software support includes software enhancements, access to comprehensive online services with SupportNet, and the optional On-Site Mentoring service. For details, see:

<http://supportnet.mentor.com/about/>

If you have questions about a software release, you can log in to SupportNet and search thousands of technical solutions, view documentation, or open a Service Request online:

<http://supportnet.mentor.com>

If your site is under current support and you do not have a SupportNet login, you can register for SupportNet by filling out a short form here:

<http://supportnet.mentor.com/user/register.cfm>

All customer support contact information is available here:

<http://supportnet.mentor.com/contacts/supportcenters/index.cfm>



## — A —

- Abort limit, [193](#)
- Aborted faults, [193](#)
  - changing the limits, [193](#)
  - reporting, [193](#)
- Ambiguity
  - edge, [228](#)
  - path, [228](#)
- ASCII WGL format, [351](#)
- ASIC Vector Interfaces, [18](#)
- ATPG
  - applications, [31](#)
  - ATPG method, [157 to 162](#)
  - basic operations, [163](#)
  - constraints, [184](#)
  - default run, [191](#)
  - defined, [30](#)
  - features, [31](#)
  - for IDDQ, [197](#)
  - for path delay, [220](#)
  - for transition fault, [206 to 211](#)
  - full scan, [31](#)
  - function, [184](#)
  - inputs and outputs, [156](#)
  - MacroTest, using, [267](#)
  - non-scan cell handling, [104 to 107](#)
  - pattern types, [158 to 162](#)
  - process, [183](#)
  - setting up faults, [176, 180](#)
  - test cycles, [158](#)
  - timing model, [158](#)
  - tool flow, [154](#)
- ATPG Commands
  - create\_patterns, [253](#)
  - set\_atpg\_timing, [256, 258](#)
- ATPG commands
  - add\_ambiguous\_paths, [224](#)
  - add\_capture\_handling, [172](#)
  - add\_cell\_constraints, [174](#)
  - add\_clocks, [173](#)
  - add\_faults, [176, 180](#)
  - add\_input\_constraints, [163, 165](#)
  - add\_lists, [177, 178](#)
  - add\_nofaults, [175](#)
  - add\_output\_masks, [165](#)
  - add\_primary\_inputs, [164](#)
  - add\_scan\_chains, [173](#)
  - add\_scan\_groups, [173](#)
  - add\_slow\_pad, [168](#)
  - add\_tied\_signals, [167](#)
  - analyze\_atpg\_constraints, [186](#)
  - analyze\_bus, [169](#)
  - analyze\_fault, [224](#)
  - analyze\_graybox, [403](#)
  - analyze\_restrictions, [186](#)
  - create\_flat\_model, [76](#)
  - delete\_atpg\_constraints, [186](#)
  - delete\_atpg\_functions, [186](#)
  - delete\_cell\_constraints, [174](#)
  - delete\_clocks, [173](#)
  - delete\_false\_paths, [220](#)
  - delete\_fault\_sites, [224](#)
  - delete\_faults, [180](#)
  - delete\_input\_constraints, [163, 168](#)
  - delete\_multicycle\_paths, [220](#)
  - delete\_nofaults, [175](#)
  - delete\_primary\_inputs, [164](#)
  - delete\_primary\_outputs, [164](#)
  - delete\_scan\_chains, [174](#)
  - delete\_scan\_groups, [173](#)
  - delete\_tied\_signals, [167](#)
  - read\_fault\_sites, [224](#)
  - read\_faults, [181](#)
  - read\_patterns, [176](#)
  - report\_aborted\_faults, [193](#)
  - report\_atpg\_constraints, [186](#)
  - report\_atpg\_functions, [186](#)
  - report\_bus\_data, [169](#)
  - report\_cell\_constraints, [174](#)

report\_clocks, 173  
 report\_false\_paths, 220  
 report\_fault\_sites, 224  
 report\_faults, 177, 180, 192  
 report\_gates, 169  
 report\_input\_constraints, 163, 168  
 report\_nofaults, 175  
 report\_nonscan\_cells, 107  
 report\_paths, 224  
 report\_primary\_inputs, 164  
 report\_primary\_outputs, 164  
 report\_scan\_chains, 174  
 report\_scan\_groups, 173  
 report\_statistics, 177  
 report\_tied\_signals, 167  
 reset\_state, 178  
 run, 191  
 set\_abort\_limit, 193, 210  
 set\_bus\_handling, 169  
 set\_capture\_clock, 177  
 set\_checkpointing\_options, 189  
 set\_clock\_restriction, 174  
 set\_contention\_check, 169  
 set\_drc\_handling, 172  
 set\_driver\_restriction, 169  
 set\_fault\_mode, 182  
 set\_fault\_type, 176, 180, 210  
 set\_learn\_report, 168  
 set\_list\_file, 177, 178  
 set\_net\_dominance, 169  
 set\_net\_resolution, 169  
 set\_pattern\_buffer, 168  
 set\_pattern\_type, 107, 160, 161, 210  
 set\_possible\_credit, 168, 182  
 set\_pulse\_generators, 168  
 set\_random\_atpg, 193  
 set\_random\_clocks, 177  
 set\_random\_patterns, 177  
 set\_split\_capture\_cycle, 172  
 set\_system\_mode, 163, 175  
 set\_tied\_signals, 167  
 set\_z\_handling, 170  
 write\_fault\_sites, 224  
 write\_faults, 181  
 write\_patterns, 194

ATPG tool, definition of, 17  
 At-speed test, 33, 205 to 243

## — B —

Batch mode  
     *see also* Appendix C  
 Bidirectional pin  
     as primary input, 164  
     as primary output, 164  
 Boundary scan, defined, 21  
 Bridge fault testing  
     bridge fault definition file keywords, 203  
     creating the test set, 218  
     static bridge fault model, 39  
 Bus  
     dominant, 80  
     float, 102  
 Bus contention, 102  
     checking during ATPG, 169  
     fault effects, 169

## — C —

Capture handling, 171  
 Capture point, 41  
 Capture procedure, *see* Named capture procedure  
 Chain test, 348  
 Checkpointing example, 190  
 Checkpoints, setting, 189  
 Clock  
     capture, 173, 261  
     list, 173  
     off-state, 173  
     scan, 173  
 Clock PO patterns, 159  
 Clock procedure, 158  
 Clock sequential patterns, 160  
 Clocked sequential test generation, 106  
 Clocks, merging chains with different, 145  
 Combinational loop, 94, 95, 96, 97, 98  
     cutting, 95  
 Constant value loops, 95  
 Constraints  
     ATPG, 184  
     pin, 167  
     scan cell, 174



Contention, bus, [84](#)

Control points  
manual identification, [133](#)

Controllability, [27](#)

Copy, scan cell element, [70](#)

Coupling loops, [98](#)

CPF commands, [370](#)

create\_patterns, [253](#)

Creating a delay test set, [206](#)

Creating patterns  
default run, [191](#)

Cycle count, [350](#)

Cycle test, [348](#)

## — D —

Data capture simulation, [171](#)

Debugging simulation mismatches, [287](#)

Defect, [32](#)

Delay test coverage, [251](#)

Design Compiler, handling pre-inserted scan  
cells, [128](#)

Design flattening, [76 to 81](#)

Design flow, delay test set, [206](#)

Design rules checking  
blocked values, [89](#)  
bus keeper analysis, [87](#)  
bus mutual-exclusivity, [84](#)  
clock rules, [87](#)  
constrained values, [89](#)  
data rules, [87](#)  
extra rules, [88](#)  
forbidden values, [89](#)  
general rules, [84](#)  
introduction, [83](#)  
procedure rules, [84](#)  
RAM rules, [87](#)  
scan chain tracing, [86](#)  
scannability rules, [88](#)  
shadow latch identification, [86](#)  
transparent latch identification, [87](#)

Design-for-Test, defined, [17](#)

Deterministic test generation, [31](#)

DFF modeling, [187](#)

DFTVisualizer  
*see "DFTVisualizer" in the Tessent Shell  
User's Manual*

Differential scan input pins, [347](#)

Distributed ATPG  
troubleshooting SSH, [307](#)

Dominant bus, [80](#)

## — E —

Edge ambiguity, [228](#)

Existing scan cells  
handling, [128](#)

External pattern generation, [31](#)

Extra, scan cell element, [71](#)

## — F —

Fault

aborted, [193](#)  
classes, [55 to 65](#)  
collapsing, [36](#)  
detection, [54](#)  
no fault setting, [175](#)  
representative, [36, 65](#)  
simulation, [176](#)  
undetected, [193](#)

Fault grading  
in multiple fault model flow, [248](#)  
pattern generation and, [248](#)

Fault models  
bridge, [39](#)  
path delay, [41](#)  
pseudo stuck-at, [38](#)  
stuck-at, [36, 39](#)  
toggle, [37](#)  
transition, [40](#)

Fault sampling, [189](#)

Faults

small delay, [250](#)

Feedback loops, [94](#)

Fixed-order file, [142](#)

Flattening, design, [76 to 81](#)

Frame, simulation, [187](#)

Full scan, [24, 122](#)

Functional test, [33](#)

## — G —

Gate duplication, [96](#)

Good simulation, [178](#)

graybox overview, [403](#)

## — H —

Head register, attaching, [143](#)  
 Hierarchical instance, definition, [76](#)  
 Hold\_pi, [258](#)

## — I —

IDDQ testing, [197](#)  
     creating the test set, [197](#)  
     defined, [33](#)  
     methodologies, [33](#)  
     pseudo stuck-at fault model, [38](#)  
     test pattern formats, [347](#)  
     vector types, [34](#)  
 Incomplete designs, [117](#)  
 Instance, definition, [76](#)  
 Internal scan, [21](#)

## — L —

Latch modeling, [187](#)  
 Latches  
     handling as non-scan cells, [103](#)  
     scannability checking of, [94](#)  
 Launch point, [41](#)  
 Layout-sensitive scan insertion, [145](#)  
 Learning analysis, [81 to 83](#)  
     dominance relationships, [83](#)  
     equivalence relationships, [81](#)  
     forbidden relationships, [83](#)  
     implied relationships, [82](#)  
     logic behavior, [81](#)  
 Line holds, [56, 57](#)  
 Loop count, [350](#)  
 Loop cutting, [95](#)  
     by constant value, [95](#)  
     by gate duplication, [96](#)  
     for coupling loops, [98](#)  
     single multiple fanout, [96](#)  
 Loop handling, [94 to ??](#)  
 LSSD, [74](#)

## — M —

Macros, [34](#)  
 MacroTest, [267](#)  
     basic flow, [267, 268](#)  
     capabilities, summary, [268](#)  
     examples, [281, 283, 285](#)

basic 1-Cycle Patterns, [281](#)  
 leading & trailing edge observation, [285](#)  
 synchronous memories, [283](#)  
 macro boundary  
     defining, [273](#)  
         with instance name, [273](#)  
         with trailing edge inputs, [276](#)  
         without instance name, [274](#)  
     reporting & specifying observation sites, [275](#)  
 overview, [267](#)  
 qualifying macros, [270](#)  
 recommendations for using, [279](#)  
 test values, [277](#)  
 when to use, [271](#)

Manufacturing defect, [32](#)  
 Mapping scan cells, [123](#)  
 Mask\_po, [258](#)  
 Masking primary outputs, [167, 168](#)  
 Master, scan cell element, [69](#)  
 Memories, testing, [267](#)  
 Merging scan chains, [145](#)  
 Module, definition, [76](#)  
 MTFI, [391](#)  
 Multiple load patterns, [160](#)

## — N —

Named capture procedure  
     at-speed test using, [230 to ??](#)  
     internal and external mode of, [232 to 238](#)  
     on-chip clocks and, [231](#)  
 No fault setting, [175](#)  
 Non-scan cell handling, [103 to 107](#)  
     ATPG, [104](#)  
     clocked sequential, [106](#)  
     sequential transparent, [105](#)  
     tie-0, [104](#)  
     tie-1, [104](#)  
     tie-X, [104](#)  
     transparent, [104](#)  
 Non-scan sequential instances  
     reporting, [136](#)

## — O —

Observability, [27](#)

Observe points  
     manual identification, 133

Off-state, 73, 127, 173

One-shot circuit, 259

Output mask, 258

## — P —

Parallel scan chain loading, 345

Partition scan, 25, 122

Path ambiguity, 228

Path definition file, 225

Path delay testing, 41, 220

    basic procedure, 229

    limitations, 230

    multiple fault models and  
         flow, 248

    path ambiguity, 228

    path definition checking, 228

    path definition file, 225

    patterns, 221

    robust detection, 222

    transition detection, 222

Path sensitization, 54

Pattern formats

    binary, 350

    text, 347

    Verilog, 350

    WGL (ASCII), 351

Pattern generation

    deterministic, 31

    external source, 31

    multiple fault models and  
         flow, 248

    random, 30

Pattern generation phase

    test procedure waveforms, example, 412

Pattern types

    basic scan, 158

    clock PO, 159

    clock sequential, 160

    multiple load, 160

    RAM sequential, 161

    sequential transparent, 162

Performing ATPG, 183 to 194

Pin constraints, 167

Possible-detect credit, 59, 182

Possible-detected faults, 59

power-aware ATPG procedure, 372

power-aware overview, 367

Pre-inserted scan cells

    handling, 128

Primary inputs

    bidirectional pins as, 164

    constraining, 167

    constraints, 167

Primary outputs

    bidirectional pins as, 164

    masking, 168

Primitives, simulation, 78

Pulse generators

    gate reporting for, 428

    introduction, 108

## — R —

RAM

    ATPG support, 111

    common read and clock lines, 113

    common write and clock lines, 114

    pass-through mode, 112

    RAM sequential mode, 112

    RAM sequential mode, read/write clock  
         requirement, 112

    read-only mode, 111

    related commands, 114 to 115

    rules checking, 115 to 116

    sequential patterns, 161

    testing, 110 to 116

Random pattern generation, 30

Registers

    head, attaching, 143

    tail, attaching, 143

report\_graybox\_statistics, 403

ROM

    ATPG support, 111

    related commands, 114 to 115

    rules checking, 115

    testing, 110 to 116

## — S —

Sampling, fault, 189

Saving patterns

    Serial versus parallel, 345

- Scan
  - basic operation, [23](#)
  - clock, [23](#)
- Scan and/or Wrapper test structures, [123](#)
- Scan cell
  - concepts, [67](#)
  - constraints, [174](#)
  - existing, [128](#)
  - mapping, [123](#)
  - pre-inserted, [128](#)
- Scan cell elements
  - copy, [70](#)
  - extra, [71](#)
  - master, [69](#)
  - shadow, [69](#)
  - slave, [69](#)
- Scan chains
  - assigning scan pins, [142](#)
  - definition, [71](#)
  - fixed-order file, [142](#)
  - head and tail registers, attaching, [143](#)
  - merging, [145](#)
  - parallel loading, [345](#)
  - serial loading, [346](#)
  - serial versus parallel loading, [345](#)
  - specifying, [173](#)
- Scan clocks, [73](#), [127](#)
  - specifying, [127](#), [173](#)
- Scan design
  - simple example, [23](#)
- Scan design, defined, [21](#)
- Scan groups, [72](#), [173](#)
- Scan insertion
  - layout-sensitive, [145](#)
  - process, [119](#)
- Scan output mapping, [123](#)
- Scan patterns, [158](#)
- Scan pins
  - assigning, [142](#)
- Scan related events, [339](#)
- Scan sub-chain, [345](#)
- Scan test, [348](#)
- Scannability checks, [93](#)
- SDF file, [256](#)
- Sequential element modeling, [187](#)
- Sequential loop, [94](#), [99](#)
- Sequential transparent latch handling, [105](#)
- Sequential transparent patterns, [162](#)
- Serial scan chain loading, [346](#)
- set\_atpg\_timing, [256](#), [258](#)
- Shadow, [69](#)
- Simulating captured data, [171](#)
- Simulation data formats, [347](#)
- Simulation formats, [344](#)
- Simulation frame, [187](#)
- Simulation mismatches, debugging, [287](#)
- Simulation primitives, [78 to 81](#)
  - AND, [78](#)
  - BUF, [78](#)
  - BUS, [80](#)
  - DFF, [79](#)
  - INV, [78](#)
  - LA, [79](#)
  - MUX, [79](#)
  - NAND, [78](#)
  - NMOS, [80](#)
  - NOR, [78](#)
  - OR, [78](#)
  - OUT, [81](#)
  - PBUS, [80](#)
  - PI, [78](#)
  - PO, [78](#)
  - RAM, [80](#)
  - ROM, [80](#)
  - STFF, [79](#)
  - STLA, [79](#)
  - SW, [80](#)
  - SWBUS, [80](#)
  - TIE gates, [79](#)
  - TLA, [79](#)
  - TSD, [79](#)
  - TSH, [79](#)
  - WIRE, [80](#)
  - XNOR, [78](#)
  - XOR, [78](#)
  - ZHOLD, [80](#)
  - ZVAL, [78](#)
- Single multiple fanout loops, [96](#)
- Sink gates, [171](#)
- Slack, [250](#)

- Slave, [69](#)
- Small delay fault model, [250](#)
- Source gates, [171](#)
- SSH protocol
  - troubleshooting, [307](#)
- Static-inactive path, [259](#)
- Structural loop, [94](#)
  - combinational, [94](#)
  - sequential, [94](#)
- Structured DFT, [18](#)
- System-class
  - non-scan instances, [134](#)
  - scan instances, [134](#)
- T —
- Tail register, attaching, [143](#)
- Tessent FastScan, available in Tessent Shell, [17](#)
- Tessent Scan
  - block-by-block scan insertion, [149 to 151](#)
  - features, [29](#)
  - inputs and outputs, [121](#)
  - invocation, [123](#)
  - process flow, [119](#)
  - supported test structures, [122](#)
- Tessent Scan commands
  - add\_clocks, [127](#)
  - add\_nonscan\_instances, [135](#)
  - add\_scan\_chains, [129](#)
  - add\_scan\_groups, [128](#)
  - add\_scan\_pins, [142, 144](#)
  - delete\_buffer\_insertion, [144](#)
  - delete\_cell\_models, [127](#)
  - delete\_clock\_groups, [147](#)
  - delete\_clocks, [127](#)
  - delete\_nonscan\_instances, [136](#)
  - delete\_nonscan\_models, [136](#)
  - delete\_scan\_instances, [136](#)
  - delete\_scan\_models, [136](#)
  - delete\_scan\_pins, [142](#)
  - report\_buffer\_insertion, [144](#)
  - report\_cell\_models, [127](#)
  - report\_clock\_groups, [147](#)
  - report\_clocks, [127](#)
  - report\_control\_signals, [137](#)
  - report\_dft\_check, [136, 147](#)
  - report\_nonscan\_models, [136](#)
  - report\_primary\_inputs, [127](#)
  - report\_scan\_cells, [147](#)
  - report\_scan\_chains, [147](#)
  - report\_scan\_groups, [147](#)
  - report\_scan\_models, [136](#)
  - report\_scan\_pins, [142](#)
  - report\_sequential\_instances, [136, 137](#)
  - report\_shift\_registers, [140](#)
  - report\_statistics, [137](#)
  - report\_test\_logic, [127](#)
  - set\_scan\_pins, [142](#)
  - set\_system\_mode, [131](#)
  - write\_primary\_inputs, [127](#)
- Tessent Scan, available in Tessent Shell, [17](#)
- Tessent Shell, providing ATPG and Scan functionality, [17](#)
- Tessent TestKompress, available in Tessent Shell, [17](#)
- Test clock, [108, 125](#)
- Test logic, [93, 108, 124](#)
- Test Pattern File Format, [355](#)
  - Functional Chain Test, [358](#)
  - Header\_Data, [355](#)
  - Scan Cell, [361](#)
  - Scan Test, [359](#)
- Test patterns, [30](#)
  - chain test block, [348](#)
  - scan test block, [349](#)
- Test points
  - definition of, [123](#)
  - understanding, [27](#)
- Test procedure file
  - in Tessent Scan, [122](#)
- Test structures
  - full scan, [24 to 25, 122](#)
  - partition scan, [25 to 27, 122](#)
  - Scan and/or Wrapper, [123](#)
  - supported by Tessent Scan, [122](#)
  - test points, [27 to 28, 123](#)
- Test types
  - at-speed, [34](#)
  - functional, [33](#)
  - IDDQ, [33](#)
- Test vectors, [30](#)

Testability, [17](#)  
 Testing  
     memories, [267](#)  
 TI TDL 91 format, [352 to ??](#)  
 Tie-0 gate, [104](#)  
 TIE0, scannable, [93](#)  
 Tie-1 gate, [104](#)  
 TIE1, scannable, [93](#)  
 Tie-X gate, [104](#)  
 Timeplate statements  
     bidi\_force\_pi, [342](#)  
     bidi\_measure\_po, [342](#)  
     force, [342](#)  
     force\_pi, [342](#)  
     measure, [342](#)  
     measure\_po, [342](#)  
     offstate, [342](#)  
     period, [343](#)  
     pulse, [343](#)  
 Timing-aware ATPG  
     detection paths, [259](#)  
     example, [254](#)  
     limitations, [252](#)  
     reducing run time, [256](#)  
     setting up, [253](#)  
     troubleshooting, [255](#)  
 Toshiba TSTL2 format, [353](#)  
 Transition ATPG, [251](#)  
 Transition fault testing, [206 to 211](#)  
 Transition testing  
     basic procedures, [211](#)  
     patterns, [208](#)  
 Transparent latch handling, [104](#)  
 Transparent slave handling, [106](#)  
 Troubleshooting  
     sdf file, [256](#)  
     timing-aware ATPG, [255](#)

## — U —

UDFM, [42](#)  
 Undetected faults, [193](#)  
 UPF commands, [370](#)  
 User-class  
     non-scan instances, [134](#)  
     scan instances, [135](#)  
     test points, [133](#)

User-Defined Fault Model, [42](#)

## — V —

Verilog, [350](#)

# Third-Party Information

For information about third-party software included with this release of Tessent products, refer to the [\*Third-Party Software for Tessent Products\*](#).







# End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:  
[www.mentor.com/eula](http://www.mentor.com/eula)

## IMPORTANT INFORMATION

**USE OF ALL SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF SOFTWARE INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.**

## END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software (as defined in Section 2) and hardware (collectively "Products") between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Software received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

### 1. ORDERS, FEES AND PAYMENT.

- 1.1. To the extent Customer (or if agreed by Mentor Graphics, Customer's appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement ("Order(s)"), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement, any applicable addenda and the applicable quotation, whether or not these documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order or presented via any electronic portal or other automated order management system will not be effective unless agreed in writing by an authorized representative of Customer and Mentor Graphics.
- 1.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice(s). Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer's sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer's behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.
- 1.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics retains a security interest in all Products delivered under this Agreement, to secure payment of the purchase price of such Products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or convenient for use in filing or perfecting such security interest. Mentor Graphics' delivery of Software by electronic means is subject to Customer's provision of both a primary and an alternate e-mail address.

2. **GRANT OF LICENSE.** The software installed, downloaded, or otherwise acquired by Customer under this Agreement, including any updates, modifications, revisions, copies, documentation and design data ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form (except as provided in Subsection 5.2); (b) for Customer's internal business purposes; (c) for the term of the license; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of employment is the site where the Software is authorized for use. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions. For the avoidance of doubt, if Customer provides any feedback or requests any change or enhancement to Products,

whether in the course of receiving support or consulting services, evaluating Products, performing beta testing or otherwise, any inventions, product improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics' sole discretion) will be the exclusive property of Mentor Graphics.

3. **ESC SOFTWARE.** If Customer purchases a license to use development or prototyping tools of Mentor Graphics' Embedded Software Channel ("ESC"), Mentor Graphics grants to Customer a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESC compilers, including the ESC run-time libraries distributed with ESC C and C++ compiler Software that are linked into a composite program as an integral part of Customer's compiled computer program, provided that Customer distributes these files only in conjunction with Customer's compiled computer program. Mentor Graphics does NOT grant Customer any right to duplicate, incorporate or embed copies of Mentor Graphics' real-time operating systems or other embedded software products into Customer's products or applications without first signing or otherwise agreeing to a separate agreement with Mentor Graphics for such purpose.

4. **BETA CODE.**

- 4.1. Portions or all of certain Software may contain code for experimental testing and evaluation (which may be either alpha or beta, collectively "Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and Customer's use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form.
- 4.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.
- 4.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 4.3 shall survive termination of this Agreement.

5. **RESTRICTIONS ON USE.**

- 5.1. Customer may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Customer shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer's employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics written notice of any unauthorized disclosure or use of the Products as soon as Customer becomes aware of such unauthorized disclosure or use. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive any source code from Software. Log files, data files, rule files and script files generated by or for the Software (collectively "Files"), including without limitation files containing Standard Verification Rule Format ("SVRF") and Tcl Verification Format ("TVF") which are Mentor Graphics' proprietary syntaxes for expressing process rules, constitute or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Software or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Software, or disclose to any third party the results of, or information pertaining to, any benchmark.
- 5.2. If any Software or portions thereof are provided in source code form, Customer will use the source code only to correct software errors and enhance or modify the Software for the authorized use. Customer shall not disclose or permit disclosure of source code, in whole or in part, including any of its methods or concepts, to anyone except Customer's employees or on-site contractors, excluding Mentor Graphics competitors, with a need to know. Customer shall not copy or compile source code in any manner except to support this authorized use.
- 5.3. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms

of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.

5.4. The provisions of this Section 5 shall survive the termination of this Agreement.

6. **SUPPORT SERVICES.** To the extent Customer purchases support services, Mentor Graphics will provide Customer with updates and technical support for the Products, at the Customer site(s) for which support is purchased, in accordance with Mentor Graphics' then current End-User Support Terms located at <http://supportnet.mentor.com/about/legal/>.

7. **LIMITED WARRANTY.**

7.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification, improper installation or Customer is not in compliance with this Agreement. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) PRODUCTS PROVIDED AT NO CHARGE; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

7.2. THE WARRANTIES SET FORTH IN THIS SECTION 7 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO PRODUCTS PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

8. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT RECEIVED FROM CUSTOMER FOR THE HARDWARE, SOFTWARE LICENSE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 8 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

9. **HAZARDOUS APPLICATIONS.** CUSTOMER ACKNOWLEDGES IT IS SOLELY RESPONSIBLE FOR TESTING ITS PRODUCTS USED IN APPLICATIONS WHERE THE FAILURE OR INACCURACY OF ITS PRODUCTS MIGHT RESULT IN DEATH OR PERSONAL INJURY ("HAZARDOUS APPLICATIONS"). EXCEPT TO THE EXTENT THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF MENTOR GRAPHICS PRODUCTS IN OR FOR HAZARDOUS APPLICATIONS. THE PROVISIONS OF THIS SECTION 9 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

10. **INDEMNIFICATION.** CUSTOMER AGREES TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH THE USE OF MENTOR GRAPHICS PRODUCTS IN OR FOR HAZARDOUS APPLICATIONS. THE PROVISIONS OF THIS SECTION 10 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

11. **INFRINGEMENT.**

- 11.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay costs and damages finally awarded against Customer that are attributable to such action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

- 11.2. If a claim is made under Subsection 11.1 Mentor Graphics may, at its option and expense: (a) replace or modify the Product so that it becomes noninfringing; (b) procure for Customer the right to continue using the Product; or (c) require the return of the Product and refund to Customer any purchase price or license fee paid, less a reasonable allowance for use.
- 11.3. Mentor Graphics has no liability to Customer if the action is based upon: (a) the combination of Software or hardware with any product not furnished by Mentor Graphics; (b) the modification of the Product other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code or Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; or (h) infringement by Customer that is deemed willful. In the case of (h), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.
- 11.4. THIS SECTION 11 IS SUBJECT TO SECTION 8 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS, AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, FOR DEFENSE, SETTLEMENT AND DAMAGES, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.

## **12. TERMINATION AND EFFECT OF TERMINATION.**

- 12.1. If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term. Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement immediately upon written notice if Customer: (a) exceeds the scope of the license or otherwise fails to comply with the licensing or confidentiality provisions of this Agreement, or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. For any other material breach of any provision of this Agreement, Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement upon 30 days written notice if Customer fails to cure the breach within the 30 day notice period. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination.
- 12.2. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination, Customer shall ensure that all use of the affected Products ceases, and shall return hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form.
13. **EXPORT.** The Products provided hereunder are subject to regulation by local laws and United States ("U.S.") government agencies, which prohibit export, re-export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export or re-export Products in any manner without first obtaining all necessary approval from appropriate local and U.S. government agencies. If Customer wishes to disclose any information to Mentor Graphics that is subject to any U.S. or other applicable export restrictions, including without limitation the U.S. International Traffic in Arms Regulations (ITAR) or special controls under the Export Administration Regulations (EAR), Customer will notify Mentor Graphics personnel, in advance of each instance of disclosure, that such information is subject to such export restrictions.
14. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. The parties agree that all Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to U.S. FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. government or a U.S. government subcontractor is subject solely to the terms and conditions set forth in this Agreement, which shall supersede any conflicting terms or conditions in any government order document, except for provisions which are contrary to applicable mandatory federal laws.
15. **THIRD PARTY BENEFICIARY.** Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.
16. **REVIEW OF LICENSE USAGE.** Customer will monitor the access to and use of Software. With prior written notice and during Customer's normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system and records deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. The provisions of this Section 16 shall survive the termination of this Agreement.
17. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** The owners of certain Mentor Graphics intellectual property licensed under this Agreement are located in Ireland and the U.S. To promote consistency around the world, disputes shall be resolved as follows: excluding conflict of laws rules, this Agreement shall be governed by and construed under the laws of the State of Oregon, U.S., if Customer is located in North or South America, and the laws of Ireland if

Customer is located outside of North or South America. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the courts of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply. Notwithstanding the foregoing, all disputes in Asia arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer in the jurisdiction where Customer's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.

18. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.
19. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements. Some Software may contain code distributed under a third party license agreement that may provide additional rights to Customer. Please see the applicable Software documentation for details. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.

Rev. 130502, Part No. 255853