

TetraMAX® ATPG and TetraMAX II User Guide

Version M-2016.12, December 2016

SYNOPSYS®

Copyright Notice and Proprietary Information

Copyright © 2016 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at <http://www.synopsys.com/Company/Pages/Trademarks.aspx>. All other product or company names may be trademarks of their respective owners. Inc.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
700 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Contents

About This User Guide	xxxi
Audience	xxxi
Related Publications	xxxi
Release Notes	xxxii
Conventions	xxxiii
Customer Support	xxxiii
Accessing SolvNet	xxxiii
Contacting the Synopsys Technical Support Center	xxxiv
Getting Started	1-1
TetraMAX ATPG and TetraMAX II	1-2
Basic TetraMAX Processes	1-2
Installing TetraMAX	1-4
Specifying the Location for TetraMAX Installation	1-4
Setting the Environment	1-4
Launching TetraMAX	1-5
Setup Command Files	1-6
Using Command Files	1-7
Using Variables	1-8
Tcl Mode	1-9
Native Mode	1-9
Running the TetraMAX GUI	1-9

Starting and Stopping the TetraMAX GUI	1-9
Interrupting a Long Process	1-10
Adjusting the Workspace Size	1-11
Saving Preferences	1-11
Setting Preferences	1-12
Saving GUI Preferences	1-13
Basic ATPG Flow	1-14
Getting Started for DFT Compiler Users	1-16
Design Flow Using DFT Compiler and TetraMAX	1-17
ATPG Design Flow	2-1
ATPG Design Flow Overview	2-2
Running the Basic ATPG Design Flow	2-2
Using Command Files	2-4
Preparing a Netlist	2-5
Configuring to Read a Netlist	2-6
Reading a Netlist	2-6
Reading Library Models	2-7
Preparing to Build the ATPG Model	2-8
Building the ATPG Model	2-8
Performing Design Rule Checking (DRC)	2-9
Specifying STIL Procedures	2-10
Specifying DRC Settings	2-11
Starting DRC	2-12
Reviewing the DRC Results	2-14
Understanding Rule Violations	2-15
Viewing DRC Violations in the GSV	2-16
Preparing for ATPG	2-18

Specifying General ATPG Settings	2-18
Specifying Fault Lists	2-20
Specifying Fault Models	2-23
Specifying the Pattern Source	2-25
Specifying the ATPG Mode	2-36
Running ATPG	2-39
Running ATPG in Basic Scan, Fast-Sequential, or Full-Sequential Mode	2-39
Using Automatic Mode to Generate Optimized Patterns	2-40
Quickly Estimating Test Coverage	2-42
Specifying a Test Coverage Target Value	2-44
Increasing ATPG Effort Over Multiple Passes	2-45
Multiple Session Test Pattern Generation	2-45
Compressing Patterns	2-48
Analyzing ATPG Output	2-51
Reviewing Test Coverage	2-57
Writing ATPG Patterns	2-59
Using TetraMAX II	3-1
About TetraMAX II	3-2
TetraMAX II and the Test Environment	3-2
Key Concepts of Multithreading	3-3
How TetraMAX II Uses Multithreading	3-4
Running TetraMAX II	3-5
Launching TetraMAX II	3-6
TetraMAX II Commands	3-7
Setting the Thread Count in TetraMAX II	3-8
Running ATPG in TetraMAX II	3-8
Two-Cycle Patterns for Stuck-At Faults	3-9

SDC Timing Exceptions for Stuck-At ATPG and Simulation	3-9
Running Simulation and Fault Simulation in TetraMAX II	3-10
Reporting Power in TetraMAX II	3-10
Running Diagnostics in TetraMAX II	3-11
TetraMAX II Messaging	3-11
TetraMAX II Command Option Support	3-13
run_atpg	3-13
run_fault_sim	3-13
run_simulation	3-13
set_atpg	3-14
set_delay	3-14
set_drc	3-15
TetraMAX II Limitations	3-15
Command Interface	4-1
TetraMAX GUI	4-2
Command Entry	4-3
Menu Bar	4-3
Command Toolbar and GSV Toolbar	4-3
Command-Line Window	4-4
Commands From a Command File	4-6
Command Logging	4-6
Transcript Window	4-7
Setting the Keyboard Focus	4-8
Using the Transcript Text	4-8
Selecting Text in the Transcript	4-9
Copying Text From the Transcript	4-9
Finding Commands and Messages in the Transcript	4-9

Saving or Printing the Transcript	4-10
Clearing the Transcript Window	4-10
Online Help	4-11
Text-Only Help	4-11
Browser-Based Help	4-12
Using the Graphical Schematic Viewer	5-1
Getting Started With the GSV	5-2
Using the SHOW Button to Start the GSV	5-2
Starting the GSV From a DRC Violation or Specific Fault	5-3
Navigating, Selecting, Hiding, and Finding Data	5-6
Expanding the Display From Net Connections	5-8
Hiding Buffers and Inverters in the GSV Schematic	5-9
ATPG Model Primitives	5-10
Displaying Symbols in Primitive or Design View	5-16
Displaying Instance Path Names	5-16
Displaying Pin Data	5-16
Using the Setup Dialog Box to Display Pin Data	5-17
Pin Data Types	5-18
Displaying Clock Cone Data	5-20
Displaying Clock Off Data	5-20
Displaying Constrain Values	5-21
Displaying Load Data	5-22
Displaying Shift Data	5-23
Displaying Test Setup Data	5-24
Displaying Pattern Data	5-24
Displaying Tie Data	5-26
Analyzing a Feedback Path	5-26

Checking Controllability and Observability	5-27
Using the Run Justification Dialog Box	5-27
Using the run_justification Command	5-28
Analyzing DRC Violations in the GSV	5-28
Troubleshooting a Scan Chain Blockage	5-29
Troubleshooting a Bidirectional Contention Problem	5-31
Analyzing Buses	5-33
BUS Contention Status	5-33
Understanding the Contention Checking Report	5-34
Reducing Aborted Bus and Wire Gates	5-34
Causes of Bus Contention	5-35
Analyzing ATPG Problems	5-36
Analyzing an AN Fault	5-37
Analyzing a UB Fault	5-38
Analyzing a NO Fault	5-39
Printing a Schematic to a File	5-40
Using the Hierarchy Browser	6-1
Launching the Hierarchy Browser	6-2
Basic Components of the Hierarchy Browser	6-4
Using the Hierarchy Pane	6-4
Viewing Data in the Instance Pane	6-6
Viewing Data in the Lib Cells/Tree Map Pane	6-9
Performing Fault Coverage Analysis	6-12
Understanding the Types of Coverage Data	6-12
Expanding the Design Hierarchy	6-13
Viewing Library Cell Data	6-16
Adjusting the Threshold Slider Bar	6-17

Identifying Fault Causes	6-18
Displaying Instance Information in the GSV	6-20
Exiting the Hierarchy Browser	6-22
Using the Simulation Waveform Viewer	7-1
Getting Started With the SWV	7-2
Supported Pin Data Types and Definitions	7-2
Invoking the SWV	7-4
Using the SWV Interface	7-6
Understanding the SWV Layout	7-6
Manipulating Signals	7-7
Identifying Signal Types in the Graphical Pane	7-10
Using the Time Scales	7-10
Using the Marker Header Area	7-11
Using the SWV With the GSV	7-13
Using the SWV Without the GSV	7-15
SWV Inputs and Outputs	7-16
Analyzing Violations	7-16
Using Tcl With TetraMAX	8-1
Converting TetraMAX Command Files to Tcl Mode	8-2
Converting a Collection to a List in Tcl Mode	8-2
Tcl Syntax and TetraMAX Commands	8-3
Specifying Lists in Tcl Mode	8-3
Abbreviating Commands and Options in Tcl Mode	8-4
Using Tcl Special Characters	8-5
Using the Result of a Tcl Command	8-6
Using Built-In Tcl Commands	8-6
TetraMAX Extensions and Restrictions in Tcl Mode	8-6

Redirecting Output in Tcl Mode	8-7
Using the redirect Command in Tcl Mode	8-7
Getting the Result of Redirected Tcl Commands	8-8
Using Redirection Operators in Tcl Mode	8-8
Using Command Aliases in Tcl Mode	8-9
Interrupting Tcl Commands	8-9
Using Command Files in Tcl Mode	8-10
Adding Comments	8-10
Controlling Command Processing When Errors Occur	8-10
Using a Setup Command File	8-11
Design Netlists and Library Models	9-1
Netlist Format Requirements	9-2
EDIF Netlist Requirements	9-2
Verilog Netlist Requirements	9-3
VHDL Netlist Requirements	9-3
About Reading a Netlist	9-4
Using Wildcards to Read Netlists	9-4
About Reading Library Models	9-5
Controlling Case-Sensitivity	9-5
Setting Parameters for Learning	9-6
About Building the ATPG Model	9-7
Processes That Occur When Building the ATPG Model	9-8
Flattening Optimization for Hierarchical Designs	9-9
Identifying Missing Modules	9-15
Removing Unused Logic	9-17
Using Black Box and Empty Box Models	9-20
Declaring Black Boxes and Empty Boxes	9-20

Behavior of RAM Black Boxes	9-22
Handling Duplicate Module Definitions	9-26
Memory Modeling	9-27
Memory Model Functions	9-27
Basic Memory Modeling Template	9-28
Initializing RAM and ROM Contents	9-28
Improving Test Coverage for RAMs	9-30
Creating Custom ATPG Models	9-31
Condensing ATPG Libraries	9-32
STIL Procedures	10-1
STIL Procedure File Guidelines	10-2
Creating a New STIL Procedure File	10-3
Declaring Primary Input Constraints	10-4
Declaring Clocks	10-5
Declaring Scan Chains and Scan Enables	10-6
Writing the SPF Template	10-7
Defining STIL Procedures	10-10
Defining Scan Chains	10-11
Defining the load_unload Procedure	10-11
Controlling Bidirectional Ports	10-12
Defining the Shift Procedure	10-14
Defining the test_setup Procedure	10-15
Predefined Signal Groups in STIL	10-17
Defining Basic Signal Timing	10-17
Defining Pulsed Ports	10-19
Selecting Strobed or Windowed Measures in STIL	10-21
Supporting Clock ON Patterns in STIL	10-22

Defining the End-of-Cycle Measure	10-24
Defining Capture Procedures in STIL	10-25
Defining Constrained Primary Inputs	10-26
Defining Equivalent Primary Inputs	10-27
Defining PO Masks	10-28
Defining System Capture Procedures	10-29
Creating Generic Capture Procedures	10-31
Defining Sequential Capture Procedures	10-39
Defining Reflective I/O Capture Procedures	10-41
Using the master_observe Procedure	10-43
Using the shadow_observe Procedure	10-44
Using the delay_capture_start Procedure	10-45
Using the delay_capture_end Procedure	10-46
Using the test_end Procedure	10-47
Scan Padding Behavior	10-47
Using the Condition Statement in STIL	10-49
Excluding Vectors From Simulation	10-50
Defining Internal Clocks for PLL Support	10-51
Specifying an On-Chip Clock Controller Inserted by DFT Compiler	10-54
Specifying Synchronized Multi Frequency Internal Clocks for an OCC Controller	10-55
ClockTiming Block Syntax	10-56
Timing and Clock Pulse Overlapping	10-57
Controlling Latency for the PLLStructures Block	10-58
ClockTiming Block Selection	10-59
ClockTiming Block Example	10-59
Specifying Internal Clocking Procedures	10-61
ClockConstraints and ClockTiming Block Syntax	10-61

Specifying the Clock Instruction Register	10-63
Specifying External Clocks	10-63
Example 1	10-64
Example 2	10-65
JTAG/TAP Controller Variations for the load_unload Procedure	10-66
Multiple Scan Groups	10-67
DFTMAX Compression with Serializer	10-73
Design Rule Checking	11-1
Understanding the DRC Process	11-2
Contention Analysis	11-2
BUS Contention Ability Checking	11-3
BUS Z State Ability Checking	11-3
Contention Prevention Checking	11-4
Simulation Contention Detection	11-4
ATPG Contention Prevention	11-4
Post-Capture Contention Checking	11-4
Settings for Contention Checking	11-5
Scan Chain Tracing	11-5
Clock Grouping	11-6
Reducing the Pattern Count Through Clock Grouping	11-6
Clock Grouping Analysis	11-7
Generating a Clock Group Report	11-9
Clock Grouping Limitations	11-9
Declaring Equivalent and Differential Input Ports	11-10
Cells With Asynchronous Set/Reset Inputs	11-11
Masking Input and Output Ports	11-12
Masking Scan Cell Inputs and Outputs	11-12

Previewing Potential Scan Cells	11-13
Using the Set Scan Ability Dialog Box	11-14
Using the <code>set_scan_ability</code> Command	11-14
Transparent Latches	11-14
Shadow Register Analysis	11-15
Feedback Paths Analysis	11-15
Procedure Simulation	11-15
Changing the Design Rule Severity	11-16
Using the Set Rules Dialog Box	11-16
Using the <code>set_rules</code> Command	11-16
Understanding the DRC Summary Report	11-17
Binary Image Files	11-20
Creating and Reading Image Files	11-21
Save/Restore in TEST Mode	11-24
Optimizing ATPG	12-1
Using ATPG Constraints	12-2
Using the Random Decision Option	12-3
Obtaining Target Test Coverage Using Fewer Patterns	12-4
Maximizing Test Coverage Using Fewer Patterns	12-4
Improving Test Coverage With Test Points	12-4
Test Points Analysis Options	12-5
Running the Test Points Analysis Flow	12-5
Limitation	12-6
Optimizing Basic Scan Patterns	12-6
Limiting the Number of Patterns	12-7
Limiting the Number of Aborted Decisions	12-7
Creating Test Patterns for Diagnosing Scan Chain Failures	12-8

Understanding DFTMAX Unload Modes and Chain Diagnosis Patterns	12-9
Generating Pattern Sets	12-10
Performing Scan Chain Diagnostics	12-11
Creating End-of-Cycle Measures in ATPG Patterns	12-12
Drawbacks of Using End-of-Cycle Measures	12-13
Requirements Needed to Produce End-of-Cycle Measures	12-13
Deleting Top-Level Ports From Output Patterns	12-14
Detecting Faults Multiple Times Using N-Detect	12-14
WGL Pattern Generation Options	12-15
How Do I Create LSI-Compatible WGL Patterns?	12-16
How Do I Create NEC-Compatible WGL Patterns?	12-17
WGL Scan Chain Padding	12-18
WGL Scan Chain Definitions	12-19
Macro Usage in WGL	12-20
Grouping Bidirectional Port Data in WGL	12-22
Controlling Port Data Order in WGL	12-23
Specifying Windowed Measures in WGL	12-24
Delayed Input Force Timing and Force Prior in WGL	12-24
Balancing Vector and Scan Statements in WGL	12-25
Mapping Bidirectional Ports Within Vector Statements in WGL	12-27
Mapping Bidirectional Ports Within Scan Statements in WGL	12-30
Adjusting Pattern Data for Serial Versus Parallel Interpretation in WGL	12-31
Selecting Scan Chain Inversion Reference in WGL	12-32
Effect of CELLDEFINE in WGL	12-34
Ambiguity of the Master Cell in WGL	12-35
Running Multicore ATPG	12-35
Comparing Multicore ATPG and Distributed ATPG	12-36

Invoking Multicore ATPG	12-36
Multicore Interrupt Handling	12-37
Understanding the Processes Summary Report	12-37
Multicore Limitations	12-38
Running Logic Simulation	12-38
Comparing Simulated and Expected Values	12-39
Patterns in the Simulation Buffer	12-40
Sequential Simulation Data	12-40
Single-Point Failure Simulation	12-40
GSV Display of a Single-Point Failure	12-41
Data Volume and Test Application Time Reduction Calculations	12-41
Test Data Volume Calculations	12-42
Test Application Time Calculations	12-42
Fault Lists and Faults	13-1
Working with Fault Lists	13-2
Using Fault List Files	13-3
Collapsed and Uncollapsed Fault Lists	13-3
Random Fault Sampling	13-4
Fault Dictionary	13-5
Fault Categories and Classes	13-5
Fault Class Hierarchy	13-5
DT (Detected) = DR + DS + DI + D2 + TP	13-6
PT (Possibly Detected) = AP + NP + P0 + P1	13-7
UD (Undetectable) = UU + UO + UT + UB + UR	13-7
AU (ATPG Untestable) = AN	13-8
ND (Not Detected) = NC + NO	13-9
Fault Summary Reports	13-9

Fault Summary Report Examples	13-9
Test Coverage	13-11
Fault Coverage	13-11
ATPG Effectiveness	13-12
Using Clock Domain-Based Faults	13-12
Using Signals That Conflict With Reserved Keywords	13-16
Finding Particular Untested Faults Per Clock Domain	13-16
Fault Simulation	14-1
Fault Simulation Design Flow	14-2
Preparing Functional Test Patterns for Fault Simulation	14-2
Pattern Compliance with ATE	14-3
Checking Patterns for Timing Insensitivity	14-3
Preparing Your Design for Fault Simulation	14-4
Preprocessing the Netlist	14-5
Reading the Design and Libraries	14-5
Building the ATPG Design Model	14-5
Declaring Clocks	14-5
Running DRC	14-6
Reading Functional Test Patterns	14-7
Using the Set Patterns Dialog Box	14-8
Using the set_patterns Command	14-8
Specifying Strobes for VCDE Pattern Input	14-8
Initializing the Fault List	14-11
Using the Add Faults Dialog Box	14-11
Using the add_faults Command	14-11
Performing Good Machine Simulation	14-11
Performing Fault Simulation	14-12

Writing the Fault List	14-13
Combining ATPG and Functional Test Patterns	14-13
Creating Independent Functional and ATPG Patterns	14-14
Creating ATPG Patterns After Functional Patterns	14-14
Creating Functional Patterns After ATPG Patterns	14-15
Running Multicore Simulation	14-16
Invoking Multicore Simulation	14-17
Interrupt Handling	14-17
Understanding the Processes Summary Report	14-17
Resimulating ATPG Patterns	14-18
Limitations	14-19
Per-Cycle Pattern Masking	14-19
Flow Options	14-19
Masks File	14-20
Running the Flow	14-20
Limitations	14-22
On-Chip Clocking Support	15-1
OCC Background	15-2
OCC Definitions, Supported Flows, Supported Patterns	15-2
OCC Limitations	15-3
DFT Compiler to TetraMAX Flow	15-5
OCC Support in TetraMAX	15-8
Design Set Up	15-8
OCC Scan ATPG Flow	15-9
Waveform and Capture Cycle Example	15-9
Using Synchronized Multi Frequency Internal Clocks	15-9
Using Internal Clocking Procedures	15-15

OCC-Specific DRC Rules	15-20
Diagnosing Manufacturing Test Failures	16-1
Diagnostics Flow Overview	16-2
Running the Diagnostics Flow	16-3
Writing and Reading Binary Image Files	16-4
Reading Pattern Files	16-4
Reading Patterns	16-5
Reading Multiple Pattern Files	16-5
Translating DFTMAX Compressed Patterns Into Normal Scan Patterns	16-6
Failure Data Files	16-7
Pattern-Based Failure Data File	16-7
Cycle-Based Failure Data File	16-10
Failure Data File Extensions	16-11
Adding Header Information to a Failure Data File	16-12
Failure Data File Limitations	16-18
Class-Based Diagnostics Reporting	16-19
Filtering Candidates	16-19
Filtering Bridge Candidates	16-20
Resetting User-Specified Filters	16-20
Reporting Detailed Candidate Information	16-20
Example Flow	16-22
Understanding the Class-Based Diagnostics Report	16-22
Class-Based Cell-Aware Diagnostics	16-24
Fault-Based Diagnostics Reporting	16-24
Failure Mapping Report for DFTMAX Patterns	16-31
Composite Fault Model Data Report	16-32
Parallel Diagnostics	16-34

Specifying Parallel Diagnostics	16-35
Converting Serial Scripts to Parallel Scripts	16-36
Using Split Datalogs to Perform Parallel Diagnostics for Split Patterns	16-36
Diagnosis Log Files	16-37
Parallel Diagnostics Limitations	16-40
Using Physical Data for Diagnostics	17-1
Physical Diagnostics Flow Overview	17-2
Using TetraMAX to Create and Validate the PHDS Database	17-3
Creating and Validating the PHDS Database	17-3
Creating and Validating the PHDS Database in Separate Runs	17-5
Reading a PHDS Database	17-5
Starting and Stopping the DAP Server Process	17-6
Setting Up a Connection to the PHDS Database	17-8
Name Matching Using a PHDS Database	17-9
Name Matching Overview	17-9
Understanding the Name Matching Coverage Report	17-10
Reporting the Name Matching Coverage	17-11
Using Name Matching Results for Diagnostics	17-12
Setting Up and Running Physical Diagnostics	17-13
Running Physical Diagnostics	17-13
Static Subnet Extraction Using a PHDS Database	17-14
Reporting Physical Subnet ID Data	17-16
Understanding Physical Subnet ID Data	17-16
Writing Physical Data for Yield Explorer	17-17
Power Aware ATPG	18-1
Input Data Requirements	18-2
Setting a Power Budget	18-2

Preparing Your Design	18-2
Reporting Clock-Gating Cells	18-3
Setting a Strict Power Budget	18-3
Setting Toggle Weights	18-4
Running Power Aware ATPG	18-4
Applying Quiet Chain Test Patterns	18-5
Testing with Asynchronous Primary Inputs	18-6
Power Reporting By Clock Domain	18-6
Setting a Capture Budget for Individual Clocks	18-10
Retention Cell Testing	18-11
Creating the chain_capture Procedure	18-11
Identifying Retention Cells	18-12
Performing Test DRC	18-12
Generating the Patterns	18-12
Running Fault Simulation	18-13
Limitations	18-13
Power Aware ATPG Limitations	18-13
Bridging Fault ATPG	19-1
Detecting Bridging Faults	19-2
How Bridging Faults are Defined	19-2
Bridge Locations	19-2
Strength-Based Patterns	19-3
Bridging Fault Flows	19-4
Bridging Faults and the Overall TetraMAX Flow	19-4
Bridging Fault Flow in TetraMAX	19-4
Using StarRC to Generate a Bridge Fault List	19-8
TCAD Characterization	19-8

Extracting Capacitance	19-9
Running TetraMAX	19-11
Bridging Fault Model Limitations	19-12
Running the Dynamic Bridging Fault ATPG Flow	19-12
Understanding the Dynamic Bridging Fault Model	19-12
Preparing to Run Dynamic Bridging Fault ATPG	19-13
Fault Simulation	19-14
Running ATPG	19-15
Analyzing Fault Detection	19-15
Example Script	19-16
Limitations	19-16
Cell-Aware Test	20-1
Cell-Aware Test Flow	20-2
Targeting Physical Cell Defects	20-3
Understanding Cell Test Models	20-5
Generating Cell Test Models	20-6
Running Cell-Aware ATPG	20-7
Cell-Aware Diagnostics	20-9
Identifying a Defect	20-10
Identifying Defects within a Cell	20-10
Running Diagnostics	20-11
Path Delay Fault and Hold Time Testing	21-1
Path Delay Fault Theory	21-1
Path Delay Fault Term Definitions	21-2
Models for Manufacturing Tests	21-4
Models for Characterization Tests	21-5
Testing I/O Paths	21-6

Path Delay Test Patterns	21-7
Path Delay Testing Flow	21-8
Obtaining Delay Paths	21-10
Hold Time ATPG Test Flow	21-11
Generating Path Delay Tests	21-13
Flow for Generating Path Delay Tests	21-14
Using set_delay Options	21-14
Reading and Reporting Path Lists	21-15
Analyzing Path Rule Violations	21-15
Viewing Delay Paths	21-15
Path Delay ATPG Options	21-15
Internal Loopback and False/Multicycle Paths	21-15
Creating At-Speed WaveformTables	21-16
Maintaining At-Speed Waveform Table Information	21-19
MUXClock Support for Path Delay Patterns	21-19
Handling Untested Paths	21-22
Understanding False Paths	21-22
Understanding Untestable Paths	21-23
Reporting Untestable Paths	21-23
Analyzing Untestable Faults	21-24
Quiescence Test Pattern Generation	22-1
Why Do IDDQ Testing?	22-2
CMOS Circuit Characteristics	22-2
IDDQ Testing Methodology	22-3
Types of Defects Detected	22-4
Number of IDDQ Strobes	22-5
About IDDQ Pattern Generation	22-5

IDDQ Limitation	22-7
Fault Models	22-7
DRC Rule Violations	22-7
Generating IDDQ Test Patterns	22-9
IDDQ Test Pattern Generation Flow	22-9
Using the iddq_capture Procedure	22-10
Off-Chip IDDQ Monitor Support	22-10
Using IDDQ Commands	22-14
Using the set_faults Command	22-15
Using the set_iddq Command	22-15
Using the add_atpg_constraints Command	22-16
IDDQ Bridging	22-16
Design Principles for IDDQ Testability	22-17
I/O Pads	22-17
Buses	22-18
RAMs and Analog Blocks	22-18
Free-Running Oscillators	22-18
Circuit Design	22-18
Power and Ground	22-19
Models With Switch/FET Primitives	22-19
Connections	22-19
IDDQ Design-for-Test Rule Summary	22-20
Transition-Delay Fault ATPG	23-1
Using the Transition-Delay Fault Model	23-2
Transition-Delay Fault ATPG Flow	23-2
Transition-Delay Fault ATPG Timing Modes	23-3
STIL Protocol for Transition Faults	23-8

Creating Transition Fault Waveform Tables	23-8
DRC for Transition Faults	23-10
Limitations of Transition-Delay Fault ATPG	23-10
Specifying Transition-Delay Faults	23-11
Selecting the Fault Model	23-11
Adding Faults to the Fault List	23-12
Reading a Fault List File	23-12
Pattern Generation for Transition-Delay Faults	23-12
Using the set_atpg Command	23-13
Using the set_delay Command	23-13
Using the run_atpg Command	23-14
Pattern Compression for Transition Faults	23-14
Using the report_faults Command	23-14
Using the write_faults Command	23-15
Pattern Formatting for Transition-Delay Faults	23-15
MUXClock Support for Transition Patterns	23-17
Specifying Timing Exceptions From an SDC File	23-17
Reading an SDC File	23-17
Interpreting an SDC File	23-18
How TetraMAX Interprets SDC File Commands	23-19
Controlling Clock Timing, ATPG, and Timing Exceptions for SDC	23-19
Reporting SDC Results	23-20
Slack-Based Transition Fault Testing	23-20
Basic Usage Flow	23-21
Special Elements of Slack-Based Transition Fault Testing	23-24
Limitations	23-26
Running Distributed ATPG	24-1

Checking Your Environment for Distributed Processing	24-2
Machine Access and Setup for Distributed ATPG	24-3
Preparing to Run Distributed Processing	24-4
Setting Up the Distributed Environment	24-5
Setting Up the Distributed Environment With Load Sharing	24-6
Verifying Your Environment	24-7
Remote Shell Considerations	24-7
Tuning Your .cshrc File	24-8
Checking the Load Sharing Setup	24-8
Starting Distributed ATPG	24-8
Saving Results	24-12
Distributed Processor Log Files	24-12
Starting Distributed Fault Simulation	24-13
Debugging Distributed ATPG Issues	24-15
Distributed ATPG Limitations	24-17
Persistent Fault Model Support	25-1
Persistent Fault Model Overview	25-2
Persistent Fault Model Operations	25-2
Switching Fault Models	25-3
Working With Internal Pattern Sets	25-3
Manipulating Fault Lists	25-3
Reporting Persistent Fault Models	25-6
Direct Fault Crediting	25-7
Example Commands Used in Persistent Fault Model Flow	25-9
Using TetraMAX and DFTMAX Ultra Compression	26-1
Generating Patterns for DFTMAX Ultra Designs	26-2
Pattern Types Required by DFTMAX Ultra	26-2

Script Example for Generating Patterns for DFTMAX Ultra	26-3
Manipulating Patterns for DFTMAX Ultra	26-5
Controlling the Peak and Average Power During Shifting	26-5
Increasing the Maximum Shift Length of Patterns	26-5
Optimizing Padding Patterns	26-6
Removing and Reordering Patterns	26-7
High Resolution Pattern Flow for DFTMAX Ultra Chain Diagnostics	26-8
Identifying Defective Chains	26-8
Generating High Resolution Patterns	26-8
Rerunning Diagnostics Using the High Resolution Patterns	26-9
Flow Example	26-9
Test Validation and VCS Simulation for DFTMAX Ultra Designs	26-10
Limitations for Using DFTMAX Ultra	26-10
Troubleshooting	27-1
Reporting Port Names	27-2
Reviewing a Module Representation	27-2
Rerunning Design Rule Checking	27-4
Troubleshooting Netlists	27-4
Troubleshooting STIL Procedures	27-5
Opening the STIL Procedure File	27-5
STIL load_unload Procedure	27-6
STIL Shift Procedure	27-6
STIL test_setup Macro	27-7
Correcting DRC Violations by Changing the Design	27-8
Analyzing the Cause of Low Test Coverage	27-8
Where Are the Faults Located?	27-8
Why Are the Faults Untestable or Difficult to Test?	27-9

Using Justification	27-11
Completing an Aborted Bus Analysis	27-11
Test Concepts	A-1
Why Perform Manufacturing Testing?	A-2
Understanding Fault Models	A-2
Stuck-At Fault Models	A-2
Detecting Stuck-At Faults	A-3
Using Fault Models to Determine Test Coverage	A-4
IDDQ Fault Model	A-5
Fault Simulation	A-5
Automatic Test Pattern Generation	A-6
Translation for the Manufacturing Test Environment	A-6
What Is Internal Scan?	A-7
Example	A-7
Applying Test Patterns	A-8
Scan Design Requirements	A-9
Full-Scan Design	A-10
Partial-Scan ATPG Design	A-10
What Is Boundary Scan?	A-11
ATPG Design Guidelines	B-1
ATPG Design Guidelines	B-2
Internally Generated Pulsed Signals	B-2
Clock Control	B-6
Pulsed Signals to Sequential Devices	B-9
Multidriver Nets	B-10
Bidirectional Port Controls	B-12
Clocking Scan Chains: Clock Sources, Trees, and Edges	B-13

Protection of RAMs During Scan Shifting	B-19
RAM and ROM Controllability During ATPG	B-20
Pulsed Signal to RAMs and ROMs	B-22
Bus Keepers	B-23
Bus Keepers	B-26
Checklists for Quick Reference	B-28
ATPG Design Guideline Checklist	B-28
Ports for Test I/O Checklist	B-29
Importing Designs From DFT Compiler	C-1
Utilities	D-1
Ltran Translation Utility	D-2
Ltran in the Shell Mode	D-2
FTDL, TDL91, and TSTL2 Configuration Files	D-3
Understanding the Configuration File	D-4
Configuration File Syntax	D-7
Generating PrimeTime Constraints	D-10
Input Requirements	D-11
Starting the Tcl Command Parser Mode	D-11
Setting Up TetraMAX	D-11
Making Adjustments for OCC Controllers	D-15
Performing an Analysis for Each Mode	D-15
Implementation	D-17
Converting Timing Violations Into Timing Exceptions	D-19
STIL Language Support	E-1
STIL Overview	E-2
IEEE Std. 1450-1999	E-2
IEEE Std. 1450.1 Design Extensions to STIL	E-3

TetraMAX ATPG and STIL	E-3
STIL Conventions in TetraMAX	E-3
Use of STIL Procedures	E-4
Context of Partial Signal Sets in Procedure Definitions	E-4
Use of STIL SignalGroups	E-5
WaveFormCharacter Interpretation	E-6
IEEE Std. 1450.1 Extensions Used in TetraMAX	E-7
Vector Data Mapping Using \m	E-8
Vector Data Mapping Using \j	E-10
Signal Constraints Using Fixed and Equivalent	E-13
ScanStructures Block	E-13
Elements of STIL Not Used by TetraMAX	E-14
TetraMAX STIL Output	E-14
TetraMAX STIL Input	E-16
STIL99 Versus STIL	F-1
Defective Chain Masking for DFTMAX	G-1
Introduction	G-2
Running the Flow	G-2
Placing Constraints on the Defective Chain	G-2
Generating Patterns	G-3
Regenerating Patterns	G-3
Examples	G-4
Limitation	G-6

Preface

This preface is comprised of the following sections:

- [About This Manual](#)
 - [Customer Support](#)
-

About This User Guide

The TetraMAX ATPG and TetraMAX II User Guide describes the usage and methodology for TetraMAX ATPG and TetraMAX II. Both products are used to check testability design rules and to automatically generate manufacturing test vectors for a logic design.

This manual provides background material on design-for-test (DFT) concepts, especially test terminology and scan design techniques. You can obtain more information on TetraMAX ATPG features and commands by accessing TetraMAX Help.

Audience

This manual is intended for design engineers who have ASIC design experience and some exposure to testability concepts and strategies.

This manual is also useful for DFT engineers who incorporate the test vectors produced by TetraMAX ATPG and TetraMAX II into test programs for a particular tester or who work with DFT netlists. Engineers involved in the testing and diagnostics of manufactured parts will also find this manual useful.

Related Publications

For additional information about TetraMAX ATPG, see Documentation on the Web, which is available through SolvNet® at the following address:

<https://solvnet.synopsys.com/DocsOnWeb>

You might also want to read the documentation for the following related Synopsys products: DFTMAX™ and Design Compiler®.

Release Notes

Information about new features, enhancements, changes, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the TetraMAX ATPG Release Notes on the SolvNet site.

To see the TetraMAX ATPG Release Notes:

1. Go to the SolvNet Download Center located at the following address:
<https://solvnet.synopsys.com/DownloadCenter>
2. Select TetraMAX ATPG, and then select a release in the list that appears.

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
Courier	Indicates command syntax.
<i>Courier italic</i>	Indicates a user-defined value in Synopsys syntax, such as <i>object_name</i> . (A user-defined value that is not Synopsys syntax, such as a user-defined value in a Verilog or VHDL statement, is indicated by regular text font italic.)
Courier bold	Indicates user input—text you type verbatim—in Synopsys syntax and examples. (User input that is not Synopsys syntax, such as a user name or password you enter in a GUI, is indicated by regular text font bold.)
[]	Denotes optional parameters, such as pin1 [pin2 ... pinN]
	Indicates a choice among alternatives, such as low medium high. (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
—	Connects terms that are read as a single term by the system, such as <code>set_environment_viewer</code>
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

Accessing SolvNet

The SolvNet site includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. The SolvNet site also gives you access to a

wide range of Synopsys online services including software downloads, documentation on the Web, and technical support.

To access the SolvNet site, go to the following address:

<https://solvnet.synopsys.com>

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.

If you need help using the SolvNet site, click HELP in the top-right menu bar.

Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a support case to your local support center online by signing in to the SolvNet site at <http://solvnet.synopsys.com>, clicking Support, and then clicking “Open a Support Case.”
- Send an e-mail message to your local support center.
 - E-mail support_center@synopsys.com from within North America.
 - Find other local support center e-mail addresses at
<http://www.synopsys.com/Support/GlobalSupportCenters/Pages>
- Telephone your local support center.
 - Call (800) 245-8005 from within the continental United States.
 - Call (650) 584-4200 from Canada.
 - Find other local support center telephone numbers at:
<http://www.synopsys.com/Support/GlobalSupportCenter/Pages>

1

Getting Started

The following sections describe how to get started with TetraMAX ATPG:

- [TetraMAX ATPG and TetraMAX II](#)
- [Basic TetraMAX Processes](#)
- [Basic ATPG Flow](#)
- [Getting Started For DFT Compiler Users](#)
- [Synopsys Reference Methodology](#)

TetraMAX ATPG and TetraMAX II

This documentation describes the processes and features associated with the TetraMAX ATPG and TetraMAX II products:

- TetraMAX ATPG, Synopsys' original ATPG product, automatically generates high quality manufacturing test patterns and is optimized for a wide range of test methodologies. It is tightly integrated with DFTMAX and DFTMAX Ultra compression. TetraMAX ATPG enables you to quickly and efficiently create test patterns for the most complex design.
- TetraMAX II is Synopsys' next generation ATPG solution. It uses advanced compression, diagnosis, and fault simulation engines that are fast, memory-efficient, and optimized for fine-grained multithreading of ATPG and diagnosis processes across multiple cores.

Both TetraMAX ATPG and TetraMAX II share many of the same flows, commands, command options. However, there are some key differences that are explained in detail in "TetraMAX II." In summary, the key differences are:

- You launch TetraMAX II using the `tmax2` executable command.
- You do not need to specify the simulation mode, since it is set by default
- The `set_atpg` and `set_simulation` commands have `-num_threads` option that sets the thread count to adjust the TetraMAX II multithread settings. The settings for both these options must match.
- When TetraMAX II is running, it replaces TetraMAX ATPG when you specify the `report_power`, `run_atpg`, `run_diagnosis`, `run_fault_sim`, or `run_simulation` commands.

Note that most references to "TetraMAX ATPG" in this documentation apply to both TetraMAX ATPG and TetraMAX II, unless specifically noted.

Basic TetraMAX Processes

The following sections describe the basic TetraMAX ATPG processes:

- [Installing TetraMAX](#)
- [Setting the Environment](#)
- [Launching TetraMAX](#)
- [Setup Command Files](#)
- [Command Files](#)

- [Variables](#)
- [Running the TetraMAX GUI](#)

Installing TetraMAX

To obtain the TetraMAX ATPG installation files, download them from Synopsys using electronic software transfer (EST) or File Transfer Protocol (FTP).

TetraMAX ATPG can be installed as a standalone product or over an existing Synopsys product installation (an “overlay” installation). An overlay installation shares certain support and licensing files with other Synopsys tools, whereas a standalone installation has its own independent set of support files. You specify the type of installation you want when you install the product.

An environment variable called **SYNOPSYS** specifies the location for the TetraMAX ATPG installation. You need to explicitly set this environment variable.

Complete installation instructions are provided in the Installation Guide that comes with each release of TetraMAX ATPG .

Specifying the Location for TetraMAX Installation

TetraMAX ATPG requires the **SYNOPSYS** environment variable, a variable typically used with all Synopsys products. For backward compatibility, **SYNOPSYS_TMAX** can be used instead of the **SYNOPSYS** variable. However, TetraMAX ATPG looks for **SYNOPSYS** and if not found, then looks for **SYNOPSYS_TMAX**. If **SYNOPSYS_TMAX** is found, then it overrides **SYNOPSYS** and issues a warning that there are differences between them.

The conditions and rules are as follows:

- **SYNOPSYS** is set and **SYNOPSYS_TMAX** is not set. This is the preferred and recommended condition.
- **SYNOPSYS_TMAX** is set and **SYNOPSYS** is not set. The tool will set **SYNOPSYS** using the value of **SYNOPSYS_TMAX** and continue.
- Both **SYNOPSYS** and **SYNOPSYS_TMAX** are set. **SYNOPSYS_TMAX** will take precedence and **SYNOPSYS** is set to match before invoking the kernel.
- Both **SYNOPSYS** and **SYNOPSYS_TMAX** are set, and are of different values, then a warning message is generated similar to the following:

```
WARNING: $SYNOPSYS and $SYNOPSYS_TMAX are set differently,  
using $SYNOPSYS_TMAX  
WARNING: SYNOPSYS_TMAX = /mount/groucho/joeuser/tmax  
WARNING: SYNOPSYS = /mount/harpo/production/synopsys  
WARNING: Use of SYNOPSYS_TMAX is outdated and support for this  
will be removed in a future release. Use SYNOPSYS instead.
```

Setting the Environment

Before invoking TetraMAX ATPG , you need to set your environment. Make sure your **PATH** environment variable includes the path to the Synopsys tools, which is typically **\$SYNOPSYS/bin**. Also set the license file using the **LM_LICENSE_FILE** setting. For example:
`setenv SYNOPSYS /synopsys/m_branch/`

```
set path =($SYNOPSYS/bin $path)
setenv LM_LICENSE_FILE synopsys_licenses/my_license.lic:$LM_
LICENSE_FILE
```

You can optionally specify the TMAX_SHELL variable to avoid the need to constantly specify the -shell switch with the `tmax` command (for TetraMAX ATPG) or the `tmax2` command (for TetraMAX II).

```
% setenv TMAX_SHELL
```

If the TMAX_SHELL environment variable is defined, you can override it by invoking:

```
% tmax -gui // overrides TMAX_SHELL=1
```

Launching TetraMAX

You can launch TetraMAX in either shell mode or using the TetraMAX GUI:

- To launch TetraMAX in shell mode, specify the `-shell` option with the `tmax` command (for TetraMAX ATPG) or the `tmax2` command (for TetraMAX II).

```
tmax my_command_file -shell
```

or

```
tmax2 my_command_file -shell
```

- To launch the TetraMAX GUI specify the `tmax` or `tmax2` command without using the `-shell` option.

```
tmax my_command_file
```

or

```
tmax2 my_command_file
```

The following table summarizes the various methods you can use to invoke TetraMAX ATPG.

Table TetraMAX Invocation Methods

Invoke With ¹	You Get	Process List
<code>tmax</code>	64-bit kernel plus GUI	<code>tmax</code> , <code>tmaxgui</code>
<code>tmax2</code>	64-bit kernel plus GUI	<code>tmax2</code> , <code>tmaxgui</code>
<code>tmax -shell</code>	64-bit kernel	<code>tmax</code>
<code>tmax2 -shell</code>	64-bit kernel	<code>tmax2</code>
<code>tmax -man</code>	Online help	<code>tmax</code> , HTML browser
<code>tmax2 -man</code>	Online help	<code>tmax2</code> , HTML browser

Invoke With¹	You Get	Process List
tmax -help	List of options	
tmax2 -help	List of options	

¹ - The order of switches is not important.

² - The tmax process or tmax2 process invokes the shell. The CPU is idle after the kernel launches, but remains open so that the kernel has a transcript window in which to display output.

Setup Command Files

A setup command file is similar to a [command file](#), except that it is executed automatically when TetraMAX starts. You can include any commands in a setup command file that are used in a command file.

TetraMAX ATPG includes a tmaxtcl.rc setup file (for Tcl mode) or a tmax.rc setup file (for legacy mode).

Upon startup, TetraMAX ATPG automatically executes command files from multiple locations based upon the following order:

1. \$SYNOPSYS/admin/setup/tmax.rc
2. \$TMAXRC, if defined (intended for use by ASIC vendors)
3. \$HOME/.tmaxrc or \$HOME/.tmaxtclrc
4. tmaxrc, tmax.rc, .tmaxtclrc, or tmaxtcl.rc in the current working directory

Setup command files are executed before any command files specified in the TetraMAX ATPG invocation line. You can specify a command file using any of the following techniques:

```
% tmax command_file [other_args]...
% tmax [other_args]... command_file
```

Within a script as a "here" document:

```
#!/bin/sh
tmax [other_args] -shell <<!
source command_file
exit -force
!
% tmax [other_args]
BUILD> source command_file
```

By default, commands in a command setup file are not echoed to the transcript. To see the commands as they are executed, place a `set_messages -display` command at the beginning of the command setup file.

To invoke TetraMAX ATPG without using a setup command file, use the `-nostartup` switch:

```
% tmax -nostartup
```

Using Command Files

A command file is a simple ASCII text file containing any command accepted by TetraMAX. You can place multiple commands into a file and execute them sequentially by running the name of the command file using the `source command_file_name` command.

There are several ways you can specify a command file:

- `% tmax command_file [other_args]...`
- `% tmax [other_args]... command_file`
- Within a script as a “here” document:

```
#!/bin/sh
tmax [other_args] -shell <<!
source command_file
exit -force
!
```
- `% tmax [other_args]`
`BUILD-T> source command_file`

You can use the `abort`, `noabort`, or `exit` options of the `set_commands` command to specify the command file execution to stop or continue when TetraMAX ATPG encounters an error.

You can specify whether comments and command output will appear in the transcript or log files using the `set_messages` command.

You can reference one command file from within another by nesting `source command_file_name` commands.

You can specify a command file to be executed when you invoke TetraMAX ATPG on a UNIX or NT machine running a command shell. The syntax is:

```
% tmax command_file
% tmax command_file -shell
```

Use of the optional `-shell` argument runs the non-GUI form of TetraMAX ATPG. This might be more convenient if no user interaction or interactive debugging is expected, or when you are running TetraMAX ATPG from a remote telnet session or other environment where a graphic display is not available.

Note: You can condition TetraMAX ATPG to pause by invoking it with the `-shell` argument followed by a command file specification; for example:

```
% tmax -shell
% source command_file
```

For more information on command files, see "[Using Command Files](#)."

Batch Files

Note: When you operate TetraMAX ATPG in batch mode using command files, you should use the `set_commands noabort` command at the beginning and the `exit -force` command

at the end of each command file. Then you can safely use commands such as the following at the shell prompt:

```
% tmax batch_command_file -shell &
```

Without these commands at the beginning and end of a command file, the situation could arise where the tool encounters an error, but there is no way to make it exit.

Using Variables

TetraMAX ATPG supports limited use of variables in commands and [command files](#). Variables are accepted only as the prefix (or first) string of a file path name argument. No other arguments or options of commands support the use of variables.

A variable is recognized by the leading dollar sign (\$), followed by the variable name, as shown in the following examples:

```
set_messages log $specLOG -replace  
read_netlist $LIBDIR/cmos/verilog/*.v  
write_patterns $tmp/testbench.v -format verilog -replace
```

TetraMAX supports two types of variables:

- UNIX environment variables

These variables are typically defined using the `setenv` command, or the `set` and `export` commands.

- User-defined environment variables

These variables are defined in the TetraMAX invocation line as shown in the following example:

```
% tmax -env specLOG save/tmax.log -env tmp /tmp
```

You can define multiple variables by repeating the `-env` argument of the `tmax` command. To view the current setting of any user-defined environment variable, specify the `report_settings` command. A variable defined with `-env` will override any existing environment variable with the same name.

TetraMAX ATPG recognizes UNIX environment variables specified within a command. You can also set variables in a script using the `set` or `setenv` commands. The `set` command can be used for most commands; the `setenv` command makes the variable available for programs called from the TetraMAX shell.

For example:

```
setenv LTRAN_SHELL 1  
setenv SNPSLMD_QUEUE
```

There are several differences in behavior between Tcl mode and native mode when using variables.

Tcl Mode

In Tcl mode, you can use the `getenv` or `get_unix_variable` commands to return the value of the variable. You can also use the `$env (VAR)` syntax.

Some usage examples are as follows:

```
set_messages -log [get_env LOG_DIR]/tmax.log  
report_rules -fail > [getenv RPTS]/violations.rpt  
set_atpg -num_processes [get_env cpu]  
source $env(SYNOPSYS)/auxx/syn/tmax/tmax2pt.tcl
```

For more information on Tcl mode, see "[Using Tcl With TetraMAX](#)."

Native Mode

In native mode, you can use variables only at the beginning of the path file names, as shown in the following examples::

```
set messages log $specLOG -replace  
read netlist $LIBDIR/cmos/verilog/*.v  
write patterns $tmp/testbench.v -format verilog -replace
```

Running the TetraMAX GUI

The following sections describe how to set up and run the TetraMAX GUI:

- [Starting and Stopping the TetraMAX GUI](#)
 - [Interrupting a Long Process](#)
 - [Setting Preferences](#)
 - [Saving GUI Preferences](#)
-

Starting and Stopping the TetraMAX GUI

If you are in shell mode, you can display the TetraMAX GUI in its current state by entering the `gui_start` command:

```
BUILD-T> gui_start
```

This command switches the context to listen-only in the GUI console.

After you start the TetraMAX GUI (using the `gui_start` command), enter the `gui_stop` command to exit the GUI:

```
BUILD-T> gui_stop
```

This command stops the TetraMAX GUI session and reverts to the TetraMAX shell command prompt. If you did not use the `gui_start` command to start the GUI, the `gui_stop` command exits the TetraMAX application. You can also use the `gui_stop` command from the pull-down menu: File > Exit GUI. If you use the `gui_stop` command before invoking TetraMAX ATPG using the `gui_start` command to start the GUI, the `gui_stop` command exits the TetraMAX application.

Interrupting a Long Process

While TetraMAX ATPG is processing commands, the Submit button in the TetraMAX window changes to a Stop button. To stop a process, click the Stop button. (Depending on the type of platform you are using, Control-c and Control-Break may also perform the Stop function.)

Note: The Stop button usually works within a few seconds. However, interrupting a large file I/O process might take longer.

You can use the Stop function to interrupt the following types of processes:

- Reading netlists
- Running ATPG, simulation, or fault simulation
- Reporting faults to the screen
- Running design rule checking (DRC)
- Building the design-level ATPG model
- Learning following an ATPG build
- Compressing patterns
- Writing patterns to a file
- Reading or writing fault lists from files
- Reporting scan cells
- Executing command files

When you use the Stop function to stop execution of a command file, TetraMAX ATPG normally stops execution of the entire file, unless the file contains the following line:

```
set_commands noabort
```

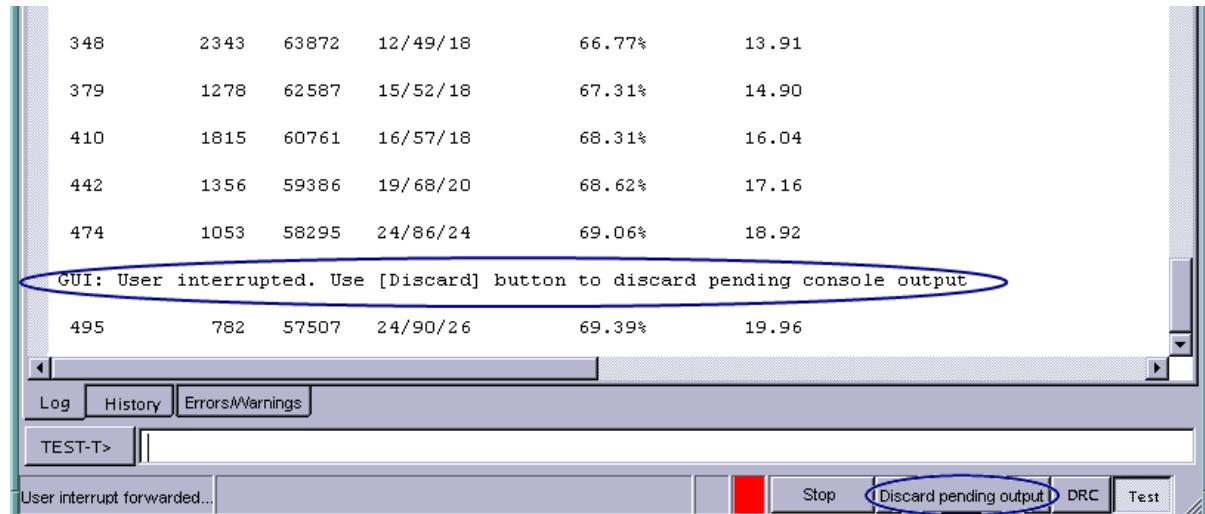
In that case, TetraMAX ATPG stops execution of only the current command in the file and continues execution of any commands following the stopped command. The `set_commands noabort` command is useful when you want a file to continue command file execution even though an error might occur.

Discarding Pending Output

There are times when the Stop button doesn't interrupt lengthy output from TetraMAX ATPG. This occurs, for example, if you enter the following command:

```
report_atpg_constraints -all
```

To discard pending output, you can use the "Discard pending output" button located at the bottom of the TetraMAX console. This button is visible only after an interrupt (via the Stop button or ESC key) is detected, as shown in Figure 1.

Figure 1: Discard Pending Output Button

Note that the Stop button will always be enabled and operational if the kernel is still processing the current command.

Adjusting the Workspace Size

In TetraMAX ATPG, you can set the maximum line length, maximum string length, and maximum number of decisions allowed during test pattern generation. If you encounter messages indicating that the limits for the line or string lengths have been reached, on the menu bar, choose Edit > Environment to display the Environment dialog box. Then click Kernel and increase the limits.

For information about the other options available to this command, see Online Help for the `set_workspace_sizes` command.

As an alternative to the Environment dialog box, you can use the `set_workspace_sizes` command to change the workspace size settings.

Saving Preferences

TetraMAX ATPG enables you to save these GUI preferences so that the settings persist the next time you invoke TetraMAX ATPG.

When you invoke the TetraMAX GUI, it reads some of the default GSV preferences from the `tmax.rc` file. The TetraMAX GUI has a Preferences dialog box to change the default settings to control the appearance and behavior of the GUI. These default settings control the size of main window, window geometry, application font, size, GSV preferences and other preferences. If you change the appearance and behavior of the GUI using the Preferences dialog box, TetraMAX ATPG saves your changes in the

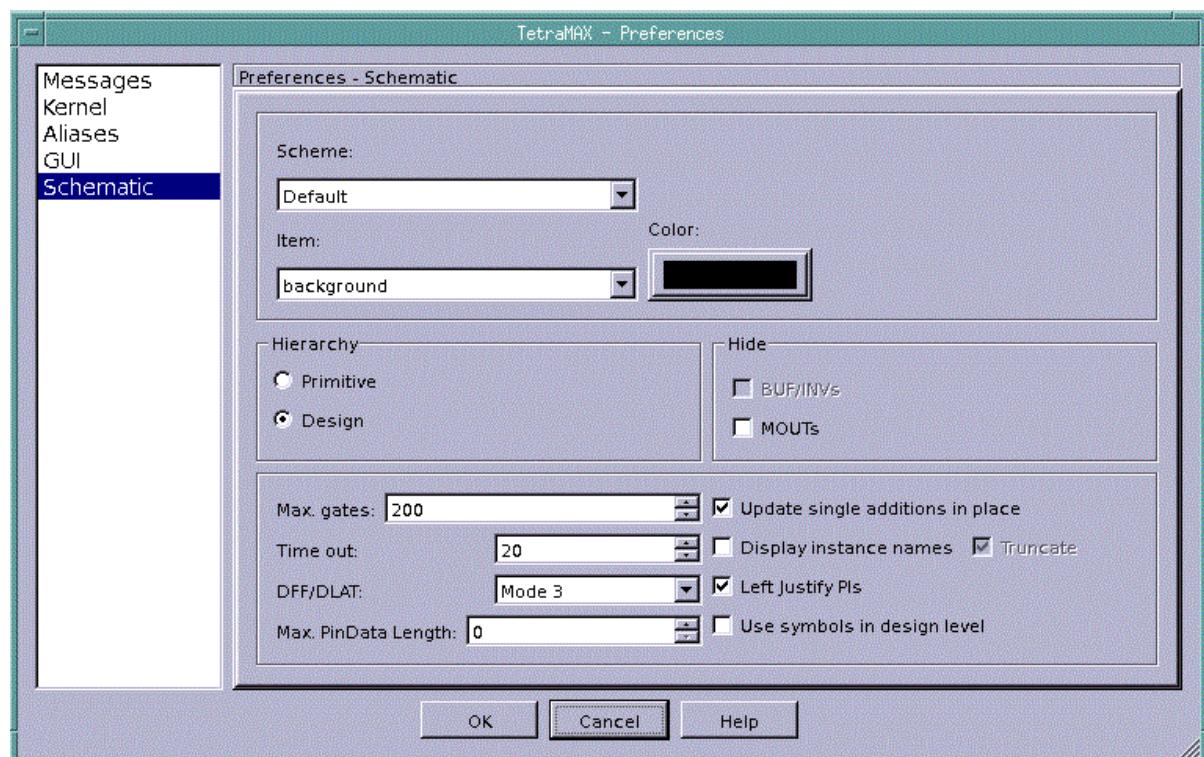
`$ (HOME) /.config/Synopsys/tmaxgui.conf` file before it exits. The next time you invoke TetraMAX ATPG, it does the following:

- Reads the default preferences from the `tmax.rc` file.
- Reads the preferences from the `$ (HOME) / .config/Synopsys/tmaxgui.conf` file. For the preferences that are listed in the `tmax.rc` file, the `$ (HOME) / .config/Synopsys/tmaxgui.conf` file has precedence over the `tmax.rc` file. For all other GUI preferences, TetraMAX ATPG uses the values from the `tmaxgui.rc` file to define the appearance and behavior of the GUI.

Setting Preferences

You can adjust settings such message formatting, workspace size, aliases, GUI font and color display, and schematic display options. To access the Preference dialog box, select Edit -> Preferences, as shown in [Figure 1](#).

Figure 1 Preferences Dialog Box



[Table 1](#) describes the various Preferences settings:

Table 1: TetraMAX GUI Preferences

Category	Description
Messages	Directs output messages to a log file, formats messages with a prefix, displays comments, sets message level to standard or expert.

Table 1: TetraMAX GUI Preferences (Continued)

Kernel	Sets maximum workspace sizes for ATPG gates, decisions, file line length, string line length, connectors.
Aliases	Adds, removes, and modifies alias names and text.
GUI	Sets the display of the toolbar, the default font size and type, and the default font color for commands and error messages.
Schematic	Sets the default color scheme, the hierarchy display, the display of gates and instance names, and the pin data length.

To save any Preferences specifications you made, select Edit -> Save Preferences or Edit -> Autosave Preferences.

Note: As an alternative to the Preferences dialog box, you can use the `set_workspace_sizes` command to change the workspace size settings.

See Also

[Displaying Symbols in Primitive or Design View](#)

Saving GUI Preferences

You can save GUI preferences so that the settings are used the next time you invoke TetraMAX ATPG.

When you invoke the TetraMAX GUI, it reads some of the default graphical schematic viewer (GSV) preferences from the `tmax.rc` file. The TetraMAX GUI has a Preferences dialog box to change the default settings to control the appearance and behavior of the GUI. These default settings control the size of main window, window geometry, application font, size, GSV preferences and other preferences. If you change the appearance and behavior of the GUI using the Preferences dialog box, TetraMAX ATPG saves your changes in the `$(HOME)/.config/Synopsys/tmaxgui.conf` file before it exits. The next time you invoke TetraMAX ATPG, it does the following:

- Reads the default preferences from the `tmax.rc` file.
- Reads the preferences from the `$(HOME)/.config/Synopsys/tmaxgui.conf` file. For the preferences that are listed in the `tmax.rc` file, the `$(HOME)/.config/Synopsys/tmaxgui.conf` file has precedence over the `tmax.rc` file. For all other GUI preferences, TetraMAX ATPG uses the values from the `tmaxgui.rc` file to define the appearance and behavior of the GUI.

See Also

[TetraMAX GUI Main Window](#)
[Using the Graphical Schematic Viewer](#)

Basic ATPG Flow

To run the basic ATPG flow:

1. Set your environment and launch TetraMAX ATPG.

To launch TetraMAX, specify the `tmax` command (for TetraMAX ATPG) or the `tmax2` command (for TetraMAX II).

```
setenv SYNOPSYS /synopsys/m_branch/
set path = ($SYNOPSYS/bin $path)
setenv LM_LICENSE_FILE synopsys_licenses/my_license.lic:$LM_
LICENSE_FILE
tmax my_command_file -shell
```

2. Prepare your netlist.

TetraMAX ATPG can read netlists in Electronic Design Interchange Format (EDIF), Verilog, and VHDL formats. You might need to make some minor edits to make the netlists compatible with TetraMAX ATPG. For more information, see "[Preparing a Netlist](#)."

3. Read your netlist into TetraMAX using either the `read_netlist` command or the Read Netlist dialog box in the TetraMAX GUI. The following example specifies the `read_netlist` command to read in all Verilog netlists in the `/tech` directory:

```
BUILD-T> read_netlist /tech/*.v
```

For more information, see "[Reading a Netlist](#)."

4. Read the Verilog library models using either the `read_netlist` command or the Read Netlist dialog box in the TetraMAX GUI. The following example reads in all Verilog library model files in the `/proj1234/shared_verilog` directory:

```
BUILD-T> read_netlist /proj1234/shared_verilog/*.v -noabort
```

For more information on reading the Verilog library models, see "[Reading Library Models](#)."

5. Create an in-memory design model using either the `run_build_model` command or the Run Build Model dialog box in the TetraMAX GUI. The following example builds a design model based on the last unreferenced module read by the `read_netlist` command or the Read Netlist dialog box:

```
BUILD-T> run_build_model
```

For more information, see "[Preparing to Build the ATPG Model](#)" and "[Building the ATPG Model](#)."

6. Create the STIL procedures file. For more information, see "[STIL Procedures](#)".

7. Define the clocks and asynchronous set and reset ports using either the STIL procedures file (SPF), the `add_clocks` command or the Add Clocks dialog box in the TetraMAX GUI. The following example use the `add_clocks` command to specify a set of clocking parameters:

```
DRC-T> add_clocks 0 CLK1 -timing 200 50 80 40 -unit ns -shift
```

For more information on specifying timing and clocks, see "[Defining Basic Signal Timing](#)" and "[Declaring Clocks](#)."

8. Set up and perform design rule checking (DRC) using the `set_drc` and `run_drc` commands, or the Run DRC dialog box in the TetraMAX GUI. The following example, prepares and runs DRC using the `set_drc` and `run_drc` commands:

```
DRC-T> set_drc -oscillation 200 -clock -any
```

```
DRC-T> run_drc spec_stil_file.spf
```

For more information, see "[Specifying DRC Settings](#)," "[Starting DRC](#)," and "[Performing Design Rule Checking](#)."

9. Set up and run ATPG using the `set_atpg` and `run_atpg` commands or the Run ATPG dialog box in the TetraMAX GUI. The following example uses the `set_atpg` and `run_atpg` command to prepare for and run ATPG:

```
TEST-T> set_atpg -patterns 500 -coverage 98
```

```
TEST-T> run_atpg
```

For more information, see "[Preparing for ATPG](#)" and "[Running ATPG](#)."

10. Using the `write_patterns` command or the Write ATPG dialog box in the TetraMAX GUI to save the ATPG patterns. The following example uses the `write_patterns` command to write a set of serial STIL patterns.

```
TEST-T> write_patterns patterns.stil -serial -format stil
```

Getting Started for DFT Compiler Users

The following steps show how to generate ATPG patterns when you start from a netlist in which DFT Compiler has performed scan insertion:

1. Before performing scan insertion, configure DFT Compiler for optimal results in TetraMAX ATPG.

```
test_default_delay 0
test_default_bidir_delay 0
test_default_strobe 40
test_default_period 100
test_stil_multiclock_capture_procedures true
```

Note: If your ASIC vendor has specific requirements, you might need to change these settings.

2. Within dc_shell or design_analyzer, write the netlist and the test protocol file.

```
test_stil_netlist_format verilog
write -hierarchy -format verilog -output name.v
write_test_protocol -format stil -out name.spf
```

3. Read the netlist and models into TetraMAX.

Use the `read_netlist` command or the Read Netlist dialog box in the TetraMAX GUI. For more information, see the "[Reading the Netlist](#)," and "[Reading Library Models](#)" topics.

4. Create the in-memory design model.

Use the `run_build_model` command or the Run Build Model dialog box in the TetraMAX GUI.

For details, see "[Building the ATPG Model](#)."

5. Define clocks and scan chains.

For details, see "[Declaring Clocks in STIL](#)." If your vendor requires LSI WGL protocols, additional edits to the DRC procedure file might be required. See "[LSI Compatible WGL](#)."

```
run_drc name.spf
```

6. Perform circuit initialization and Design Rule Checking (DRC) using the STIL procedure file.

For details, see "[Performing Test Design Rule Checking](#)."

7. Initialize the fault list and set ATPG options, effort, and dynamic compression.

For details, see "[Preparing for ATPG](#)."

8. Run ATPG to develop patterns.

For details, see "[Running ATPG](#)."

- Write the fault lists using the `write_faults` command.

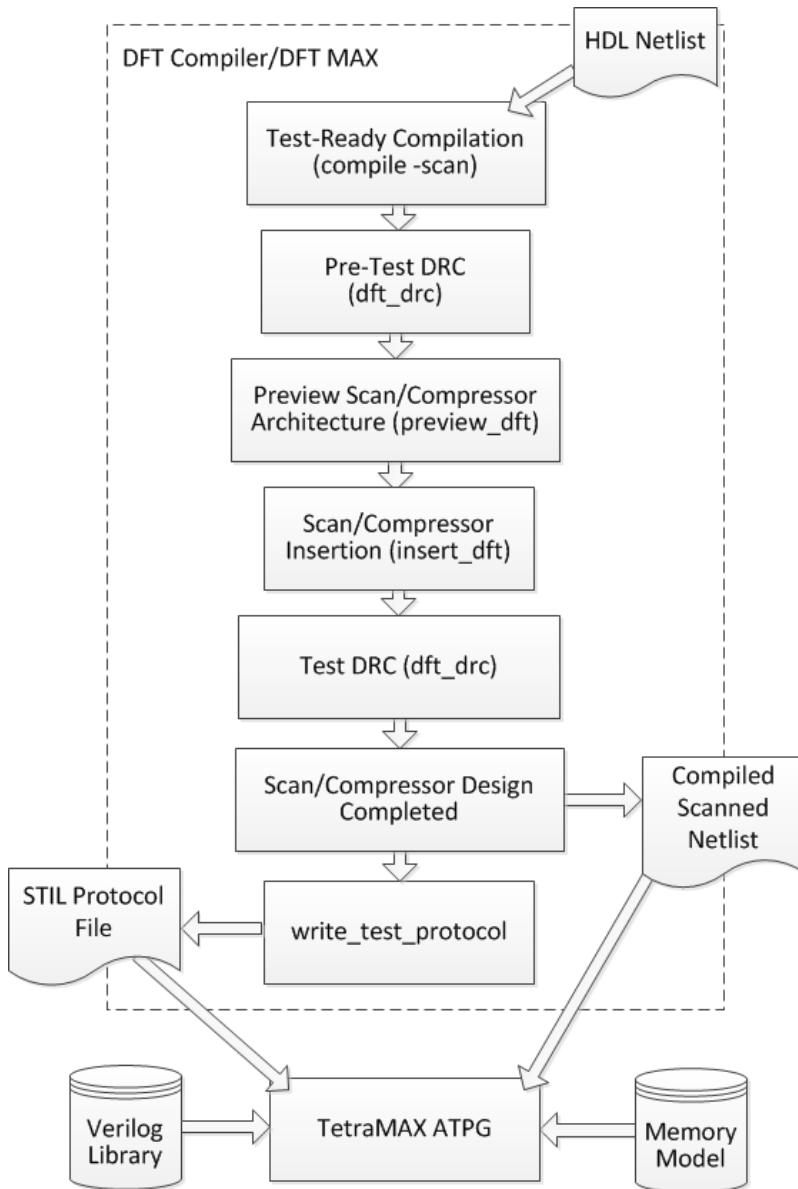
For details, see "[Writing Fault Lists](#)."

Design Flow Using DFT Compiler and TetraMAX

TetraMAX ATPG is compatible with a wide range of design-for-test tools, such as DFT Compiler.

[Figure 1](#) shows how TetraMAX ATPG fits into the DFT Compiler design-for-test flow for a module or a medium-sized design of less than 750K gates.

Figure 1 Design Flow for a Module or Medium-Sized Design



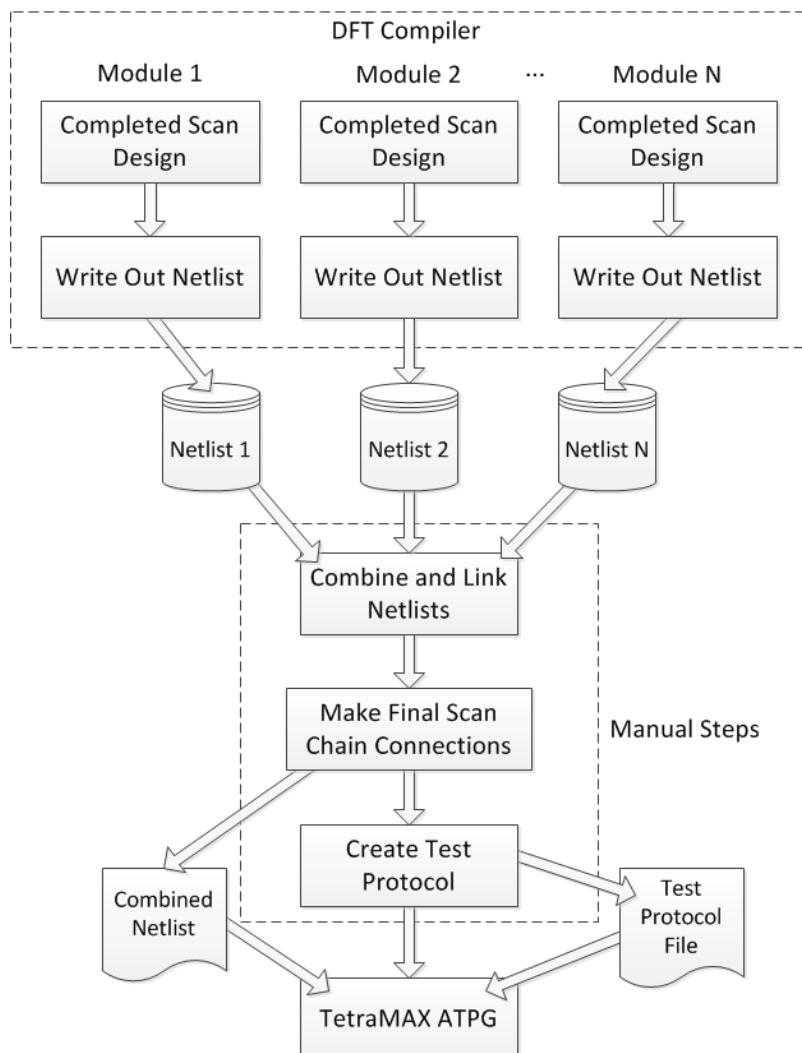
The design flow shown in [Figure 1](#) is as follows:

1. Starting with an HDL netlist at the register transfer level (RTL) within DFT Compiler, run a test-ready compilation, which integrates logic optimization and scan replacement.
The `compile -scan` command maps all sequential cells directly to their scan equivalents. At this point, you still don't know whether the sequential cells meet the test design rules.
2. Perform test design rule checking; the `check_scan` command reports any sequential cells that violate test design rules.
3. After you resolve the DRC violations, run the `preview_scan` command to examine the scan architecture that is synthesized by the `insert_scan` command. Repeat this procedure until you are satisfied with the scan architecture, then run the `insert_scan` command, which implements the scan architecture.
4. Rerun the `check_scan` command to identify any remaining DRC violations and to infer a test protocol. For details about the DFT Compiler design flow through completion of the scan design, see the *DFT Compiler Scan Synthesis User Guide*.
5. When your netlist is free of DRC violations, it is ready for ATPG. For medium-sized and smaller designs, DFT Compiler provides the `write_test_protocol` command, which allows you to write out a STL procedure file. TetraMAX ATPG reads the STL procedure file and design netlist.

For details of the TetraMAX ATPG portion of the design flow, see "[ATPG Design Flow](#)."

[Figure 2](#) shows the design flow for a design that is too large for test protocol file generation from a single netlist (about 750K gates or larger).

Figure 2 Design Flow for a Very Large Design



For large designs, you initially follow the design flow shown in [Figure 1](#) at the module level, using modules of 200K gates or fewer, to get the completed scan design for each module.

Then, as shown in [Figure 2](#), you start with the completed scan design for each module. You write the netlists, combine and link the netlists, and make the final scan chain connections, thus generating a combined netlist for the entire design. A test protocol file is created automatically.

For information on creating a test procedure file, see “[STIL Procedure Files](#).” You use the combined netlist and the manually generated test protocol file as inputs to TetraMAX ATPG.

See Also

[ATPG Design Flow](#)

2

AUTOMATIC TEST PATTERN GENERATION (ATPG) DESIGN FLOW

The ATPG process creates a sequence of test patterns that enable an ATE to distinguish between the correct circuit behavior and the faulty circuit behavior caused by the defects. The generated patterns are used to test devices and to determine the cause of failure. ATPG effectiveness is measured by the amount of modeled defects, or fault models, that are detected and the number of generated patterns.

The following sections describe the basic ATPG design flow:

- [ATPG Design Flow Overview](#)
- [Preparing a Netlist](#)
- [Reading a Netlist](#)
- [Reading Library Models](#)
- [Setting Up and Building the ATPG Model](#)
- [Performing Test Design Rule Checking \(DRC\)](#)
- [Preparing for ATPG](#)
- [Running ATPG](#)
- [Analyzing ATPG Output](#)
- [Reviewing Test Coverage](#)
- [Writing ATPG Patterns](#)

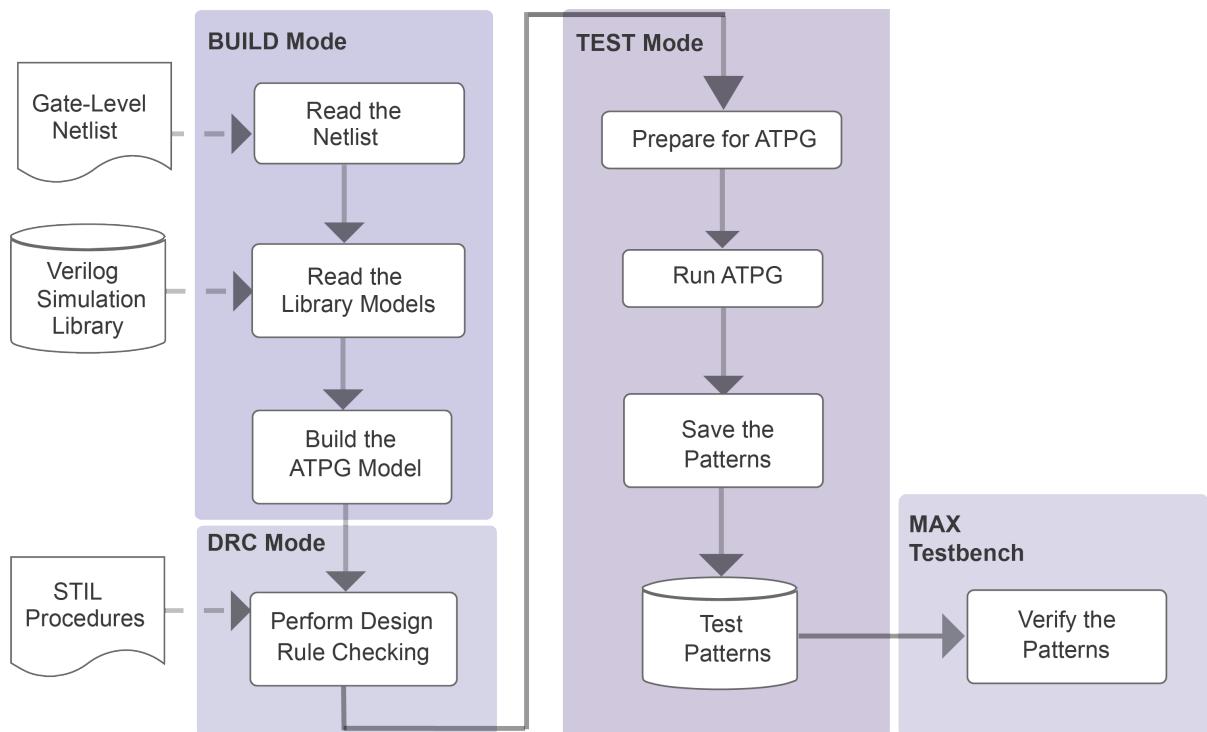
ATPG Design Flow Overview

The basic ATPG flow applies to most designs. To get started running ATPG, you must provide a supported netlist, a model library, and a set of STIL procedures used for design rule checking.

STIL procedures are usually provided via a STIL procedures file generated from the DFT Compiler tool or an equivalent tool. You can also provide many of the parameters via TetraMAX commands. For complete information on STIL procedures, see "[STIL Procedures](#)."

[Figure 1](#) shows the basic ATPG design flow. For a step-by-step overview of the ATPG design flow, see "[Running the Basic ATPG Design Flow](#)."

Figure 1 Basic ATPG Design Flow



If you encounter problems with your design, see "[Using the GSV for Review and Analysis](#)," which provides information on graphical analysis and troubleshooting.

Running the Basic ATPG Design Flow

The basic ATPG design flow consists of the following steps:

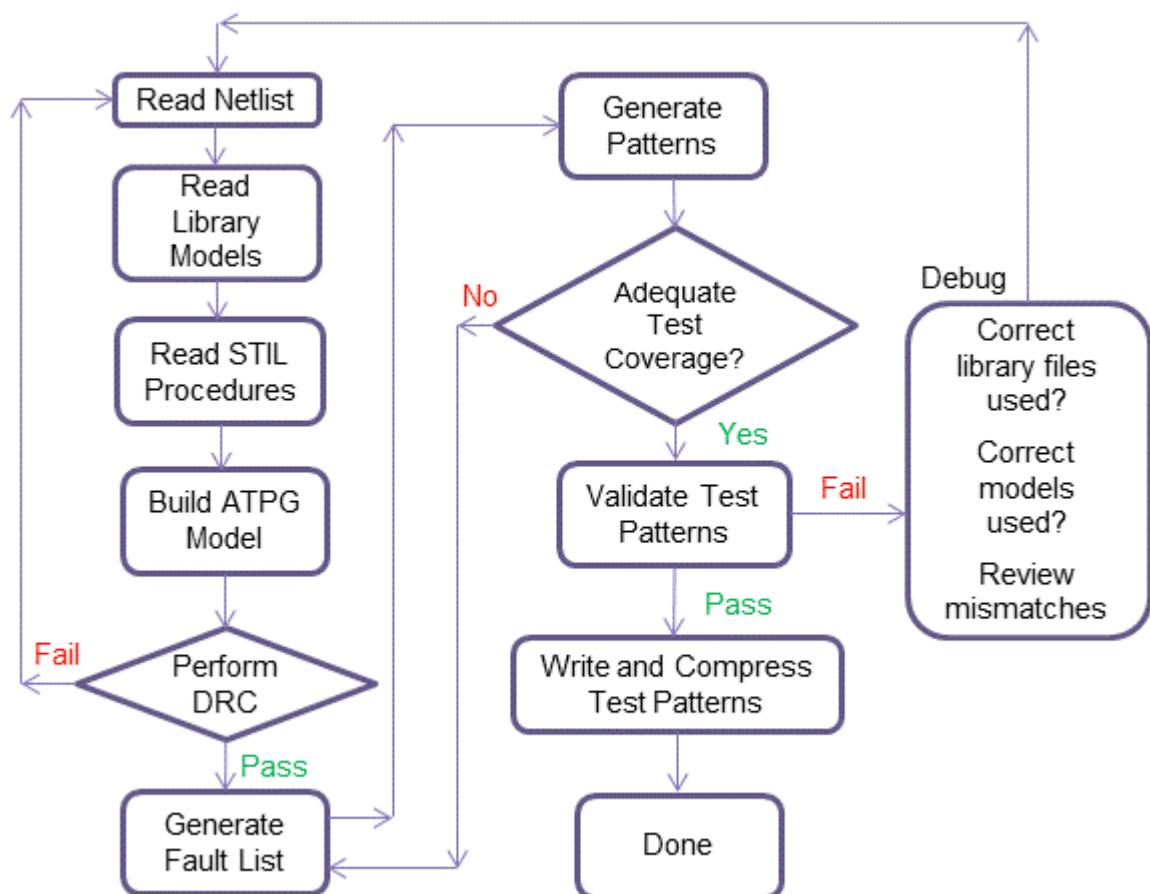
1. Prepare your netlist or netlists (see "[Preparing a Netlist](#)").
2. Read the netlist (see "[Reading a Netlist](#)").
3. Read the library models (see "[Reading Library Modules](#)")
4. Build the ATPG design model (see "[Setting Up and Building the ATPG Model](#)")

5. Perform test DRC and make any necessary corrections (see "[Performing Test Design Rule Checking](#)").
6. Prepare the design for ATPG, set up the fault list, analyze buses for contention, and set the ATPG options (see "[Preparing for ATPG](#)").
7. Run automatic test pattern generation (see "[Running ATPG](#)").
8. Analyze the ATPG pattern generation output (see "[Analyze ATPG Output](#)").
9. Review the test coverage (see "[Reviewing Test Coverage](#)").
10. Rerun ATPG, as needed.
11. Write and save the test patterns (see "[Writing ATPG Patterns](#)").

For an example of a typical command file used for running a basic ATPG design flow in TetraMAX ATPG, see "[Using Command Files](#)".

[Figure 2](#) shows a typical ATPG design flowchart.

Figure 2 ATPG Design Flowchart



See Also

[Design Flow Using DFT Compiler and TetraMAX](#)
[ATPG Design Guidelines](#)

Using Command Files

A command file is a simple ASCII text file containing any command accepted by TetraMAX ATPG. You can accomplish many of the tasks described in "[ATPG Design Flow](#)" using a command file. The following example launches TetraMAX ATPG using a command file:

```
% tmax -shell spec_command_file.cmd
```

[Example 1](#) shows a typical command file, which reads in a design that has been debugged to eliminate DRC problems. The commands in this file create and store ATPG patterns and fault lists while saving the execution log.

Example 1: Typical Command File

```
# --- basic ATPG command sequence
#
set_messages log last_run.log -replace
#
# --- read design and libraries
#
read_netlist spec_design.v -delete
read_netlist /home/vendor_A/tech_B/verilog/*.v -noabort
report_modules -summary
report_modules -error
#
# --- build design model
#
run_build_model spec_top_level_name
report_rules -fail
#
# --- define clocks and pin constraints
#
add_clocks 1 CLK MCLK SCLK
add_clocks 0 resetn ioscl4m
add_pi_constraints 1 testmode
#
# --- define scan chains & STIL procedures, perform DRC checks
#
run_drc spec_design.spf
report_rules -fail
report_nonscan_cells -summary
report_buses -summary
report_feedback_paths -summary
#
# --- create patterns
```

```

#
set_atpg -abort 20 -pat 1500 -merge high
add_faults -all
run_atpg -auto_compression
report_summaries
#
# --- save fault list and patterns
#
report_faults -level 5 64 -class au -collapse -verbose
write_faults faults.all -all -replace
write_patterns patterns.v -format verilog -parallel 2 -replace
#
exit
#

```

See Also

[Command Files](#)

[Using Command Files in Tcl Mode](#)

[Command Entry](#)

[Invoking TetraMAX](#)

Preparing a Netlist

TetraMAX ATPG accepts netlists in Verilog, EDIF, and VHDL formats. For more information on these formats, see ["Netlist Format Requirements."](#)

Netlists can be flat or hierarchical and can be in standard ASCII format or GZIP format.

TetraMAX ATPG automatically detects compressed files and decompresses them during the read operation.

Before reading in a netlist or library models, you should compare the names of the modules in your netlist to the names of the Verilog library models you are using. If there are duplicate module definitions, TetraMAX ATPG uses the last definition it encounters. If you read in your netlist and then read in library models, the modules in your netlist are overwritten by any library models using the same names.

You can specify the following options in preparation for reading a netlist:

- Set the maximum number of parsing errors allowed before terminating the parsing of the current netlist file
- Use the last module read if your design has duplicate modules. This allows TetraMAX ATPG to reread a file if you edit a module or read multiple files if there is a duplication of module definitions.
- Accept or ignore the 'celldefine', 'enable_portfaults', and 'suppress_faults' Verilog compiler directives
- Set check and warning behavior for reading netlists and designs
- Specify if conservative or combinational MUX gates are extracted from conservative UDP models of a MUX

- Set parameters for handling dominance behavior between set, reset, and clock pins
- Specify behavior for escape characters, redefined modules, scalar nets, and X modeling

For complete descriptions of these options, see the description of the `set_netlist` command in TetraMAX Help.

See Also

[Configuring to Read a Netlist](#)
[Reading a Netlist](#)

Configuring to Read a Netlist

You can use either the `set_netlist` command or the Set Netlist dialog box or Read Netlist dialog box to specify options for reading a netlist into TetraMAX ATPG.

The following example shows how to use the `set_netlist` command to allow a maximum of 15 parsing errors, extract combinational MUX Gates from conservative MUX UDP models, and use the remaining default parameters for reading a netlist:

```
BUILD> set_netlist -max_errors 15 -conservative_mux combo_udp
```

To use the TetraMAX GUI to set the parameters specified in the previous example:

1. Do one of the following:
 - From the menu bar, select Netlist > Set Netlist Options.
The Set Netlist dialog box appears.
 - From the command toolbar, click the Netlist button.
The Read Netlist dialog box appears.
2. In either the Set Netlist dialog box or the Read Netlist dialog box, enter 15 in the Maximum Errors text field, and select Combinational UDP in the Conversative MUX drop-down menu.
3. Click OK.

For a complete description of the requirements and contents of netlists used for TetraMAX ATPG, see "[Design Netlists and Libraries](#)."

Reading a Netlist

You can read one or more netlists associated with your design using the `read_netlist` command or the Read Netlist dialog box in the TetraMAX GUI.

The following example specifies the `read_netlist` command to read in all Verilog netlists in the `/tech` directory:

```
BUILD-T> read_netlist /tech/*.v
```

You can specify as many `read_netlist` commands as necessary to read in all portions of a design. You can read multiple files from the same directory using wildcards (for example, *.v). For more information, see "[Using Wildcards to Read Netlists](#)".

To read a netlist using the Read Netlist dialog box:

1. Do one of the following:
 - From the menu bar, select Netlist > Read Netlist.
 - From the command toolbar, click the Netlist button.

In both cases, the Read Netlist dialog box appears with the selected default values.

2. Change or select any values to meet your requirements. The options in the Read Netlist dialog box are equivalent to the options for the `read_netlist` command (see the description in TetraMAX Help).
3. Click OK.

See Also

[Working with Design Netlists and Models](#)

[About Reading a Netlist](#)

[Netlist Requirements](#)

[Reading the Library Models](#)

Reading Library Models

To read library models, use the the `read_netlist` command or the Read Netlist dialog box.

The following example uses the `read_netlist` command to read in all Verilog library model files in the /proj1234/shared_verilog directory and to not terminate the process if there is an error reported for a model:

```
BUILD-T> read_netlist /proj1234/shared_verilog/*.v -noabort
```

You can specify as many `read_netlist` commands as necessary to read in all portions of a design. You can read multiple files from the same directory using wildcards (for example, *.v). For more information, see "[Using Wildcards to Read Netlists](#)".

To read library models using the Read Netlist dialog box:

1. Do one of the following:
 - From the menu bar, select Netlist > Read Netlist.
 - From the command toolbar, click the Netlist button.

In both cases, the Read Netlist dialog box appears with the selected default values.

2. Change or select any values to meet your requirements. To duplicate the previous command line example, make sure the Abort on Error check box is not selected. The options in the Read Netlist dialog box are equivalent to the options for the

`read_netlist` command (see the description in TetraMAX Help).

3. Click OK.

See Also

[About Reading a Library Model](#)

[Reading a Netlist](#)

[Building the ATPG Model](#)

Preparing to Build the ATPG Model

You can use either the `set_build` command or the Set Build dialog box to set parameters for building the ATPG model.

The following example uses the `set_build` command to set the parameters to build an ATPG model. In this case, it specifies TetraMAX ATPG to use a period (.) as a hierarchical delimiter and to not to delete any unused gates:

```
DRC-T> set_build -hierarchical_delimiter . -nodelete_unused_gates
```

You can make the same settings from the previous example using the Set Build dialog box, as shown in the following steps:

1. Do one of the following:
 - From the menu bar, select Netlist > Set Build Options.
 - From the command toolbar, click the Build button. When the Build Model dialog box appears, click the Set Build Options button.In both cases, the Set Build dialog box appears with the selected default values.
2. Change or select the values in the Set Build dialog box to meet your requirements.
The options in this dialog box are equivalent to the options for the `set_build` command (see the description in TetraMAX Help). To match the options specified in the previous example, enter a period (.) in the Hierarchical text field and unselect the Delete Unused Gates checkbox.
3. Click OK.

See Also

[Building the ATPG Model](#)

Building the ATPG Model

You can use the `run_build_model` command or the Run Build dialog box to build the ATPG model.

TetraMAX ATPG builds a model based on the last unreferenced module read by the `read_netlist` command or the Read Netlist dialog box. This means you do not need to specify any options with the `run_build_model` command, as shown in the following example:

```
BUILD-T> run_build_model
```

You can also specify a particular module, as shown in the following example, which includes the command transcript:

```
BUILD-T> run_build_model spec_asic
-----
Begin build model for topcut = spec_asic ...
-----
End build model: #primitives=101004, CPU_time=13.90 sec,
Memory=34702381
-----
Begin learning analyses...
End learning analyses, total learning CPU time=33.02
-----
```

To build the ATPG model using the Run Build Model dialog box:

1. Do one of the following:
 - From the menu bar, select Netlist > Run Build Model.
 - From the command toolbar, click the Build button.
- In both cases, the Run Build dialog box appears with the selected default values.
2. Change or select the additional values in the Run Build dialog box to meet your requirements. The options in this dialog box are equivalent to the options for the `run_build_model` command and the `set_learning` command (see the descriptions for both commands in TetraMAX Help). To match the option specified in the previous example, enter `spec_asic` in the Top Module name text field.
3. Click OK.

The build model process begins.

See Also

- [About Building a Library Model](#)
- [Processes That Occur When Building the ATPG Model](#)

Performing Design Rule Checking (DRC)

During DRC, TetraMAX ATPG performs a set of checks to ensure that the scan structure is correct and to determine how to use the scan structure for test generation and fault simulation. These checks include ensuring that the scan chains operate properly, identifying scan cells, identifying nonscan cell behavior, and ensuring that clocks obey the required rules.

The following sections describe how to prepare for and perform DRC:

- [Specifying STIL Procedures](#)
- [Specifying DRC Settings](#)
- [Starting DRC](#)
- [Reviewing the DRC Results](#)

- [Understanding Rule Violations](#)
 - [Viewing Violations in the GSV](#)
-

Specifying STIL Procedures

The STIL language describes scan-shifting protocol, test procedures, and ATPG signal, timing, and data information. STIL procedures provide information TetraMAX ATPG uses as a basis to perform design rule checking (DRC).

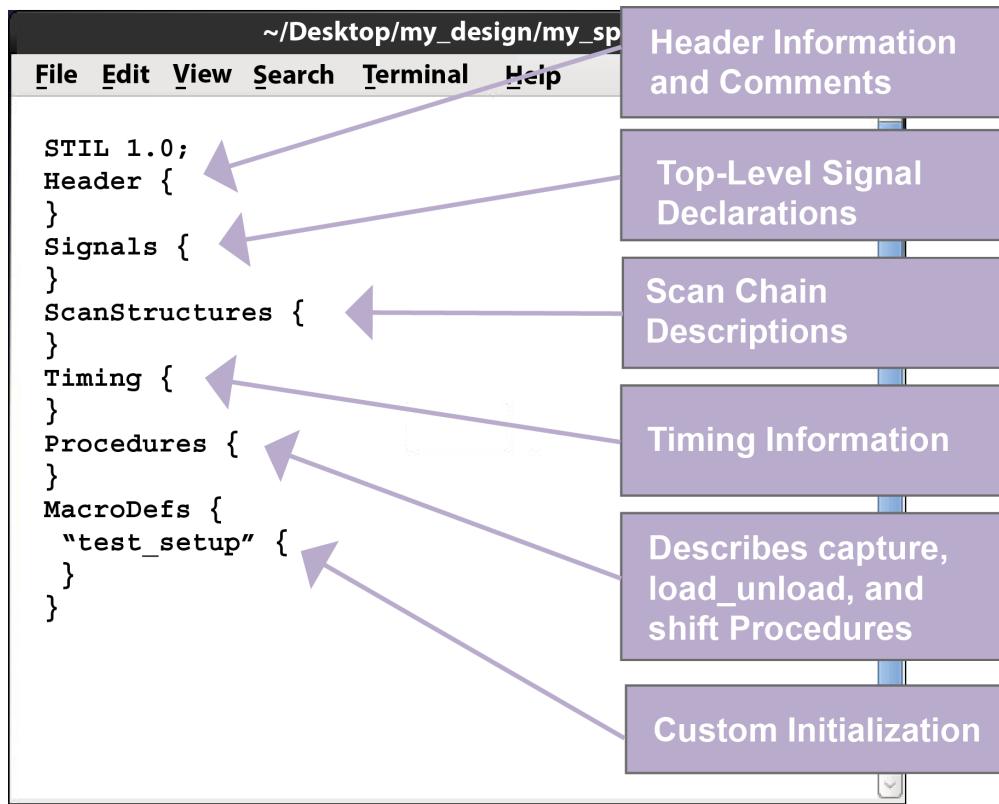
TetraMAX ATPG supports a subset of STIL syntax that describe:

- Scan chain inputs and outputs
- Pin constraints for test modes
- Clock ports and waveform definitions
- Shifting and capturing protocols
- Initialization sequences

There are several ways you can provide STIL procedures to TetraMAX for DRC:

- Create an SPF using Synopsys' DFT Compiler tool.
- Create an SPF template file using the `write_drc_file` command. For details, see "[Creating a New SPF](#)."
- Use the QuickSTIL tab in the DRC dialog box of the TetraMAX GUI.
- Use TetraMAX commands, such as `add_clocks`, `add_scan_chains`, and `add_pi_constraints`.

[Figure 1](#) provides a brief description of the major sections of a STIL procedures file.

Figure 1 STIL Procedures File

Specifying DRC Settings

Prior to performing DRC, make sure you specified a set of STIL procedures as described in ["Specifying STIL Procedures."](#) These procedures provide key information that TetraMAX ATPG needs to perform DRC.

You can set a variety of parameters that control the DRC process, including:

- Specify clock grouping and skew values
- Set the number of simulation passes before oscillation
- Define restrictions on clock usage for pattern generation
- Specify DLAT clock checks and DRC violation parameters for DFF and DLAT devices with unstable sets or resets
- Display primitives in scan chains that were sensitized during scan chain tracing
- Generate patterns with capture cycles that always have clock pulses from a controller clock
- Limit the reporting of shadows
- Specify the top-level port that globally enables or disables bidirectional pins
- Define the number of PLL clock pulses supported per load and the number of pulses to extend the simulation of the load procedure
- Allow patterns to have more than one capture clock procedure per load

- Store simulated time periods of the test_setup procedure, stability patterns, and unload mode data

Options for Specifying DRC Settings

You can use the `set_drc` command or the DRC dialog box to specify DRC parameters.

The following example shows how to specify the `set_drc` command:

```
DRC-T> set_drc -oscillation 200 -clock -any
```

This example uses the `-oscillation` option to specify that 200 simulation passes are allowed during DRC simulation before oscillation is declared. It also uses the `-clock -any` setting to allow pattern generation using any single clock, including patterns that don't use clocks.

To use the Run DRC dialog box to set DRC parameters:

1. Do one of the following:
 - From the menu bar, select Rules > Run DRC
 - From the command toolbar, click the DRC button

In both cases, the DRC dialog box appears with the Run tab active.

2. In the Set field of the DRC dialog box, specify the options you want to apply to the DRC process. To duplicate the settings of the `set_drc` command in the previous example:
 - a. Enter 200 in the Oscillation Passes text field
 - b. Select -Any from the Capture Clock drop-down menu.
3. Click the Set button to save your settings.

For more information on specifying DRC options, see "[Design Rule Checking](#)".

See Also

[Starting Test DRC](#)

[Reviewing the DRC Results](#)

Starting DRC

Before starting DRC, make sure you specified the appropriate STIL procedures (see "[Specifying STIL Procedures](#)") and DRC settings (see "[Specifying DRC Settings](#)").

To start DRC, use the `run_drc` command or the Run DRC dialog box in the TetraMAX GUI.

The following example uses the `run_drc` command to start DRC:

```
DRC-T> run_drc spec_stil_file.spf
```

The argument in the example, `spec_stil_file.spf`, is the name of the STIL procedure file.

To use the Run DRC dialog box to perform DRC:

1. Do one of the following:
 - From the menu bar, select Rules > Run DRC.
 - From the command toolbar, click the DRC button
- In both cases, the DRC dialog box appears with Run tab active.
2. In the Test Protocol File Name field, enter the path name of the STIL procedure file previously created, or use the Browse button to navigate and select the file.
3. Click Run.

As TetraMAX performs the DRC checks, it produces a status report and lists the DRC violations, as shown in [Example 1](#).

See "DRC Rules" in TetraMAX Help for a list of the rule categories, including links to each category.

Example 1 Typical DRC Run

```
BUILD-T> run_drc spec_stil_file.spf
-----
Begin scan design rule checking...
-----
Begin reading test protocol file lander.spf...
End parsing STIL file lander.spf with 0 errors.
Test protocol file reading completed, CPU time=0.10 sec.
-----
Begin Bus/Wire contention ability checking...
Bus summary: #bus_gates=40, #bidi=40, #weak=0, #pull=0,
#keepers=0
Contention status: #pass=0, #bidi=40, #fail=0, #abort=0,
#not_analyzed=0
Z-state status : #pass=0, #bidi=40, #fail=0, #abort=0,
#not_analyzed=0
Bus/Wire contention ability checking completed, CPU time=0.04 sec.
-----
Begin simulating test protocol procedures...
Nonscan cell constant value results: #constant0 = 4, #constant1 =
7
Nonscan cell load value results : #load0 = 4, #load1 = 7
Warning: Rule Z4 (bus contention in test procedure) failed 48
times.
Test protocol simulation completed, CPU time=0.14 sec.
-----
Begin scan chain operation checking...
Chain c1 successfully traced with 31 scan_cells.
Chain c2 successfully traced with 31 scan_cells.
Scan chain operation checking completed, CPU time=0.34 sec.
-----
Begin clock rules checking...
Warning: Rule C17 (clock connected to PO) failed 16 times.
```

```
Warning: Rule C19 (clock connected to non-contention-free BUS)
failed 1 times.
Clock rules checking completed, CPU time=0.14 sec.
-----
Begin nonscan rules checking...
Nonscan cell summary: #DFF=201 #DLAT=0 tla_usage_type=none
Nonscan behavior: #C0=4 #C1=7 #LE=11 #TE=179
Nonscan rules checking completed, CPU time=0.05 sec.
-----
Begin contention prevention rules checking...
26 scan cells are connected to bidirectional BUS gates.
Warning: Rule Z9 (bidi bus driver enable affected by scan cell)
failed 24 times.
Contention prevention checking completed, CPU time=0.02 sec.
-----
Begin DRC dependent learning...
DRC dependent learning completed, CPU time=0.97 sec.
-----
DRC Summary Report
-----
Warning: Rule C17 (clock connected to PO) failed 16 times.
Warning: Rule Z4 (bus contention in test procedure) failed 48
times.
Warning: Rule Z9 (bidi bus driver enable affected by scan cell)
failed 24 times.
There were 72 violations that occurred during DRC process.
Design rules checking was successful, total CPU time=2.01 sec.
```

Reviewing the DRC Results

After you run DRC (see "[Starting Test DRC](#)"), TetraMAX ATPG generates a summary report that provides a starting point for reviewing the DRC results. To view a description of the summary report, see "[Understanding the DRC Summary Report](#)".

You should inspect and correct all DRC violations that are classified as errors. If you ignore or overlook these violations, the ATPG patterns might fail in simulation or on the real device. For more information about DRC rule violations and how to fix them, see "[Understanding DRC Rule Violations](#)".

To view descriptions of specific violations and how to fix them, see "DRC Rules by Category" in TetraMAX Help).

If you want to view a summary of failing rule messages, enter the following command:

```
DRC-T> report_rules -fail
```

The DRC summary report in [Example 2](#) shows one class of clock rule warnings (C17) and two classes of bus rule warnings (Z4 and Z9).

[Example 2](#) shows an example of the `report_rules -fail` output.

Example 2 Reporting Rules That Fail

```
TEST-T> report_rules -fail
```

```
// C16: #fails=190 severity=warning  
// C17: #fails=16 severity=warning  
// C19: #fails=1 severity=warning  
// Z4: #fails=128 severity=warning  
// Z9: #fails=24 severity=warning
```

For more detailed information about specific [DRC violations](#) in the design, use the `report_violations` command. You can identify a single violation, all violations of a single type, all violations within a class, or all violations, as in the following examples:

```
DRC-T> report_violations c17-2  
DRC-T> report_violations c17  
DRC-T> report_violations c  
DRC-T> report_violations -all
```

See Also

[Understanding run_drc Output](#)

[Starting Test DRC](#)

[Viewing Violations in the GSV](#)

Understanding Rule Violations

The test design rules are organized by category. Each rule has an identification code (rule ID) consisting of a single character followed by a number. The first character defines the major category of the rule.

The rules are organized functionally into nine major categories:

- B (Build rules)
- C (Clock rules)
- N (Netlist rules)
- P (Path Delay rules)
- S (Scan Chain rules)
- V (Vector rules)
- X (X-state rules)
- Z (Tristate rules)

Links to descriptions of individual rules are provided in the "Rules Violation Messages" topic in TetraMAX Help.

When a rule is violated, each violation is assigned a unique violation ID, which is the rule ID followed by a dash and then a sequence number. For example, the rule violation ID for the 24th violation of a Z4 rule is Z4-24. You can use this number to identify a specific violation for reporting or analysis.

Some violation IDs show an abort indicator suffix, which appears as Z7-12.A or Z6-3 (Abort). This means that the ATPG analysis of the violation was aborted. In such cases, you might want to increase the ATPG abort limit.

Each rule violation also includes a brief description of what is checked by the rule. For example, a B8 rule violation explains that the circuit contains an unconnected module input pin.

The effects of a rule violation vary depending on the rule's severity level. For example, rule N5, "redefined module," has a default Warning severity level and in most cases notifies you that a module was defined and then redefined, and that the last definition encountered is being used. In contrast, the rule S1, "scan chain blockage," has a default Fatal severity level. The scan chain is not usable in its current state, and you must correct the problem before trying further pattern generation.

The default severity level for each rule reflects a conservative approach to ATPG efforts. When an error or warning is produced, review the potential problem and determine whether you need to change the design or the ATPG procedures. You might be able to adjust the severity level downward and continue ATPG.

TetraMAX Help provides a complete description of each rule violation. The "What Next" section in the description for a rule suggests an action you can take to analyze the cause of the rule violation and determine whether the violation may be fixed by changing a procedure or setup, or whether the design may have to be changed.

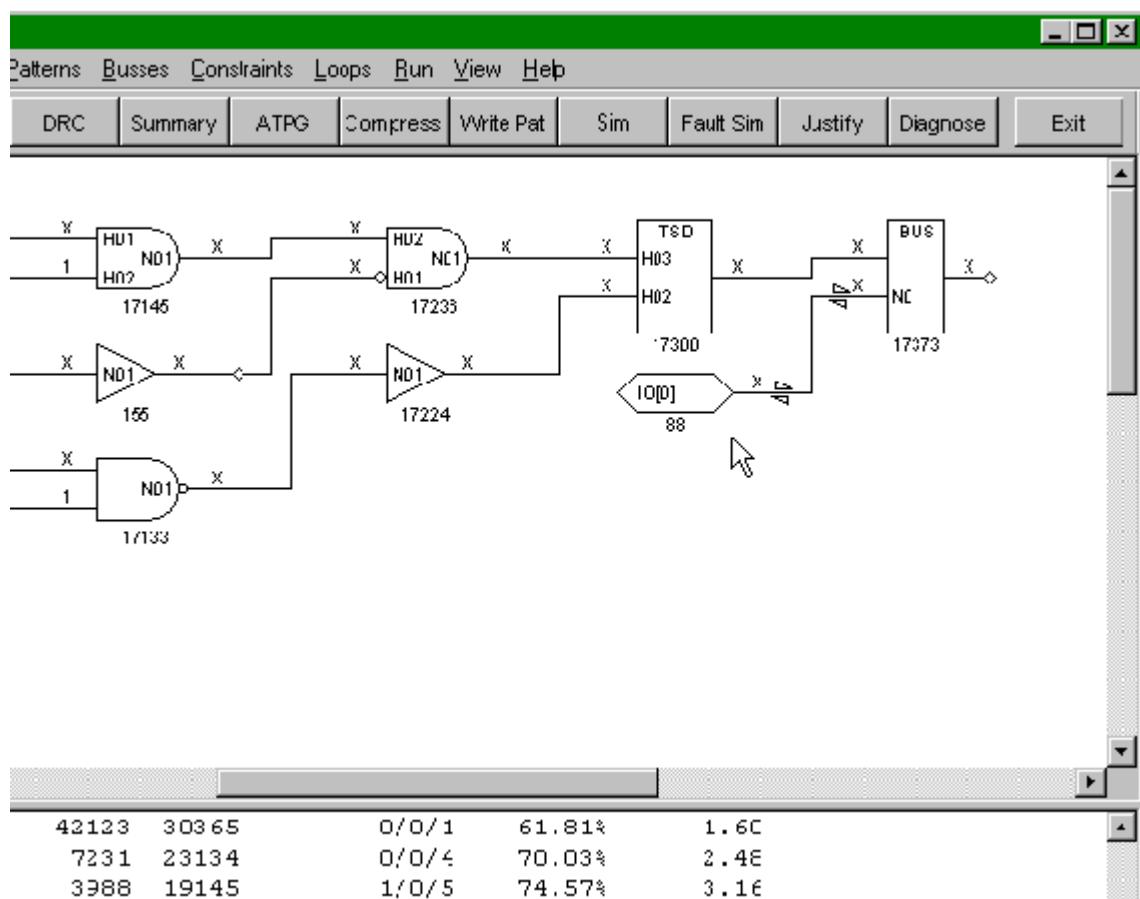
For rule violations with an error severity, the occurrence message is displayed when the rule violation occurs. For rule violations with a warning severity, the summary message is displayed at the end of the process. You can selectively display the occurrence messages for a warning using the `report_violations` command.

Viewing DRC Violations in the GSV

You can visually inspect many of the rule violations using the graphical schematic viewer (GSV). The GSV displays a subset of the design showing the logic gates involved in the [DRC violation](#), along with appropriate diagnostic data such as logic values, constrained ports, or clock cones. For more information on using the GSV, see "[Using the Graphical Schematic Viewer](#)."

To analyze a warning message in the GSV:

1. Click the Analyze button in the command toolbar at the top of the [TetraMAX GUI main window](#).
The Analyze dialog box appears.
2. Click the Rules tab if it is not already active.
A dialog lists all the most recent violations. All violations are numbered. For example, Z4-1:12 means there are 12 violations of rule Z4, designated Z4-1 through Z4-12.
3. Select a violation from the list or type a specific violation occurrence number in the Rule Violation field.
4. Click OK.
The GSV opens and displays the violation. The transcript window also displays the error message.

Figure 1 Schematic Display of DRC Violation

An example Z4 violation message is shown in the following example:

Warning: Bus contention on /bixcr (17373) occurred at time 0 of test_setup procedure. (Z4-1)

A simple and fast way to view the schematic for a violation message is to point to the red-highlighted error message in the transcript window, click the right mouse button, and select Analyze in the pop-up menu.

[Figure 1](#) shows the gates involved in the Z4 violation, along with the logic values resulting from simulation of the `test_setup` macro. The `test_setup` macro is described in more detail in the section “[Defining the test_setup Macro](#).”

In this example, most of the logic values are X (unknown). The violation might be caused by failing to force a Z on a bidirectional port called `IO[0]` in the `test_setup` procedure. You can choose to ignore or correct this violation. If you choose to ignore it, fault coverage is lowered because the ATPG algorithm will not generate any pattern that would cause contention.

Some messages can be safely ignored. Others can be resolved through adjustment of a procedure definition; and others require a change to the design.

See Also

[Using the Graphical Schematic Viewer](#)

Preparing for ATPG

ATPG creates a sequence of test patterns that enable an ATE to distinguish between the correct circuit behavior and the faulty circuit behavior caused by the defects. The generated patterns are used to test devices and to determine the cause of failure.

To prepare for ATPG, you can specify general ATPG settings, set up the fault list, select the fault model (stuck-at, IDDQ, path delay, hold time, transition, or bridging), select the pattern source (internal, external, or random patterns), and select the ATPG mode (basic-scan, fast-sequential, or full-sequential).

The following tasks show you how to prepare for ATPG:

- [Specifying General ATPG Settings](#)
- [Specifying Fault Lists](#)
- [Specifying Fault Models](#)
- [Specifying the Pattern Source](#)
- [Specifying the ATPG Mode](#)

See Also

[Running ATPG](#)

Specifying General ATPG Settings

There are a variety of general parameters you can use to control pattern generation by TetraMAX ATPG. For example, you can:

- Specify the maximum number of patterns to generate before terminating ATPG
- Set the maximum CPU time allowed per fault before terminating fault detection
- Limit the maximum coverage for ATPG to attain before terminating
- Set the minimum number of system cycles for each pattern
- Use fill options for running internal scan and compressed scan patterns
- Establish checkpoints to save patterns and fault lists to files

For complete details on these settings and all other settings that control ATPG, see "[ATPG Settings](#)"

Options for Specifying ATPG Settings

You can specify several types of general settings for ATPG using the `set_atpg` command or the TetraMAX GUI, as shown in the following examples:

- The following command specifies TetraMAX ATPG to generate a maximum of 500 patterns and to terminate ATPG when the coverage reaches 98 percent:

```
set_atpg -patterns 500 -coverage 98
```

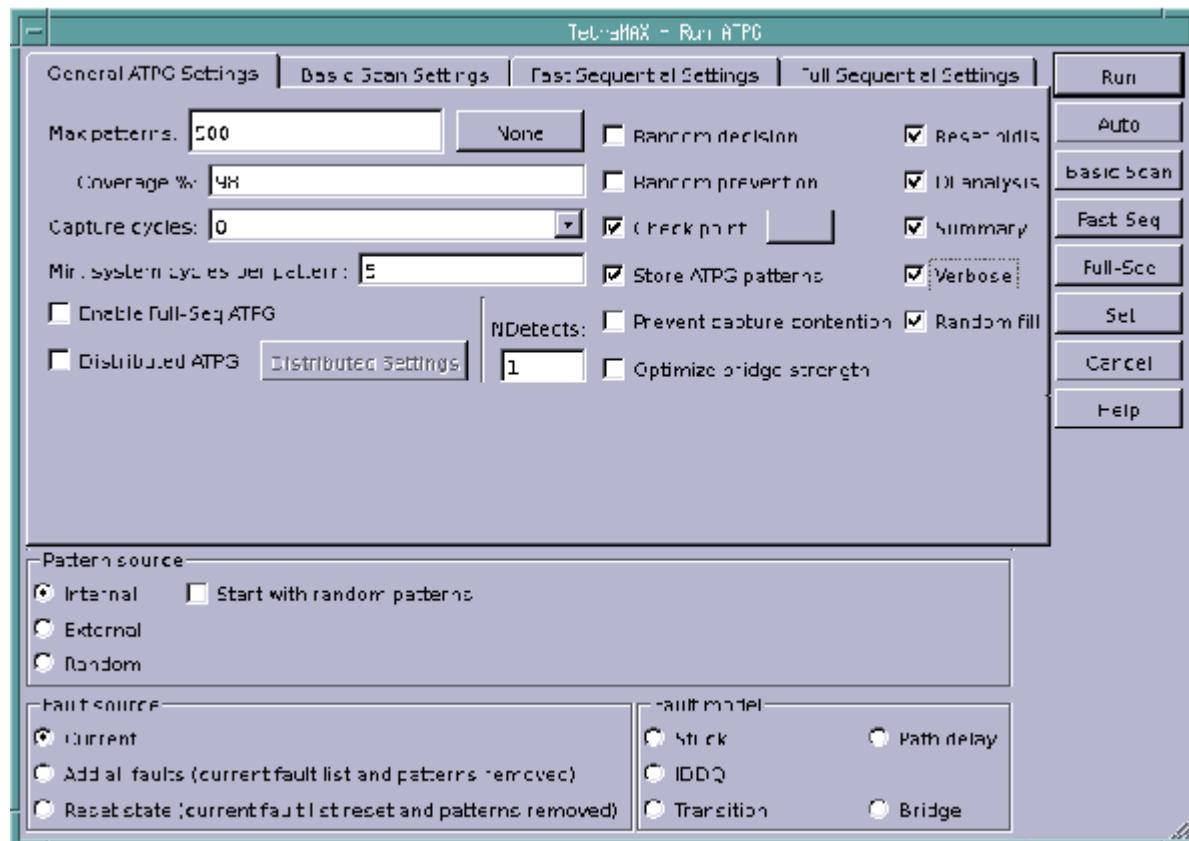
- The following command specifies that each pattern must have minimum of 5 system cycles and to report extras messages during the pattern merge operation:
`set_atpg -min_ateclock_cycles 5 -verbose`
- The following command specifies TetraMAX to use the random decision method when compressing patterns and to save patterns to the chkp_patt file every 360 CPU seconds:
`set_atpg -checkpoint {360 chkp_patt}`

The following steps make the same settings specified in the previous examples using the Run ATPG dialog box:

1. Do one of the following:
 - Select Run > Run ATPG
 - Click the ATPG button in command barIn both cases, the Run ATPG dialog box appears.
2. Click the General ATPG Settings tab, then do the following:
 - a. Enter 500 in the Max patterns text field.
 - b. Enter 98 in the Coverage % text field.
 - c. Enter 5 in the Min. system cycles per pattern text field.
 - d. Click the Verbose check box.
 - e. Click the Random fill check box.
 - f. Click the Check point check box. In the Set Check Point dialog box, enter chkp_patt in the Pattern file name text field and 360 in the Time interval text field.
 - g. Click OK.

[Figure 1](#) shows the appearance of the Run ATPG dialog box after entering the specifications from the previous steps (note that the default settings are also selected).

Figure 1 General Pattern Generation Options in the Run ATPG Dialog Box



Specifying Fault Lists

TetraMAX ATPG maintains a list of potential faults for a design. You can specify TetraMAX ATPG to use an existing fault list provided in a formatted ASCII file, create a fault list, or use only particular faults.

For complete information on using faults and fault lists, see "[Working with Faults and Fault Lists](#)."

The following sections show several methods for specifying and creating fault lists:

- [Selecting an Existing Fault List File](#)
- [Generating a Fault List Containing All Fault Sites](#)
- [Including Specific Faults in a Fault List](#)
- [Writing Faults to a File](#)
- [Example Fault Lists](#)

Selecting an Existing Fault List File

To specify TetraMAX ATPG to use an existing fault list file, do one of the following:

- Use the `read_faults` command, as shown in the following example:

```
TEST-T> read_faults spec_faults.all
```

- Use the Add Faults dialog box by doing the following:

1. Select Faults > Add Faults

The Add Faults dialog box appears.

2. Click the Read File button, and enter or browse and select the name of the fault file.
3. Click OK

Generating a Fault List Containing All Fault Sites

To generate a fault list that includes all possible fault sites in the ATPG design model, do one of the following:

- Specify the `add_faults` command, as shown in the following example:

```
TEST-T> add_faults -all
```

- Use the Add Faults dialog box by doing the following:

1. Select Faults > Add Faults

The Add Faults dialog box appears.

2. Click the All button.

3. Click OK

Including Specific Faults in a Fault List

You can exclude specific blocks, instances, gates, or pins from the fault list using any of the following methods:

- Specify objects to be excluded using the `add_nofaults` command and then execute the `add_faults -all` command, as shown in the following example:

```
TEST-T> add_nofaults /sub_block_A/adder  
TEST-T> add_nofaults /io/demux/alu  
TEST-T> add_faults -all
```

- Remove faults based on fault locations in a fault list file specified by the `add_faults -all` command, as shown in the following example:

```
TEST-T> add_faults -all  
TEST-T> read_faults fault_list_file -delete
```

- Remove faults using the `remove_faults` command after executing the `add_faults -all` command, as shown in the following example:

```
TEST-T> add_faults -all  
TEST-T> remove_faults /sub_block_A/adder  
TEST-T> remove_faults /io/demux/alu
```

- If you have a small number of faults, you can add them explicitly using the `add_faults` command:

```
TEST-T> remove_faults -all
TEST-T> add_faults /proc/io
TEST-T> add_faults /demux
TEST-T> add_faults /reg_bank/bank2/reg5/Q
```

Note: You can perform these same tasks in the TetraMAX GUI using the Add Faults, Add No Faults, Remove Faults dialog boxes.

Writing Faults to a File

You can use the `write_faults` command or the Report Faults dialog box to write a fault list to a file for analysis or to read back in for future ATPG sessions:

- Write a fault list containing only AU class faults, as shown in the following example:
`TEST-T> write_faults faults.AU -class au -replace`
- Write a fault list for all faults:
`TEST-T> write_faults filename -all -replace`
- Write only the undetectable blocked (UB) and undetectable redundant (UR) fault classes:
`TEST-T> write_faults filename -class UB -class UR -replace`
- Write only the faults down one hierarchical path:
`TEST-T> write_faults filename /top/demux/core/mul8x8 -replace`
- By default, the list of faults is either collapsed or uncollapsed as determined by the last `set_faults -report` command. The following command overrides the default by using the `-collapsed` option:
`TEST-T> write_faults filename -all -replace -collapsed`
- Generate a fault list using the Report Faults dialog box:
 1. From the menu bar, choose Faults > Report Faults.
 The Report Faults dialog box appears.
 2. Use the Report Type list box to select the type of fault report that you want. A set of additional options might appear to the right of the Report Type list box, depending on your selection.
 3. Select the options you want.
 4. Click OK.

Example Fault Lists

[Example 1](#) shows a typical uncollapsed fault list. The equivalent faults always immediately follow the primary fault and are identified by two dashes (--) in the second column.

Example 1 Uncollapsed Fault List

```
sa0 NP /moby/bus/Logic0206/N01
sa0 -- /moby/bus/Logic0206/H01
sa0 -- /xyz_nwr
sa0 NP /moby/i278/N01
sa0 -- /moby/i278/H01
```

```
sa0 -- /moby/i337/N01
sa0 -- /moby/i337/H02
sa1 -- /moby/i337/H01
sa0 -- /moby/i222/N01
sa0 -- /moby/i222/H01
sa0 -- /moby/i222/H02
sa0 NP /moby/core/PER/PRT_1/POUTMUX_1/i411/N01
sa0 -- /moby/core/PER/PRT_1/POUTMUX_1/i411/H03
sa0 -- /moby/core/PER/PRT_1/POUTMUX_1/i411/H04
sa1 -- /moby/core/PER/PRT_1/POUTMUX_1/i411/H01
sa1 -- /moby/core/PER/PRT_1/POUTMUX_1/i411/H02
```

For comparison, [Example 2](#) shows the same fault list written with the `-collapsed` option specified.

Example 2 Collapsed Fault List

```
sa0 NP /moby/bus/Logic0206/N01
sa0 NP /moby/i278/N01
sa0 NP /moby/core/PER/PRT_1/POUTMUX_1/i411/N01
```

See Also

[Fault Lists and Faults](#)

Specifying Fault Models

Effective testing requires an accurate behavioral description of a design containing defects. Fault models represent how a manufacturing defect affects a design, and are crucial in identifying target faults and performing fault analysis. You can run TetraMAX ATPG using any of the following fault models:

- **Stuck-At** — This is the default model used by TetraMAX ATPG, and is the industry standard model used for generating test patterns. This model assumes that a circuit defect behaves as a node stuck at either 0 or 1. The test pattern generator attempts to propagate the effects of these faults to the primary outputs and scan cells of the device, where they can be observed at a device output or captured in a scan chain. For more information on the stuck-at fault model and fault models in general, see "[Understanding Fault Models](#)".
- **Transition Delay** — Generates test patterns to detect single-node slow-to-rise and slow-to-fall faults. Using this model, TetraMAX ATPG launches a logical transition upon completion of a scan load operation and uses a capture clock procedure to observe the transition results. For more information, see "[Transition-Delay Fault ATPG](#)".
- **Path Delay** — Tests and characterizes critical timing paths in a design. Path delay fault tests exercise the critical paths at-speed (the full operating speed of the chip) to detect whether the path is too slow because of manufacturing defects or variations. For more information, see "[Path Delay Fault and Hold Time Testing](#)".

- **Hold Time** — This model is similar to the transition delay and path delay models, except that it detects a fault through the shortest possible path to increase the probability of finding small delay defects or process variations. For more information, see "[Hold Time ATPG Test Flow](#)."
- **IDDQ** — Assumes that a circuit defect causes excessive current drain due to an internal short circuit from a node to ground or to a power supply. For this model, TetraMAX ATPG does not attempt to observe the logical results at the device outputs. Instead, it tries to toggle as many nodes as possible into both states while avoiding conditions that violate quiescence, so that defects can be detected by the excessive current drain that they cause. For more information, see "[Quiescence Test Pattern Generation](#)."
- **Bridging** — Detects shorts that cause a connection between two normally unconnected signals. These defects can be detected if one of the nets (the aggressor) causes the other net (the victim) to take on a faulty value, which can then be propagated to an observable location. For more information, see "[Bridging Fault ATPG](#)."
- **IDDQ Bridging** — Uses the IDDQ model to generate additional patterns and increase the IDDQ coverage. The IDDQ bridging model uses only the toggle version of the standard IDDQ model, which means that the fault site at a gate input does not require propagation to an output of the same gate to be identified as a fault. For more information, see "[IDDQ Bridging](#)."
- **Dynamic Bridging** — Combines components of the static bridging fault model and the transition fault model to analyze transition effects in the presence of a specified value on a bridge aggressor node. For more information, see "[Running the Dynamic Bridging Fault ATPG Flow](#)."

Selecting a Fault Model

TetraMAX ATPG uses the stuck-at fault model by default. You can select any supported fault model using the `-model` option of the `set_faults` command or the Set Faults or Run ATPG dialog boxes.

The following example shows how to use the `set_faults` command to specify the transition-delay fault model:

```
TEST-T > set_faults -model transition
```

The following table shows the keywords used with the `-model` option to specify the various fault models:

Keyword	Fault Model
<code>stuck</code>	Stuck-At
<code>iddq</code>	IDDQ
<code>iddq_bridging</code>	IDDQ Bridging
<code>transition</code>	Transition-Delay

Keyword	Fault Model
path_delay	Path Delay
hold_time	Hold Time
bridging	Bridging
dynamic_bridging	Dynamic Bridging

The following sets of steps show you how to specify a fault model using the Set Faults dialog box:

1. Select Faults > Set Fault Options from the menu bar.
The Set Faults dialog box appears.
2. In the Model section, click the button associated with the fault model you want to use.
3. Click OK.

To use the Run ATPG dialog box to specify a fault model:

1. Do one of the following:
 - Select Run > Run ATPG from the menu bar
 - Click the ATPG button in the command toolbar
 In both cases, the Run ATPG dialog box appears.
2. In the Fault model section, click the button associated with the fault model you want to use.
3. Click the Set button to save your specification.

Specifying the Pattern Source

You can configure TetraMAX ATPG to use the following pattern sources:

- **Internal patterns** - These patterns are stored in memory and generated internally by TetraMAX ATPG. You can identify internal patterns by running the `report_patterns` command.
- **External patterns** - These patterns are stored in a file. TetraMAX ATPG can read external pattern files in several formats, including Verilog, VHDL, STIL, and WGL. These patterns must use the same syntax TetraMAX uses when it writes patterns.
- **Random patterns** - These patterns are defined by parameters set by the `set_random_patterns` command)

Patterns should be stored in a binary file, if possible. The WGL and STIL formats are unable to accurately store all the pattern data required by TetraMAX ATPG. When you read back a STIL or WGL pattern file, the fast-sequential patterns might be interpreted as a full-sequential patterns, and errors are reported. Do not assume that TetraMAX ATPG can correctly read STIL or WGL patterns created by tools other than TetraMAX ATPG.

The following sections describe the pattern types and formats accepted by TetraMAX ATPG, including how to select the pattern source:

- [Scan and Nonscan Functional Patterns](#)
- [STIL Functional Pattern Format](#)
- [Verilog Functional Pattern Format](#)
- [WGL Functional Pattern Format](#)
- [VCDE Functional Pattern Format](#)
- [Options for Selecting the Pattern Source](#)

Scan and Nonscan Functional Patterns

TetraMAX ATPG accepts two primary types of functional pattern files:

- **Scan functional patterns** - These patterns contain scan chain load and unload sequences and define structures and procedures that can be recognized as scan-chain related. They must use the same style and format that TetraMAX ATPG uses to write ATPG patterns. All scan chains, clocks, and primary input constraints must match the usage in the patterns. The load_unload, Shift, and other test procedures must be consistent with the patterns.
- **Nonscan functional patterns** - These patterns have no recognizable structure and do not contain procedures of a standard scan pattern. Nonscan patterns may exercise scan chains, may be completely functional, or may perform a combination of scan chain and functional testing. They must use a simple, sequential application of input stimulus and output measures and they must not define scan chains or any ATPG-related procedures (for example, load_unload or Shift).

If the functional nonscan patterns do not contain timing information, you can use a STIL procedure file to define pin timing, and reference the STIL procedure file using the `run_drc` command or the Run DRC dialog box. The following steps describe this process:

1. For your current design, use the `add_clocks` command or the Add Clock dialog box to define as clocks all ports in the input data that function as clocks or pulsed ports.
Note that defining the clocks is optional. Some clock violations found during the `run_drc` process can affect the simulator and it might be necessary to remove `add_clocks` commands.
2. Switch to TEST mode without the use of an STIL procedure file. Typically, you must change the severity of many of the rules from their defaults to either warning or ignore.
3. After you achieve TEST mode, execute `run_atpg` to generate at least one pattern.
4. Write out a few patterns. Because no scan chains have been defined, this pattern file represents a template for nonscan functional pattern input.

STIL Functional Pattern Input

TetraMAX ATPG accepts pattern input in STIL format using some limited variations of the example shown in [Example 1](#).

The supported format has the following characteristics:

- The Header block is optional.
- The Signals block is required.
- The SignalGroups block is optional.
- The Timing block, with at least one WaveformTable, is required to define the point in the cycle where the clocks pulse and the outputs are measured.
- The PatternBurst, PatternExec, and Pattern blocks are used to set up a single block of functional patterns.
- The Pattern block consists only of w and v statements.

Example 1 Functional Pattern Input in STIL

```

STIL 0.23;
Header { Title "Functional Patterns for Design-X"; }
Signals {
    d11 In;    d10 In;    d9 In;    d8 In;    d7 In;
    d6 In;    d5 In;    d4 In;    d3 In;    d2 In;
    d1 In;    d0 In;    i3 In;    i2 In;    i1 In;
    i0 In;    oe In;    rld In;    ccen In;    ci In;
    cp In;    cc In;    sdi1 In;    sdi2 In;    se In;
    tsel In;    y11 Out;    y10 Out;    y9 Out;    y8 Out;
    y7 Out;    y6 Out;    y5 Out;    y4 Out;    y3 Out;
    y2 Out;    y1 Out;    y0 Out;    full Out;    pl Out;
    map Out;    vect Out;    sdol Out;    sdo2 Out;    tout Out;
    vcoctl Out;
}
SignalGroups {
    input_ports = 'd11 + d10 + d9 + d8 + d7 + d6 + d5 + d4 + d3 + d2
                  + d1 + d0 + i3 + i2 + i1 + i0 + oe + rld + ccen + ci
                  + cp + cc + sdi1 + sdi2 + se + tsel';
    output_ports = 'y11 + y10 + y9 + y8 + y7 + y6 + y5 + y4 + y3 +
y2
                  + y1 + y0 + full + pl + map + vect + sdol + sdo2 + tout
                  + vcoctl';
}
Timing {
    WaveformTable TSET1 {
        Period '250ns';
        Waveforms {
            input_ports { 01Z { '0ns' D/U/Z; } }
            cp { P { '0ns' D; '62ns' U; '187ns' D; } }
            output_ports { X { '0ns' X; } }
            output_ports { LHT { '0ns' X; '240ns' L/H/T; } }
        }
    }
}
PatternBurst functional_burst { FUNC_BLOCK_1; }
PatternExec { Timing; PatternBurst functional_burst; }

```

```

Pattern FUNC_BLOCK_1 {
    W TSET1;
    V {
        d1=0; d9=0; sdo2=X; sdi2=0; y9=X; y1=X; d6=0; cp=0; i3=0; cc=0;
        vcoctl=X; y6=X; ci=1; d3=0; i0=0; d11=0; y3=X; y11=X; oe=0;
        d0=0;
        d8=0; vect=H; map=H; y8=X; y0=X; i2=0; d5=0; sdo1=X;
        y5=X; sdi1=0;
        tout=X; d2=0; y2=X; d7=0; d10=0; full=X; y7=X; tsel=0; ccen=0;
        se=0;
        y10=X; rld=0; i1=0; d4=0; y4=X; }
        V {tsel=1; tout=T;}
        V {d3=1; y3=H; map=L; i1=1;}
        V {sdo2=L; d3=0; i0=0; y0=H; i2=1;}
        V {sdo2=H; y1=L; i0=1; y3=L; i2=0; d4=1; y4=H;}
        V {y1=H; i3=0; cc=1; y0=L; d4=0;}
        V {y0=H; d10=1;}
        V {sdo2=L; y1=H; i0=0; i2=1;}
        V {y0=H;}
        V {y0=H;}
        V {y1=H; y0=L;}
        V {y0=H;}
        V {y1=L; y3=H; y0=L; sdo1=L; y2=L;}
        V {y0=H; full=L;}
        V {y1=L; y0=L; y2=H;}
        V {y1=H; y0=L;}
        V {y1=L; y3=L; y0=L; y2=L; y4=H;}
        V {y0=H;}
    }
}

```

Verilog Functional Pattern Input

TetraMAX ATPG accepts pattern input in Verilog format using some limited variations of the example shown in [Example 2](#).

The supported format has the following characteristics:

- `timescale is optional.
- A vector is used for primary outputs, expected data, and mask.
- Each clock capture cycle that can perform a measure is defined in an event procedure.
- Cycles with a measure and no clocks are defined in event procedures.
- Cycles with a clock and no measures are defined in event procedures.
- The data stream occurs within an initial/end block.
- Assignment to the variable pattern allows TetraMAX ATPG to track the pattern boundaries.

Example 2 Functional Pattern Input in Verilog

```
`timescale 1 ns / 100 ps
module amd2910_test;
```

```

reg [0:8*9] POnames [19:0];
integernofails, bit, pattern;
wire [11:0] d;
wire [3:0] i;
wire oe, tsel, ci, rld, ccen, cc, sdil, sdi2, se, cp;
wire [11:0] y;
wire full, pl, map, vect, tout, vcoctl, sdo1, sdo2;
wire [19:0] PO;           // primary output vector
reg [19:0] XPCT;         // expected data vector
reg [19:0] MASK;         // compare mask vector
assign PO[0] = y[0];
assign PO[1] = y[1];
assign PO[2] = y[2];
assign PO[3] = y[3];
assign PO[4] = y[4];
assign PO[5] = y[5];
assign PO[6] = y[6];
assign PO[7] = y[7];
assign PO[8] = y[8];
assign PO[9] = y[9];
assign PO[10] = y[10];
assign PO[11] = y[11];
assign PO[12] = full;
assign PO[13] = pl;
assign PO[14] = map;
assign PO[15] = vect;
assign PO[16] = tout;
assign PO[17] = vcoctl;
assign PO[18] = sdo1;
assign PO[19] = sdo2;

// instantiate the device under test

amd2910 dut (.o_y11( y[11] ), .o_y10( y[10] ), .o_y9( y[9] ),
.o_y8( y[8] ), .o_y7( y[7] ), .o_y6( y[6] ), .o_y5( y[5] ),
.o_y4( y[4] ), .o_y3( y[3] ), .o_y2( y[2] ), .o_y1( y[1] ),
.o_y0( y[0] ), .o_full(full), .o_pl(pl), .o_map(map),
.o_vect(vect), .o_sdo1(sdo1), .o_sdo2(sdo2), .tout(tout),
.vcoctl(vcoctl), .i_d11( d[11] ), .i_d10( d[10] ), .i_d9(
d[9] ),
.i_d8( d[8] ), .i_d7( d[7] ), .i_d6( d[6] ), .i_d5( d[5] ),
.i_d4( d[4] ), .i_d3( d[3] ), .i_d2( d[2] ), .i_d1( d[1] ),
.i_d0( d[0] ), .i_i3( i[3] ), .i_i2( i[2] ), .i_i1( i[1] ),
.i_i0( i[0] ), .i_oe(oe), .i_rld(rld), .i_ccen(ccen),
.i_ci(ci), .i_cp(cp), .i_cc(cc), .i_sdil(sdil),
.i_sdi2(sdi2),
.i_se(se), .tsel(tsel) );

// define pulse on "i_cp"
event pulse_i_cp;
always @ pulse_i_cp begin

```

```

#500 cp = 1;
#100 cp = 0;
end

// define capture event without a clock
event capture;
always @ capture begin
#0;
#950; ->measurePO;
end

// define how to measure outputs
event measurePO;
always @ measurePO begin
if ((XPCT&MASK) !== (PO&MASK)) begin
$display($time," ----- ERROR(S) during pattern %0d ---",
--",pattern);
for (bit = 0; bit < 20; bit=bit + 1) begin
if((XPCT[bit]&MASK[bit]) !== (PO[bit]&MASK[bit]))
begin
$display($time, " : %0s (output %0d), expected %b,
got %b",
POnames[bit], bit, XPCT[bit],
PO[bit]);
nofails =nofails + 1;
end
end
end
end

event capture_i_cp;
always @ capture_i_cp begin
#0;
#500 cp = 1; // i_cp
#100 cp = 0;
#350; ->measurePO;
end

initial begin
nofails = 0;
// --- initialize port name table
POnames[0] = "Y0"; POnames[1] = "Y1"; POnames[2] = "Y2";
POnames[3] = "Y3"; POnames[4] = "Y4"; POnames[5] = "Y5";
POnames[6] = "Y6"; POnames[7] = "Y7"; POnames[8] = "Y8";
POnames[9] = "Y9"; POnames[10] = "Y10"; POnames[11] = "Y11";
POnames[12] = "full"; POnames[13] = "pl"; POnames[14] = "map";

POnames[15] = "vect"; POnames[16] = "tout"; POnames[17] =
"vcoctl";
POnames[18] = "sdo1"; POnames[19] = "sdo2";

```

```

#0; pattern= 0;
se=0; sdi2=0; sdi1=0; cc=0; ccen=0; ci=0; tsel=0; oe=0;
cp = 0; i=4'b0010; rld=1; d=12'b0000000000111;
XPCT=20'bXXXX101100000000001; MASK=20'b0000000000000000000000000000000;
->pulse_i_cp;

#1000; pattern= 1; i=4'b1110; d=12'b0000000000000;
->pulse_i_cp;

#1000; pattern= 2; i=4'b0000; oe=0;
->capture;

#1000; pattern= 3; i=4'b0010; oe=1;
d=12'b000000000001; XPCT=20'bXXXX1011000000000001;
MASK=20'b00001111111111111111;
->capture_i_cp;

#1000; pattern= 4;
d=12'b000000000010; XPCT=20'bXXXX1011000000000010;
MASK=20'b00001111111111111111;
->capture_i_cp;

#1000; pattern= 5;
d=12'b0000000000100; XPCT=20'bXXXX1011000000000100;
MASK=20'b00001111111111111111;
->capture_i_cp;

#1000;
$display("Simulation of %0d cycles completed with %0d errors",
         pattern, nofails );
$finish;
end
endmodule

```

WGL Functional Pattern Input

TetraMAX ATPG accepts pattern input in WGL format using some limited variations of the example shown in [Example 3](#).

The supported format has the following characteristics:

- The `waveform` function is required.
- The `pmode` function is optional.
- The `signal` block is required.
- The `timeplate` block is required.
- The `pattern` block consists of simple vectors applied sequentially.

Example 3 Functional Pattern Input in WGL

```

waveform funct_1
pmode[last_drive];
signal

```

```

TEST : input; RESET_B : input; EXTS1 : input; EXTS0 : input;
LOBAT : input; SS_B : input; SCK : input; MOSI : input;
EXTAL : input; TOUTEN : input; TOUTSEL : input;
XTAL : output; MISO : output; READY_B : output;
CLKOUT : output; SYMCLK : output; S7 : output;
S6 : output; S5 : output; S4 : output;
S3 : output; S2 : output; S1 : output;
S0 : output; TOUT3 : output; TOUT2 : output;
TOUT1 : output; TOUT0 : output;
end

timeplate tts0 period 500ns
TEST := input[0pS:P, 200nS:S];
RESET_B := input[0pS:P, 200nS:S];
EXTS1 := input[0pS:P, 200nS:S];
EXTS0 := input[0pS:P, 200nS:S];
LOBAT := input[0pS:P, 200nS:S];
SS_B := input[0pS:P, 200nS:S];
SCK := input[0pS:P, 200nS:S];
MOSI := input[0pS:P, 200nS:S];
EXTAL := input[0pS:P, 100nS:S];
TOUTEN := input[0pS:P, 200nS:S];
TOUTSEL := input[0pS:P, 200nS:S];
XTAL := output[0pS:X, 450nS:Q, 451nS:X];
MISO := output[0pS:X, 450nS:Q, 451nS:X];
READY_B := output[0pS:X, 450nS:Q, 451nS:X];
CLKOUT := output[0pS:X, 450nS:Q, 451nS:X];
SYMCLK := output[0pS:X, 450nS:Q, 451nS:X];
S7 := output[0pS:X, 450nS:Q, 451nS:X];
S6 := output[0pS:X, 450nS:Q, 451nS:X];
S5 := output[0pS:X, 450nS:Q, 451nS:X];
S4 := output[0pS:X, 450nS:Q, 451nS:X];
S3 := output[0pS:X, 450nS:Q, 451nS:X];
S2 := output[0pS:X, 450nS:Q, 451nS:X];
S1 := output[0pS:X, 450nS:Q, 451nS:X];
S0 := output[0pS:X, 450nS:Q, 451nS:X];
TOUT3 := output[0pS:X, 450nS:Q, 451nS:X];
TOUT2 := output[0pS:X, 450nS:Q, 451nS:X];
TOUT1 := output[0pS:X, 450nS:Q, 451nS:X];
TOUT0 := output[0pS:X, 450nS:Q, 451nS:X];
end

pattern group_ALL (TEST,RESET_B,EXTS1,EXTS0,LOBAT,SS_B,
                    SCK,MOSI,EXTAL,TOUTEN,TOUTSEL,XTAL,
                    MISO,READY_B,CLKOUT,SYMCLK,S7,S6,
                    S5,S4,S3,S2,S1,S0,TOUT3,TOUT2,
                    TOUT1,TOUT0)
vector(0, 0pS, tts0)  := [0 0 0 0 0 0 0 0 0 0 0 0 X X X X X X X X
X X X X X X X X X ] (0pS);
vector(1, 500nS, tts0) := [0 0 0 0 0 0 0 0 0 0 0 0 X X X X X X X X
X X X

```

```

X X
    X X X X ] (500ns);
vector(2, 1uS, tts0) := [0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 X 0 Z Z
Z Z Z Z Z Z Z Z Z Z ] (1uS);
vector(3, 1.5uS, tts0) := [0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 X 0 Z Z Z
Z Z Z
Z Z
    Z Z Z Z ] (1.5uS);
vector(4, 2uS, tts0) := [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 X 0 Z Z
Z Z Z Z Z Z Z Z Z Z ] (2uS);
vector(5, 2.5uS, tts0) := [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 X 0 Z Z Z
Z Z Z
Z Z
    Z Z Z Z ] (2.5uS);
vector(6, 3uS, tts0) := [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 X 0 Z Z
Z Z Z Z Z Z Z Z Z Z ] (3uS);
vector(7, 3.5uS, tts0) := [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 X 0 Z Z Z
Z Z Z
Z Z
    Z Z Z Z ] (3.5uS);
vector(8, 4uS, tts0) := [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 X 0 Z Z
Z Z Z Z Z Z Z Z Z Z ] (4uS);
vector(9, 4.5uS, tts0) := [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 X 0 Z Z Z
Z Z Z
Z Z
    Z Z Z Z ] (4.5uS);
vector(10, 5uS, tts0) := [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 X 0 Z Z
Z Z Z Z Z Z Z Z Z Z ] (5uS);
end
end

```

VCDE Functional Pattern Input

TetraMAX ATPG can read patterns in extended VCD (VCDE) format. This format is not the same as traditional VCD. To create a VCDE data file, you need a Verilog-compatible simulator that supports the IEEE draft definition of VCDE. (For details, refer to IEEE P1364.1-1999, *Draft Standard for Verilog Register Transfer Level Synthesis*.) The Synopsys VCS simulator, version 5.1 or later, supports this standard.

Creating a VCDE data file is fairly simple for most Verilog simulators. You need to add a single `$dumpports()` system task to the `initial` block of the top-level module. The syntax is similar to the following:

```

initial begin
    //
    // --- other variable inits here
    //
    $dumpports( testbench.DUT, "vcde_output_file");
    ...
end

```

In this example, the simulator captures all of the I/O events for the simulation instance `testbench.DUT` into a file called `vcde_output_file`. If your simulation is performed directly on your design, the path to this file may be `DUT`. If your design is instantiated in a testbench, then this path is more likely to be `testbench.DUT`, where `testbench` is the top-level module name and `DUT` is the instance name of the design found within the module `testbench`.

If you want to generate a VCDE file from a TetraMAX Verilog testbench, you can use the `+define+tmx_vcde` variable to help generate that file. Do this by adding the `+define+tmx_vcde` variable to your VCS command line when you simulate the TetraMAX ATPG-generated Verilog testbench. An VCDE file called `sim_vcde.out` is automatically created.

Do not create a VCDE file with complex timing events. The most efficient functional patterns are those most closely resembling what would be applied on a tester. Within a cycle, use as few separate events as possible as in the following sequence:

1. Force all inputs at the same time.
2. Pulse the clock.
3. Measure all outputs at the same time.

Functional patterns in VCDE format do not need to have any measures defined. TetraMAX ATPG decides what values to expect on output and bidirectional pins by keeping a running tally of the most recently reported values in the VCDE event stream. For an output port, all values other than X are measurable. For a bidirectional port, the values L, H, T, I, and h are measurable; the value X is not measured; and the values 0, 1, and Z indicate input mode (which is not measurable).

In TetraMAX ATPG, when you read in VCDE patterns, you specify the cycle period and measure points within each cycle. TetraMAX ATPG uses this information to construct internal measure points and expected data. For more information, see ["Specifying Strobes for VCDE Pattern Input."](#)

Options for Selecting the Pattern Source

You can select the pattern source using the `set_patterns` command, the Set Patterns dialog box, or the Run ATPG dialog box. In addition, you can specify various options that affect how TetraMAX ATPG uses the patterns.

The following examples show how to use the `set_patterns` command to specify the pattern source:

- To use internal patterns, specify the `-internal` option, as shown in the following example:

```
TEST-T> set_patterns -internal
```
- To use external patterns, specify the `-external` option and the name of the file containing the patterns (in the following example, the file name is `b010.vi1`). You can also use the `-append` option to append the external patterns to any existing internal patterns, and use the `-load_summary` option to enable the `report_summaries` command to display the total number of scan loads used by the basic-scan and fast-sequential

patterns, as shown in the following example:

```
TEST-T> set_patterns -external b010.vil -append -load_summary
```

- To use random patterns, do the following:

1. Specify the parameters for defining the random patterns using the `set_random_patterns` command, as shown in the following example:

```
TEST-T> set_random_patterns -length 3000 -observe_master
```

2. Specify the `-random` option of the `set_patterns` command, as shown in the following example:

```
TEST-T> set_patterns -random
```

The Set Patterns dialog box generally uses the same options specified by the `set_patterns` command. To select the pattern source using the Set Patterns dialog box:

1. From the menu bar, choose Patterns > Set Pattern Options.
The Set Patterns dialog box appears.
2. Do one of the following:
 - To select internal patterns as the pattern source, click the Internal button in Pattern source section.
 - To select external patterns, do the following:
 - a. Click the External button in the Pattern source section.
 - b. Enter the pattern file name you want to use as the pattern source, or use the Browse button to navigate and select the file.
 - To select random patterns, click the Random button.
3. Select or enter any applicable options in the Set Patterns dialog box.
4. Click OK.

To select the pattern source using the Run ATPG dialog box:

1. Click the ATPG button in the command toolbar.
The Run ATPG dialog box appears.
2. Do one of the following:
 - To select internal patterns as the pattern source, click the Internal button in Pattern source section.
 - To select external patterns, do the following:
 - a. Click the External button in the Pattern source section.
 - b. Enter the pattern file name you want to use as the pattern source, or use the Browse button to navigate and select the file.
 - To select random patterns, click the Random button.
3. Select or enter any applicable options in the Set Patterns dialog box.
4. Click OK.

Specifying the ATPG Mode

TetraMAX ATPG can use three different modes when performing pattern generation. Each mode provides different types and levels of optimization. Since ATPG normally requires multiple runs, the mode you select depends on your particular pattern generation goals and where you are in the ATPG process.

TetraMAX ATPG supports the following ATPG modes:

- **Basic Scan Mode** - This is the default mode for TetraMAX ATPG, and is usually the first mode you run. It enables TetraMAX ATPG to operate as a full-scan, combinational-only ATPG tool. To get high test coverage, the sequential elements must be scan elements.
- **Fast-Sequential Mode** - This mode provides limited support for partial-scan designs, and accommodates multiple capture procedures between scan load and scan unload. Fast-sequential mode allows data to be propagated through nonscan sequential elements in the design, such as functional latches, nonscan flops, and RAMs and ROMs. However, all clock and reset signals to these nonscan elements must be directly controllable at the primary inputs of the device.
- **Full-Sequential Mode** - This mode supports multiple capture cycles between scan load and unload, which increases test coverage in partial-scan designs. Clock and reset signals to the nonscan elements do not need to be controllable at the primary inputs and there is no specific limit on the number of capture cycles used between scan load and unload.

Each mode is described in more detail in the following sections:

- [Setting Basic Scan Mode](#)
- [Setting Fast-Sequential Mode](#)
- [Setting Full-Sequential Mode](#)

Basic Scan Mode Settings

The basic scan mode is the default mode for running ATPG. This mode uses the combinational ATPG method, which tests the individual nodes (or flip-flops) of a logic circuit without concern to the overall operation of the circuit. During test, basic-scan mode forces a simplified connection of flip-flops that effectively bypasses their normal interconnections. This allows TetraMAX ATPG to use a relatively simple vector matrix to quickly test all the comprising flip-flops and to trace failures to specific flip-flops.

You can use the `set_atpg` command or the Run ATPG dialog box to set several options specific to basic-scan mode. For example, you can do the following:

- Specify the `-abort_limit` option to set the maximum number of remade decisions before terminating a basic-scan test generation effort.
- Specify the `-resim_atpg_patterns` option to enable and disable the resimulation of patterns generated by basic-scan ATPG to increase the robustness of patterns.

The following example shows how to specify both options:

```
TEST> set_atpg -abort_limit 8 -resim_atpg_patternsnofault_sim
```

To perform these same tasks using the Run ATPG dialog box:

1. Do one of the following:
 - Select Run > Run ATPG from the command menu
 - Click the ATPG button in the command toolbar.

In both cases, the Run ATPG dialog box appears.

2. Click the Basic Scan Settings tab.
3. Enter 8 in the Abort limit field, and select Mask in the drop-down menu of the Resim basic scan patterns field.
4. Click the Set button.

After making the appropriate settings, you can run ATPG in basic scan mode. For details on this process, see "[Running TetraMAX in Basic Scan, Fast-Sequential, or Full-Sequential Mode](#)".

Fast-Sequential Mode Settings

Fast-sequential ATPG provides limited support for partial-scan designs (designs containing some nonscan sequential elements). This mode is particularly useful when there are AU (ATPG Undetectable) faults remaining after you run ATPG in basic-scan mode.

You can use the `-capture_cycles` option of the `set_atpg` command to specify an integer between 2 and 10. This specification sets the level of effort used by the ATPG algorithm based on the number of capture procedures allowed between scan load and unload.

You should not set the `-capture_cycles` option value too high since it can cause excessive runtimes. In most cases, you should use a starting value of 4 (the default), and generate an initial set of patterns. You can then incrementally increase the value following each pattern generation until you achieve your required coverage.

You can use the `sequential_depths` options of the `report_summaries` command to identify the maximum depth for controlling, observing, and detecting faults, as shown in the following example:

```
TEST> report_summaries sequential_depths
      type depth gate_id
      -----
Control    1      21569
Observe    2      6866
Detect     3      6859
```

Based on this report, to obtain optimal runtime you should set the `-capture_cycles` option to 3 as shown in the following example:

```
TEST> set_atpg -capture_cycles 3
```

For optimal coverage, set the `-capture_cycles` option to 10:

```
TEST> set_atpg -capture_cycles 10
```

To perform these same tasks using the TetraMAX GUI:

1. Do one of the following:
 - Select Report > Report Summaries from the command menu
 - Click the Summary button in the command toolbar.In both cases, the Report Summaries dialog box appears.
2. Select the Sequential depths button.
3. In the Output to section, select either the Report window, Transcript, or File.
4. Click OK
The Report Summaries dialog prints a report that shows the sequential depths.
5. Do one of the following:
 - Select Run > Run ATPG from the command menu
 - Click the ATPG button in the command toolbar.In both cases, the Run ATPG dialog box appears.
6. Click the Fast Sequential Settings tab.
7. Enter a value in the Capture cycle field (for example, enter 4).
8. Click the Set button.

Setting Full-Sequential Mode

Full-sequential ATPG supports multiple capture cycles between scan load and unload, and supports RAM and ROM models, which increases the test coverage in partial-scan designs (similar to fast-sequential ATPG). However, in full-sequential mode, clock and reset signals to the nonscan elements do not need to be controllable at the primary inputs and there is no specific limit on the number of capture cycles used between scan load and unload.

To enable TetraMAX ATPG to use the full-sequential mode, specify the `-full_seq_atpg` option of the `set_atpg` command, as shown in the following example:

```
TEST-T> set_atpg -full_seq_atpg
```

Full-sequential mode supports a feature called *sequential capture*. If you define a sequential capture procedure in the STIL procedure file, you can customize the capture clock sequence applied to the device during full-sequential ATPG. For example, you can define the clocking sequence for a two-phase latch design, in which CLKP1 is followed by CLKP2. Otherwise, the tool creates its own sequence of clocks and other signals to target the as-yet-undetected faults in the design. For more information, see [“Defining a Sequential Capture Procedure”](#).

The following limitations apply to full-sequential ATPG:

- It supports stuck-at faults, transition faults, and path delay faults, but not IDQ or bridging faults.
- It does not support the `-fault_contention` option of the `set_buses` command.
- It does not support the `-nocapture`, `-nopreclock`, and `-retain_bidi` options of the `set_contention` command.
- Patterns generated by Full-Sequential ATPG are not compatible with failure diagnosis

using the `run_diagnosis` command.

- The following options of the `set_simulation` command are not implemented for Full-Sequential simulation:

```
-bidi_fill | -strong_bidi_fill -measure <sim|pat>  
-oscillation
```

Running ATPG

ATPG generates a sequence of test patterns that enable an ATE to distinguish between the correct circuit behavior and the faulty circuit behavior caused by the defects. You use these patterns to test devices and to determine the cause of failure. Before running ATPG, make sure you have completed the recommended processes described in "[Preparing for ATPG](#)."

Basic scan mode (the default) is usually the first mode first you run in the ATPG process, followed by fast-sequential mode, then full-sequential mode. For detailed descriptions of various settings for these modes, see "[ATPG Modes](#)".

After running ATPG, you can review a set of output reports that provide coverage information on primitives, faults, patterns, library cells, memories, and other data relevant to ATPG. Based on these reports, you can make incremental adjustments to meet your ATPG goals, such as obtaining a good balance between pattern compaction and execution speed.

The following sections describe how to run ATPG:

- [Running ATPG in Basic Scan, Fast-Sequential, or Full-Sequential Mode](#)
- [Using Automatic Mode to Generate Optimized Patterns](#)
- [Quickly Estimating Test Coverage](#)
- [Specifying a Test Coverage Target Value](#)
- [Increasing Effort Over Multiple Passes](#)
- [Using Multiple-Session Test Generation](#)
- [Compressing Patterns](#)

Note: You can also set a variety of optimization parameters for running ATPG. For details on these settings, see "[Optimizing ATPG](#)".

Running ATPG in Basic Scan, Fast-Sequential, or Full-Sequential Mode

You can run ATPG using either the `run_atpg` command or the Run ATPG dialog box. This topic explains how to run ATPG in the basic scan, fast-sequential, or full-sequential modes. You can also run ATPG in automatic mode, which automatically selects the best settings and algorithms to provide reasonably good results. For details on automatic mode, see "[Using Automatic Mode to Generate Optimized Patterns](#)".

To run ATPG using the basic scan mode (the default), specify the `run_atpg` command without any options. This mode uses default two-clock transition ATPG when running distributed ATPG

for system clock transition, and is usually the first mode you use during the ATPG process. The following example runs ATPG in basic scan mode:

```
TEST-T> run_atpg
```

For information on specifying settings for basic scan mode, see "[Basic Scan Mode Settings](#)."

You can run ATPG in fast-sequential mode using the `fast_sequential_only` option of the `run_atpg` command. This mode provides limited support for partial-scan designs, and accommodates multiple capture procedures between scan load and scan unload. The following example runs ATPG in fast-sequential mode:

```
TEST-T> run_atpg fast_sequential_only
```

You can also use the `-capture_cycles` option of the `set_atpg` command to set a level of effort used by fast-sequential mode. For information on specifying settings for fast-sequential mode, see "[Fast-Sequential Mode Settings](#)."

You can set ATPG to run full-sequential mode using the `full_sequential_only` option of the `run_atpg` command. This mode supports multiple capture cycles between scan load and unload, which increases test coverage in partial-scan designs. The following example runs ATPG in full-sequential mode:

```
TEST-T> run_atpg full_sequential_only
```

You can also set full-sequential mode using the `-full_seq_atpg` option of the `set_atpg` command. In this case, you can run full-sequential mode by running the `run_atpg` command without using the `full_sequential_only` option. For information on specifying setting for full-sequential mode, see "[Full-Sequential Mode Settings](#)."

To use the Run ATPG dialog box to specify the ATPG mode and start the ATPG process, do the following:

1. Do one of the following:

- Select ATPG > Run ATPG
- Click the ATPG button

In both cases, the Run ATPG dialog box appears.

2. On the right side of the Run ATPG dialog box, click the button associated with the ATPG mode you want to run:

- To run basic scan mode, click the Basic Scan button.
- To run fast-sequential mode, click the Fast-Seq button.
- To run full-sequential mode, click the Full-Seq button .

Using Automatic Mode to Generate Optimized Patterns

You can specify TetraMAX ATPG to use an automatic mode that optimally generates compact sets of ATPG patterns. This mode automatically selects the best settings and algorithms to provide reasonably good results. Automatic mode is a good starting point for most ATPG flows. Although this mode uses a set of default parameters, you can still make manual adjustments as necessary.

Automatic pattern compression uses a combination of algorithms to achieve optimal results: a fast test generation algorithm that results in a lower pattern count and a secondary algorithm that produces excellent fault detection results in a slower runtime.

TetraMAX ATPG performs the following tasks in automatic mode:

- **Fault Population**

If there is no existing fault population, TetraMAX ATPG automatically populates a fault list (the same as running the `add_faults -all` command). If a fault population exists, the faults are left undisturbed and used for the remainder of the automatic mode process. For more information, on setting the fault population, see "[Specifying Fault Lists](#)."

- **Pattern Source**

Internal patterns are used as the pattern source (the default). For more information on internal patterns, see "[Setting the Pattern Source](#)."

- **Pattern Generation**

TetraMAX ATPG automatically uses basic-scan mode (the default) to generate an initial set of patterns. All patterns are stored and dynamic merge is enabled. The merge effort is automatically set to high, unless you have set the merge parameter to some other value. TetraMAX ATPG adheres to any other `set_atpg` command settings you specified (see "[Specifying General ATPG Settings](#)" for details). If you set the `-capture_cycles` option of the `set_atpg` command to a value greater than 1, fast-sequential ATPG is performed after basic-scan ATPG. Also, if you set the `-full_seq_atpg` option of the `set_atpg` command, full-sequential ATPG is performed after basic-scan ATPG or fast-sequential ATPG.

- **Reports**

After TetraMAX ATPG generates the patterns, it produces a fault summaries report, a test coverage report, and a pattern count report. In addition, the total CPU time is reported.

- **Restoration**

After completing the automatic mode process, TetraMAX ATPG restores all settings to their original values.

Setting Automatic Mode

To run ATPG in automatic mode:

1. Use the `set_atpg` command or the Run ATPG Dialog box to select the ATPG abort limit, ATPG verbose mode, and the ATPG merge effort, if necessary. You can also create a non-default fault population, or you can use defaults for any or all of these settings. For more information, see "[Specifying General ATPG Settings](#)."
2. Do one of the following to initiate automatic mode:
 - Specify the `-auto_compression` option of the `run_atpg` command, as shown in the following example:

```
run_atpg -auto_compression
```

- Use the Run ATPG dialog box, as shown in the following steps:

- a. Do one of the following:
 - Select ATPG > Run ATPG
 - Click the ATPG button

In both cases, the Run ATPG dialog box appears.

- b. Click the Auto button.

Note the following:

- Multiple fault sensitization is only available if you use the `-auto_compression` option.
- You can use the `-optimize_patterns` option of the `run_atpg` command to produce a very compact set of patterns with high test coverage. The trade-off is a longer runtime. For details, see [Optimizing Patterns during the run_atpg Process](#).

Quickly Estimating Test Coverage

You can quickly estimate the final test coverage by setting a low abort limit and low merge effort before running ATPG.

To quickly estimate coverage, use the `-abort_limit` and `-merge` option of the `set_atpg` command, as shown in the following example:

```
TEST-T> set_atpg -abort_limit 5 -merge off
TEST-T> run_atpg
```

To estimate coverage using the Run ATPG dialog box:

1. Select ATPG > Run ATPG or click the ATPG button in the command toolbar.
The Run ATPG dialog box appears.
2. Set the Abort Limit to 5.
3. Set the Merge Effort to Off.
4. Click Set.
5. For details about these and other settings, see the description of the `set_atpg` and `run_atpg` commands in TetraMAX Help.
6. Click Run.

Examples

[Example 1](#) shows a transcript produced by these commands. The reported test coverage is usually within 1 percent of the final answer, and the number of patterns with merge effort turned off is usually two to three times the number of patterns produced by a final pattern generation run with the merge effort set to high.

Example 1: Run ATPG Transcript, Merge Effort Turned Off

```
TEST-T> set_atpg -abort 5 -merge off
TEST-T> run_atpg
ATPG performed for 71800 faults using internal pattern source.
```

#patterns stored	#faults detect/active	#ATPG faults red/au/abort	test coverage	process CPU time
32	41288	30512	0/0/2	60.92% 1.35
64	7135	23377	0/0/3	69.04% 2.17
96	3231	20146	0/0/6	72.73% 2.81
128	2643	17503	0/0/7	75.74% 3.33
160	1976	15527	0/0/11	78.00% 3.91
192	1977	13550	0/0/13	80.26% 4.43
224	1450	12100	0/0/16	81.92% 4.85
256	1246	10854	0/0/21	83.35% 5.32
288	1101	9753	0/0/24	84.61% 5.77
319	683	9070	0/0/26	85.39% 6.13
351	748	8322	0/0/27	86.24% 6.46
383	620	7702	0/0/29	86.95% 6.77
:	:	:	:	:
1617	41	348	0/0/170	95.37% 22.02
1648	51	297	0/0/171	95.43% 22.34
1652	12	285	0/0/171	95.45% 22.43

TEST-T>

For comparison, [Example 2](#) shows a transcript from an ATPG run on the same design with the merge effort set to high.

Example 2: Run ATPG Transcript, Merge Effort Set to High

```
TEST-T> set_atpg -abort 5 -merge high
TEST-T> run_atpg
ATPG performed for 71800 faults using internal pattern source.
```

#patterns stored	#faults detect/active	#ATPG faults red/au/abort	test coverage	process CPU time
<hr/>				
Begin deterministic ATPG: abort_limit = 5...				
32	52694	19106	0/0/2	73.93% 39.05
64	6363	12743	0/0/6	81.21% 58.29
96	3200	9543	0/0/10	84.88% 74.35
128	2082	7461	0/0/13	87.26% 91.86
160	1234	6227	0/0/15	88.65% 105.62
192	1182	5045	0/0/17	90.00% 117.14
224	849	4196	0/0/21	90.97% 127.18
256	610	3586	0/0/25	91.67% 136.52
288	572	3014	0/0/29	92.32% 145.44
320	514	2500	0/0/34	92.91% 154.06
352	420	2080	0/0/37	93.39% 161.81
383	327	1753	0/0/43	93.77% 169.07
415	320	1433	0/0/49	94.13% 176.13
447	253	1180	0/0/72	94.42% 183.10
479	212	968	0/0/80	94.67% 189.54
511	176	792	0/0/90	94.87% 195.15
543	110	682	0/0/111	94.99% 200.98

```

575      97      585      0/0/133    95.11%    205.85
607      60      525      0/0/145    95.17%    210.38
639      90      435      0/0/175    95.28%    214.81
671      84      351      0/0/177    95.37%    218.10
695      46      305      0/0/177    95.43%    220.55
TEST-T>

```

The columns in the Run ATPG transcript are described as follows:

- `#patterns stored` – The total cumulative number of stored patterns (patterns that TetraMAX ATPG keeps).
- `#faults detect` – The number of faults detected by the current group of 32 patterns
- `#faults active` – The number of faults remaining active
- `#ATPG faults red/au/abort` – The cumulative number of faults found to be redundant, ATPG untestable, or aborted
- `test coverage` – The cumulative test coverage
- `process CPU time` – The cumulative CPU runtime, in seconds

With merge effort turned off, the design example produced the following results:

- Test coverage = 95.45 percent
- Number of patterns stored = 1652
- CPU time = 22 seconds

With merge effort set to high, the same design produced the following results:

- Test coverage = 95.43 percent
- Number of patterns stored = 695
- CPU time = 221 seconds

For a compromise between pattern compactness and CPU runtime, you can use the `-auto_compression` option of the `run_atpg` command. This option selects an automatic algorithm designed to produce reasonably compact patterns and high test coverage, with very little user effort and a reasonable amount of CPU time. To use this option, the fault source must be internal.

Specifying a Test Coverage Target Value

By default, TetraMAX ATPG processes faults and generates patterns in an attempt to achieve 100 percent test coverage. You can specify a lower test coverage target value by entering a decimal number between 0 and 100.0 in the Coverage % field of the Run ATPG dialog box or by issuing a command similar to the following example:

```
TEST-T> set_atpg -coverage 88.5
```

You might want to specify a test coverage lower than 100 percent if you want to produce fewer patterns, your design requirements are satisfied with a lower coverage, or you want an alternative to using a pattern limit for decreasing CPU time.

Increasing ATPG Effort Over Multiple Passes

Determining an appropriate setting for the abort limit is an iterative process. The following multipass approach produces reasonable results without using excessive CPU time:

1. Set the abort limit to 10 or less.
2. Set the merge effort to Off.
3. Generate test patterns (`run_atpg`).
4. Examine the results. If there are too many ND (not detected) faults remaining, increase the abort limit and generate test patterns again.
5. Repeat as necessary to determine the minimum abort limit necessary to achieve the required results.

The following example sequence shows how to specify these settings:

```
TEST-T> set_atpg -abort_limit 10 -merge off
TEST-T> run_atpg
TEST-T> set_atpg -abort 50
TEST-T> run_atpg
TEST-T> set_atpg -abort 250
TEST-T> run_atpg
```

Note: Increasing the abort limit might decrease the number of ND faults, but it will not decrease the number of AU (ATPG untestable) faults.

Multiple Session Test Pattern Generation

You can create patterns using multiple sessions as well as using multiple passes. For an example of using multiple passes, see "[Increasing Effort Over Multiple Passes](#)."

The following examples describe situations where you might use multiple sessions:

- Your pattern set is too large for the tester, so you try an additional compression effort. If that is unsuccessful, you truncate the pattern set to a size that fits the tester.
- Your pattern set is too large for the tester, so you split the pattern set into two or more smaller sets.
- You have 2,000 patterns and a simulation failure occurs around pattern 1,800. You want to look at the problem in more detail but do not want to take the time to resimulate 1,799 patterns, so you read in the original patterns and write out the pattern with the error, plus one pattern before and after for good measure.
- You have three separate pattern files from previous attempts, and you want to merge them all into a single pattern file that eliminates duplications.
- Your design has asymmetrical scan chains or other irregularities, and you want to create separate pattern files with different environments of scan chains, clocks, and PI constraints.
- You have changed the conditions under which your existing patterns were generated (for

example, by using a different fault list). You want to see how the existing patterns perform with the new fault list.

These examples are explained in more detail in the following sections:

- [Splitting Patterns](#)
- [Extracting a Pattern Sub-Range](#)
- [Merging Multiple Pattern Files](#)
- [Using Pattern Files Generated Separately](#)

Splitting Patterns

To split patterns, reestablish the exact environment under which the patterns were generated. You do not need to restore a fault list. After achieving test mode, you can split the patterns at the 500-pattern mark by using a command sequence similar to the following example:

```
TEST-T> set_patterns -external session_1_patterns
TEST-T> write_patterns pat_file1 -last 499 -external
TEST-T> write_patterns pat_file2 -first 500 -external
```

Extracting a Pattern Sub-Range

To extract part of the pattern, you use the same environment setup rules as for splitting patterns, except that you use the `-first` and `-last` options of the `write_patterns` command when writing patterns. After achieving test mode, you can extract a subrange of three patterns using a command sequence similar to the following example:

```
TEST-T> set_patterns -external session_1_patterns
TEST-T> write_patterns subset_file -first 198 -last 200 -ext
```

Merging Multiple Pattern Files

You can merge multiple pattern files only if all the files were generated under the same conditions of clocks and constraints and have identical scan chains. The fault lists do not have to match. To accomplish the merge, reestablish the environment and choose the final fault list to be used. Patterns in the external files are eliminated during the merge effort if they do not detect any new faults based on the current fault list.

After you achieve test mode and initialize a starting fault list, execute commands similar to the following example:

```
TEST-T> set_patterns -external patterns_1
TEST-T> run_atpg
TEST-T> set_patterns -external patterns_2
TEST-T> run_atpg
TEST-T> set_patterns -external patterns_3
TEST-T> run_atpg
TEST-T> report_summaries
```

Alternatively, if you want to avoid running ATPG repeatedly or want to avoid potentially dropping patterns, then you can replace the `run_atpg` commands with `run_simulation -store` commands:

```
TEST-T> set_patterns -delete
TEST-T> set_patterns -external patterns_1
TEST-T> run_simulation -store
TEST-T> set_patterns -external patterns_2
TEST-T> run_simulation -store
TEST-T> set_patterns -external patterns_3
TEST-T> run_simulation -store
TEST-T> report_summaries
```

This alternative approach copies and appends the patterns from an external buffer into an internal one without performing ATPG and without any potential dropping of patterns.

Using Pattern Files Generated Separately

Using multiple sessions to generate patterns, you can use different definitions for clocks, PI constraints, or even scan chains to obtain two or more separate sets of ATPG patterns that achieve a cumulative test coverage effect. The key to determining cumulative test coverage is sharing and reusing the fault list from one session to another.

For example, suppose that you want to create separate pattern files for a design that has the following characteristics:

- 20 scan chains, evenly distributed so that they all are between 240 and 250 bits in length
- 1 boundary scan chain that is 400 bits in length
- 1,500 patterns that have been run through ATPG and produced 98 percent test coverage
- A tester cycle budget of 500,000 cycles

Some rough calculations indicate that the 1,500 patterns require approximately 600,000 tester cycles ($400 \times 1,500$), which exceeds the tester cycle budget. One possible solution is to set up two different environments, one that uses all scan chains and another that eliminates the definition of the 400-bit long scan chain.

Your two ATPG sessions are organized in the following manner:

- Session 1: You create an STL procedure file that defines all scan chains except the 400-bit chain. You proceed to generate maximum coverage using minimum patterns. After saving the patterns and before exiting, you save the final fault list, as in the following command:

```
TEST-T> write_faults sess1_faults.gz -all -uncollapsed -
compress gzip
```

- Session 2: You create an STL procedure file that defines all scan chains. You read in the fault list saved in Session 1, as in the following command:

```
TEST-T> read_faults sess1_faults.gz -retain_code
```

The first session probably achieves less than the original 98 percent coverage, but still consumes approximately 1,500 patterns. More important, the combination of the two sessions matches the original 98 percent test coverage but generates fewer than 20 percent of the

original patterns for the second session (about 300 patterns). The total test cycles for both sets of patterns are now as follows:

$$(1,500 \times 250) + (300 \times 400) = 495,000 \text{ tester cycles}$$

The number of patterns has increased from 1,500 to 1,800, but the number of tester cycles has decreased by more than 100,000 and the original test coverage has been maintained.

Note: When you pass a fault list from one session to another and perform pattern compression, you will see different test coverage results before and after pattern compression. Pattern compression performs a fault grade on the patterns that exist only at that point in time. After pattern compression, the test coverage statistics reflect the coverage of the current set of patterns. The correct cumulative test coverage for both sessions is the output from the last `report_summaries` command executed before any pattern compression.

Compressing Patterns

Test patterns produced by ATPG techniques usually have some amount of redundancy. You can usually reduce the number of patterns significantly by compressing them, which means eliminating some patterns that provide no additional test coverage beyond what has been achieved by other patterns.

Dynamic pattern compression is performed while patterns are being created. With this technique, each time a new pattern is created, an attempt is made to merge the pattern with one of the existing patterns within the current cluster of 32 patterns in the pattern simulation buffer.

To enable dynamic pattern compression, use the `-merge` option of the `set_atpg` command or the equivalent options in the Run ATPG dialog box.

The following sections describe the process for compressing patterns:

- [Balancing Pattern Compaction and CPU Runtime](#)
- [Compression Reports](#)

Balancing Pattern Compaction and CPU Runtime

Normally, a reasonable number of passes of static compression produces a smaller number of patterns. However, this reduced pattern count results in a CPU runtime penalty.

For a compromise between pattern compactness and CPU runtime, you can use the `-auto_compression` option of the `run_atpg` command. This option selects an automatic algorithm designed to produce reasonably compact patterns and high test coverage, using a reasonable amount of CPU time. For more information, look in the online help under the index topic “Automatic ATPG.”

To obtain the maximum test coverage while achieving a reasonable balance of CPU time and patterns:

1. Obtain an estimate of test coverage using the Quick Test Coverage technique (see [“Quickly Estimating Test Coverage”](#)). If you are not satisfied with the estimate, determine the cause of the problem and obtain satisfactory test coverage before you attempt to achieve minimum patterns.
2. Set Abort Limit to 100–300.

3. Set Merge Effort to High.
4. Specify the `run_atpg -auto_compression` command.
5. Examine the results. If there are still some NC or NO faults remaining, increase the Abort Limit by a factor of 2 and execute `run_atpg` again.

Compression Reports

[Example 1](#) shows a dynamic compression report generated using the `-verbose` option of the `set_atpg` command. The `-verbose` option produces the following additional information:

- The pattern number within the current group of 32 patterns
- The number of fault detections successfully merged into the pattern (#merges)
- The number of faults that were attempted but could not be merged into the current pattern, which matches the merge iteration limit unless the number of faults remaining is less than this limit (#failed_merges)
- The number of faults remaining in the active fault list (#faults)
- The CPU time used in the merge process

If you monitor the verbose information, you will eventually see a point at which the number of merges approaches zero. At this point, stop the process and reduce the merge effort or disable it because the effect is not producing sufficient benefit to justify the CPU effort expended.

Example 1 Verbose Dynamic Compression Report

```
TEST-T> set_atpg -patterns 150 -merge medium -verbose
TEST-T> run_atpg
ATPG performed for 72440 faults using internal pattern source.
-----
#patterns      #faults      #ATPG faults    test      process
stored        detect/active   red/au/abort  coverage   CPU time
----- ----- ----- ----- -----
Begin deterministic ATPG: abort_limit = 5...
Patn 0: #merges=452 #failed_merges=100 #faults=40083  CPU=1.51 sec

Patn 1: #merges=637 #failed_merges=100 #faults=33938  CPU=2.82 sec

Patn 2: #merges=380 #failed_merges=100 #faults=30325  CPU=3.67 sec

Patn 3: #merges=211 #failed_merges=100 #faults=27403  CPU=4.52 sec

Patn 4: #merges=115 #failed_merges=100 #faults=25827  CPU=5.16 sec

Patn 5: #merges=798 #failed_merges=100 #faults=24633  CPU=6.66 sec

Patn 6: #merges=97  #failed_merges=100 #faults=23436  CPU=7.19 sec

Patn 7: #merges=82  #failed_merges=100 #faults=22431  CPU=7.69 sec
```

```

Patn 8: #merges=73 #failed_merges=100 #faults=21348 CPU=8.27 sec
Patn 9: #merges=77 #failed_merges=100 #faults=20340 CPU=8.83 sec
Patn 10: #merges=58 #failed_merges=100 #faults=19906 CPU=9.34 sec
Patn 11: #merges=65 #failed_merges=100 #faults=18231 CPU=9.97 sec
Patn 12: #merges=39 #failed_merges=100 #faults=17414 CPU=10.44 sec
Patn 13: #merges=50 #failed_merges=100 #faults=16759 CPU=10.96 sec
Patn 14: #merges=35 #failed_merges=100 #faults=16383 CPU=11.28 sec
Patn 15: #merges=36 #failed_merges=100 #faults=15994 CPU=11.62 sec
Patn 16: #merges=29 #failed_merges=100 #faults=15588 CPU=11.99 sec
Patn 17: #merges=28 #failed_merges=100 #faults=15112 CPU=12.36 sec
Patn 18: #merges=36 #failed_merges=100 #faults=14763 CPU=12.69 sec
Patn 19: #merges=34 #failed_merges=100 #faults=14510 CPU=13.02 sec
Patn 20: #merges=21 #failed_merges=100 #faults=14289 CPU=13.35 sec
Patn 21: #merges=342 #failed_merges=100#faults=13933 CPU=14.18 sec
Patn 22: #merges=37 #failed_merges=100 #faults=13711 CPU=14.50 sec
Patn 23: #merges=24 #failed_merges=100 #faults=13570 CPU=14.79 sec
Patn 24: #merges=24 #failed_merges=100 #faults=13438 CPU=15.05 sec
Patn 25: #merges=20 #failed_merges=100 #faults=13294 CPU=15.32 sec
Patn 26: #merges=23 #failed_merges=100 #faults=13145 CPU=15.59 sec
Patn 27: #merges=134 #failed_merges=57 #faults=12687 CPU=16.93 sec
Patn 28: #merges=27 #failed_merges=100 #faults=12552 CPU=17.28 sec
Patn 29: #merges=23 #failed_merges=100 #faults=12410 CPU=17.54 sec
Patn 30: #merges=29 #failed_merges=100 #faults=12296 CPU=17.82 sec
Patn 31: #merges=22 #failed_merges=100 #faults=12202 CPU=18.09 sec

```

32	51756 20684	0/0/1	72.80%	19.37
----	-------------	-------	--------	-------

```

Patn 0: #merges=19 #failed_merges=100 #faults=11909 CPU=19.65 sec
Patn 1: #merges=34 #failed_merges=100 #faults=11755 CPU=19.93 sec
Patn 2: #merges=17 #failed_merges=100 #faults=11666 CPU=20.22 sec

```

Analyzing ATPG Output

You can analyze ATPG pattern generation output from the `run_atpg` command. This output includes the following formats:

- [Standard Format](#)
- [Expert Format](#)
- [Verbose Format with Merge Without -auto_compression](#)
- [Verbose Format with Merge Without -auto_compression](#)

Standard Format

```

TEST> run_atpg
ATPG performed for 72436 faults using internal pattern source.
-----
#patterns #faults      #ATPG faults test      process
stored    detect/active red/au/abort coverage CPU time
-----
Begin deterministic ATPG: abort_limit = 5...
 32       49465 22971  0/0/1      70.05%   6.50
 64       6808  16163  0/0/3      77.82%   10.52
 96       3779  12380  1/1/4      82.13%   13.48
128       2220  10156  2/2/6      84.66%   16.02
160       1264  8890   4/2/7      86.11%   18.54
192       1415  7474   4/3/11     87.73%   20.87
224     1021   6450   6/4/13     88.89%   23.04
256       835   5610   9/6/17     89.85%   25.17
288       722   4881  13/8/19     90.68%   27.20
320       653   4223  15/11/21    91.43%   29.16
352       572   3648  16/13/26    92.08%   31.15
:         :     :   :   :       :       :
831       78    378   176/105/132  95.69%   62.35
862       73    295   184/107/142  95.78%   64.08
889       49    212   205/113/143  95.87%   65.35

```

#patterns stored

This indicates the current number of patterns which are stored in the internal pattern set. These patterns were created during the ATPG process and are selected only if they are required for fault detection.

#faults detect/active

The first field indicates the number of faults that were detected in the current simulation pass. The second field indicates the number of faults that still remain active in the fault list. Depending on the fault-report setting, the fault counts may be either uncollapsed (default) or collapsed.

#ATPG faults red/au/abort

The first field indicates the cumulative number of faults identified as redundant in the current ATPG process. The second field indicates the cumulative number of faults identified as ATPG untestable in the current ATPG process. The third field indicates the cumulative number of faults that were aborted in the current ATPG process. All of these fault counts are collapsed fault counts.

test coverage

This indicates the current value of the test coverage considering the current fault list and patterns previously evaluated. There is a user selectable credit given for possible-detected faults (default 50%) and ATPG untestable faults (default 0%). Depending on the fault-report setting, the test coverage is calculated using fault counts which are either uncollapsed (default) or collapsed.

process CPU time

This indicates the cumulative number of CPU seconds that have been used up to this point in the current ATPG process.

Expert Format

```
TEST> run_atpg
ATPG performed for stuck fault model using internal pattern source.
Fast-seq simulation is used to verify Basic-Scan patterns.
-----
#patterns #patterns      #faults      #ATPG faults test      process
simulated eff/total detect/active red/au/abort coverage CPU time
-----
Begin deterministic ATPG: #uncollapsed_faults=72346, abort_limit=10...
32        32 32      49273 23072      1/0/1      69.89%    7.40
64        32 64      6890 16182      1/0/2      77.75%    11.59
96        32 96      3233 12948      2/0/6      81.45%    15.06
128       32 128     2295 10651      3/1/7      84.07%    18.04
160       32 160     1986 8662       4/2/7      86.33%    20.80
192       32 192     1256 7403       6/3/7      87.77%    23.37
224       32 224     971 6429       8/4/8      88.88%    25.76
256       32 256     842 5583       10/6/10     89.84%    28.12
288       32 288     702 4875       14/7/12     90.65%    30.44
320       32 320     639 4235       14/8/13     91.38%    32.73
352       32 352     514 3718       16/9/16     91.97%    35.02
:         :  :  :      :          :          :          :
832       32 830     80 294       143/92/58     95.78%    68.51
864       32 862     59 212       163/93/65     95.87%    70.85
896       32 894     58 133       179/94/70     95.96%    72.91
909       13 907     29 83       197/96/70     96.01%    73.72
Begin fast-seq ATPG: #uncollapsed_faults=179, abort_limit=10, depth=4...
910      1 908      5 174       0/0/9      96.02%    73.87
911      1 909      1 172       0/1/38     96.02%    74.43
912      1 910      1 171       0/1/47     96.02%    74.61
913      1 911      2 169       0/1/48     96.02%    74.67
```

This form is generated when set messages-level expert is in effect.

#patterns eff/total

This report is identical to the Standard Form with the exception that an additional information appears as the 2nd and 3rd columns. The 2nd column is the number of patterns in the current working group of 32 which were effective and kept. The 3rd column is the cumulative total number of patterns kept.

Verbose Format with Merge (without -auto_compression)

```
run_atpg
ATPG performed for stuck fault model using internal pattern source.
Fast-seq simulation is used to verify Basic-Scan patterns.
-----
#patterns #patterns      #faults      #ATPG faults test      process
simulated eff/total detect/active red/au/abort coverage CPU time
-----
```

```

Begin deterministic ATPG: #uncollapsed_faults=266012, abort_limit=10...
Patn 1: #merges=537 #failed_merges=20 #faults=154875 #det=8693 CPU=1.14 sec
Patn 2: #merges=316 #failed_merges=20 #faults=124914 #det=53161 CPU=1.71sec
Patn 3: #merges=256 #failed_merges=20 #faults=107012 #det=29741 CPU=2.19sec
Patn 4: #merges=83 #failed_merges=20 #faults=95837 #det=17827 CPU=2.48sec
Patn 5: #merges=235 #failed_merges=20 #faults=85826 #det=15586 CPU=2.92sec
.....
Patn 28: #merges=40 #failed_merges=20 #faults=34518 #det=1181 CPU=9.10 sec
Patn 29: #merges=44 #failed_merges=20 #faults=33872 #det=959 CPU=9.28 sec
Patn 30: #merges=56 #failed_merges=20 #faults=33223 #det=1009 CPU=9.47 sec
Patn 31: #merges=32 #failed_merges=20 #faults=32730 #det=829 CPU=9.63 sec
32          32
32      210799      55209           2/0/0      80.42%      10.23
Patn 0: #merges=43 #failed_merges=20 #faults=32127 #det=1179 CPU=10.39sec
Patn 1: #merges=28 #failed_merges=20 #faults=31515 #det=1059 CPU=10.54 sec
.....
Patn 31: #merges=33 #failed_merges=20 #faults=21284 #det=353
CPU=15.18 sec
64          32   64   18839 36366           4/0/0      86.43%      15.43
Patn 0: #merges=18 #failed_merges=20 #faults=21145 #det=232
CPU=15.55 sec
.....
Patn 31: #merges=32 #failed_merges=20 #faults=15576 #det=225
CPU=19.81 sec
96          32   96   9508 26846           7/0/2      89.47%      20.02

```

This form is generated when `set_atpg -verbose -merge` is in effect.

#merges

This indicates the number of additional patterns merged with the original pattern. Each pattern successfully detects a fault on at least one target fault (primary fault). The combined pattern can also detect additional faults (secondary faults). A merge count of 10 means the single pattern is doing the work of 11 patterns and it will detect at least the 11 target faults (primary faults) and might also detect many more faults that were not the original targets (secondary faults).

#failed_merges

This indicates the number of faults for which a pattern was generated but that new pattern could not be merged with the existing pattern for the primary fault site. When the count is less than the merge limit (low=20, medium=100, high=500) then TetraMAX ATPG ran out of active faults/patterns to merge into the primary pattern before it reached the iteration limit. When the count is equal to the merge limit, then the limit was reached and there were still faults that could have attempted to be merged.

When you see the merge limit being reached over and over again, there can be value in increasing the merge effort using the `-merge` option of the `set_atpg` command. However, this increase in merge effort will come at a cost of additional CPU time.

When the failed merge count is consistently less than the limit on each pattern attempt, then the optimal setting for the merge effort is just higher than the maximum failure count.

If you assume in the previous example that the merge effort was 300, then the majority of patterns showed a #failed_merges count less than 300 and this value is reasonably good. Increasing the merge effort to 400 or 500 can improve the number of patterns merged for patterns 0, 1, and 4 in the first group of 32, but at a cost of increased runtime. The optimal value for merge effort often requires repeated ATPG runs and seeking the optimal value can often take more time than is efficient. One shortcut approach is to set the merge effort high, say 3000, and then watch the progress for the first 32 patterns and then stop the run. Using the information learned in the first 32 patterns to set the merge effort for a more complete run. When -auto_compression is not used, only the first parameter of set_atpg -merge is in effect.

#faults

If single-pattern fault simulation is performed, this number represents the number of faults detected by the pattern. If single-pattern fault simulation is not performed, this number represents the number of faults targeted by the test generator (i.e., primary, secondary and side-path detection faults). In either case, fault simulation at the end of the interval ultimately decides which faults are truly detected and which are not.

A heuristic algorithm is used to decide whether or not to perform single-pattern fault simulation. This algorithm attempts to simultaneously maximize coverage and minimize pattern count and CPU time.

In some cases, single-pattern fault simulation is performed on some patterns in an interval (thus the #faults can be high). But simulation may not be performed on other patterns (thus the #faults is low). This does not indicate incorrect behavior and is not a cause for cone timern.

CPU=

This indicates the cumulative number of CPU seconds that have been used up to this point in the current ATPG process.

Verbose Format with Merge and -auto_compression

```
run_atpg -auto_compression
ATPG performed for stuck fault model using internal pattern
source.
Fast-seq simulation is used to verify Basic-Scan patterns.
-----
#patterns #patterns      #faults      #ATPG faults test      process
simulated eff/total detect/active red/au/abort coverage CPU time
-----
Begin deterministic ATPG: #uncollapsed_faults=3199364, abort_
limit=10...
Patn 1: #merges=0/922 (0%) #failed_merges=0/34 #faults=1783833
```

```
#det=14023 CPU=32.58 sec clocks=
Patn 2: #merges=0/86808(3%) #failed_merges=0/3484 #faults=1435411
#det=642019 CPU=58.32 sec clocks= cclk pclk

Patn 3: #merges=0/46986(0%) #failed_merges=0/125 #faults=1366319
#det=101465 CPU=81.48 sec clocks= cclk crst_ pclk
.....
Patn 31: #merges=0/2110(0%) #failed_merges=0/272 #faults=411938
#det=12251 CPU=324.24 sec clocks= pclk
Warning: 3 (4) basic-scan patterns failed current pass simulation
check and is treated as ignored measures. (M212)
32 32 32 2492119 707234 2/4/6 77.30% 362.76
Local redundancy analysis results: #redundant_faults=4398, CPU_
time=3.00 sec

Patn 0: #merges=0/1761(0%) #failed_merges=0/242 #faults=400847
#det=9765 CPU=370.68 sec clocks= cclk pclk
.....
Patn 31: #merges=0/869(0%) #failed_merges=0/62 #faults=276129
#det=3269 CPU=484.81 sec clocks= ZXIN ZADCK
Warning: 1 (1) basic-scan patterns failed current pass simulation
check and is treated as ignored measures. (M212)
64 32 64 238165 462910 3/6/10 83.51% 499.63

Patn 0: #merges=0/1437(0%) #failed_merges=0/231 #faults=272195
#det=6860 CPU=502.74 sec clocks= inclk zxin ad[0]

Patn 1: #merges=0/1253(0%) #failed_merges=0/155 #faults=268677
#det=6429 CPU=505.99 sec clocks= clk inclk crst_ pclk
.....
Patn 31: #merges=0/568(0%) #failed_merges=0/39 #faults=221584
#det=2037 CPU=587.17 sec clocks= inclk
```

This form is generated when set atpg -verbose -merge is in effect.

#merges=d1/d2 (d3%)

This indicates the number of additional patterns merged with the original pattern. Each pattern successfully detects a fault on at least one target fault (primary fault). The combined pattern can also detect additional faults (secondary faults). A merge count of 10 means the single pattern is doing the work of 11 patterns and it will detect at least the 11 target faults (primary faults) and may also detect many more faults that were not the original targets.

o d1 is number of secondary faults detected and merged into the pattern.

- o d2 is number of faults detected and merged through multiple fault sensitization.
 - o d3 is the percentage of of multiple fault sensitization merges that did not detect any faults (d2 and d3 are printed only when using `-auto_compression` and verbose mode is turned on)
- The first and second values passed to the `set_atpg -merge` command control the secondary fault merge effort and multiple fault sensitization merge effort, respectively.

`#failed_merges=d4/d5`

Where d4 is the number of failed merges of secondary faults and d5 is the number of failed merges of multiple fault sensitization (d5 is printed only when `-auto_compression` is used and verbose mode is turned on)

`#faults`

This indicates the calculated number of collapsed faults that are still active in the fault list.

`#detects`

This indicates the number detected faults.

`CPU=`

This indicates the cumulative number of CPU seconds that have been used up to this point in the current ATPG process.

`clock=s`

This is a list of clocks pulsed during the capture cycle. With dynamic clock grouping, you can have multiple clocks pulsing together in the same capture cycle, which results in a considerable reduction in the pattern count. This field is printed only when `-auto_compression` is used.

Note: For d3, #faults and #detects, you might sometimes see "---". When the design is large and only multiple fault sensitization is in progress, it is more efficient and productive to run fault simulation at the end of an interval (that is, 32 patterns). For these conditions, because each pattern is not fault simulated as soon as it is generated, some information required in verbose messages is not available.

Reviewing Test Coverage

You can view the results of the test coverage and the number of patterns generated using the `report_summaries` command or the Report Summaries dialog box.

The following example shows how to generate a fault summary report using the `report_summaries` command:

TEST-T> `report_summaries`

For the complete syntax and option descriptions, see the description of the `report_summaries` command in TetraMAX Help.

To use the Report Summaries dialog box to generate a fault summary report:

1. From the command toolbar, click the Summary button. The Report Summaries dialog box appears.
2. Select the appropriate summary settings.
For details about available settings, see the description of the `report_summaries` command in TetraMAX Help.
3. Click OK.

An example output report showing the fault counts and the test coverage obtained by using the uncollapsed fault list is shown in [Example 3](#). A detailed description of each fault class is shown in [“Fault Lists and Faults.”](#)

Example 3: Uncollapsed Fault Summary Report

```
TEST-T> report_summaries
Uncollapsed Fault Summary Report
-----
fault class          code #faults
-----
Detected            DT    83348
Possibly detected   PT    324
Undetectable        UD    1071
ATPG untestable     AU    3453
Not detected        ND    212
-----
total faults         88408
test coverage        95.62%
-----
Pattern Summary Report
-----
#internal patterns  1636
```

[Example 4](#) shows the same report with collapsed fault reporting. Notice that there are fewer total faults, and fewer individual fault categories.

Example 4: Collapsed Fault Summary Report

```
TEST-T> set_faults -report collapsed
TEST-T> report_summaries
Collapsed Fault Summary Report
-----
fault class          code #faults
-----
Detected            DT    50993
Possibly detected   PT    214
Undetectable        UD    1035
ATPG untestable     AU    2370
Not detected        ND    122
-----
total faults         54734
test coverage        95.16%
```

```
-----  
          Pattern Summary Report  
-----  
#internal patterns           1636  
-----
```

To find out where the faults are located in the design, see "[Analyzing the Cause of Low Test Coverage](#)"

Writing ATPG Patterns

TetraMAX ATPG can write pattern files in binary, STIL, and WGL. By default, TetraMAX ATPG generates new internal patterns. To save the test patterns, you can use the `write_patterns` command or the Write Patterns dialog box.

Note: For information on translating adaptive scan patterns into normal scan-mode patterns, see "[Reading Pattern Files](#)."

By default, TetraMAX ATPG writes parallel patterns in the unified STIL flow format when the `-format` option of the `write_patterns` command is specified with the `stil` or `stil99` arguments.

The following examples show how to use the `write_patterns` command to write serial STIL patterns:

```
write_patterns patterns.stil -serial -format stil
```

The following example writes patterns in a proprietary binary format that can be read by TetraMAX ATPG:

```
write_patterns patterns.bin -format binary -replace
```

To use the Write Patterns dialog box to format and save test patterns:

1. From the command toolbar, click the Write Pat button.
The Write Patterns dialog box appears.
2. In the Pattern File Name field, enter the name of the pattern file to be written or use the Browse button to find the directory you want to use or to view a list of existing files.
3. Accept the default settings unless you require more.
4. Click OK.

For descriptions of all the options for writing patterns, see the description of the `write_patterns` command in TetraMAX Help.

For information on generating patterns for DFTMAX Ultra, see "[Pattern Types Accepted by DFTMAX Ultra](#)".

3

Using TetraMAX II

TetraMAX II is Synopsys' next-generation ATPG product. It uses advanced compression, diagnosis, and fault simulation engines that are fast, memory-efficient, and optimized for fine-grained multithreading of ATPG and diagnosis processes across multiple cores.

The following topics describe the essentials for understanding and using TetraMAX II:

- [About TetraMAX II](#)

Provides an overview of TetraMAX II and how it fits into the existing Synopsys test environment. This section includes a brief explanation of multithreading, and how TetraMAX II implements multithreading.

- [Running TetraMAX II](#)

Describes the primary flow and user interface for using TetraMAX II. Includes a description of how to start TetraMAX II, the commands that are specific for running TetraMAX II, how to set the thread count, run simulation and fault simulation, report power, and run diagnostics.

- [Messaging](#)

Describes the messages reported when using TetraMAX II.

- [Command Option Support](#)

How TetraMAX II handles various command options compared to TetraMAX ATPG.

- [Limitations](#)

Lists known limitations or features not supported by TetraMAX II.

About TetraMAX II

TetraMAX II uses existing libraries and fits seamlessly into existing flows. It is intended to produce significant runtime improvement, consistent test coverage, and pattern reduction across various server configurations and machines. TetraMAX II retains the production-proven rule checking, design modeling, and fault modeling of the existing TetraMAX ATPG product.

The following topics provide an introduction to TetraMAX II:

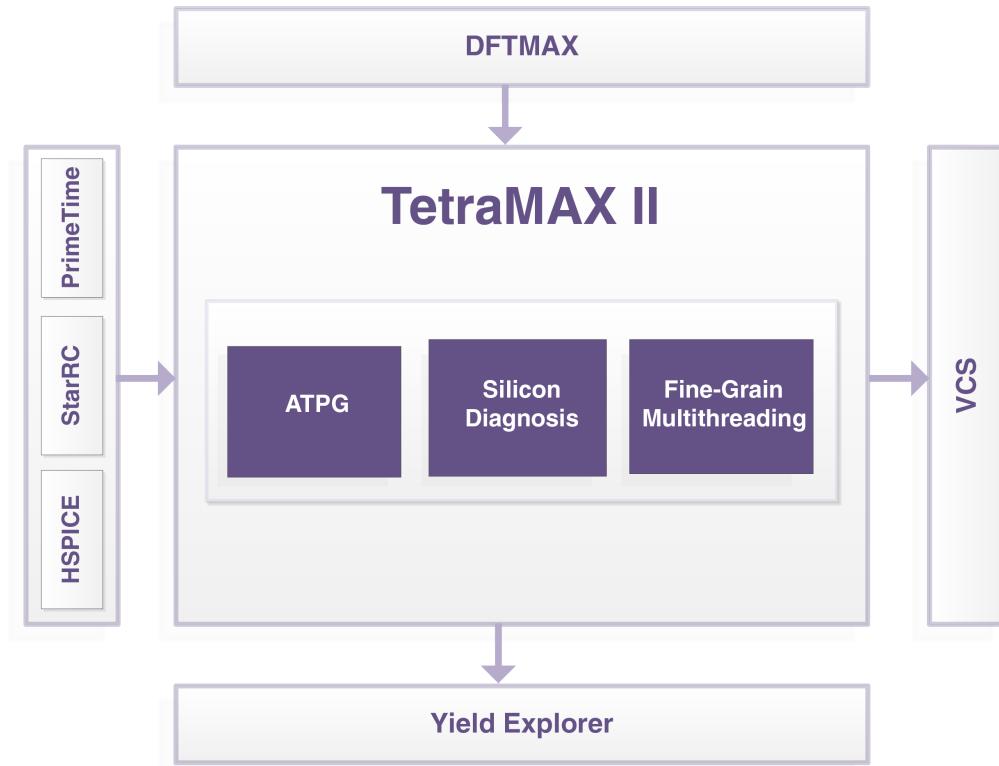
- [TetraMAX II and the Test Environment](#)
 - [Key Concepts of Multithreading](#)
 - [How TetraMAX II Uses Multithreading](#)
-

TetraMAX II and the Test Environment

TetraMAX II fits seamlessly into the existing Synopsys test environment:

- Integration with both DFTMAX and DFTMAX Ultra optimize timing, power, area, and congestion for test and functional logic.
- Integration with Yield Explorer provides volume diagnostics and yield analysis.
- PrimeTime provides TetraMAX II crucial slack data to accurately target timing-critical defects
- StarRC provides TetraMAX II key physical extraction and timing information from HSPICE simulations for cell-aware ATPG, which targets defects inside a cell.

Figure 1 : TetraMAX II and the Synopsys Test Environment



Key Concepts of Multithreading

TetraMAX II is built upon the use of multithreading. This technique concurrently executes small, multiple tasks or threads based upon instructions from a single central controlling scheduler.

A *thread* is the flow of execution through a single process. Each thread is comprised of its own program counter to track instructions, system registers to store its current working variables, and a stack which contains the execution history.

Some of the key benefits of multithreading include:

- **Improve Responsiveness**

Each activity is defined as a thread, which can be replicated as many times as required.

- **Efficient Use of Multiprocessors**

Because performance improves transparently with additional processors, there is no need to account for the number of available processors. Numerical algorithms and the use of parallelism run much faster when implemented with threads on a multiprocessor.

- **Efficient and Adaptive Program Structure**

Multithreaded architecture is more efficiently structured as a series of multiple independent execution units rather than a single, monolithic thread.

- **Minimizes the Use of System Resources**

The use of two or more processes that access common data through shared memory usually requires more than one thread of control. But in multithreaded architecture, each process has a full address space and operating environment state, which minimizes overall system resources.

For more information on multithreading, see the "What is the Difference Between Multicore Processing and Multithreading?" topic in TetraMAX Help.

Also, note that TetraMAX II does not require you to turn on hyperthreading. This is explained in detail in the next section, "[How TetraMAX Uses Multithreading.](#)"

How TetraMAX II Uses Multithreading

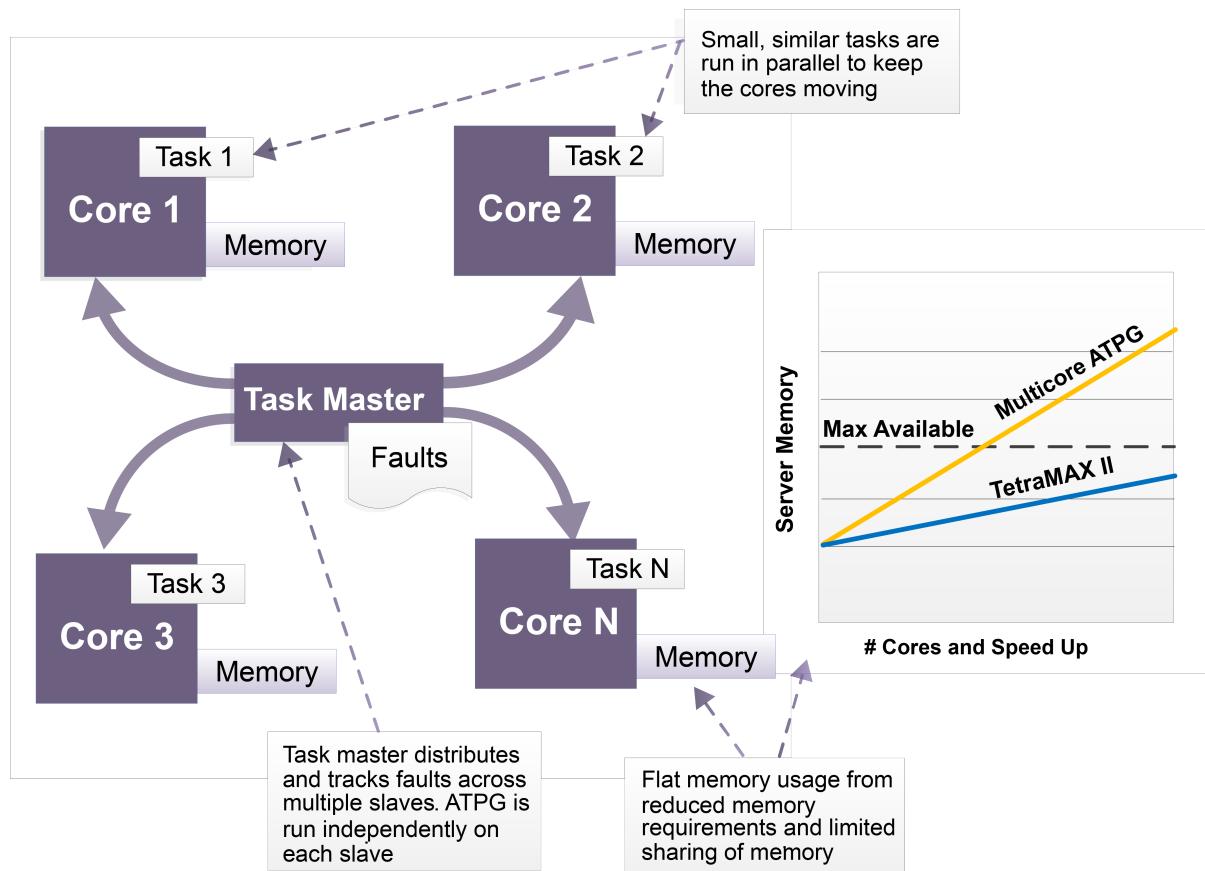
In conventional multicore implementations, cores are underutilized because existing ATPG technologies require a large amount of memory per core, which limits the actual number of cores that can be used. TetraMAX II uses fine-grained, multithreaded architecture to provide virtually unlimited memory usage for its core engines. This enables the full use of multiple cores and results in extremely fast runtime. TetraMAX II achieves higher diagnostics throughput by transparently running multiple simulations.

In the TetraMAX II multithreaded architecture, the task master distributes and tracks faults across multiple slave processes and triggers independent, parallel ATPG runs on each slave. As each task completes, the task master makes a copy of the program counter and instruction registers from each slave, and compiles this data as resources become available. Each slave moves onto the next task without interruption.

Multithreading eliminates the continuous stopping and restarting of the cores, which is the underlying cause of the inconsistent results common in multicore processing. For a complete explanation, see "What is the Difference Between Multicore Processing and Multithreading?" in the FAQ.

Figure 2 shows how TetraMAX II uses multithreading

Figure 2: How TetraMAX II Uses Multithreading



As shown in Figure 2, multithreaded ATPG runs a series of small tasks in parallel. The cores are continuously active and scalability is predictable. Reduced memory requirements and limited sharing of memory in TetraMAX II results in flat memory usage.

TetraMAX II does not require you to turn on hyperthreading. This methodology is effective when there are considerably more threads than CPUs, and the threads often wake up and fall asleep. This means they need to effectively move on and off the CPUs. TetraMAX II, creates more threads than are specified by the `-num_threads` setting of the `set_atpg` and `set_simulation` commands. However, because these threads can stay active for long periods, they can remain in a CPU for an extended period. The only exception occurs when CPUs are overutilized, which always causes the delay of all active jobs.

Running TetraMAX II

The user interface of TetraMAX II is nearly identical as TetraMAX ATPG. Both products share the same basic flows and most of the same commands and command options.

The following topics describe how to run TetraMAX II, including the specific differences between TetraMAX II and TetraMAX ATPG:

- [Launching TetraMAX II](#)

How to use the `tmax2` command to start TetraMAX II.

- [TetraMAX II Commands](#)

Lists the specific commands that are affected by TetraMAX II.

- [Setting the Thread Count](#)

Describes the process for setting the thread count using the `-num_threads` option of the `set_atpg` command and the `set_simulation` command.

- [Running ATPG](#)

Guidelines for running ATPG in TetraMAX II, including how to handle parallel strobe file generation and SDC Timing exceptions for stuck-at fault ATPG.

- [Running Simulation and Fault Simulation](#)

How to avoid mismatches when running simulation or fault simulation in TetraMAX II.

- [Reporting Power](#)

Describes how to achieve optimal power reduction in TetraMAX II using the `report_power` command.

- [Running Diagnostics](#)

The diagnostics process uses multithreading for good machine simulation. All other diagnostics operations use a single process.

Note the following:

- Some TetraMAX ATPG command options, such as those that control pattern merging effort, are not relevant for TetraMAX II and are ignored. All other commands within the TetraMAX environment are shared between TetraMAX II and TetraMAX ATPG.
- For a complete description of how TetraMAX II handles various command options compared to TetraMAX ATPG, see "[Command Option Support](#)."

Launching TetraMAX II

To start TetraMAX II, specify the `tmax2` executable located in the `$SYNOPSYS/bin/` directory. You can set up the same environment and include the same options and input used by the `tmax` executable, as shown in the following steps:

1. Set your environment.

```
setenv SYNOPSYS /synopsys/m_branch/
set path =($SYNOPSYS/bin $path)
setenv LM_LICENSE_FILE synopsys_licenses/my_license.lic:$LM_
LICENSE_FILE
```

2. Do one of the following:

- Launch TetraMAX in shell mode using the `-shell` option with the `tmax2` command.
`tmax2 my_command_file -shell`
- Launch the TetraMAX GUI by specifying the `tmax2` command without the `-shell` option.
`tmax2 my_command_file`

The following table summarizes the various methods you can use to invoke TetraMAX ATPG.

Table TetraMAX II Invocation Methods

Invoke With¹	You Get	Process List
<code>tmax2</code>	64-bit kernel plus GUI	<code>tmax2, tmax64, tmaxgui</code>
<code>tmax2 -shell</code>	64-bit kernel	<code>tmax2, tmax64</code>
<code>tmax2 -man</code>	Online help	<code>tmax2, HTML browser</code>
<code>tmax2 -help</code>	List of options	

¹ - The order of switches is not important.

2 - The `tmax2` process invokes the shell. The CPU is idle after the kernel launches, but remains open so that the kernel has a transcript window in which to display output.

3 - The `tmax2` and `tmax2 -shell` commands sometimes invoke the `stil2verilog` command.

For a complete list of options associated with TetraMAX II, see "[tmax and tmax2 Command Syntax](#)."

TetraMAX II Commands

TetraMAX II replaces the existing TetraMAX ATPG application when you use the following commands:

- `report_power` – See "[Reporting Power in TetraMAX II](#)" for more information.
- `run_atpg` – See "[Running ATPG in TetraMAX II](#)" for more information.
- `run_diagnosis` – See "[Running Diagnostics in TetraMAX II](#)" for more information.
- `run_fault_sim` – See "[Running Simulation and Fault Simulation](#)" for more information.
- `run_simulation` – See "[Running Simulation and Fault Simulation](#)" for more information.

All other commands within the TetraMAX environment are shared between TetraMAX II and TetraMAX ATPG.

Note that because TetraMAX II uses a multithreaded architecture, it is not necessary to specify the scan mode, such as basic scan mode, full-sequential scan mode, or fast-sequential scan mode.

See Also

[TetraMAX II Command Option Support](#)

Setting the Thread Count in TetraMAX II

When TetraMAX II starts, it uses eight threads by default.

To change the default values, use the `-num_threads` option with the `set_atpg` command and the `set_simulation` command. Make sure to specify a positive integer, and always use the same number of threads for both commands. Also, make sure to specify this option before the `run_drc` or `read_image` commands.

```
set_atpg -num_threads 12  
set_simulation -num_threads 12
```

There is no limit to the number of cores you can use. However, performance has not been optimized for more than 20 cores, so this is a good limit.

You can display the number of threads currently in use for ATPG or simulation using the `report_settings` command. If you specify the `report_settings atpg` command or the `report_settings simulation` command, the following message is displayed:

```
num_threads=8
```

Note: Image files written after changing the `-num_threads` settings cannot be read into TetraMAX ATPG, although TetraMAX II can read image files written by TetraMAX ATPG.

When TetraMAX II is enabled, multicore settings for TetraMAX ATPG are not used for commands that use threads. They are used for commands that are not supported with threads. Multicore settings are saved in image files and can be disabled after they are read by setting the `-num_processes` option to 0 in the `set_atpg` and `set_simulation` commands, as shown in the following example:

```
set_atpg -num_processes 0  
set_simulation -num_processes 0
```

Running ATPG in TetraMAX II

When running TetraMAX II, you can use the `run_atpg` command without any options. This generally results in the best overall performance and pattern count. You can also use the –

`auto_compression` option, but its only function is to add all faults to the fault list if the fault list is empty.

TetraMAX II does not generate chain test or diagnostics patterns. When TetraMAX II is enabled, chain test or diagnostics patterns are created by TetraMAX ATPG. The remaining patterns are generated in TetraMAX II.

TetraMAX II supports parallel strobe data (PSD) file generation with the `run_simulation` command. However, a PSD file should not be generated using the `run_atpg` command in TetraMAX II. If you attempt to run this process in TetraMAX II, warning messages are printed. Although TetraMAX II can create a PSD file using the `run_atpg` command, it is unusable.

For more information on generating the PSD file, see "Using the `run_simulation` command to create the PSD File."

For complete information on the ATPG process, see "[Running ATPG](#)."

See Also

[Two-Cycle Patterns for Stuck-At Faults](#)

[SDC Timing Exceptions for Stuck-AT ATPG and Simulation](#)

Two-Cycle Patterns for Stuck-At Faults

If you use TetraMAX II to run ATPG on stuck-at faults, most of the patterns that it generates use a single clock cycle. Some patterns may contain two clock cycles even when fast-sequential ATPG is not enabled. These two-cycle patterns make it difficult to compare TetraMAX II results to TetraMAX ATPG results.

To prevent the generation of two-cycle patterns, use the `-noextra_cycle` option `set_atpg` command:

```
set_atpg -noextra_cycle
```

To force TetraMAX II ATPG to always use the same clocking for both cycles, use the `-common_launch_capture_clock` option of the `set_delay` command:

```
set_delay -common_launch_capture_clock
```

Note that this option has no effect on TetraMAX ATPG when targeting stuck-at faults.

SDC Timing Exceptions for Stuck-At ATPG and Simulation

When using the [stuck-at fault model](#), TetraMAX II simulates the effect of Synopsys design constraints (SDC) timing exceptions differently than TetraMAX ATPG.

In TetraMAX ATPG, transitions that occur on the last scan shift are simulated to determine if they create setup violations. This behavior is inconsistent with the normal [transition delay fault model](#) behavior, and is inconsistent with most design setups.

TetraMAX II simulates SDC timing exceptions the same way for both the stuck-at and transition delay fault models. The last scan shift is not considered, and only capture clocks are considered as launch clocks for setup violations. If a timing problem exists between the last scan shift and the first capture clock, you can fix it using the `write_patterns -use_delay_capture_start` command.

See Also

[Specifying Timing Exceptions From an SDC File](#)

Running Simulation and Fault Simulation in TetraMAX II

The TetraMAX II simulator resolves 0 or 1 values in situations where the TetraMAX ATPG non-threaded simulator reports an X value. This means that patterns generated using threaded ATPG often mismatch if they are simulated using the non-threaded simulator. In this case, mismatches are reported, as follows:

`got=X`

To avoid mismatches, make sure that patterns generated using threaded ATPG are always simulated using the TetraMAX II threaded simulator. Patterns generated using non-threaded ATPG may be safely simulated using the TetraMAX II threaded simulator.

See Also

[Fault Simulation](#)

Reporting Power in TetraMAX II

TetraMAX power-aware ATPG calculates the fanout of clock-gating structures and other clock sources during DRC. This approach enables you to use this `report_power` command to specify capture and shift power budgets for generating power-aware ATPG vectors.

To ensure optimal accuracy, the `report_power` command uses the TetraMAX II threaded simulator to resimulate patterns generated by TetraMAX II. Consequently, the `set_atpg -calculate_power` command is ignored by TetraMAX II. In TetraMAX ATPG, this command improves the runtime performance of the `report_power` command by enabling the ATPG process to calculate the switching activity of scan flip-flops. However, this switching calculation is unnecessary in TetraMAX II.

TetraMAX II also supports hardware-assisted shift power reduction, which decreases average shift power and pattern count compared to an ATPG-only approach through the use of independently controlled scan chain groups implemented in DFTMAX or DFTMAX Ultra.

See Also

[Power-Aware ATPG](#)

Running Diagnostics in TetraMAX II

The `run_diagnosis` command uses TetraMAX II threading for the good machine simulation only. All other diagnostics operations use a single process.

Threaded simulation is not used for good machine simulation of chain defects. If a failure log includes failing chain test patterns, threading is not used.

You should avoid using patterns generated from multithreaded ATPG with single-process diagnosis. Otherwise, M266 warning messages, indicating that failures were ignored due to X measures, may be printed and diagnosis may be less accurate.

See Also

[Diagnosing Manufacturing Test Defects](#)

TetraMAX II Messaging

This section describes the messages that are unique to TetraMAX II. When running TetraMAX II, make sure you set the following commands for better support if a question or issue arises:

```
set_messages -level expert  
set_atpg -verbose
```

The log file produced by TetraMAX II is nearly identical to the file produced by TetraMAX ATPG. The exceptions are the `report_power`, `run_atpg`, `run_diagnosis`, `run_fault_sim`, and `run_simulation` commands, since these commands are run in TetraMAX II.

To ensure that TetraMAX II is enabled, look for messages similar to the following example after the first time you specify the `run_atpg`, `run_diagnosis`, `run_fault_sim`, or the `run_simulation` commands:

```
Parallel simulation data created for 16 threads #TLA_input_  
gates=2312 #levels=1 CPU_time=0.00 sec.  
Parallel simulation data created for 16 threads #internal_  
gates=1876366 #levels=6 CPU_time=1.02 sec.
```

The numbers will differ depending on run-specific details. The essential information is that threads are being used, as shown in bold in the previous example (for 16 threads).

Each TetraMAX II run command is followed by a particular message. For example, the following message is printed when you specify the `run_atpg` command:

```
ATPG will be performed on 16 threads using words of size 64  
#collapsed_faults=3100718/5730594.
```

The following message is printed when you specify the `run_simulation` command:

```
Simulation will be performed on 16 threads using words of size 64.
```

When using X-tolerant DFTMAX compression, TetraMAX II reduces the need to insert padding patterns until after the ATPG process completes. As pattern generation continues, TetraMAX II checks the newly generated patterns to determine if they satisfy pending X-tolerant mode requirements. Patterns that satisfy the requirements are used instead of a padding pattern.

In some cases, patterns are stored with different pattern numbers than the original numbers they were assigned when they were generated. You can report the number of these patterns by setting the `set_messages -level expert` command and the `set_atpg -verbose` command. The M988 message is printed at the end of each pattern interval:

```
There are 8 patterns out of 3084 that have pending observe needs  
(M988) .
```

When you specify the `set_atpg -patterns` command, the number of pending observe needs is counted when TetraMAX II determines if the pattern limit has been reached. As a result, the interval pattern counts printed by the default messaging may provide inaccurate data about how many more patterns will be generated.

Note that TetraMAX II uses a smaller pattern interval than the 32 patterns used by TetraMAX ATPG. Usually the pattern interval used by TetraMAX II is 21 patterns for the stuck-at fault model, or 12 patterns for the transition delay fault model.

The M724 message warns you that certain settings will be ignored. In this case, threaded ATPG is run without considering the settings, as shown in the following example:

```
TEST-T> run_atpg  
The following settings are ignored for threaded ATPG. (M724)  
set_atpg -decision random  
set_atpg -lete_fastseq  
set_atpg -basic_min_detects_per_pattern 17 40  
Starting threaded ATPG with 8 threads. (M733)
```

The M729 message warns you that certain settings require the use of single-process or multicore ATPG. In this case, threaded ATPG is not run because it ignores the settings. For example:

```
TEST-T> run_atpg  
Warning: The following setting requires single-process/multi-core  
ATPG. (M729)  
set_delay -launch_cycle last_shift  
Starting single-process ATPG. (M733)
```

The M733 message informs you which ATPG or simulation engine is being used by the current command.

```
Starting threaded simulation with N threads. (M733)  
Starting threaded ATPG with N threads. (M733)  
Starting threaded fault simulation with N threads. (M733)
```

TetraMAX II Command Option Support

Some command options have a different effect in TetraMAX II than TetraMAX ATPG or are not supported in TetraMAX II. This section describes how TetraMAX II handles various command options. These limitations apply to the M-2016.12 release and are subject to change in future releases.

run_atpg

The following `run_atpg` command option is not relevant to TetraMAX II and is silently ignored:

- `basic_scan_only`

These options switch to the TetraMAX ATPG engine, even though TetraMAX II is running:

- `full_sequential_only`
- `-distributed`
- `-ndetects`
- `-random`
- `-resolve_differences`

These options are ignored by TetraMAX II:

- `-auto_compression` (the only effect of this option is to add all faults if the fault list is empty)
 - `-optimize_patterns`
 - `-rerun_for_timing_exceptions`
-

run_fault_sim

The following `run_fault_sim` command options switch to the TetraMAX ATPG engine:

- `-checkpoint`
 - `-detected_pattern_storage`
 - `-distributed`
 - `-nodrop_faults`
 - `-sequential`
 - `-store`
-

run_simulation

The following `run_simulation` command options switch to the TetraMAX ATPG engine:

- `-bridge`
- `-fast`
- `-slow`
- `-stuck` (this option is supported if only one instance name argument is specified)

- `-sequential`
 - `-sequential_update`
 - `-store`
 - `-update`
-

set_atpg

The following `set_atpg` command options are silently ignored by TetraMAX II:

- `-capture_cycles 0` (two-cycle patterns may be generated)
- `-shared_io_analysis`

The following option has a different effect in TetraMAX II than TetraMAX ATPG. In this case, ATPG uses threading provided by TetraMAX II, but resimulation uses a single-process simulation provided by TetraMAX ATPG:

- `-resim_atpg_patterns`

These options switch to the TetraMAX ATPG engine:

- `-allow_clockon_measures`
- `-checkpoint`
- `-fast_path_delay`
- `-full *` (all full sequential settings are unsupported)

These options are ignored by TetraMAX II:

- `-abort_limit` (values up to 100 are used, but values greater than 100 are ignored)
 - `-basic_min_detects_per_pattern`
 - `-calculate_power`
 - `-decision`
 - `-fast_min_detects_per_pattern`
 - `-lete_fastseq`
 - `-merge`
 - `-new_capture`
 - `-num_processes`
 - `-parallel_strobe_data_file` (the PSD file is created but it is incorrect)
 - `-save_patterns`
-

set_delay

Many `set_delay` options are used only for slack-based test or only for path delay faults. Neither of these features is supported by TetraMAX II, so the corresponding options are not supported. Other options are listed here.

These options are silently ignored by TetraMAX II:

- `-nodisturb_clock_grouping`
- `-extra_force`

This option has the same effect in both TetraMAX ATPG and TetraMAX II. But in TetraMAX II it is used during stuck-at ATPG to constrain two-cycle patterns:

- `-common_launch_capture_clock`

This option switches to the TetraMAX ATPG engine, except when `system_clock` is specified as a value, which is supported by TetraMAX II:

- `-launch_cycle`
-

set_drc

The following `set_drc` command option is silently ignored by TetraMAX II, except when the `-dynamic` option is specified, which is supported by TetraMAX II:

- `-clock`

These options switch to TetraMAX ATPG:

- `-clock_constraints`
 - `-internal_clock_timing`
-

TetraMAX II Limitations

The following list of limitations for TetraMAX II includes features of the design or protocol, but does not include limitations of command options. These limitations apply to the M-2016.12 release and are subject to change in future releases. For command option limitations, see "[TetraMAX II Command Option Support](#)."

- **DFTMAX Ultra compression**

TetraMAX II runs on designs with DFTMAX Ultra compression, but does not generate correct patterns. You should avoid using designs with DFTMAX Ultra compression.

- **Compression technologies other than high X-tolerance DFTMAX**

When compression mode is used, and the compression technology is not DFTMAX with high X-tolerance, an M729 message is printed and TetraMAX ATPG is run instead.

Shared CODEC I/O is supported. Unsupported technologies include DFTMAX with default X-tolerance, DFTMAX Serializer, and DFTMAX LogicBIST.

- **Memory models**

If the design contains memory models, TetraMAX II treats them as black boxes. When fast-sequential ATPG starts and the design includes memory models, a message like this is printed:

Fast-sequential ATPG with no RAM support will be performed on
4 threads using words of size 64 #collapsed_faults=2623/4127

- **Launch on Extra Shift (LOES) and Launch on Last Shift**

TetraMAX II does not generate patterns with an LOES protocol. If this is attempted, an M729 message is printed and TetraMAX ATPG is run instead.

- **Synchronous LOES and Clocking Procedures**

TetraMAX II does not run with either synchronous LOES or internal clocking procedures. If this is attempted, TetraMAX ATPG is run instead and an M729 message is printed. However, the `run_simulation` and `run_fault_sim` commands are supported for LOES patterns.

- **Fault models that switch to TetraMAX ATPG:**

The following fault models are not supported in TetraMAX II:

- Path Delay
- IDDQ
- Hold Time

- **Full-sequential ATPG and full-sequential simulation**

Full-sequential ATPG and full-sequential simulation cannot be run in TetraMAX .

- **Fault models that switch to TetraMAX ATPG**

Dynamic bridging (supported by the `run_fault_sim` command).

4

Command Interface

TetraMAX ATPG provides an interactive command interface in the TetraMAX GUI, and menus and buttons in the TetraMAX GUI. The command interface includes a command language that you can use to execute command sequences in batch mode.

Online Help is available on commands, error messages, design flows, and many other TetraMAX topics.

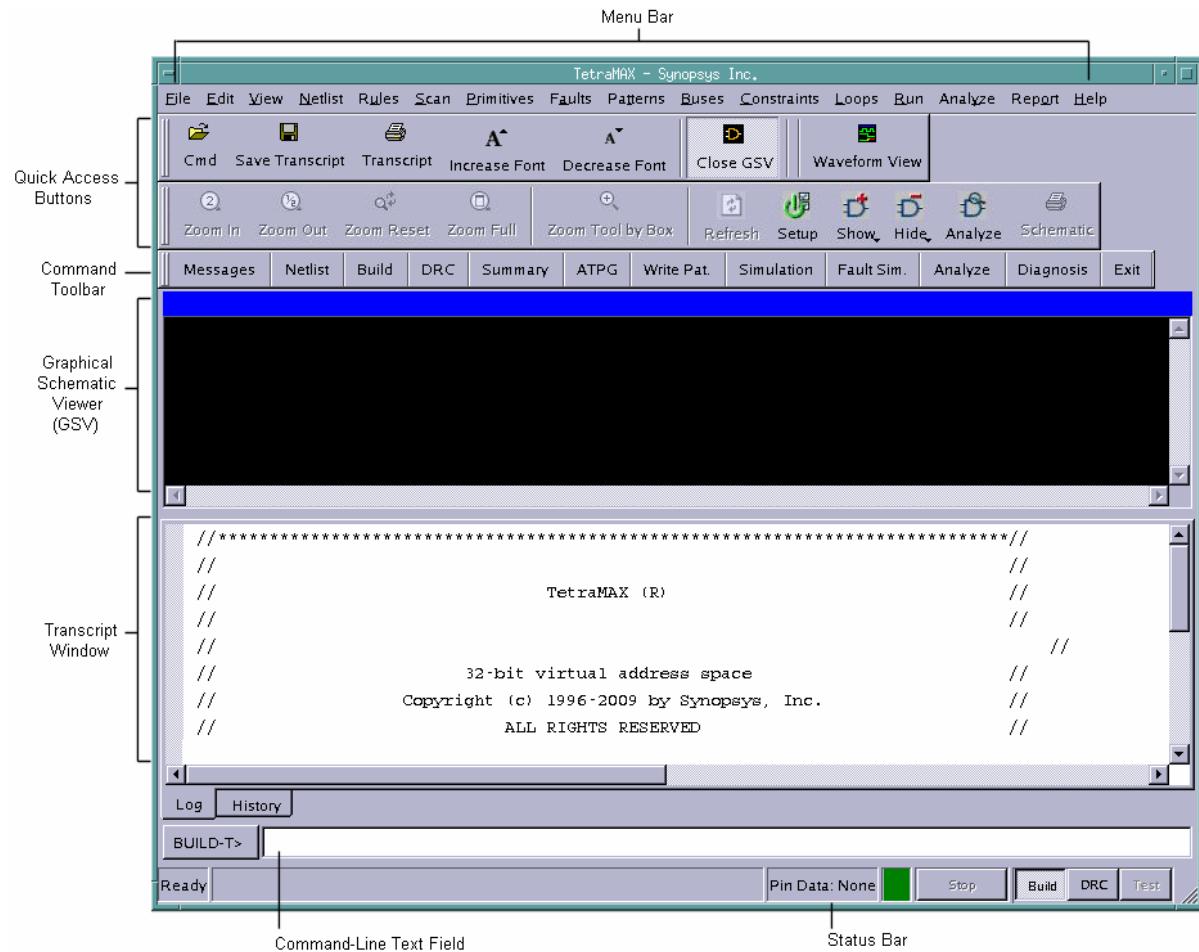
The following sections describe the components of the TetraMAX command interface:

- [TetraMAX GUI Main Window](#)
- [Command Entry](#)
- [Transcript Window](#)
- [Online Help](#)

TetraMAX GUI

Figure 1 shows the main window of the TetraMAX graphical user interface (GUI). The major components in this window are (from top to bottom): the menu bar, the quick access buttons, the command toolbar, the graphical schematic viewer (GSV) toolbar and window, the transcript window, the command-line text field, and the status bar.

Figure 1: TetraMAX GUI Main Window



The GSV window is not displayed when you start TetraMAX ATPG. It first appears when you execute a command that requests a schematic display. For more information on the GSV window, see "[Using the Graphical Schematic Viewer](#)."

The status bar, located at the very bottom of the main window, contains the STOP button and displays the state of TetraMAX ATPG (Kernel Busy/Ready), pin/block reference data, Pin Data details, red/green signal indicating the kernel busy/ready status, and the command mode indicator. For more information, see "[Command Mode Indicator](#)."

See Also

[Using the Graphical Schematic Viewer](#)

[Using the Hierarchy Browser](#)[Using the Simulation Waveform Viewer](#)

Command Entry

The main window provides three ways to interactively enter commands:

- Select from pull-down menus at the top of the main window.
- Use the command buttons in the command toolbar or the graphical schematic viewer (GSV) toolbar.
- Type commands in the command-line window.

The pull-down menus and command buttons let you specify the command options in dialog boxes. The command-line window uses a command-line-based entry method.

The following sections describe how to perform command entry:

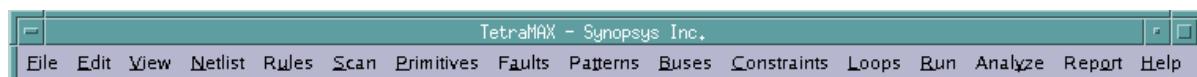
- [Menu Bar](#)
 - [Command Toolbar and GSV Toolbar](#)
 - [Command-Line Window](#)
 - [Commands From a Command File](#)
 - [Command Logging](#)
-

Menu Bar

The menu bar consists of a set of pull-down menus you use to select a required action. These menus provide the most comprehensive set of command selections.

Figure 1 shows the menu bar.

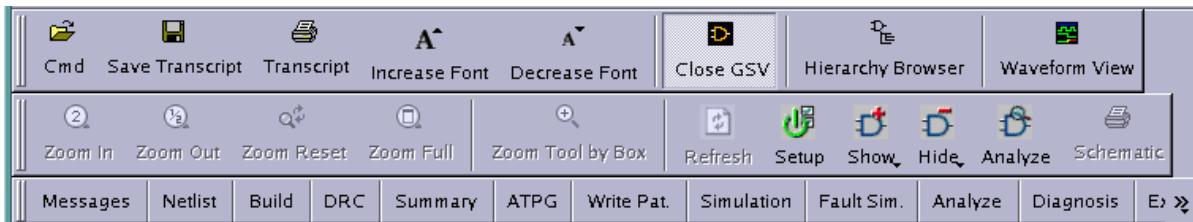
Figure 1: Menu Bar



Command Toolbar and GSV Toolbar

The command toolbar is a collection of buttons you use to run TetraMAX commands. These buttons provide a fast and convenient alternative to using the pull-down menus or command-line window. Similarly, the GSV toolbar provides a fast way to control the contents of the GSV window.

Figure 2 shows the command toolbar and GSV toolbar.

Figure 2: Command Toolbar and GSV Toolbar

By default, the command toolbar is displayed at the top of the main window, just below the menu bar. The GSV toolbar is displayed on the left side of the GSV window. Both toolbars are “dockable” — that is, you can move and “dock” them to any four sides of the GSV window, or use them as free-standing windows.

To move the toolbar, position the pointer on the border of the toolbar (outside any of the buttons), press and hold the mouse button, drag the toolbar to the required location, and release the mouse button.

Command-Line Window

The command-line window is located between the transcript window and the status bar. The following components comprise the command-line window:

- [Command Mode Indicator](#)
- [Command-Line Entry Field](#)
- [Command Continuation](#)
- [Command History](#)
- [Stop Button](#)

[Figure 3](#) shows the command-line window.

Figure 3: Command-Line Window

Command Mode Indicator

The command mode indicator, located to the left of the status bar, displays either `BUILD-T>`, `DRC-T>`, or `TEST-T>`, depending on the operating mode currently enabled.

To change the command mode, use the Build, DRC, and Test buttons, located at the far right. The buttons are dimmed if you cannot change to that mode in the current context. To change to one of these modes, click the corresponding button. If an attempt to change the current mode fails, the command-line window remains unchanged and an error message appears in the transcript window.

Command-Line Entry Field

You type TetraMAX commands in the command-line text field at the bottom of the screen. To enter a command, click in the text field, type the command, and either click Submit or press Enter. After it has been entered, the command is echoed to the transcript, stored in the command history, and sent to TetraMAX ATPG for execution.

You can use the editing features Cut (Control-x), Copy (Control-c), and Paste (Control-v) in the command-line text field. If the command is too long for the text field, the text field automatically scrolls so that you can continue to see the end of the command entry.

The command line supports multiple commands. You can enter more than one command on the command line by separating commands with a semicolon.

You can enter two exclamation characters (!!) to repeat the last command. Entering !!xyz repeats the most recent command that begins with the string xyz.

You can use the arrow keys to queue the command line. If you are in Tcl mode, TetraMAX ATPG includes automatic command completion feature. This feature also applies to directory and file name completion in both native mode and Tcl mode.

Command Continuation

To continue a long command line over multiple lines, place at least one space followed by a backslash character (\) at the end of each line.

[Example 1](#) shows an example of the add_atpg_primitives command in Tcl mode. This command defines an ATPG primitive connected to multiple pins. Note the use of curly brackets in Tcl mode for specifying lists. Each pin path name is presented on a separate line using the backslash character. All five lines are treated as a single command. [Example 2](#) shows the same command example in native mode.

Example 1: Command continuation across multiple lines (Tcl mode)

```
BUILD-T> add_atpg_primitives spec_atpg_prim1 equiv \
{ /BLASTER/MAIN/CPU/TP/CYCL/CDEC/U1936/in1 \
/BLASTER/MAIN/ALU_CORE/TP/CYCL/CDEC/U1936/in1 \
/BLASTER/MAIN/ALU_CORE/TP/CYCL/CDEC/U16/in2 \
/BLASTER/MAIN/ALU_CORE/TP/CYCL/CDEC/U13/in0 }
```

Example 2: Command continuation across multiple lines (native mode)

```
BUILD> add atpg primitives spec_atpg_prim1 equiv \
/BLASTER/MAIN/CPU/TP/CYCL/CDEC/U1936/in1 \
/BLASTER/MAIN/ALU_CORE/TP/CYCL/CDEC/U1936/in1 \
/BLASTER/MAIN/ALU_CORE/TP/CYCL/CDEC/U16/in2 \
/BLASTER/MAIN/ALU_CORE/TP/CYCL/CDEC/U13/in0
```

Command History

The command history contains commands you have entered at the command line. To run a previous command, use the arrow keys to highlight the required command, and then press Enter.

Another way to view a list of recent commands is to use the `report_commands -history` command.

Stop Button

The Stop button is located to the right of the status bar. If TetraMAX ATPG is idle, the Stop button is dimmed. If TetraMAX ATPG is busy processing a command (and the command mode indicator displays <Busy>), the button is active and is labeled "Stop." Click this button to halt processing of the current command. TetraMAX ATPG might take several seconds to halt the activity.

You can interrupt a multicore ATPG process by clicking the Stop button. At this point, the master process sends an abort signal to the slave processes and waits for the slaves to finish any ongoing interval tasks. If this takes an extended period of time, you can click the Stop button twice; this action causes the master process to send a kill signal to the slaves, and the prompt will immediately return. Note that the clicking the Stop button twice will terminate all slave processes without saving any data gathered since the last communication with the master. For more information on multicore ATPG, see "[Running Multicore ATPG](#)."

Commands From a Command File

You can submit a list of commands as a file and have TetraMAX ATPG execute those commands in batch mode. In Tcl mode, any line starting with # is treated as a comment and is ignored. In native mode, any line starting with a double-slash (//) is ignored.

Although a command file can have any legal file name, for easy identification, you might want to use the standard extension .cmd (for example, specfile.cmd).

To run a command file, click the Cmd File button in the command toolbar, or enter the following in the command-line window:

```
> source filename
```

Command files can be nested. In other words, a command file can contain a source command that invokes another command file.

For an example of a command file, see "[Using Command Files](#)".

Note: The history list shows only the source filename command, not the commands executed in the command file.

Command Logging

Commands that you enter through menus, buttons, and the command-line window can be logged to a file along with all information reported to the transcript. By default, the command log contains the same information as the saved transcript. In addition, the command log contains comments from any command files that were used.

To turn on command logging (also called message logging), click the Set Msg button in the command toolbar to open the Set Messages dialog box, or type the following command:

```
> set_messages log spec_logfile.log
```

If the log file already exists, an error message is displayed unless you add the optional -replace or -append options, as follows:

```
> set_messages log spec_logfile.log -replace  
> set_messages log spec_logfile.log -append
```

If you intend to use the log file as an executable command file, use the `-leading_comment` option of the `set_messages` command. In this case, TetraMAX ATPG writes out the comment lines starting with either a pound sign (#) in Tcl mode, or a double slash in Native mode, so that those lines are ignored when you use the log file as a command file.

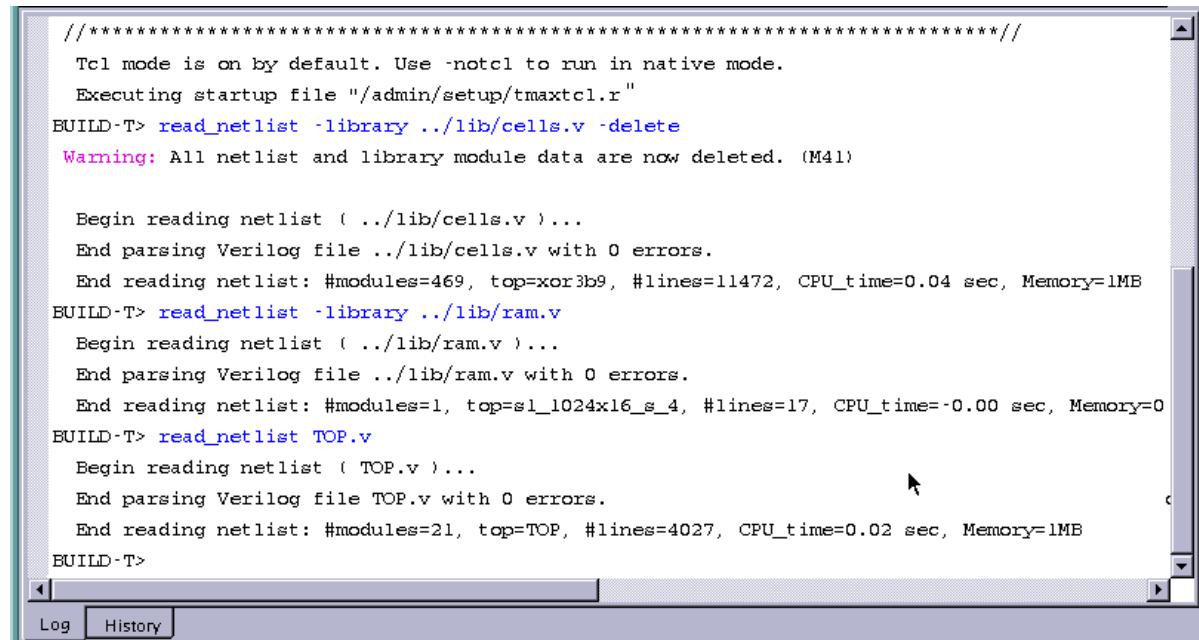
Transcript Window

The transcript window is a read-only, scrollable window that displays the session transcript, including text produced by TetraMAX ATPG and commands entered in the command line and from the GUI. The transcript provides a record of all activities carried out in the TetraMAX session.

The following sections describe the transcript window:

- [Setting the Keyboard Focus](#)
- [Using the Transcript Text](#)
- [Selecting Text in the Transcript](#)
- [Copying Text From the Transcript](#)
- [Finding Commands and Messages in the Transcript](#)
- [Saving or Printing the Transcript](#)
- [Clearing the Transcript Window](#)

Figure 1: Transcript Window



The screenshot shows a transcript window with a light gray background and a dark gray border. The window contains a log of command execution:

```
//*****  
Tcl mode is on by default. Use -notcl to run in native mode.  
Executing startup file "/admin/setup/tmaxtcl.r"  
BUILD-T> read_netlist -library ..\lib\cells.v -delete  
Warning: All netlist and library module data are now deleted. (M41)  
  
Begin reading netlist ( ..\lib\cells.v )...  
End parsing Verilog file ..\lib\cells.v with 0 errors.  
End reading netlist: #modules=469, top=xor3b9, #lines=11472, CPU_time=0.04 sec, Memory=1MB  
BUILD-T> read_netlist -library ..\lib\ram.v  
Begin reading netlist ( ..\lib\ram.v )...  
End parsing Verilog file ..\lib\ram.v with 0 errors.  
End reading netlist: #modules=1, top=s1_1024x16_s_4, #lines=17, CPU_time=-0.00 sec, Memory=0  
BUILD-T> read_netlist TOP.v  
Begin reading netlist ( TOP.v )...  
End parsing Verilog file TOP.v with 0 errors.  
End reading netlist: #modules=21, top=TOP, #lines=4027, CPU_time=0.02 sec, Memory=1MB  
BUILD-T>
```

At the bottom of the window, there is a toolbar with two buttons: "Log" and "History".

Setting the Keyboard Focus

Setting the keyboard focus in the transcript window allows you to use keyboard shortcuts and some keypad keys in the transcript window. You set the keyboard focus by clicking anywhere in the transcript window. A blinking vertical-bar cursor appears in the text where you have set the focus.

Using the Transcript Text

The transcript window has the editing features Copy, Find, Find Next, Save, Print, and Clear. If the cursor is in the transcript window, you can use keyboard shortcuts for these editing features. Otherwise, open the transcript window pop-up menu by clicking anywhere in the transcript window with the right mouse button.

You can look at any part of the entire transcript by using the horizontal and vertical scroll bars. Notice that if you scroll up, you will not be able to see new text being added to the bottom of the transcript.

If the cursor is in the transcript window, you can use the following keypad keys:

- **Up / Down arrow** -- Moves the cursor up or down one line.
- **Left / Right arrow** -- Moves the cursor left or right one character position.
- **Page Up / Page Down** -- Scrolls the transcript up or down one page. A “page” is the amount of text that can be displayed in the transcript window at one time.
- **Home / End** -- Moves the cursor to the beginning or end of the current line.

- **Control-Page Up / Control-Page Down** -- Moves the cursor to the top or bottom of the current transcript page.
 - **Control-Home** -- Moves the cursor to the beginning of the first line in the transcript.
 - **Control-End** -- Moves the cursor to the end of the last line in the transcript.
-

Selecting Text in the Transcript

To select part or all of the text in the transcript window, press the left mouse button at the beginning of the required text, drag to the end of the required text, and release the mouse button. The selected text is highlighted.

Copying Text From the Transcript

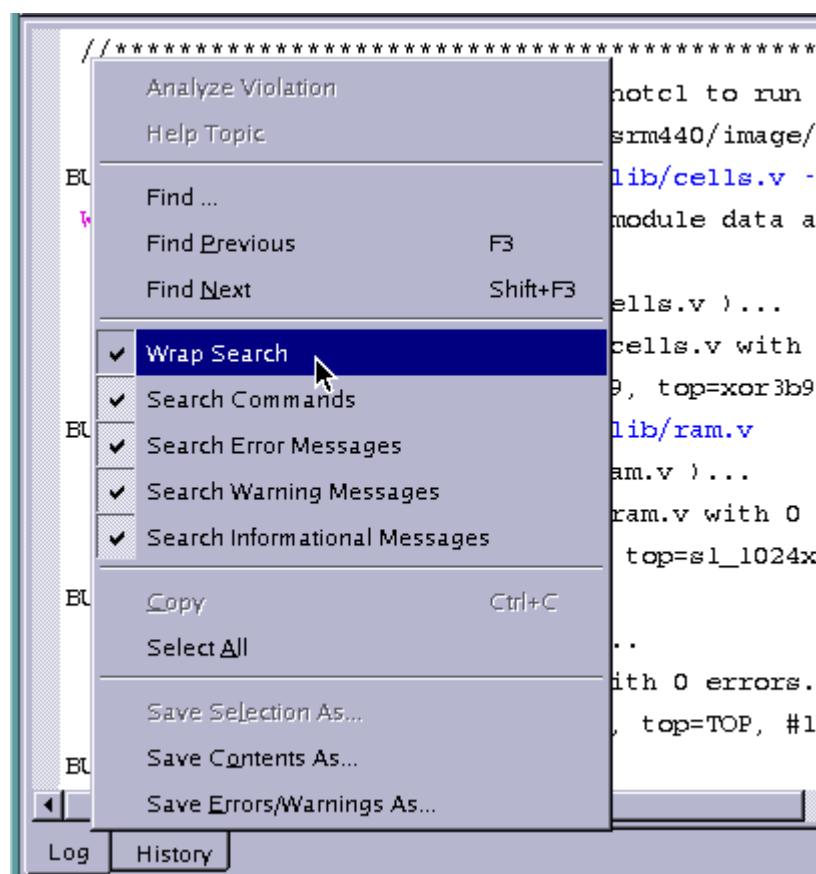
You can copy selected text from the transcript window to the Clipboard. Use the keyboard shortcut Control-c, or choose Copy from the pop-up menu that appears when you press the right mouse button. The Copy command is disabled if no text has been selected.

Finding Commands and Messages in the Transcript

To find commands and messages in the transcript window, right-click inside the Transcript window, and then click the box or boxes that reference the type of search you want to perform (i.e., Wrap Search, Search Commands, Search Error Messages, Search Informational Messages). Click “Find Next” to find the next occurrence of a command or message in the transcript after the current cursor position or “Find Previous” to find the previous occurrence of a command or message.

[Figure 2](#) shows how to find text in the transcript.

Figure 2: Finding Text in the Transcript



Saving or Printing the Transcript

To save selected text from the transcript window, select the text you want to save, then right-click anywhere in the Transcript window, and choose “Save Selection As...” in the pop-up dialog. To save the entire contents of the Transcript, right-click in the Transcript window, and choose “Save Contents As...” To print the transcript, use the keyboard shortcut Control-p.

Clearing the Transcript Window

To clear the transcript window, choose Edit> Clear All. If the cursor is in the transcript window, you can use the keyboard shortcut Control-Delete. This removes all of the existing text from the window.

Online Help

TetraMAX ATPG provides Online Help in the following forms:

- [Text-only Help](#) on TetraMAX commands, displayed in the transcript window. In Tcl mode, enter the command name, followed by the `-help` option. In non-Tcl (native) mode, use the `help` command in the command-line window.
 - [Browser-Based Help](#) on commands, design flows, error messages, design rules, fault classes, and many other topics.
-

Text-Only Help

To obtain text-only help on a command in Tcl mode, enter the name of the command, followed by the `-help` option of the command-line window. In non-Tcl (native) mode, use the `help` command, followed by the name of the command.

A Tcl mode text-only help example is as follows:

```
BUILD-T> set_workspace_sizes -help
Usage: set_workspace_sizes Set WOrkspace Sizes
[-connectors] (maximum number of fanout connections supported)
[-decisions] (maximum active decisions)
[-drc_buffer_size] (maximum DRC buffer size)
[-line] (maximum line length)
[-string] (maximum string length)
[-command_line] (command line length)
[-command_words] (command line words)
```

A native mode text-only help example is as follows:

```
BUILD> help set workspace sizes
Set WOrkspace Sizes [-Atpg_gates d]
[-CONnectors d] [-Decisions d] [-DRC_buffer_size d] [-Line d]
[-String d] [-COMMAND_Line d] [-COMMAND_Words d]
```

For a list of available command help topics, type the following command in the command-line window:

```
BUILD-T> report_commands -all
add_atpg_constraints add_atpg_primitives
add_capture_masks add_cell_constraints
add_clocks add_display_proc
add_delay_paths add_display_gates
add_distributed_processors add_equivalent_nofaults
add_faults add_net_connections
add_nofaults add_pi_constraints
...
```

Browser-Based Help

You can view detailed help on a wide range of TetraMAX topics by using browser-based Online Help. This section describes the following topics related to Online Help:

- [Launching Online Help](#)
- [Basic Components of Help](#)

Launching Online Help

You can launch TetraMAX Help by doing any of the following:

Selecting GUI Help Menu

From the menu bar in the GUI, choose Help > Table of Contents, or select a particular topic to open (i.e., Command Summary, Getting Started, Fault Classes, etc.). See [Figure 1](#).

Figure 1: Accessing Help Through the GUI Menu Bar



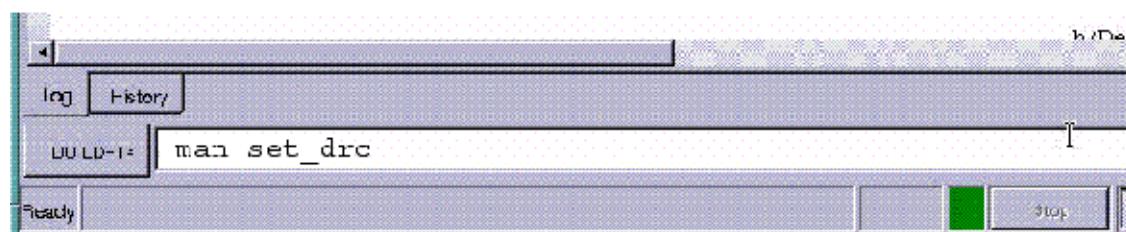
Entering Command in GUI Text Field

In the command-line text field of the GUI, use the following syntax to open a topic related to either a specific command (`set_drc`) or a message (i.e., M401):

```
> man command | message_id
```

See [Figure 2](#) for an example.

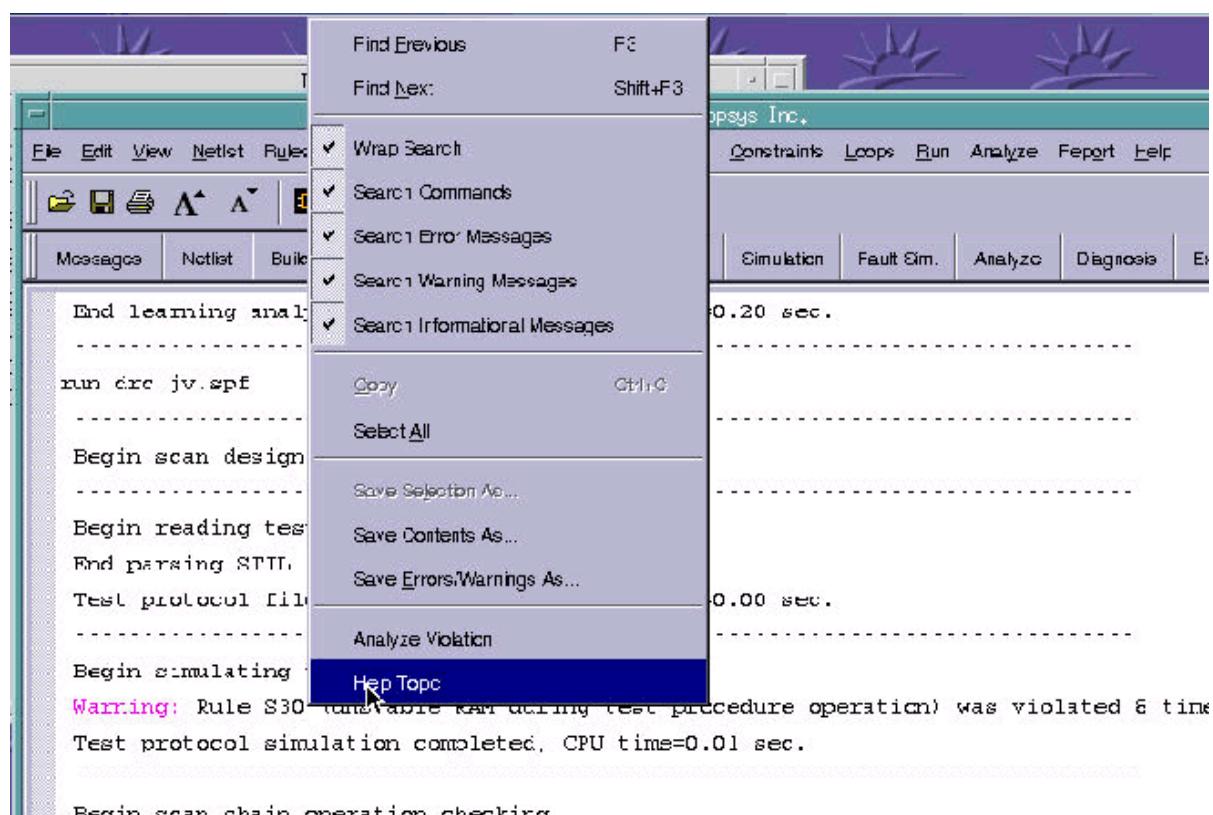
Figure 2: Opening a Specific Topic in TetraMAX Help From the Command-line Text Field



Right-Clicking On a Command Or Message

Right-click a particular command or message in the console window, then select Help Topic. The Help topic for the command or message will appear. See Figure 3 for an example.

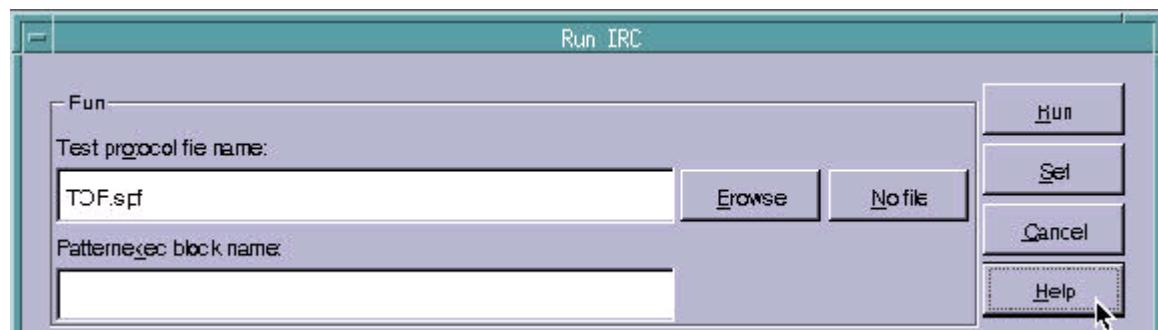
Figure 3: Right-Clicking on a Message (i.e., S30) to Bring Up Online Help



Click the Help Button in Dialog Box

Click the Help button within a dialog box to bring up a Help topic that describes the active dialog box. See Figure 4.

Figure 4: Accessing Help From a Dialog Box



Basic Components of Help

The TetraMAX Help system is displayed in a frameset that consists of three main components:

- The menu and button bars at the top.
- The navigation frame (for the table of contents, index, bookmarks, and full-text search) below the button bar on the left.
- The contents frame (where Help topics are displayed) below the button bar on the right.

5

Using the Graphical Schematic Viewer

The graphical schematic viewer (GSV) displays design information in schematic form for review and analysis. It selectively displays a portion of the design related to a test design rule violation, a particular fault, or some other design-for-test (DFT) condition. You use the GSV to find out how to correct violations and debug the design.

The following sections describe how to use the GSV for interactive analysis and correction of test design rule checking (DRC) violations and test pattern generation problems:

- [Getting Started With the GSV](#)
- [Displaying Symbols in Primitive or Design View](#)
- [Displaying Instance Path Names](#)
- [Displaying Pin Data](#)
- [Analyzing a Feedback Path](#)
- [Checking Controllability and Observability](#)
- [Analyzing DRC Violations](#)
- [Analyzing Buses](#)
- [Analyzing ATPG Problems](#)
- [Printing a Schematic to a File](#)

Getting Started With the GSV

The following sections describe how to get started using the GSV:

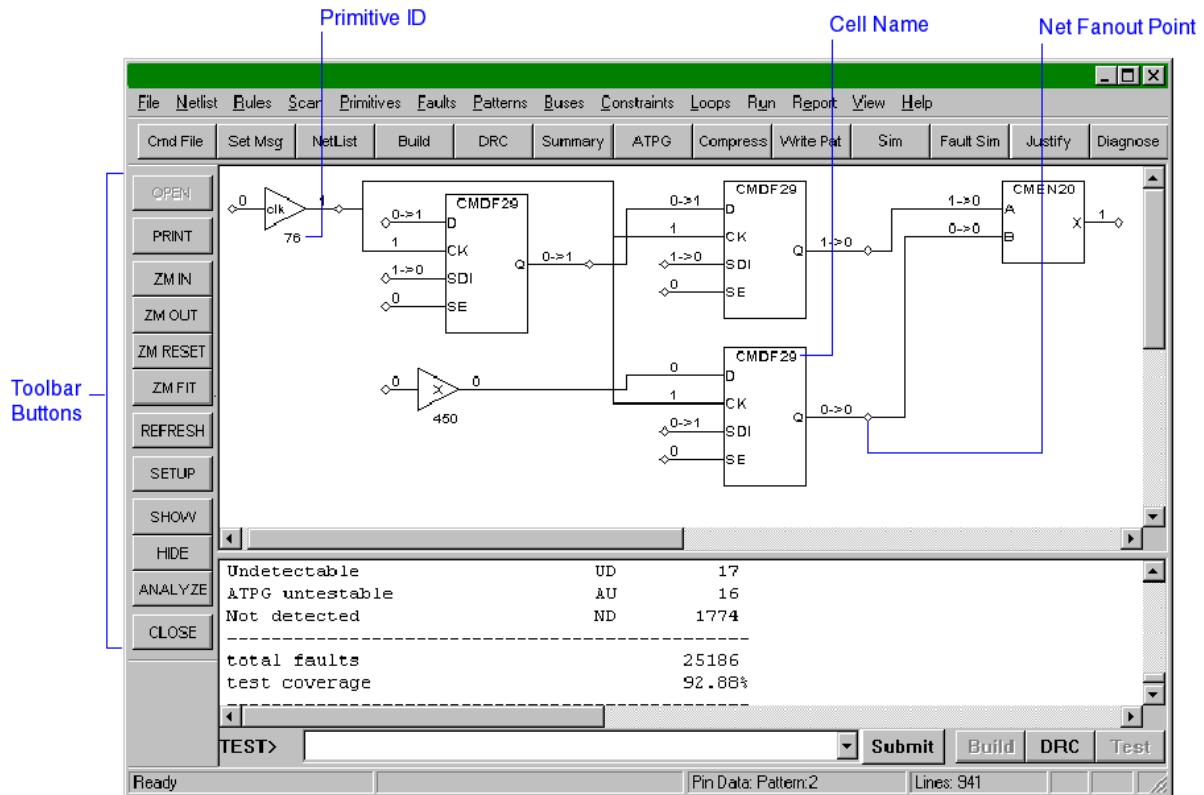
- [Using the SHOW Button to Start the GSV](#)
 - [Starting the GSV From a DRC Violation or Specific Fault](#)
 - [Navigating, Selecting, Hiding, and Finding Data](#)
 - [Expanding the Display From Net Connections](#)
 - [Hiding Buffers and Inverters in the GSV Schematic](#)
 - [ATPG Model Primitives](#)
-

Using the SHOW Button to Start the GSV

The following steps describe how to start the GSV and display a particular part of the design:

1. Click the SHOW button.
The SHOW menu appears, which lets you choose what to show: a named object, trace, scan path, and so on.
2. To display a named object, select Named.
The Show Block dialog box appears.
3. In the Block ID/PinPath Name text field, enter a primitive ID, instance, or pin path name to the object to display. (If you do not know what instance or pin names are available, enter 0; this is the primitive ID of the first primary input port to the top level.)
For information on the design's port names and hierarchy, review the list of top-level ports using the `report_primitives -ports` command.
4. Click the Add button.
Your entry is added to the list box.
5. Repeat steps 3 and 4 to add all the parts of the design that you want to view.
6. Click OK

[Figure 1](#) shows the TetraMAX GUI main window split by the movable divider. The top window shows a GSV schematic containing the specified objects. The bottom window contains the transcript.

Figure 1: GSV in the TetraMAX GUI Main Window

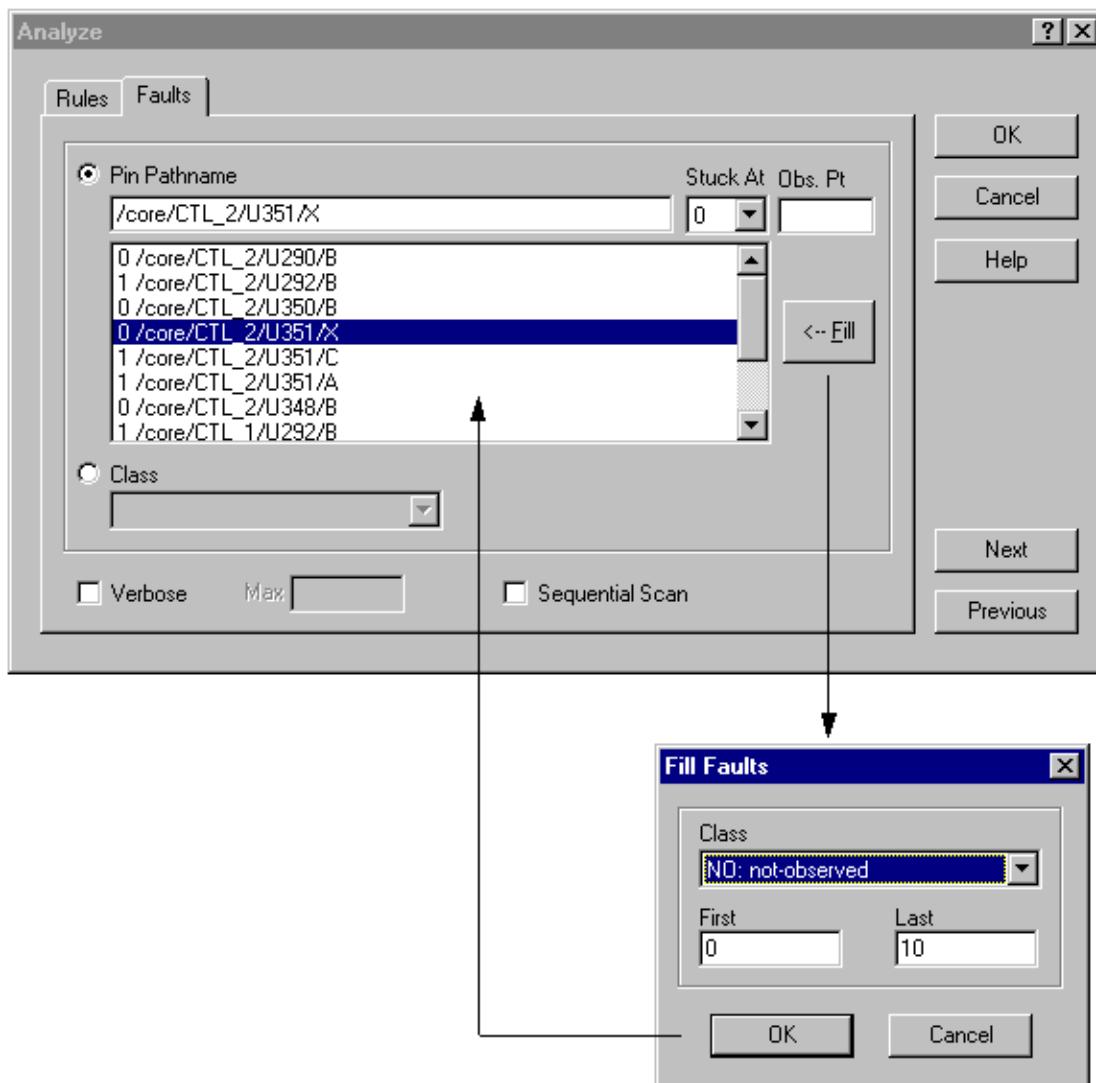
Starting the GSV From a DRC Violation or Specific Fault

You can start the GSV and view a specific DRC violation by using the Analyze dialog box, as shown in the following steps:

1. Click the ANALYZE button in the GSV toolbar.

The Analyze dialog box appears as shown in [Figure 1](#).

Figure 1: Analyze and Fill Faults Dialog Boxes

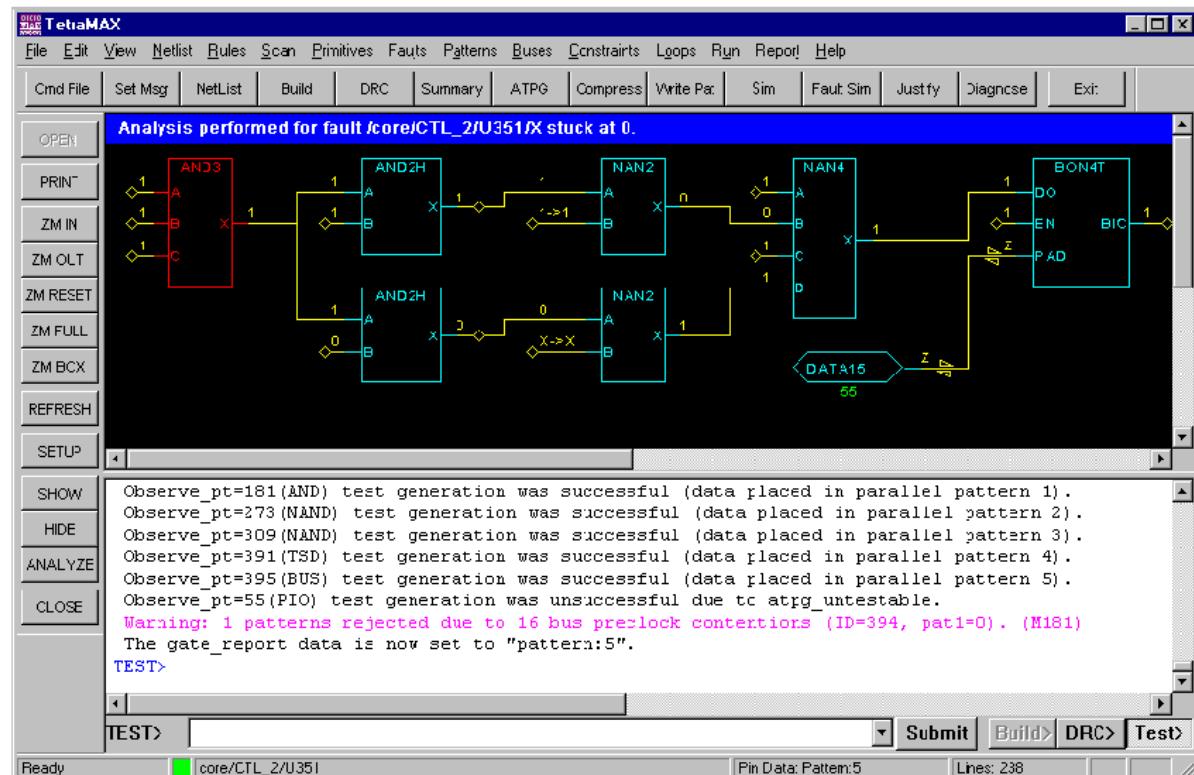


2. Click the Faults tab if it is not already active.
3. Select the Pin Pathname option, if it is not already selected.
4. Click the Fill button.
The Fill Faults dialog box opens.
5. Using the Class field, select the class of faults that you would like to see listed, such as "NO: not-observed." You can also specify the range of faults within that class that are to be listed.
6. Click OK to fill in the list box in the Analyze window, as shown in [Figure 2](#).
7. From the list, select the specific fault you would like displayed, such as "0 /core/CTL_2/U351/X".
The fields at the top of the dialog box are filled in automatically from your selection.
8. Click OK.

The Analyze dialog box closes and the GSV displays the logic associated with the selected fault location.

[Figure 2](#) shows the schematic displayed for a selected fault. The title at the top of the GSV window indicates the fault location displayed and appears on any printouts of the GSV.

Figure 2: GSV Window With a Fault Displayed



The command-line equivalent to the Analyze dialog box is the `analyze_faults` command. This command and the resulting report appear in the transcript window, as shown in [Example 1](#).

Example 1: Transcript of Not-Observed Analysis

```
TEST-T> analyze_faults /core/CTL_2/U351/X -stuck 0 -display
-----
Fault analysis performed for /core/CTL_2/U351/X stuck at 0 (output
of AND gate 178).
Current fault classification = NO (not-observed).
-----
Connection data: to=CLKPO,MASTER from=CLOCK
Fault site control to 1 was successful (data placed in parallel
pattern 0).
Observe_pt=any test generation was unsuccessful due to abort.
Observe_pt=181(AND) test generation was successful (data placed in
parallel pattern 1).
Observe_pt=273(NAND) test generation was successful (data placed
in parallel pattern 2).
```

```
Observe_pt=309(NAND) test generation was successful (data placed  
in parallel pattern 3).  
Observe_pt=391(TSD) test generation was successful (data placed in  
parallel pattern 4).  
Observe_pt=395(BUS) test generation was successful (data placed in  
parallel pattern 5).  
Observe_pt=55(PIO) test generation was unsuccessful due to atpg_  
untestable.  
Warning: 1 patterns rejected due to 16 bus preclock contentions  
(ID=394, pat1=0). (M181)  
The gate_report data is now set to "pattern:5".
```

The details of this type of report are described in the [Example: Analyzing a NO Fault](#) section.

See Also

[Performing Design Rule Checking](#)
[Fault Lists and Faults](#)

Navigating, Selecting, Hiding, and Finding Data

Within the GSV, you can navigate to different locations and views, select objects, hide objects, and find specific data for various objects. The following sections describe each of these actions:

- [Navigating Within the GSV](#)
- [Selecting Objects in the GSV Schematic](#)
- [Hiding Objects in the GSV Schematic](#)
- [Using the Block ID Window](#)

Navigating Within the GSV

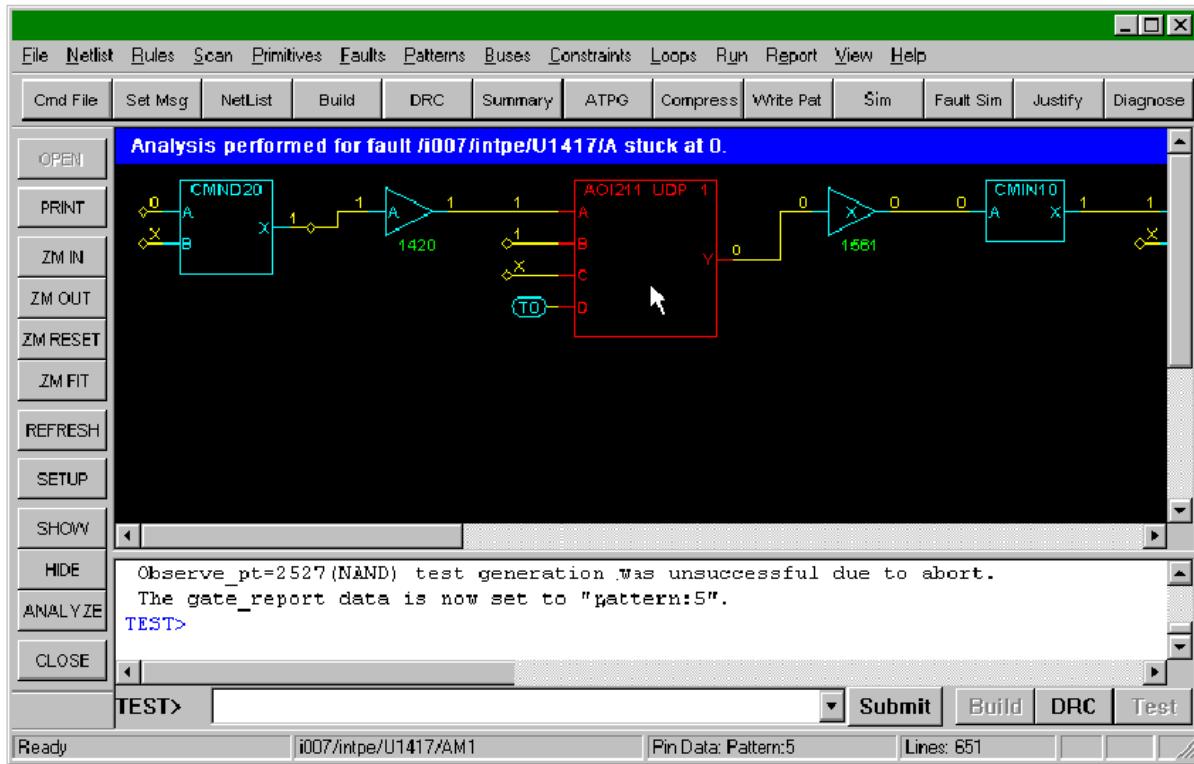
To navigate within the GSV window, use the horizontal or vertical slider; the arrow keys on the keyboard; and the ZM IN, ZM OUT, ZM RESET, ZM FULL, and ZM BOX buttons.

To zoom in to a specific area, click the ZM BOX button and then drag a box around the area to be magnified.

Selecting Objects in the GSV Schematic

To select an object, click it. The selected object color changes to red. The net or instance name of the selected object appears in the lower status bar, as shown in [Figure 1](#).

To deselect the object, click it again. To select more than one object, hold down the Shift key and click each object.

Figure 1: Selected Object Name

Hiding Objects in the GSV Schematic

The following steps describe how to hide an object in the GSV:

1. Select the object by clicking it.
2. Click the HIDE button.
The HIDE menu appears.
3. Choose Selected.
The selected object is hidden. Alternatively, you can choose Named to hide a named object, or All to hide all objects. You can also press the Delete key to hide selected objects.

Using the Block ID Window

You can find out the instance name, parent module, and connection data for any displayed object using the Block ID window. The following steps show you how to open the Block ID window:

1. Click the object of interest; the object color changes to red.
2. With the right mouse button, click the object again.
A menu appears.
3. With the left mouse button, click the Display Gate Info option of the menu.
The Block ID window appears with information about the selected object.

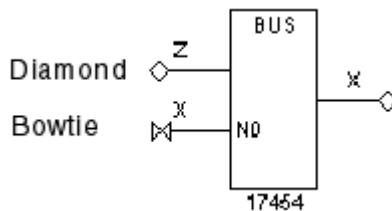
4. To display information for other objects, with the Block ID window still open, click each object with the right mouse button while holding down the Control key.

Expanding the Display From Net Connections

In the schematic display, net connections to undisplayed nets appear with one of two termination symbols, as shown in [Figure 1](#):

- The diamond symbol represents a unidirectional net connection
- The bow tie symbol represents a bidirectional net connection

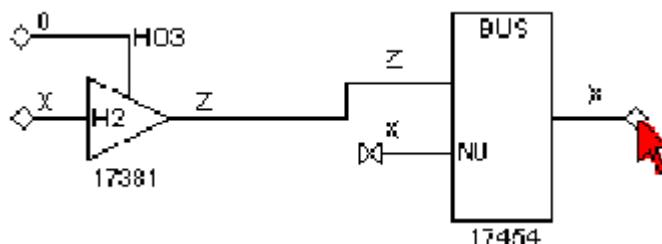
Figure 1: Net Expansion Symbols: Diamond and Bow Tie



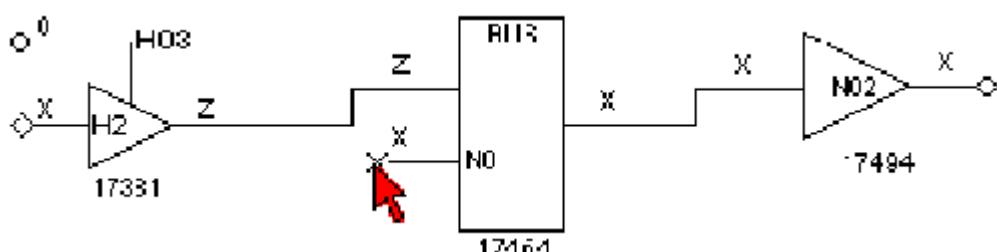
To expand the display from a specific connection, click the diamond or bow tie that represents the connection of interest. The schematic expands to include the next gate or component forward or backward from the selected connection. Each click adds one component to the display. If a net has multiple additional components, you can click repeatedly and display more components until the diamond or bow tie no longer appears.

The following steps show an example of the results obtained by clicking the diamond and bow tie connection points:

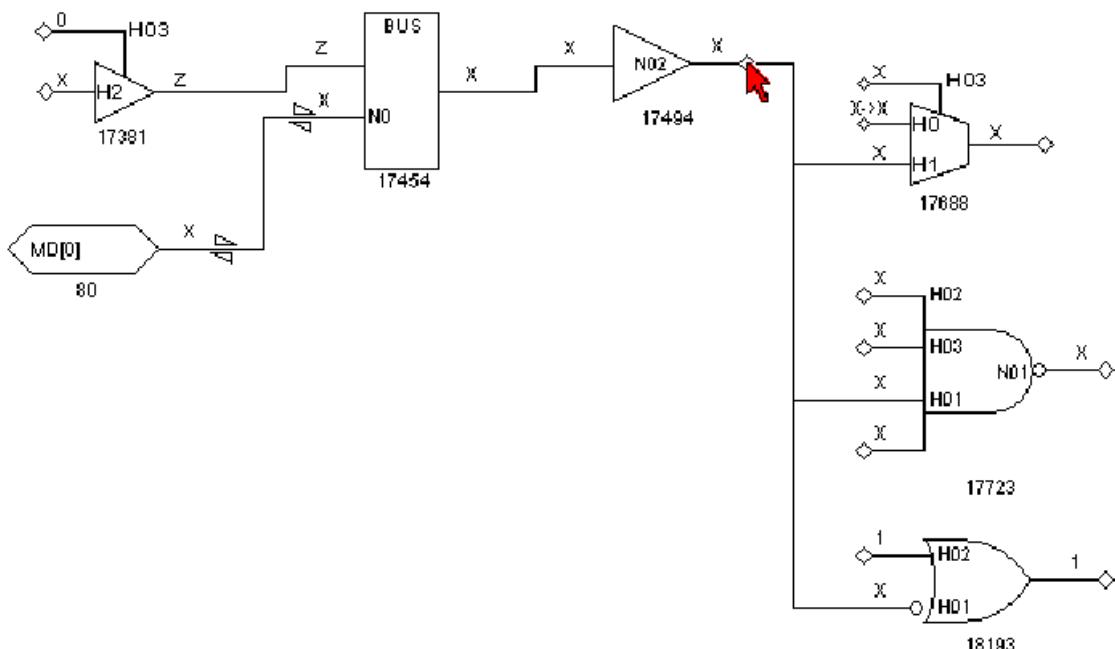
1. Click the diamond.



2. Click the boxtie on gate 17454.



3. Click three times on diamond on gate 17454.



The following steps describe how to traverse a specific route from output pin to input pin without displaying all the fanout connections:

1. Right-click the net diamond.
2. From the pop-up menu, select Show Unconnected Fanout.
The Unconnected Fanout dialog box appears, which lists all of the paths from the net that are not currently shown in the schematic.
3. Select from the list the path you want to traverse.
4. Click OK. The GSV adds the selected path to the GSV display.

See Also

[add_net_connections](#)

Hiding Buffers and Inverters in the GSV Schematic

When you display a design at the primitive level, you can save display space by removing the buffer and inverter gates and instead display them as double slashes and bubbles.

The following steps describe how to hide buffer and inverter gates:

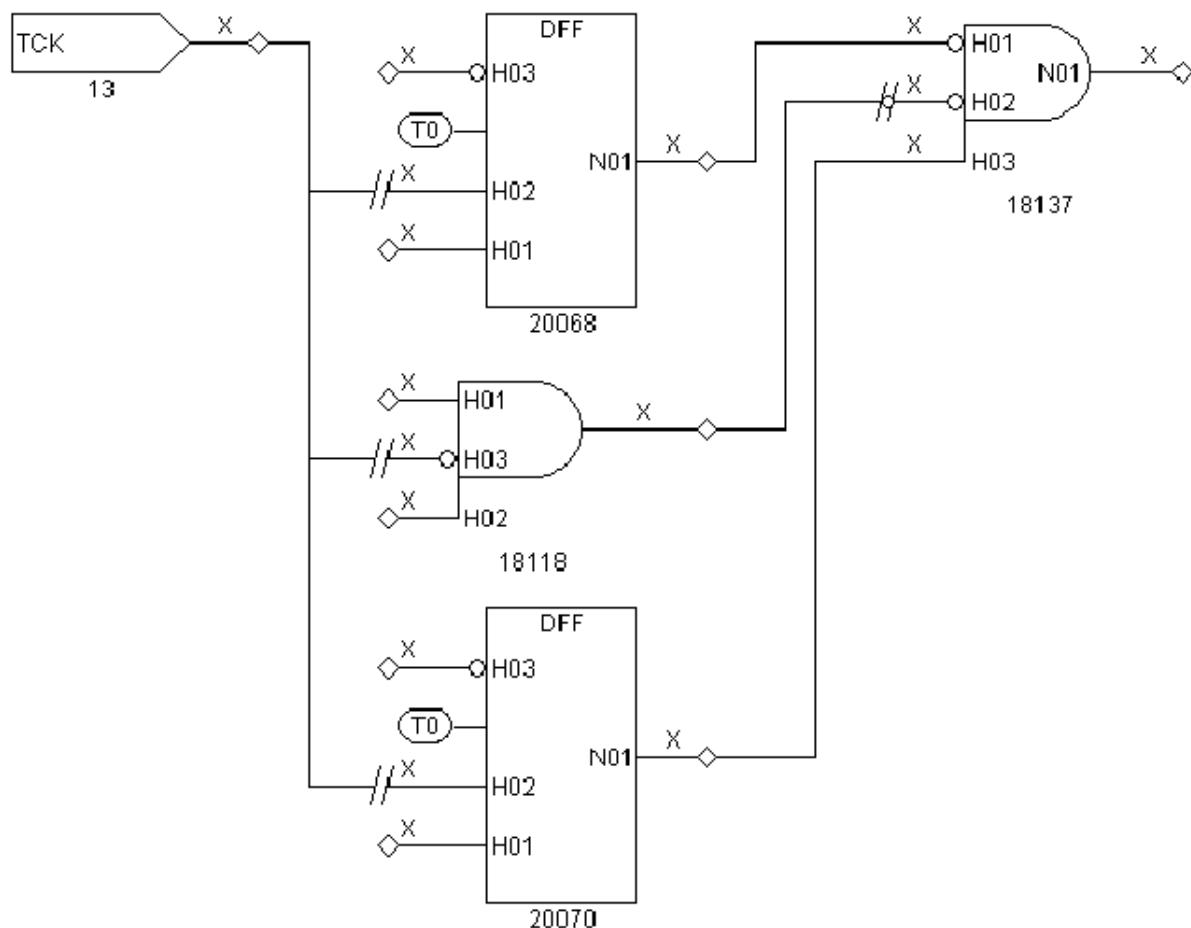
1. Click the SETUP button on the GSV toolbar.
The GSV Setup dialog box appears. The Hierarchy selection lets you specify whether to display primitives or design components. (For a discussion of primitives, see "[ATPG Model Primitives](#).")
2. Select the BUF/INVs check box in the Hide section.

3. Click OK.

TetraMAX ATPG redraws the schematic without the usual buffer and inverter symbols.

As you redraw items in the schematic, the buffers and inverters are displayed as double slashes and bubbles, as shown in [Figure 1](#). Double slashes across the net represent a hidden gate with no logic inversion; double slashes around a bubble represent a hidden gate with logic inversion. When you look at schematics that contain hidden gates, be aware of any hidden gates that invert logic.

Figure 1: Schematic With Buffers and Inverters Hidden



ATPG Model Primitives

This section describes the set of TetraMAX primitives that are used in GSV displays when Primitive is selected in the GSV Setup dialog box. If Design is selected, see [“Displaying Symbols in Primitive or Design View.”](#)

The primitives include the following:

- [Tied Pins](#)
- [Primary Inputs and Outputs](#)

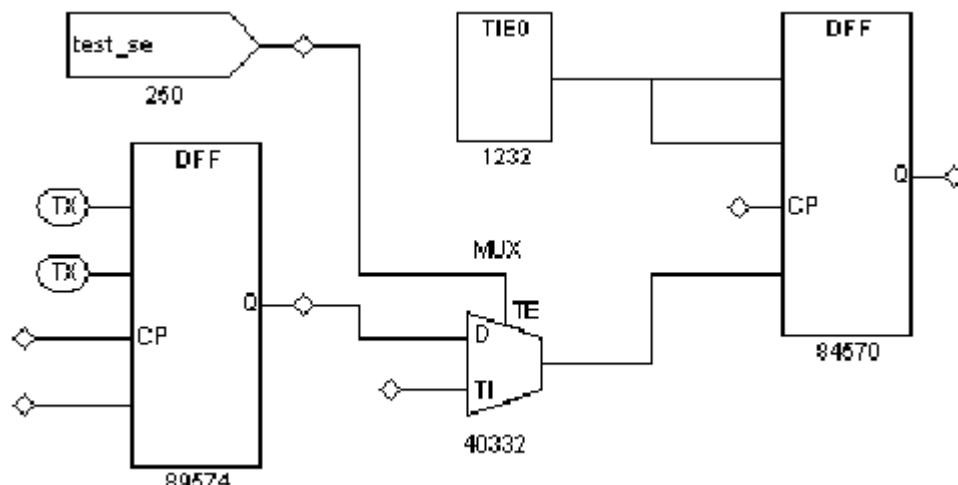
- [Basic Gate Primitives](#)
- [Additional Visual Characteristics](#)
- [RAM and ROM Primitives](#)

Tied Pins

A pin can be tied to 0, 1, X, or Z and can be represented in one of the following two ways:

- By an oval containing the label T0, T1, TX, or TZ connected to the pin. For example, in [Figure 1](#), the DFF on the left has two of its input pins connected to ovals labeled TX, indicating that the two pins are tied to X (unknown).
- By a separate connection to a TIE primitive. For example, in [Figure 1](#), the DFF on the right has two of its input pins connected to the TIE0 primitive, indicating that the two pins are tied to 0.

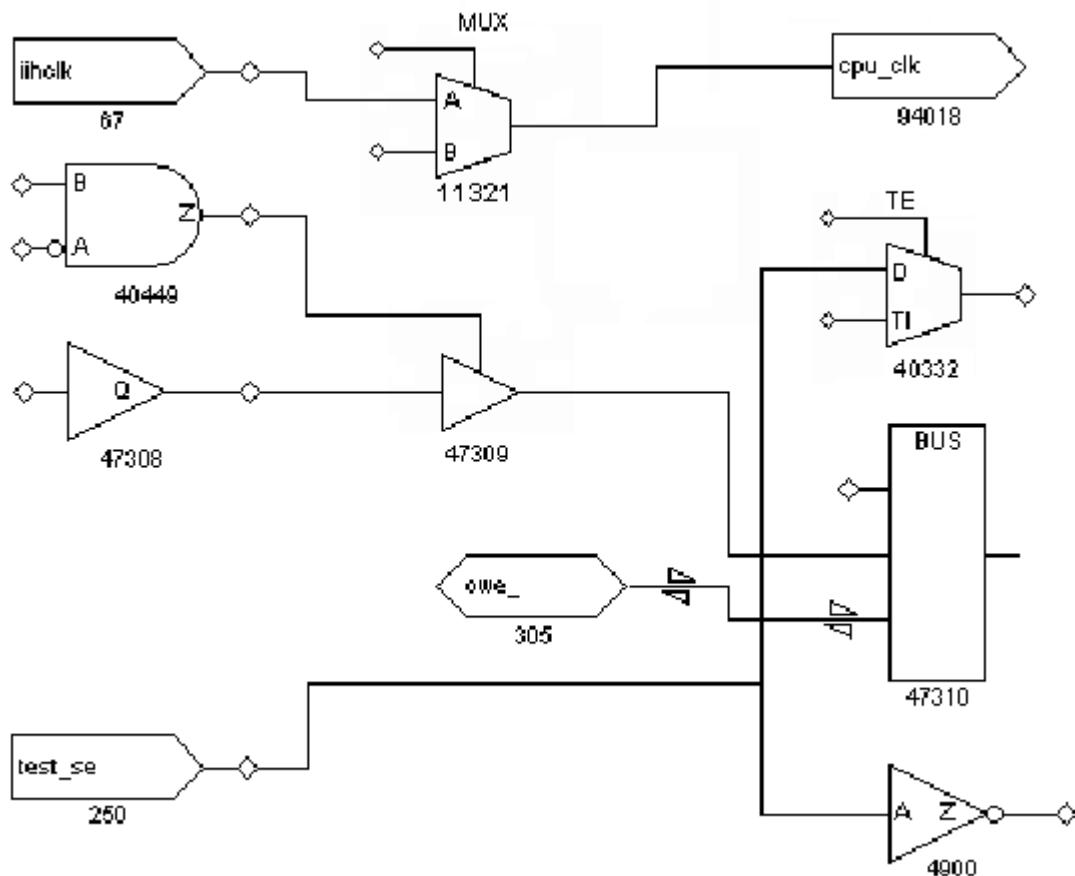
Figure 1: GSV Representation of Tied Pins



Primary Inputs and Outputs

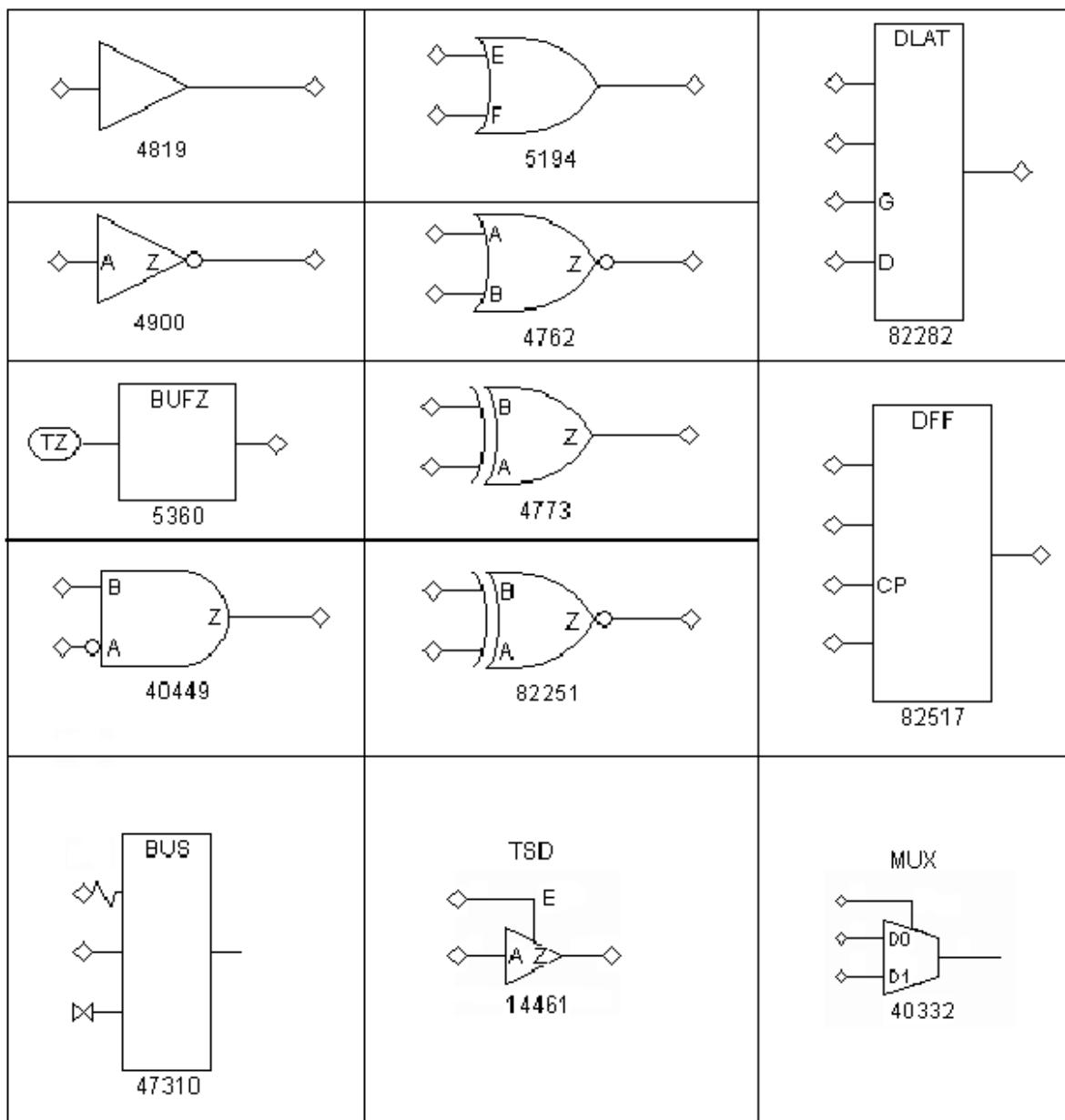
Primary (top-level) inputs and outputs are identified in the following ways:

- Primary inputs are identified with the symbol shown for gates 67 and 250 in [Figure 2](#). Primary input ports always appear at the left of the schematic, and the symbol contains the port label (for example, iihclk and test_se).
- Primary outputs are identified with the symbol shown for gate 94018 in [Figure 2](#). Primary outputs always appear at the right of the schematic, and the symbol contains the port label (for example, cpu_clk).
- Primary bidirectional ports are identified with the symbol shown for gate 305 in [Figure 2](#). Primary bidirectional ports can appear anywhere in the schematic, and the symbol contains the port label (for example, owe_). The two bidirectional triangular wedges on bidirectional nets distinguish them from unidirectional nets.

Figure 2: Primary I/O and Bidirectional Port Symbols

Basic Gate Primitives

[Figure 3](#) shows representative symbols for many of the more commonly used TetraMAX primitives. The combinational gates AND, OR, NOR, XOR, and XNOR are shown with two inputs, but can have any number of inputs. For a complete list, refer to the Online Help reference topic “ATPG Simulation Primitives.”

Figure 3: Some Basic Gate Primitives

Additional Visual Characteristics

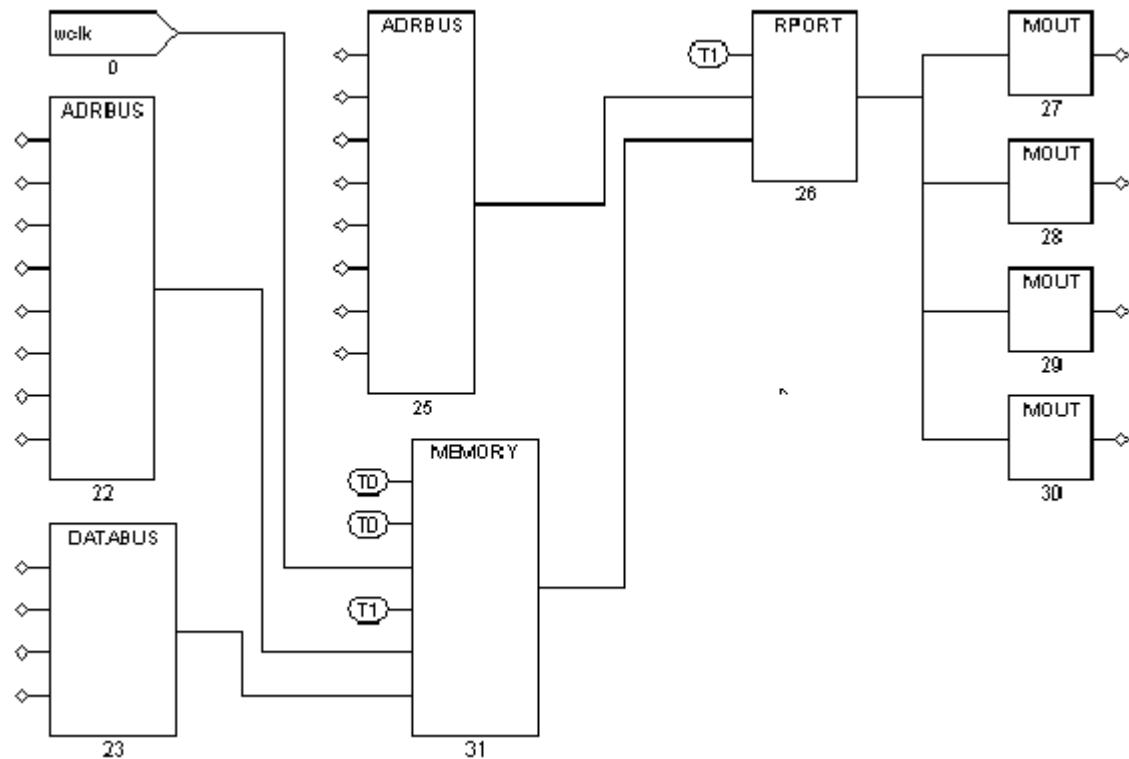
Some additional visual characteristics of ATPG primitives are described as follows:

- Merged inverters: Inverters can be merged into the drawn symbol to make the schematic more compact. For example, in [Figure 3](#), the AND gate (ID 40449) shows an inversion bubble on the A input, indicating that an inverter that preceded this pin has been merged into the AND gate.

- Merged resistors: Resistors can be merged into the drawn symbol to show a gate that has a weak output drive strength. For example, in [Figure 3](#), the BUS gate (ID 47310) shows a resistor on one of its input pins, indicating a resistive input.
- Pin Name labels: Some pins are labeled with pin names, and some are not. A pin name on a primitive indicates that the pin maps directly to the identical pin on the defining module in the library cell. For example, in [Figure 3](#) the TSD or three-state device (ID 14461) shows pins labeled A, E, and Z, meaning that those pins are all directly mapped to the pins of the defining module. However, the DFF (ID 82517) shows only pin CP labeled, meaning that CP is mapped directly to the defining module's pin called CP, but the unnamed pins are connected to other TetraMAX primitives. A single module can be represented by several TetraMAX primitives; in that case, the labels do not all appear on the same TetraMAX primitive.
- Pin order: The order of pins on the TSD, DLAT, DFF, and MUX primitives is significant. Refer to [Figure 3](#); pins are displayed in the following order, starting at the top:
 - For the DLAT (level-sensitive latch) primitive: asynchronous set, asynchronous reset, active-high enable, and data inputs.
 - For the DFF (edge-triggered flip-flop) primitive: asynchronous set, asynchronous reset, positive triggered clock, and data inputs.
 - When the display mode is set to Primitive, you can control the appearance of DFF/DLAT symbols in the Environment dialog box (Edit > Environment). In the dialog box, click the Viewer tab and set the DFF/DLAT option to Mode 1, Mode 2, or Mode 3. For details, see “[Displaying Symbols in Primitive or Design View](#)” on and “DFF Primitive.”

RAM and ROM Primitives

For readability, instead of a single rectangle with numerous pins, a RAM or ROM block is represented as a collection of the special primitives shown in [Figure 4](#). The example represents a simple 256x4 RAM with a single write port and a single read port, each with its own address and control pins. Other RAMs can have multiple read and write ports. Although the RAM and ROM primitives are shown with specific bit-widths (for example, ADRBUS has eight bits and DATABUS has four bits), all bit-widths are supported, as required by the design.

Figure 4: RAM and ROM Primitives

The RAM and ROM primitives are described as follows:

- ADRBUS: Merges the eight individual address lines at the left into the single 8-bit address bus at the right. In this example, the write port uses a separate address from the read port.
- DATABUS: Merges the four individual data write lines at the left into the single 4-bit data bus at the right.
- MEMORY: The core of the RAM or ROM; holds the stored contents. Starting from the top left, pins are as follows: an active-high set; an active-high reset (both tied to 0 in the example); a single data write port consisting of a write clock (wclk); a write enable (tied to 1); the write port address bus (8 bits); and the write port data bus (4 bits). A memory block can have multiple read and write ports; a memory without a write port represents a ROM. The module where the ROM is defined must give a path name to a memory initialization file.
- REPORT: Provides a single read port. It has a read clock or read enable pin (tied to 1 in the example), an 8-bit address bus input, and a 4-bit data bus input from the memory core. Its output is a 4-bit data bus.
- MOUT: Splits a single bit from the 4-bit REPORT data bus.

See Also

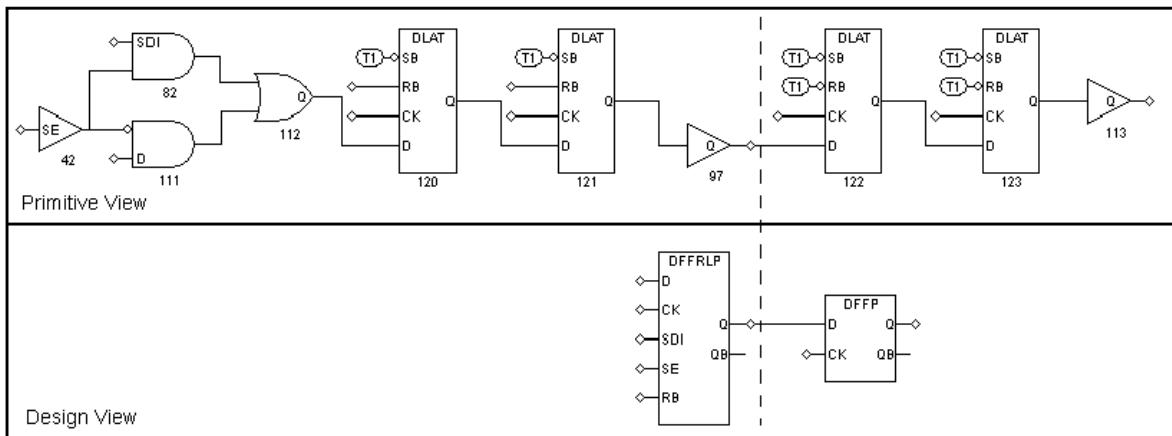
[Creating Custom ATPG Models](#)

Displaying Symbols in Primitive or Design View

You can choose to display a schematic using the TetraMAX primitives (Primitive view) or using the higher-level symbols that represent the library cells (Design view).

To specify the type of view, in the GSV Setup dialog box, select either Primitive or Design in the Hierarchy box and click OK. Figure 1 shows two different views of a design.

Figure 1: Comparison of Primitive and Design Views



Note that the schematic labeled *Primitive View* uses TetraMAX primitives; the schematic labeled *Design View* uses cells in the technology library.

Displaying Instance Path Names

You can display the instance path name above each instance in the schematic, as described in the following steps:

1. Select Edit > Environment in the menu bar.
The Environment dialog box appears.
2. In the Environment dialog box, click the Viewer tab.
3. Select the Display Instance Names check box.
4. Click OK.

See Also

[Masking Scan Cell Input and Outputs](#)

Displaying Pin Data

You can display various types of pin data on the schematic to help you analyze DRC problems or view logic states for specific patterns, constrained and blocked values, or simulation results. For

example, you might want to see the ripple effects of pins tied to 0 or 1, identify all nets that are part of a clock distribution, or see logic values on nets resulting from a STIL shift procedure.

The data values displayed are generated either by DRC or by ATPG. Data values generated by DRC correspond to the simulation values used by DRC in simulating the STIL protocol to check conformance to the test rules. Data values generated by ATPG are the actual logic values resulting from a specific ATPG pattern.

When you analyze a rule violation or a fault, TetraMAX ATPG automatically selects and displays the appropriate type of pin data. You can also manually select the type of pin data to be displayed by using the SETUP button in the GSV toolbar, or you can use the `set_pindata` command at the command line.

The following sections describe how to display pin data:

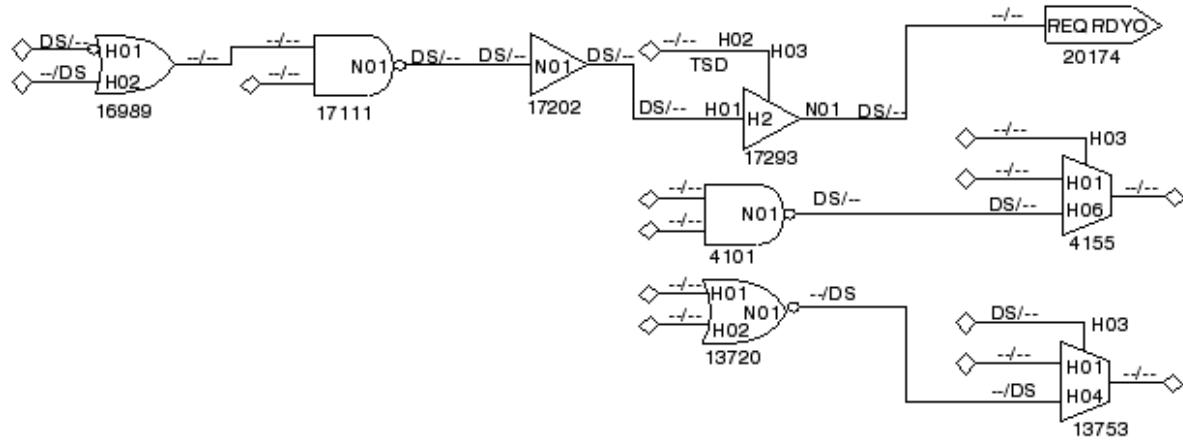
- [Using the Setup Dialog Box](#)
- [Pin Data Types](#)
- [Displaying Clock Cone Data](#)
- [Displaying Clock Off Data](#)
- [Displaying Constrain Values](#)
- [Displaying Load Data](#)
- [Displaying Shift Data](#)
- [Displaying Test Setup Data](#)
- [Displaying Pattern Data](#)
- [Displaying Tie Data](#)

Using the Setup Dialog Box to Display Pin Data

The following steps describe how to display pin data on the schematic using the Setup dialog box:

1. With a schematic displayed in the GSV (for example, as shown in [Figure 1](#)), click the SETUP button on the GSV toolbar.
The Setup dialog box opens.
2. Using the Pin Data Type pull-down menu, select the type of pin data you want to display.
3. Click OK.

TetraMAX ATPG redraws the schematic using the new pin data type.

Figure 1: GSV Display With Pin Data Type Set to Tie Data

To set the pin data display mode from the command line, use the `set_pindata` command. For example:

```
TEST-T> set_pindata -clock_cone CLK
```

For complete syntax and option descriptions, see Online Help for the `set_pindata` command.

Pin Data Types

[Table 1](#) lists each pin data type, a description of the data displayed in the GSV, and its typical use. You can find additional related information in the description of the `set_pindata` command in Online Help.

Table 1: Pin Data Types

Pin Data Type	Data Displayed	Typical Use
Clock Cone	Cone of influence and effect cones for the selected clock	Debugging clock (C) violations
Clock On	Simulated values when all clocks are held in on state	Debugging clock (C) violations
Clock Off	Simulated values when all clocks are held in off state	Debugging clock (C) violations
Constrain Value	Simulated values that result from tied circuitry and ATPG constraints	Analysis of the effects of constrained signals
Debug Sim Data	Imported external simulator values	Debugging golden simulation vector mismatches
Error Data	Simulated values associated with the current DRC error	Analysis of DRC violations with severity of error

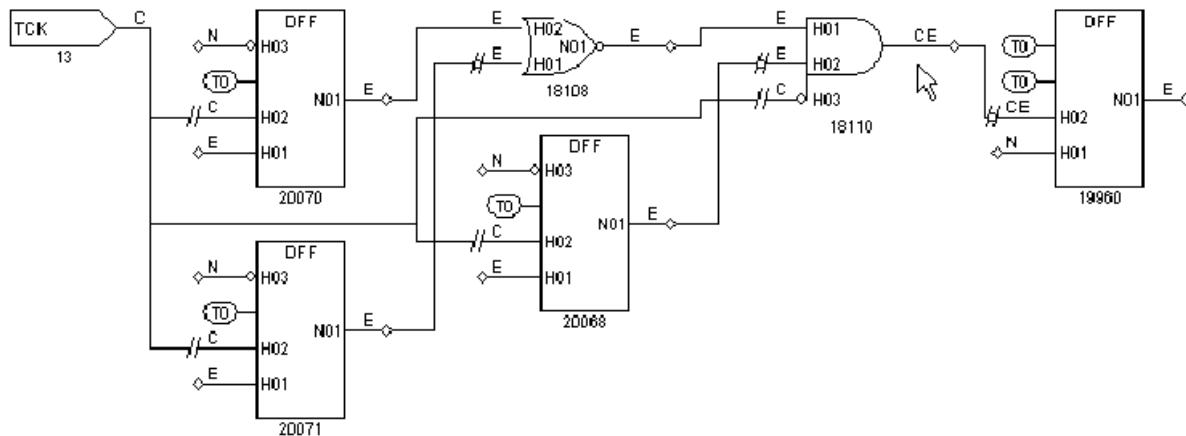
Pin Data Type	Data Displayed	Typical Use
Fault Data	Current fault codes	Analysis of fault coverage (for advanced users of fault simulation)
Fault Sim Results	Good machine and faulty machine values for a selected fault	Displaying results of Basic-Scan fault simulation (for advanced users of fault simulation)
Full-Seq SCOAP Data	SCOAP controllability and observability measures using Full-Sequential ATPG	Identification of logic that is difficult to test with Full-Sequential ATPG
Full-Seq TG Data	Full-Sequential test generator logic values, showing the sequence of logic values used to achieve justification	Analysis of logic controllability using Full-Sequential ATPG
Good Sim Results	The good machine value for the selected ATPG pattern	Displaying ATPG pattern values
Load	Simulated values for the <code>load_unload</code> procedure	Debugging problems in a STIL <code>load_unload</code> macro
Master Observe	Simulated values for the <code>master_observe</code> procedure	Debugging problems in a STIL <code>master_observe</code> procedure
Pattern	Simulated values for a selected pattern	Fault analysis; displays ATPG generated values
SCOAP Data	SCOAP controllability and observability measures	Identification of logic that is difficult to test
Sequential Sim Data	Currently stored sequential simulation data	Displaying results of sequential fault simulation (for advanced users of fault simulation)
Shadow Observe	Simulated values for the <code>shadow_observe</code> procedure	Debugging problems in a STIL <code>shadow_observe</code> procedure
Shift	Simulated values for the <code>shift</code> procedure	Debugging DRC T (scan chain tracing) violations
Stability Patterns	Simulated values for the <code>load_unload, Shift, and capture</code> procedures	Analysis of classification of nonscan cells

Pin Data Type	Data Displayed	Typical Use
Test Setup	Simulated values for the <code>test_setup</code> macro	Debugging problems in a STIL <code>test_setup</code> macro
Tie Data	Simulated values that result from tied circuitry	Analysis of the effects of tied signals

Displaying Clock Cone Data

To display clock cone data, select Clock Cone as the Pin Data type in the GSV Setup dialog box and click OK. The schematic is redrawn as shown [Figure 2](#). This example shows the clock cones and effect cones of the TCK clock port.

Figure 2: GSV Display: Pin Data Type Set to Clock Cone

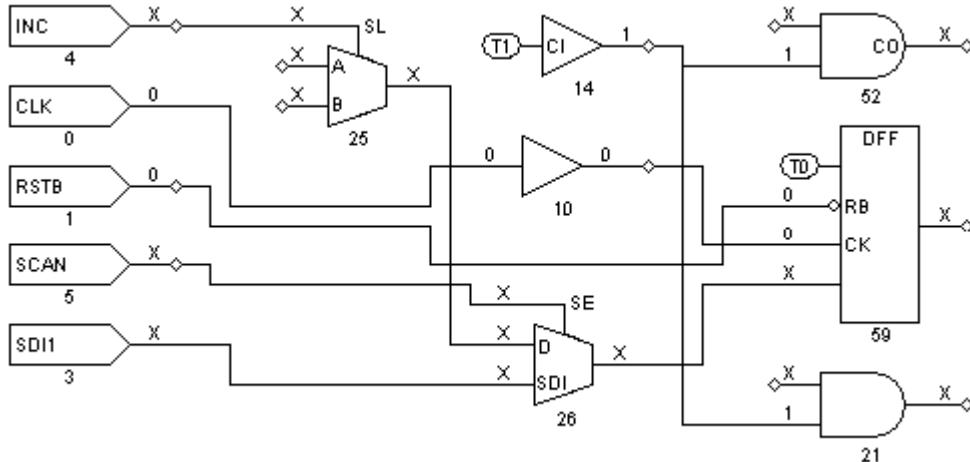


Note the following:

- Nets labeled “C” are in the clock’s clock cone. A clock cone is an area of influence that begins at a single point, spreads outward as it passes through combinational gates, and terminates at a clock input to a sequential gate.
- Nets labeled “E” are in the clock’s effect cone. An effect cone begins at the output of the sequential gate affected by the clock, spreads outward as it passes through combinational gates, and also terminates at a sequential gate.
- Nets labeled “CE” are in both the clock and effect cones because of a feedback path through a common gate that allows the effect cone to merge with the clock cone.
- Nets labeled “N” are in neither the clock nor effect cones.

Displaying Clock Off Data

To display clock off data, select Clock Off as the Pin Data type in the GSV Setup dialog box and click OK. The schematic is redrawn as shown in [Figure 3](#).

Figure 3: GSV Display: Pin Data Type Set to Clock Off

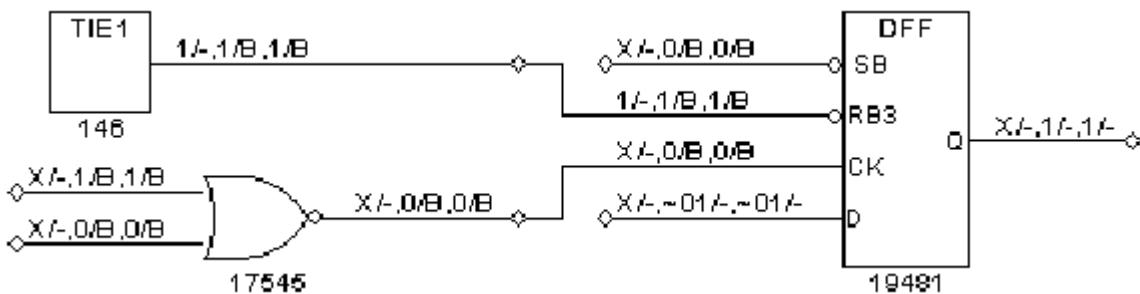
In [Figure 3](#), nets that are part of a clock distribution are shown with the logic values they have when the clocks are at their defined off states. Nets not affected by clocks are shown with Xs.

In this design, the clock ports are CLK and RSTB and their nets have values of 0. The 0 value from the CLK net is propagated to the input of gate 10, the output of gate 10, and the CK input of gate 59 (the DFF). The 0 value of RSTB is propagated to the RB input of the same DFF, gate 59. Notice that the RB pin has an inversion bubble; this is an active-low reset. When the clocks are off, there is a logic 0 value on this pin, which results in a C1 violation (unstable scan cells when clocks off).

The solution to the problem detected here is to delete the clock RSTB and redefine it with the opposite polarity. Then, execute `run_drc` again and verify that this particular DRC violation is no longer reported.

Displaying Constrain Values

To display constrain values, select Constrain Value as the Pin Data type in the GSV Setup dialog box and click OK. [Figure 4](#) shows a schematic displaying the constrain values.

Figure 4: GSV Display: Pin Data Type Set to Constrain Value

Constrain values are shown as three pairs of characters in the format T/B1, C/B2, S/B3:

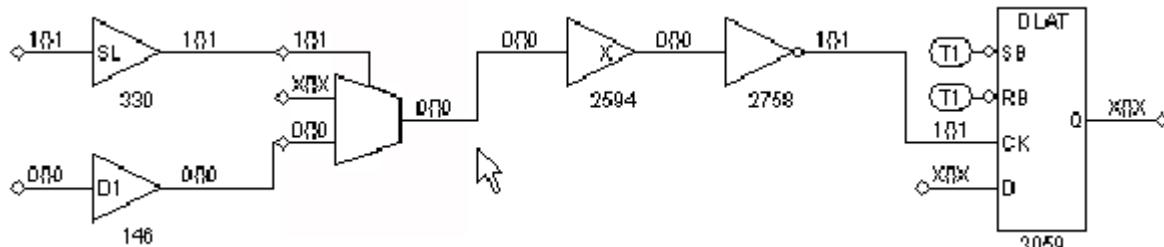
- T is the pin's value that is a result of tied circuitry, if any exists. An "X" indicates that there is no value due to tied logic.

- B1 indicates whether faults are blocked on the pin because of the tied value “T.” A value of “B” indicates that the fault is blocked; a dash (-) indicates that the fault is not blocked.
- C is the constant value on the pin that results from constrained circuitry during Basic-Scan ATPG, if any. A tilde (~) preceding the character indicates values that cannot be achieved. For example, ~1 means that a value of 1 cannot be achieved, so the value is either 0 or X. An “X” indicates that there is no constant value due to constraints during Basic-Scan ATPG.
- B2 indicates whether faults are blocked on the pin because of the constrained value “C.” A value of “B” indicates that the fault is blocked; a dash (-) indicates that the fault is not blocked.
- S is similar to C, except that it is the constant value on the pin that results from constrained circuitry during sequential ATPG.
- B3 is similar to B2, except that it indicates whether faults are blocked on the pin because of the constrained value “S.”

Displaying Load Data

To display logic values during the load_unload procedure, select Load as the Pin Data type in the GSV Setup dialog box and click OK. [Figure 5](#) shows a schematic displaying the load data.

Figure 5: GSV Display: Pin Data Type Set to Load



The logic values are shown in the format “AAA{ }SBB”:

- AAA is one or more logic states associated with test cycles defined at the beginning of the load_unload procedure.

For each test cycle defined before the Shift procedure within the load_unload procedure, AAA has only one logic state if there were no events during that cycle.

For example, if three test cycles within the load_unload procedure precede the Shift procedure and an input port is forced to a 1 in the first cycle, the input port might show logic values 111{ }1. If, however, the port is pulsed and an active-low pulse is applied in the third test cycle, the port would show logic values 11101{ }1. In this case, the third test cycle is expanded into three time events and produces the third, fourth, and fifth characters, --101{ }-.

Curly braces { } represent application of the Shift procedure as many times as needed to shift the longest scan chain. For single-bit shift chains, the actual data simulated for the shift pattern is used rather than the { } placeholder.

- S represents the final logic value at the end of the Shift procedure.

- BB represents the logic values from cycles in the load_unload procedure that occur after the Shift procedure. TetraMAX ATPG determines the logic values for multibit shift chains as follows:
 - It places all constrained primary inputs at their constrained states.
 - It simulates all test cycles within the load_unload procedure before the Shift procedure, in the order that they occur.
 - It sets to X all other input ports and scan inputs that are not constrained or explicitly set.
 - It pulses the shift clock repeatedly until the circuit comes to a stable state.
 - It simulates all test cycles that are defined within the load_unload procedure that occur after the Shift procedure.

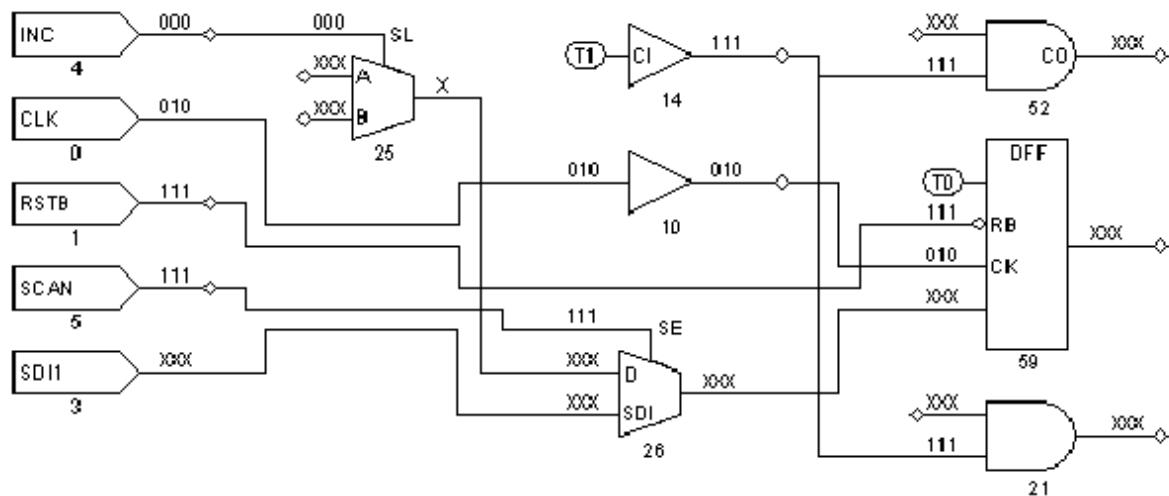
If no test cycles in the load_unload procedure occur after the Shift procedure, BB is an empty string. Otherwise, the string displayed for BB contains characters: one character for each test cycle that can be represented with a single time event, and multiple characters for any test cycles that require multiple time events. This is similar to how a single cycle in A is expanded into three characters when the port is pulsed; see the preceding discussion of AAA.

Displaying Shift Data

To display logic values during the Shift procedure, select Shift as the Pin Data type in the GSV Setup dialog box and click OK. The schematic is redrawn as shown in [Figure 6](#).

In [Figure 6](#), the pins show logic values that result from simulating the Shift procedure. The CLK port shows a simulation sequence of 010, and during the same three time periods, the RSTB pin is 111 and the SCAN pin is 111.

Figure 6: GSV Display: Pin Data Type Set to Shift



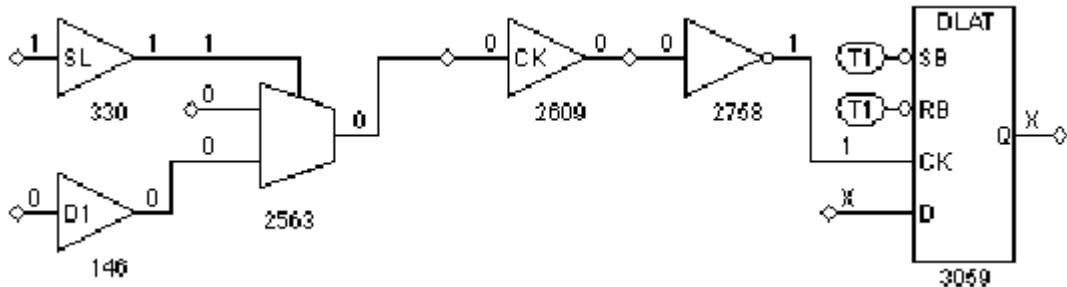
These are all appropriate values for the STIL Shift procedure shown in the following example:

```
Shift
V { _so = #; _si = #; INC = 0; CLK = P; RSTB = 1; SCAN = 1; }
```

Displaying Test Setup Data

To display logic values simulated during the test_setup macro, select Test Setup as the Pin Data type in the GSV Setup dialog box and click OK. An example schematic with Test Setup data is shown in [Figure 7](#).

Figure 7: GSV Display: Pin Data Type Set to Test Setup



By default, only a single logic value is shown, which corresponds to the final logic value at the exit of the test_setup macro. To show all logic values of the test_setup macro, you must change a DRC setting using the `set_drc` command, then rerun the DRC analysis as follows:

```
TEST-T> drc
DRC-T> set_drc -store_setup
DRC-T> run_drc
```

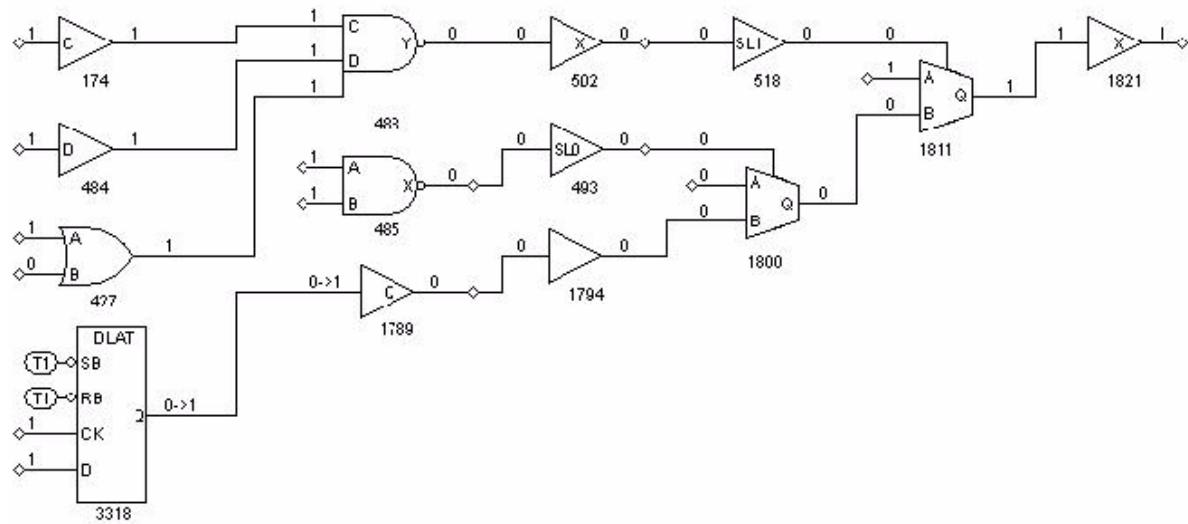
Displaying Pattern Data

You can use the GSV to display logic values for a specific ATPG pattern within the last 32 patterns processed. The GSV can also show the values for all 32 patterns simultaneously.

To display logic values for a specific pattern:

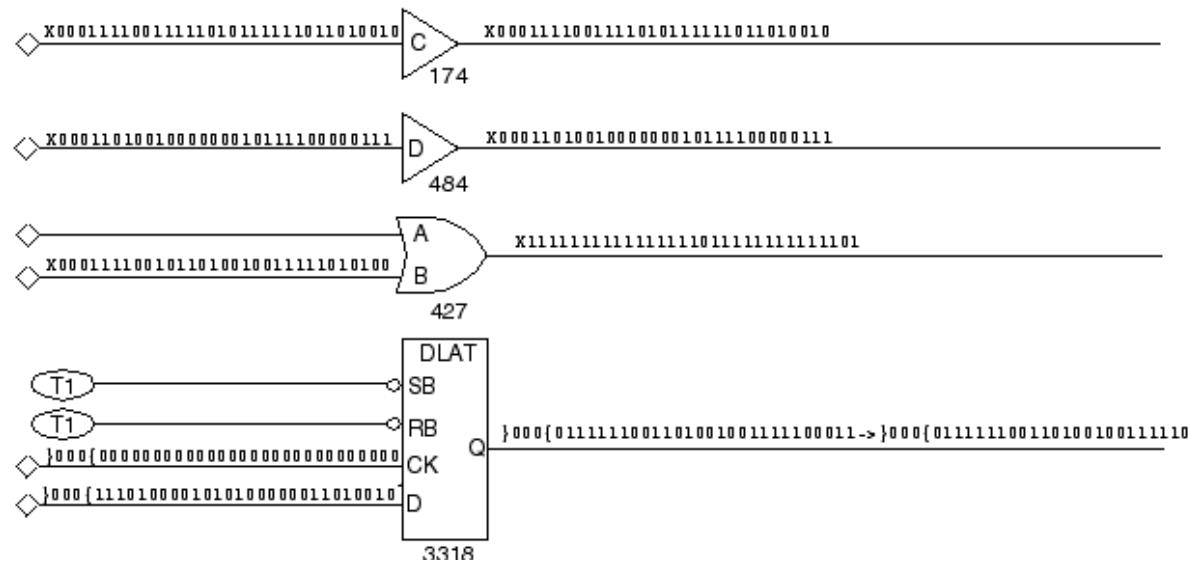
1. Display some design gates in the schematic window.
2. Click the SETUP button in the GSV toolbar.
3. In the Setup dialog box, set the Pin Data type to Pattern.
4. In the Pattern No. text box, choose the specific pattern number to be displayed.
5. Click OK.

The logic values that result from the selected ATPG pattern are displayed on the nets of the schematic, as shown in [Figure 8](#).

Figure 8: GSV Display: Pin Data Type Set to a Pattern Number

The logic values shown with an arrow, as in 0->1, show the pre-clock state on the left and the post-clock state on the right. A logic state shown as a single character represents the pre-clock state. For a clock pin, a single character represents the clock-on state.

To display logic values for all patterns, choose All Patterns in the GSV Setup dialog box and click OK. [Figure 9](#) shows all 32 patterns on the pins. You read the values from left to right. The leftmost character is the logic value resulting from pattern 0, and the rightmost character is the logic value resulting from pattern 31.

Figure 9: GSV Display: Pin Data Type Set to Pattern All

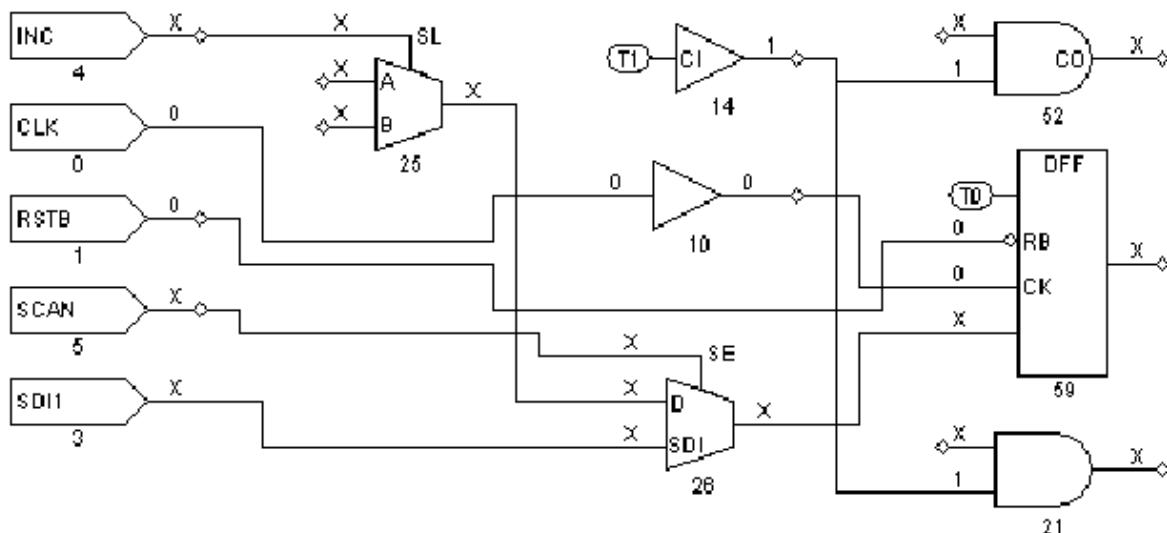
To examine a pattern that is not in the final 32 patterns processed, choose Good Sim Results in the GSV Setup dialog box and click OK.

For an additional method of viewing the logic values from a specific pattern, see "[Running Logic Simulation](#)." By simulating a fault on an output port, you can display the logic values for any pattern.

Displaying Tie Data

To display tie data, select Tie Data as the Pin Data type in the GSV Setup dialog box and click OK. The schematic is redrawn as shown in [Figure 10](#).

Figure 10: GSV Display: Pin Data Displayed



In [Figure 10](#), logic values are shown on nets affected by pins tied to 0 or 1. Thus, the output of gate 14 is shown with logic value 1, because its input is tied to 1. The tied value of 1 is propagated to the inputs of gates 52 and 21. Nets not affected by tied values are shown with Xs.

Analyzing a Feedback Path

You can use the GSV to review combinational feedback loops in the design. [Example 1](#) shows the use of the report feedback paths command to obtain a summary of all combinational feedback paths and details about a specified feedback path. The five gates involved in this feedback path example are identified by their instance path names (under “id#”) and gate IDs.

Example 1: Report Feedback Paths Transcript

```
TEST-T> report_feedback_paths -all
id# #gates #sources sensitization_status
--- -----
0 2 1 pass
1 10 1 pass
2 10 1 pass
3 10 1 pass
4 10 1 pass
5 10 1 pass
6 5 1 pass
7 10 1 pass
8 8 1 pass
TEST-T> report_feedback_paths 6 -verbose
```

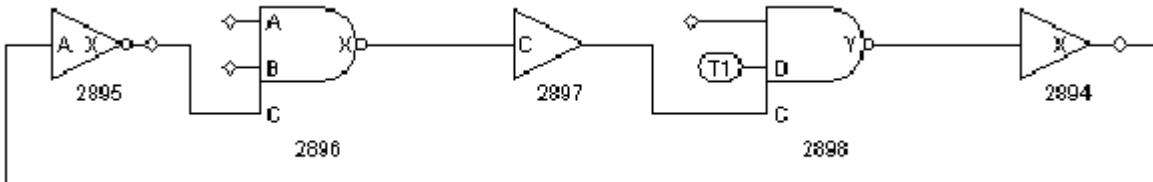
```

id# #gates #sources sensitization_status
-----
6 5 1 pass
BUF /amd2910/register/U70 (2894), cell=CMOA02
INV /amd2910/register/sub_23/U11 (2895), cell=CMIN20
NAND /amd2910/register/U86 (2896), cell=CMND30
BUF /amd2910/register/U70 (2897), cell=CMOA02
NAND /amd2910/register/U70/M1 (2898), cell=OAI211_UDP_1

```

To view a particular feedback path in the GSV, click the SHOW button, select Feedback Path, and specify the feedback path in the Show Feedback Path dialog box. [Figure 1](#) shows the resulting schematic display for feedback path number 6 in [Example 1](#).

Figure 1: GSV Display: A Feedback Path



Checking Controllability and Observability

You can use the Run Justification dialog box or the `run_justification` command, along with the GSV's ability to display pattern data, to determine if:

- a single internal point is controllable and observable
- a single internal point is controllable and observable within existing ATPG constraints
- multiple points can be set to required states simultaneously

You specify one or more internal pin states to achieve. TetraMAX ATPG attempts to find a pattern that achieves the specified logic states. If a pattern can be found, it is placed in the internal pattern buffer, and you can write it out or display it in the schematic by running the Pattern pin display format in the Setup dialog box.

By default, the `run_justification` command uses Basic-Scan ATPG; or if you have enabled Fast-Sequential ATPG with the `set_atpg -capture_cycles` command, it uses Fast-Sequential ATPG. If you want justification performed with Full-Sequential ATPG, use the `-full_sequential` option of the `run_justification` command, or enable the Full Sequential option of the Run Justification dialog box.

Using the Run Justification Dialog Box

To specify pin states using the Run Justification dialog box:

1. From the menu bar, choose Run > Run Justification. The Run Justification dialog box appears.

2. In the Gate ID/Pin path name text field, type the gate ID number of a gate whose state you want to specify or the pin path name of the pin you want to specify.
 3. In the Value field, use the drop-down menu to choose the value you want to specify for that gate or pin (0, 1, or Z).
 4. Click Add.
- The value and gate ID are added to the list in the dialog box.
5. Repeat steps 2, 3, and 4 for each gate or pin that you want to specify. When you are finished, click OK.
- The Run Justification dialog box closes. TetraMAX ATPG attempts the justification and reports the results.

Using the run_justification Command

[Example 2](#) shows the use of the `run_justification` command to request that gate ID 330 be set to 1 while gate ID 146 is simultaneously set to 0. The message indicates that the operation was successful and that the pattern is stored as pattern 0, available for pattern display.

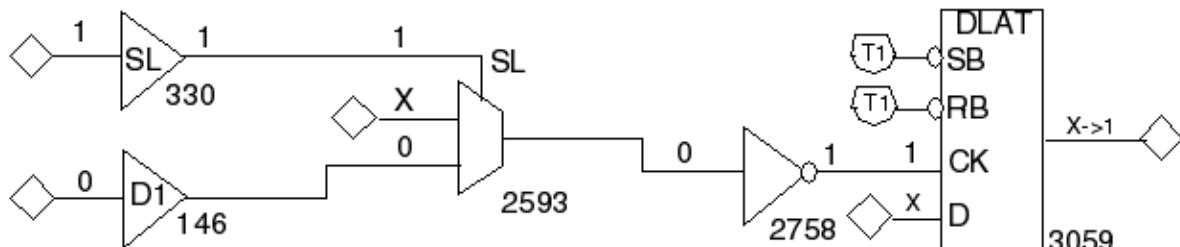
Example 2: Using the run_justification Command

```
TEST-T> run_justification -set 330 1 -set 146 0 -store
```

Successful justification: pattern values available in pattern 0.

[Figure 2](#) shows a schematic that displays the data for pattern number 0 in [Example 2](#). Gate 146 is at logic 0 state and gate 330 is at logic 1, as requested. Justification was successful, and TetraMAX ATPG was able to create a pattern to satisfy the list of set points.

Figure 2: GSV Display: Logic Values From Run Justification



See Also

[Analyzing the Cause of Low Test Coverage](#)

Analyzing DRC Violations in the GSV

To analyze DRC violations in the GSV:

1. Run DRC. For details, see "[Starting Test DRC](#)."
2. Click the ANALYZE button in the command toolbar of the GSV.

3. Click the Rules tab and select a violation from the displayed list or enter a specific violation occurrence number in the Rule Violation field.
4. Click OK.
5. Determine the cause of the violation and correct it. For details, see "[Output from the run drc Command](#)."
6. Run DRC again using the Run DRC Dialog Box.
7. List the violations of the same rule, verify the absence of the violation you just corrected, and examine the remaining violations. (Sometimes, correcting a violation corrects others as well. But it also might create new violations.)
8. Return to [Step 2](#) and repeat the same process until all violations of the rule have been corrected.

The following topics show how to troubleshoot some typical DRC violations:

- [Troubleshooting a Scan Chain Blockage](#)
- [Troubleshooting a Bidirectional Contention Problem](#)

Troubleshooting a Scan Chain Blockage

An S1 rule violation is referred to as a scan chain blockage and is a common DRC violation. The S1 violation occurs when DRC cannot successfully trace the scan chain because a signal somewhere in the circuit is in an incorrect state and is blocking the scan chain.

[Example 1](#) shows the transcript message for violation S1-13.

Example 1: S1-13 Violation Message

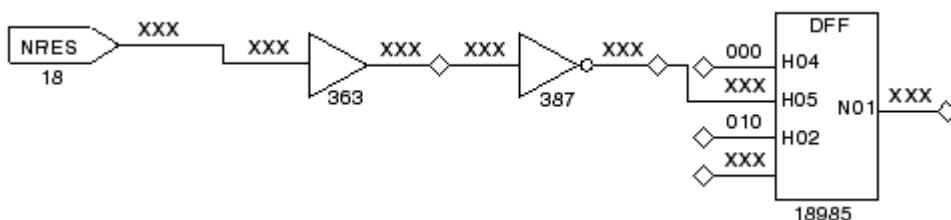
```
Error: Chain c16 blocked at DFF gate /spec_asic/alu/bits/AD_DATIN/
ff_reg (18985)
after tracing 3 cells. (S1-13)
```

The following steps show you how to view the violation:

1. Click the ANALYZE button on the GSV toolbar. The Analyze dialog box opens.
2. Click the Rules tab if it is not already active.
3. Type S1-13 in the Rule Violation box.
4. Click OK.

The schematic in [Figure 1](#) displays the violation. The pin data type has been automatically set to Shift, and the shift data is displayed. The schematic shows the gate identified in the S1-13 violation message and the gates feeding its second pin (the reset pin).

Figure 1: GSV Display: DRC Violation S1-13



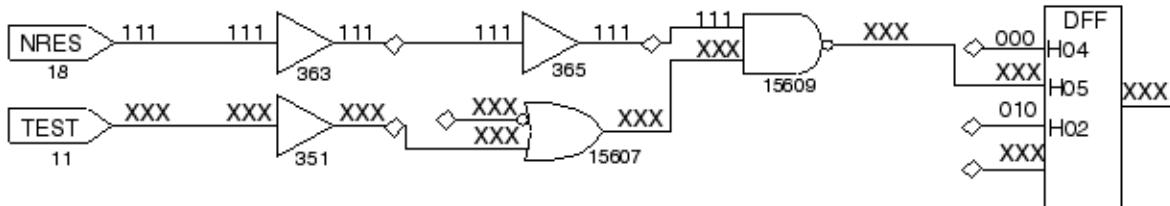
The following steps show you how to find the signal blocking the scan chain at gate 18985:

1. Check the clock and asynchronous pins, starting with the DFF clock pin (H02); it has a 010 simulated state from the shift procedure, which is correct.
2. Check the DFF reset pin (H05); it has an XXX value, which is unacceptable. For a successful shift, H05 must be held inactive.
3. Trace the XXX value back from the H05 pin. The source is the primary input NRES.

The NRES input has an unknown value, either because it was not declared as a clock (as it should have been because of its asynchronous reset capability) or because the STIL load_unload procedure does not force NRES to an off state. You should investigate these possibilities and correct the problem, then execute the `run_drc` command and examine the new list of violations.

After correcting the NRES input problem and executing the `run_drc` command, you select violation S1-9 from the list of remaining S1 violations. As before, you display the violation using the GSV. [Figure 2](#) shows the resulting schematic with Shift pin data displayed.

Figure 2: GSV Display: DRC Violation S1-9

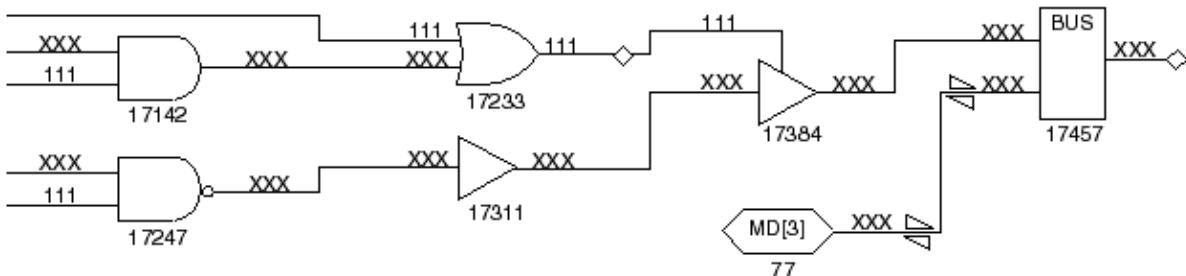


In the [Figure 2](#), although the NRES input is now correctly defined as a clock with an off state of 1, there is a problem with the reset pin, pin H05, on gate 19766 (DFF). Tracing the XXX values back as in the previous example, you find that the source is the primary input TEST. In this case, TEST was not defined as a constrained port in the STIL file.

To correct the problem, you need to edit the STIL file to define TEST as a primary input constrained to a logic 1, make entries in the STIL procedures for `load_unload` and `test_setup` to initialize this primary input, and execute `run_drc` again.

The number of DRC violations decreases with each iteration, but there are still S1 violations. You select another violation and display it in the GSV as shown in [Figure 3](#).

Figure 3: GSV Display: Another S1 Violation



This time, the problem is associated with the bus device, which is a gate inserted by TetraMAX ATPG during ATPG design building to resolve multidriver nets. Both potential sources for the bus inputs appear to be driving, and both have values of X. One of the sources that has an X

value is the MD[3] bidirectional port; you can correct this by driving the port to a Z state. You edit the STIL file to add the declaration MD[3] = Z to one of the V{..} vectors at the start of the load_unload procedure (see “[STIL Procedure Files](#)”).

After you make this correction, you will need to execute the `run_drc` command again and find no further S1 violations.

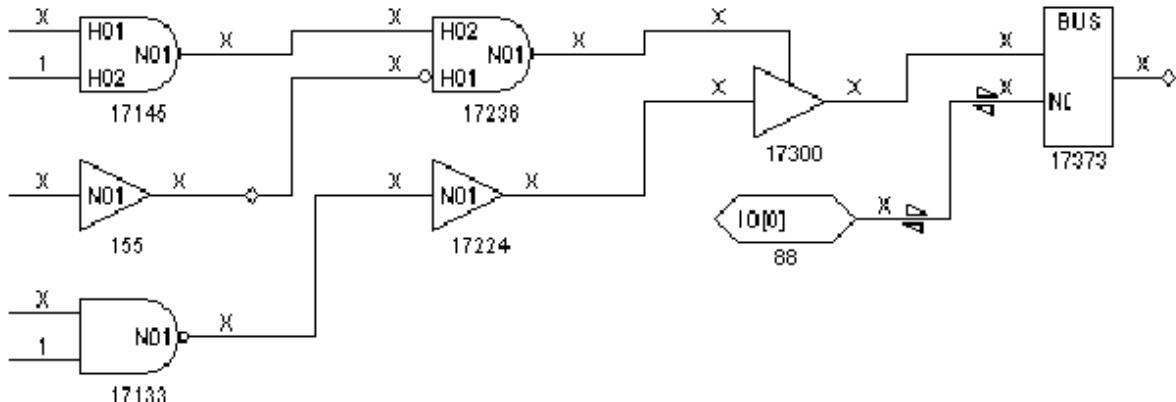
Troubleshooting a Bidirectional Contention Problem

Bidirectional contention issues on ports and internal pins are checked by the Z rules. In Example 1, you use the `report_rules` command to get a listing of Z rules that have failed. This particular report shows 108 Z4 failures and 24 Z9 failures. Suppose you decide to troubleshoot the Z4 failures. You use the `report_violations` command and get a list of five violations, as shown in [Example 1](#). From those, you select the Z4-1 violation to troubleshoot first.

Example 1: report_rules Listing of Violation Messages

```
TEST-T> report_rules -fail
rule severity #fails description
----- -----
S19 warning 201 nonscan cell disturb
C2 warning 201 unstable nonscan DFF when clocks off
C17 warning 17 clock connected to PO
C19 warning 1 clock connected to non-contention-free BUS
Z4 warning 108 bus contention in test procedure
Z9 warning 24 bidi bus driver enable affected by scan cell
TEST-T> report_violations z4 -max 5
Warning: Bus contention on /spec_asic/L030 (17373)
occurred at time 0 of test_setup procedure. (Z4-1)
Warning: Bus contention on /spec_asic/L032 (17374)
occurred at time 0 of test_setup procedure. (Z4-2)
Warning: Bus contention on /spec_asic/L034 (17375)
occurred at time 0 of test_setup procedure. (Z4-3)
Warning: Bus contention on /spec_asic/L036 (17376)
occurred at time 0 of test_setup procedure. (Z4-4)
Warning: Bus contention on /spec_asic/L038 (17377)
occurred at time 0 of test_setup procedure. (Z4-5)
```

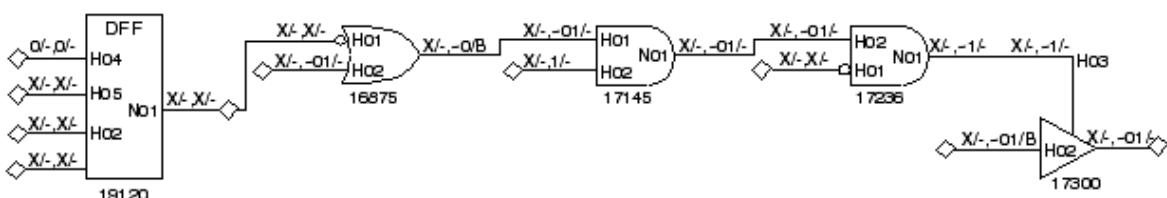
According to the violation error message in [Example 1](#), the problem is bus contention at time 0 of the `test_setup` macro. You display the violation using the GSV, as shown in [Figure 1](#). The schematic shows the `test_setup` data.

Figure 1: GSV Display: DRC Violation Z4-1

The schematic display shows a bidirectional port, IO[0], which is at an X state. In addition, BUS has both inputs driven at X; at least one should be a Z value. Tracing back from BUS, you find a three-state driver TSD (gate 17300) whose enable and data values are both X. There appear to be numerous potential causes of the contention.

The violation message indicates that the violation occurred at time 0 of the test_setup macro. Therefore, you examine the test_setup macro in the STIL procedure file and find that the IO[0] port has not been explicitly set to the Z state. You edit the test_setup macro in the STIL file to add lines that set IO[0] and all the other bidirectional ports to Z.

After eliminating the bus contention, you execute run_drc and find no Z4 violations. However, Z9 violations are still reported. You select Z9-1 for analysis. The pin data is changed to Constrain Value, and the schematic display of the Z9 violation appears as shown in [Figure 2](#).

Figure 2: GSV Display: DRC Violation Z9-1

The Z9-1 violation indicates that the control line to a three-state enable gate is affected by the contents of a scan chain cell. Thus, if a scan chain is loaded with a known value and then a capture clock or reset strobe is applied, the state of the scan cell probably changes and therefore the three-state driver control changes. Depending on the states of the other drivers on this multidriver net, the result might be a driver contention.

You can deal with this violation in one of the following ways:

1. Accept the potential contention, especially if the only other driver of the net is the top-level bidirectional port. In this case, you can set the Z9 rule to ignore for future runs.
2. Alter the design to provide additional controls on the three-state enable. In test mode you might block the path from the scan cell or redirect the control to some top-level port by means of a MUX.
3. Adjust the contention checking to monitor bus contention both before and after clock events. TetraMAX ATPG then discards patterns that result in contention and tries new

patterns in an attempt to find a pattern to detect faults without causing contention. To set bus contention checking, you enter the following command:

```
SETUP> set_contention bus -capture
```

Analyzing Buses

During the DRC process, TetraMAX ATPG analyzes bus and wire gates to determine if they can be in contention.

All bus and wire gates are analyzed to determine if two or more drivers can drive different states at the same time. Bus gates are also analyzed to determine whether they can be placed at a Z state. Drivers that have weak drive outputs are not considered for contention.

This analysis is performed before a DRC analysis of the defined STIL procedures. The data from the analysis is used to prevent issuing false contention violations for the STIL procedures.

The following sections describe how to analyze buses:

- [Bus Contention Status](#)
- [Understanding the Contention Checking Report](#)
- [Reducing Aborted Bus and Wire Gates](#)
- [Causes of Bus Contention](#)

BUS Contention Status

Based on the results of DRC contention analysis, a BUS or wire gate is assigned one of the following contention status types:

- **Pass:** Indicates that the BUS or wire gate can never be in contention. These gates do not have to be checked further.
- **Fail:** Indicates that the BUS or wire gate can be in contention. These gates must be monitored by TetraMAX ATPG during ATPG to avoid patterns with contention.
- **Abort:** Indicates that the analysis for determining a pass/fail classification was aborted. Because these gates were not identified as “pass,” they must be monitored during ATPG.
- **Bidi:** Indicates a BUS gate that has an external bidirectional connection; any internal drivers are not capable of contention. TetraMAX ATPG can avoid contention by controlling the bidirectional ports.

In addition to a contention status, BUS gates undergo an additional analysis to determine whether the driver can achieve a Z state. This produces a Z-state status for each pass, fail, abort, or bidirectional gate.

See Also

[Contention Analysis](#)

Understanding the Contention Checking Report

After the contention check is complete, TetraMAX ATPG displays a report similar to [Example 1](#). This report identifies the number of bus and wire gates and the number of gates that were placed into each contention and Z-state category.

Example 1: DRC Report for Contention Checking

```
SETUP> run_drc spec_stil_file.spf
# -----
# Begin scan design rule checking...
# -----
# Begin reading test protocol file spec_stil_file.spf...
# End parsing STIL file spec_stil_file.spf with 0 errors.
# Test protocol file reading completed, CPU time=0.05 sec.
#
# Begin Bus/Wire contention ability checking...
# Bus summary: #bus_gates=577, #bidi=128, #weak=0, #pull=0,
keepers=0
# Contention status: #pass=257, #bidi=31, #fail=289, #abort=2,
not_analyzed=0
# Z-state status : #pass=160, #bidi=128, #fail=286, #abort=3, not_
analyzed=0
# Warning: Rule Z1 (bus contention ability check) failed 289
times.
# Warning: Rule Z2 (Z-state ability check) failed 289 times.
# Bus/Wire contention ability checking completed, CPU time=7.19
sec.
```

The “Bus summary” line in the report provides the following information:

- #bus_gates: the total number of bus gates in the circuit
- #bidi: the number of bus gates with an external bidirectional port
- #weak: the number of bus gates that have only weak inputs
- #pull: the number of bus gates that have both strong and weak inputs
- #keepers: the number of bus gates connected to a bus keeper

Reducing Aborted Bus and Wire Gates

Bus gates associated with aborted contention checking are still checked for contention during ATPG. If contention checking is aborted for some gates, you should increase the effort used to classify as pass, fail, or bidirectional, rather than abort. You can do this using the Analyze Buses dialog box, or you can use the `set_atpg -abort_limit` and `analyze_buses` commands on the command line, as shown in [Example 1](#).

Example 1: Using `set_atpg -abort_limit` and `analyze_buses`

```
TEST-T> report_buses -summary
Bus summary: #bus_gates=577, #bidi=128, #weak=0, #pull=0,
#keepers=0
```

```

Contention status: #pass=257, #bidi=31, #fail=89, #abort=200,
#not_analyzed=0
Z-state status : #pass=160, #bidi=128, #fail=231, #abort=58,
#not_analyzed=0
TEST-T> set_atpg -abort 50
TEST-T> analyze_buses -all -update
Bus Contention results: #pass=257, #bidi=31, #fail=289, #abort=0,
CPU time=0.00
TEST-T> analyze_buses -zstate -all -update
Bus Zstate ability results: #pass=160, #bidi=128, #fail=289,
#abort=0, CPU
time=0.80
TEST-T> report_buses -summary
Bus summary: #bus_gates=577, #bidi=128, #weak=0, #pull=0,
#keepers=0
Contention status: #pass=257, #bidi=31, #fail=289, #abort=0,
#not_analyzed=0
Z-state status : #pass=160, #bidi=128, #fail=289, #abort=0,
#not_analyzed=0
Learned behavior : none

```

Using the Analyze Buses Dialog Box

To reduce the number of aborted bus and wire gates:

1. From the menu bar, choose Buses > Analyze Buses.
The Analyze Buses dialog box appears.
2. In the Gate ID text field, choose -All.
3. In the Analysis Type text field, choose Prevention.
4. Enable the Update Status option.
5. Click OK.

Using the `set_atpg` and `analyze_buses` Commands

To reduce the number of aborted bus and wire gates from the command, use the `set_atpg -abort_limit` and `analyze_buses` commands.

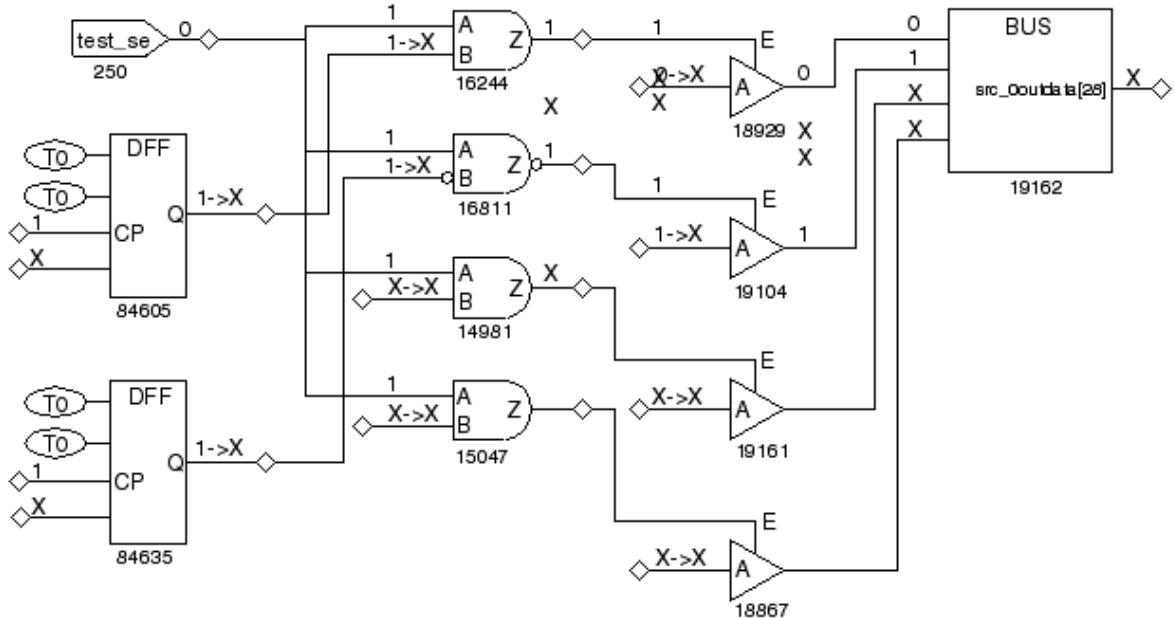
Causes of Bus Contention

After attempting to eliminate bus or wire gates originally classified as aborted, you might want to review some of the bus or wire gates that were classified as failing. To review these gates, view a violation ID from the Z1 or Z2 class. The Z1 class deals with buses that can potentially be in contention, and the Z2 class deals with buses that can potentially be floating.

[Figure 1](#) shows the GSV display of a Z1 violation and the logic that contributes to the three-state driver control. In this case, a pattern was found to cause contention on the bus device, gate 19162. The first two input pins of the bus device are conflicting non-Z values. The remaining two inputs are X. If a conflict is found, TetraMAX ATPG does not fill in the details of the remaining inputs. The source of the potential contention is inherent in the design; with the `test_se` port at 0,

the TSD driver enables are controlled by the contents of the two independent DFF devices on the left. Although there might not be any problem during normal design operation, contention is almost certain to occur under the influence of random patterns.

Figure 1: GSV Display: DRC Violation Z1



You can deal with the Z1 and Z2 violations in one of the following ways:

- Ignore the warnings, with the following consequences:
 - a. Contention will probably occur during pattern generation, and TetraMAX ATPG will discard those patterns that result in contention, possibly increasing the runtime.
 - b. The resulting test coverage could be reduced. TetraMAX ATPG might be forced to discard patterns that would otherwise detect certain faults.
 - c. Floating conditions will probably occur. Although floating conditions might have very little impact on ATPG patterns, internal Z states quickly become X states after passing through a gate, leading to an increased propagation of Xs throughout the design. These Xs eventually propagate to observe points and must be masked off, thus potentially increasing the demands on tester mask resources.
- Modify the design to attempt to eliminate potential contention or, in the case of the Z2 violation, a potential floating internal net. You accomplish this by using DFT logic that ensures that one and only one driver is on at all times, even when the logic is initialized to a random state of 1s and 0s.

Analyzing ATPG Problems

The following steps show you how to analyze ATPG problems that appear as fault sites classified as untestable:

1. View the fault list by opening the Analyze dialog box and clicking the Faults tab. You can also use the `report_faults` command or write a fault list.
2. Select a specific fault class and fault location from the fault list.
3. Display the fault in the GSV using the Analyze dialog box, or use the `analyze_faults` command.
4. View the schematic and transcript to determine the cause of the problem.

The following examples demonstrate the process of analyzing ATPG problems:

- [Analyzing an AN Fault](#)
- [Analyzing a UB Fault](#)
- [Analyzing a NO Fault](#)

Analyzing an AN Fault

This example shows how to perform an analysis on an AN (ATPG untestable—not detected) fault identified as follows:

```
/amd2910/stack/U948/D1
```

[Example 1](#) shows a transcript of an `analyze_faults` command for this fault. TetraMAX ATPG analyzes the fault, draws its location in the GSV, generates one or more patterns, and places them in the internal pattern buffer. You can examine these patterns to determine the controllability and observability issues encountered in classifying the fault.

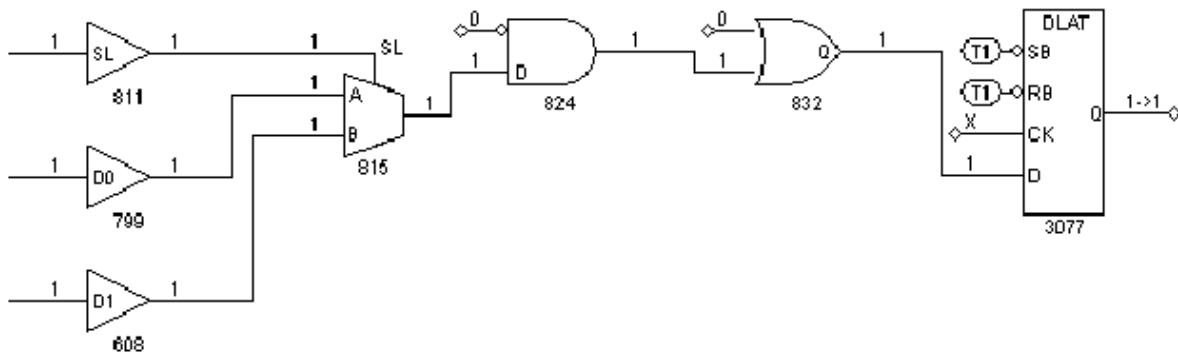
Example 1: Transcript of analyze_faults Results for an AN Fault

```
TEST-T> analyze_faults /amd2910/stack/U948/D1 -stuck 0 -display
-----
Fault analysis performed for /amd2910/stack/U948/D1 stuck at 0
(input 0 of BUF gate 608).
Current fault classification = AN (atpg_untestable-not_det).
-----
Connection data: to=TLA
Fault site control to 1 was successful (data placed in parallel
pattern 0).
Observe_pt=any test generation was unsuccessful due to atpg_
untestable.
Observe_pt=815(MUX) test generation was successful (data placed in
parallel pattern 1).
Observe_pt=824(AND) test generation was successful (data placed in
parallel pattern 2).
Observe_pt=832(OR) test generation was successful (data placed in
parallel pattern 3).
Observe_pt=3077(DLAT) test generation was unsuccessful due to
atpg_untestable.
```

[Figure 1](#) shows the GSV schematic display of the untestable fault location. From the schematic and the messages in [Example 1](#), you can make the following conclusions:

- The fault site was controllable; it could be set to 1. Therefore, controllability is not the reason the fault is untestable.
- Attempts to observe the fault at gates 815, 824, and 832 were successful; therefore, observability at these gates is not the reason the fault is untestable.
- Attempts to observe the fault at gate 3077 (DLAT) were unsuccessful, so observability at this gate could be the reason the fault is untestable. The DLAT is not in a scan chain and is not in transparent mode with this particular pattern (CK pin = X), so the fault cannot be propagated to an observe site.

Figure 1: GSV Display: An AN Fault



The source of the problem seems to be an observability blockage at the DLAT device. You could now explore whether you can place the DLAT in a transparent state using the `run_justification` command, following the method described in "[Checking Controllability and Observability](#)".

Analyzing a UB Fault

This example shows how to analyze a UB (undetectable-blocked) fault using the Analyze dialog box:

- Click the ANALYZE button in the GSV toolbar.
The Analyze dialog box opens.
- Click the Faults tab.
- Click Pin Pathname if it is not already selected.
- Click the Fill button.
- In the Fill Faults dialog box, select “UB: undetectable-blocked” as the Class type.
- Enter 100 in the Last field.
- Click OK in the Fill Faults dialog box.
In the Analyze dialog box, the first 100 UB faults appear in the scrolling window under the Faults tab. Scroll through the list and select a fault to analyze.
- Click OK.

TetraMAX ATPG analyzes the fault selected and displays in the transcript window the equivalent analyze_faults command and the results of the analysis, as shown in [Example 1](#).

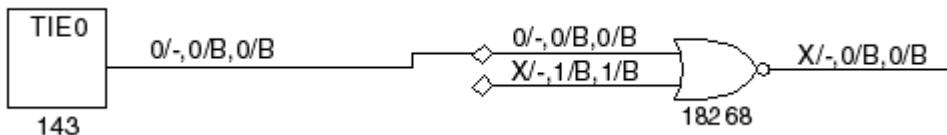
Example 1: Transcript of analyze_faults Results for a UB Fault

```
TEST-T> analyze_faults /JTAG_IR/U51/H02 -stuck 0 -display
-----
Fault analysis performed for /JTAG_IR/U51/H02 stuck at 0 \
(input 1 of OR gate 18268).
Current fault classification = UB \
(undetectable-blocked).

Fault is blocked from detection due to tied values.
Blockage point is gate /MAIN/JTAG_IR/U51 (18268).
Source of blockage is gate /MAIN/U354 (143).
```

[Figure 1](#) shows the graphical representation of the section of the design associated with the fault.

Figure 1: GSV Display: A UB Fault



The fault analysis message provides information similar to that in the schematic display. A stuck-at-0 fault at pin H02 of gate 18268 cannot be detected because the input to pin H01 of this OR gate comes from a tied-to-0 source.

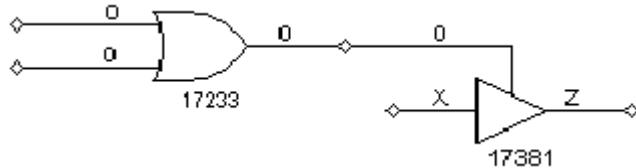
Notice that the schematic contains the fault site as well as the gates involved with the source of the blockage. In addition, the pin data type has been set to Constrain Data and the constraint information is displayed directly on the schematic. For an interpretation of constrain values, see ["Displaying Constrain Values."](#)

In the example in [Figure 1](#), TetraMAX ATPG has analyzed a stuck-at-0 fault on the H02 pin in the schematic. The transcript shows that this fault is UB, that the blockage point is gate 18268, and that the source of the blockage is gate 143.

You review the GSV display in [Figure 1](#) to gain some additional insight. Gate 143 is a tie-off cell that ties the H01 input of gate 18268 to 0, forcing the output of gate 18268 to a logic 1 and blocking the propagation of faults from pin H02.

Analyzing a NO Fault

[Figure 1](#) shows the schematic for an NO (not-observed) fault class.

Figure 1: GSV Display: A NO Fault

The fault report in [Example 1](#) states that the fault site is controllable but not observable. A pattern that controlled the fault site to 0 to detect a stuck-at-1 fault was placed in the internal pattern buffer as pattern 0, but was later rejected because the pattern failed bus contention checks.

Example 1: analyze_faults Report for a NO Fault

```

TEST-T> analyze_faults /U317/H02 -stuck 1 -display
-----
Fault analysis performed for /U317/H02 stuck at 1 (output of OR
gate 17233).
Current fault classification = NO (not-observed).
-----
Connection data: to=REGPO,MASTER,TS_ENABLE
Fault site control to 0 was successful (data placed in parallel
pattern 0).
Observe_pt=any test generation was unsuccessful due to abort.
Observe_pt=17381(TSD) test generation was unsuccessful due to
atpg_untestable.
Warning: 1 pattern rejected due to 32 bus contentions (ID=17373,
pat1=0). (M181)

```

The fault report mentions two observe points. The first, “any test generation,” was unsuccessful because an abort limit was reached. The second observe point at gate 17381 was unsuccessful because of ATPG-untestable conditions at that gate. The first observe point might succeed if you increase the abort limit and try again.

Printing a Schematic to a File

You can use the `gsv_print` command to create a grayscale PostScript file, which captures the schematic displayed in the graphical schematic viewer (GSV):

```
gsv_print -file file -banner string Y
```

You can add the `gsv_print` command to your TetraMAX scripts and automatically capture schematic output. The computing host must have PostScript drivers installed (usually with lpr/lp). You can enclose the arguments in double quotation marks ("").

6

Using the Hierarchy Browser

The Hierarchy Browser displays a design's basic hierarchy and enables graphical analysis of coverage issues. It is launched as a standalone window that sits on top of the TetraMAX GUI main window.

Before you start using the Hierarchy Browser, you should familiarize yourself with the graphical schematic viewer (GSV). See "[Using the Graphical Schematic Viewer](#)" for more information.

The Hierarchy Browser does not display layout data. It is intended to supplement the GSV so you can analyze graphical test coverage information while browsing through a design's hierarchy.

The following topics describe how to use the Hierarchy Browser:

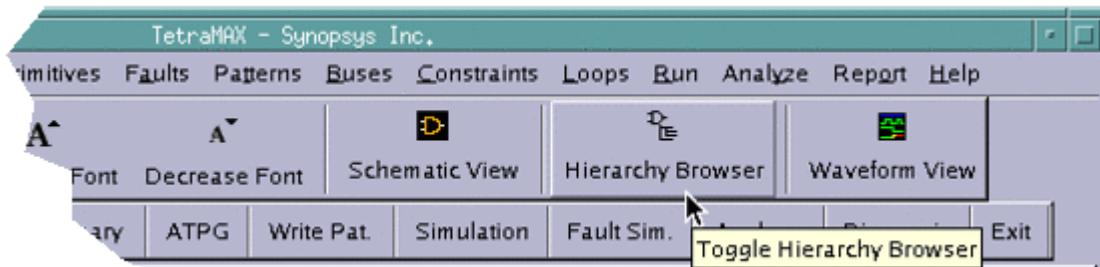
- [Launching the Hierarchy Browser](#)
- [Basic Components of the Hierarchy Browser](#)
- [Performing Fault Coverage Analysis](#)
- [Exiting the Hierarchy Browser](#)

Launching the Hierarchy Browser

To launch the Hierarchy Browser, you first need to begin the ATPG flow and start the TetraMAX GUI, as described in the following steps:

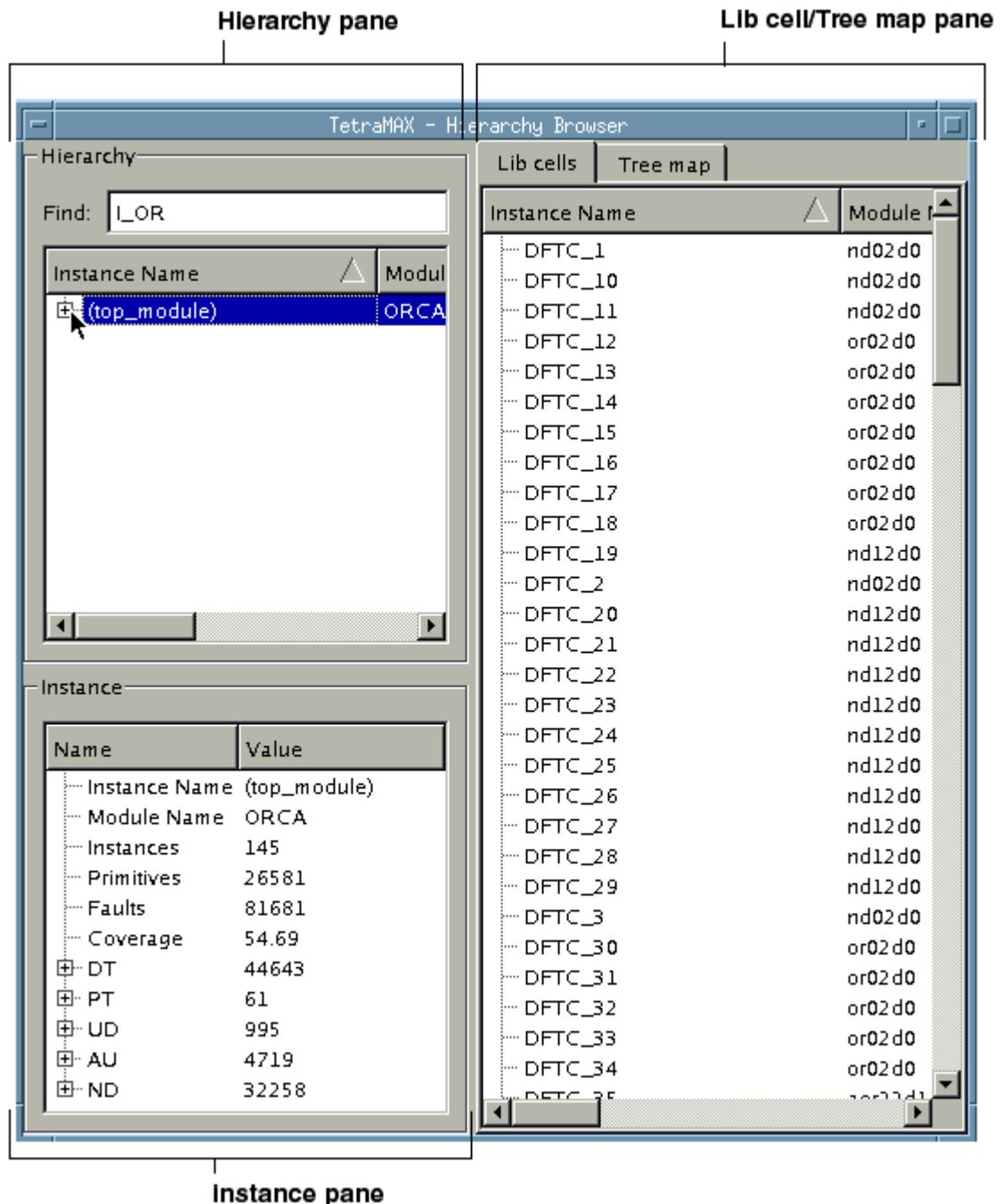
1. Follow the initial test pattern generation steps described in "[ATPG Design Flow](#)."
2. Launch the TetraMAX GUI. For details, see "[Controlling TetraMAX Processes](#)."
3. After the DRC process is completed, start the Hierarchy Browser by clicking the Hierarchy Browser button in the TetraMAX GUI, as shown in [Figure 1](#).

Figure 1: Hierarchy Browser Button in the TetraMAX GUI



The Hierarchy Browser will appear as a new window, as shown in [Figure 2](#).

Figure 2: Initial Display of the Hierarchy Browser



See Also

[Exiting the Hierarchy Browser](#)

Basic Components of the Hierarchy Browser

The Hierarchy Browser is comprised of the following main components:

- **Hierarchy Pane** — Located in the top left portion of the browser, this area displays an expandable view of the design hierarchy and test coverage data.
 - **Instance Pane** — Located in the bottom left portion of the browser, this area displays test coverage data associated with the module selected in the Hierarchy pane.
 - **Lib Cell/Tree Map Pane** — Located in the right portion of the browser, this area toggles between library cell data and a graphical display of all submodules associated with the selected instance in the Hierarchy pane.
-

Using the Hierarchy Pane

The Hierarchy pane displays the overall design hierarchy, including the number of instances, the test coverage, the number of faults, and the main fault types. It also controls the data displayed in the Instance pane and Lib cell/Tree map pane.

Using the Hierarchy pane, you can expand and collapse the hierarchical display of a design's submodules and view the associated test coverage information.

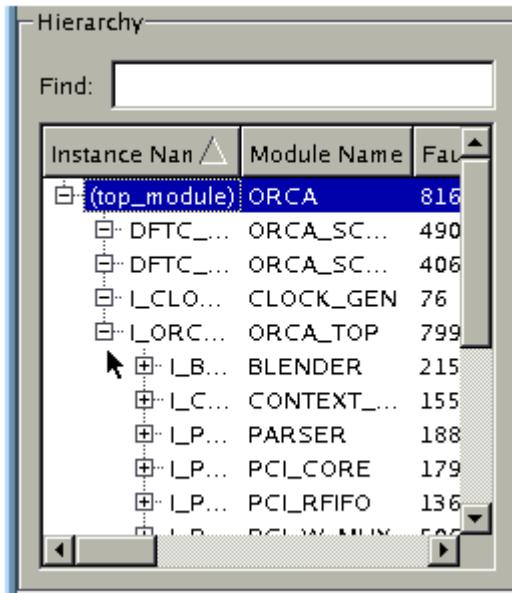
The following steps describe how to display coverage data in the Hierarchy pane:

1. Click the symbol next to the top instance name.

The design hierarchy expands, and the submodules and related test coverage information for the top instance name are displayed.

Instance Name	Module Name	Faults
(top_module)	ORCA	81681
DFTC_...	ORCA_SC...	490
DFTC_...	ORCA_SC...	406
I_CLO...	CLOCK_GEN	76
I_ORC...	ORCA_TOP	79942

2. Continue to expand the hierarchy, as needed.

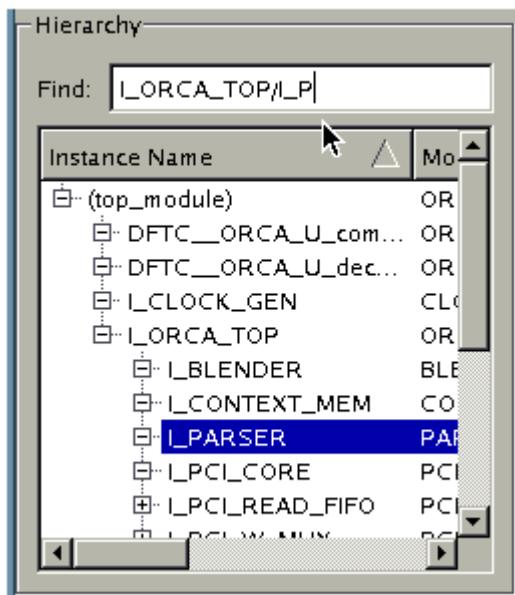


3. Move the slider bar, located at the bottom of the pane, to view coverage details and fault information associated with the instance names in the left column.

The screenshot shows the same 'Hierarchy' window, but the table has expanded to show coverage details for each instance. The columns are labeled 'Coverage', 'DT', 'PT', 'UD', and 'AU'. The data is as follows:

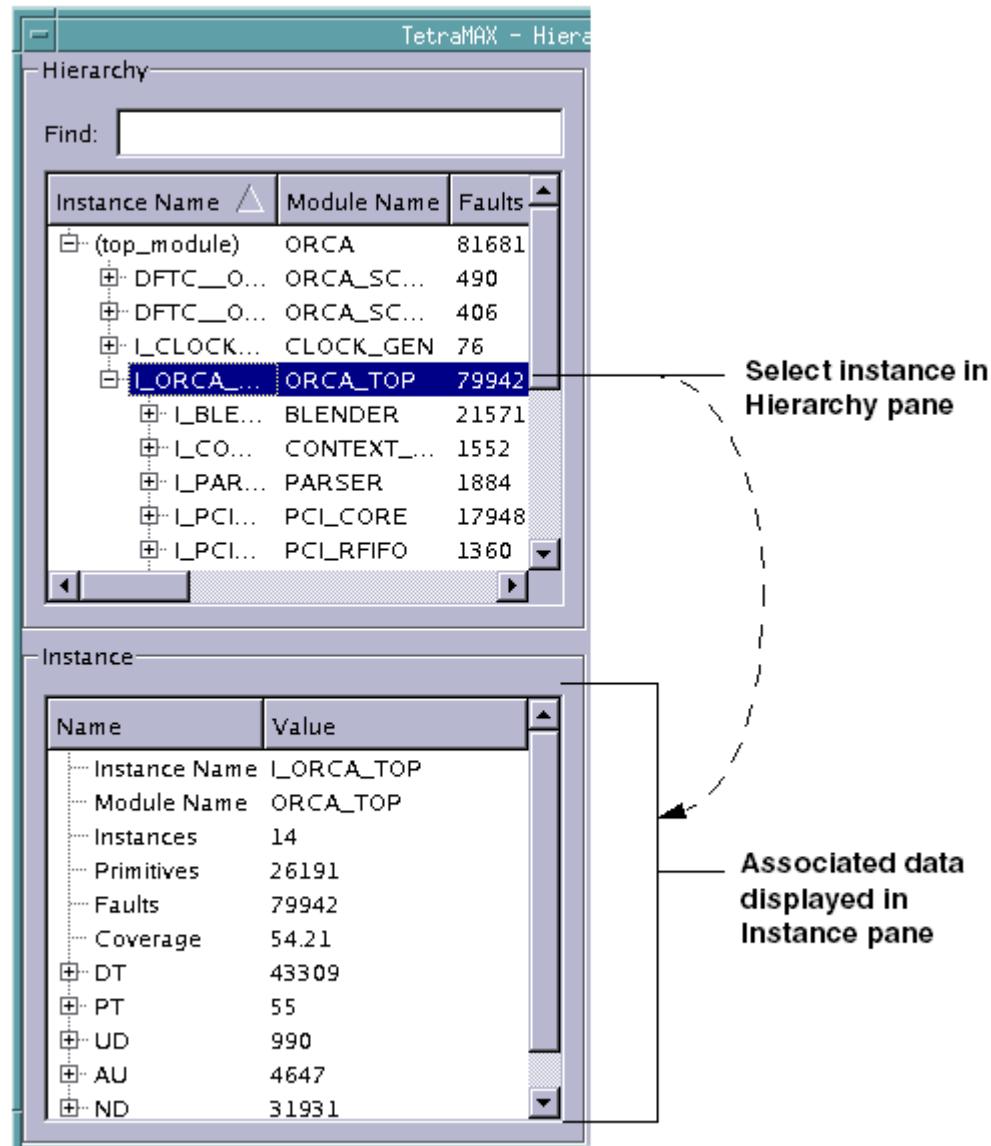
Coverage	DT	PT	UD	AU
54.69	44643	61	995	4719
93.88	460	0	0	0
45.32	184	0	0	0
30.26	20	6	0	48
54.21	43309	55	990	4647
33.73	7276	0	5	798
8.51	132	0	342	778
50.74	956	0	0	0
73.87	13259	0	0	604
52.35	712	0	0	32
0.00	0	0	0	505

4. Find data for a particular instance using the Find text field.



Viewing Data in the Instance Pane

The Instance pane displays coverage data for the instance selected in the Hierarchy pane, as shown in [Figure 1](#).

Figure 1: Displaying Information in the Instance Pane

You can expand the display of information for a fault type in the Instance pane by clicking the symbol next to a fault class, as shown in [Figure 2](#).

Figure 2: Expanding the Display of Data for the DT Fault Class

The screenshot shows a software interface titled "Instance". A tree view on the left lists categories like "Instance Name", "Module Name", "Instances", "Primitives", "Faults", "Coverage", and "DT". The "DT" node is expanded, revealing sub-items: "Constraints" (44), "To CLKPO" (386), "To SHA..." (6), and "To DSLA..." (21). To the right of the tree is a table with two columns, "Name" and "Value". The first row shows "Instance Name" as "L_ORCA_TOP" and "Module Name" as "ORCA_TOP". Subsequent rows show "Instances" (14), "Primitives" (26191), "Faults" (79942), and "Coverage" (54.21). The "DT" row is expanded, showing its sub-data.

Name	Value
Instance Name	L_ORCA_TOP
Module Name	ORCA_TOP
Instances	14
Primitives	26191
Faults	79942
Coverage	54.21
DT	43309
Constraints	44
To CLKPO	386
To SHA...	6
To DSLA...	21

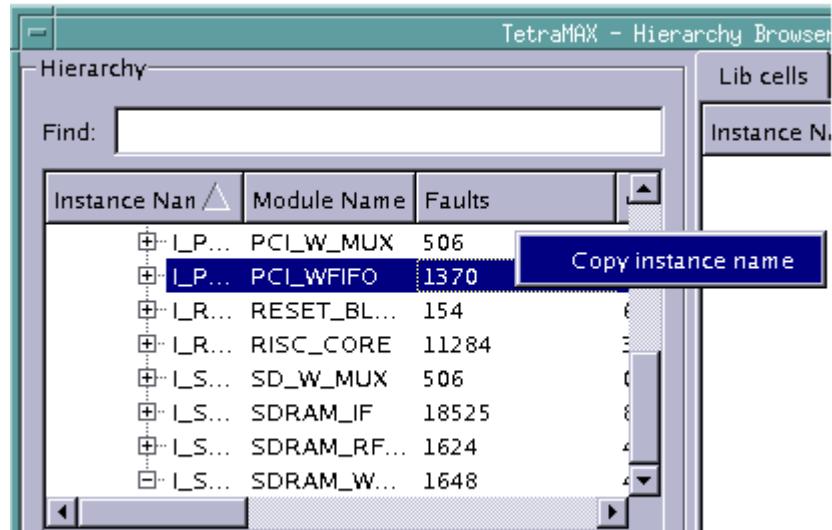
Copying an Instance Name

You can copy the full instance name from anywhere in the Hierarchy Browser and paste it in the Find text field, or use it for reference purposes in other applications.

To copy an instance name:

- Right-click on an instance name anywhere in the Hierarchy Browser, and select Copy Instance Name.

Figure 3: Copying An Instance Name



Viewing Data in the Lib Cells/Tree Map Pane

The Lib cells/Tree map pane toggles between two tabs:

- **Lib cells** — Displays module names, primitives, faults, test coverage, and fault class data for library cells, which include all non-hierarchical cells in a selected module. An example is shown in [Figure 4](#).
- **Tree map** — Displays a design's hierarchical graphical test coverage. An example is shown in [Figure 5](#).

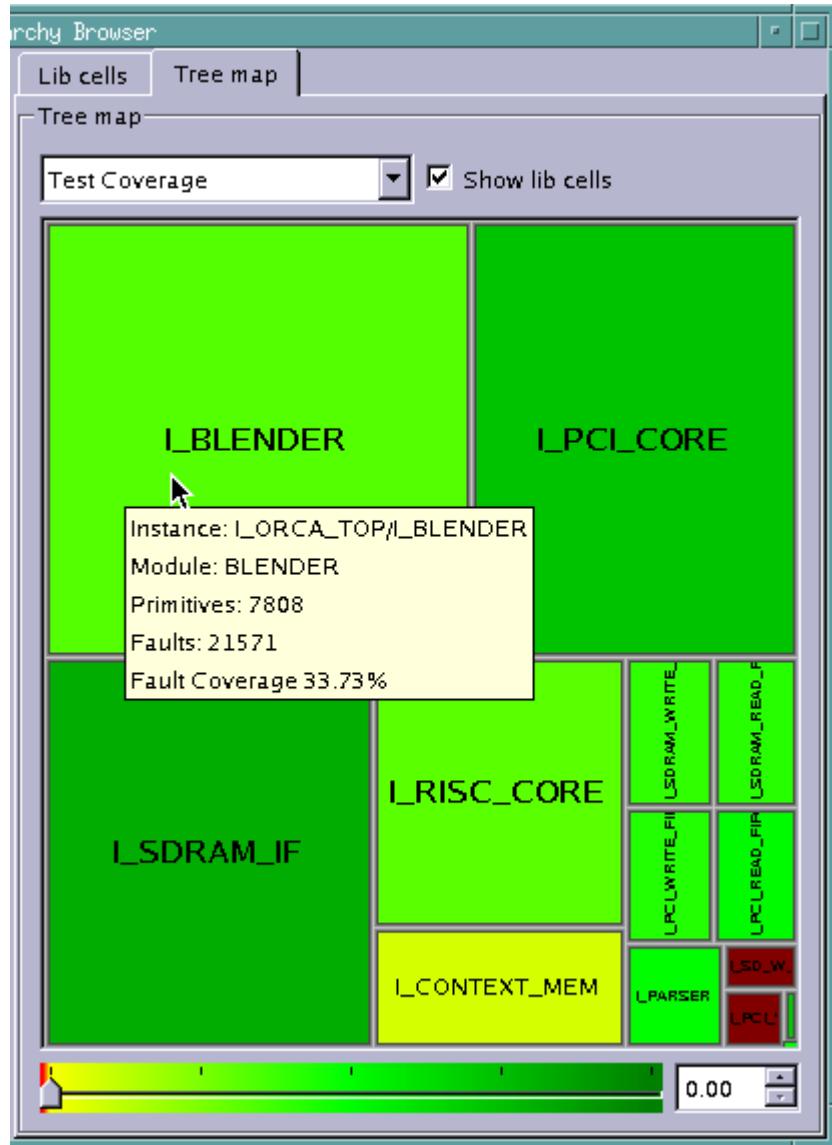
Figure 4: Display of Library Cell Information

Hierarchy Browser

Lib cells Tree map

Instance Name	Module Name	Primitives	Faults	Coverage	DT	PT	UD	AU	ND
count_int_reg_0_	sdcrlq1	2	12	75.00	9	0	0	0	3
count_int_reg_0_1	sdcrlq1	2	12	66.67	8	0	0	0	4
count_int_reg_1_	sdcrlq1	2	12	75.00	9	0	0	0	3
count_int_reg_1_1	sdcrlq1	2	12	66.67	8	0	0	0	4
count_int_reg_2_	sdcrlq1	2	12	75.00	9	0	0	0	3
count_int_reg_2_1	sdcrlq1	2	12	66.67	8	0	0	0	4
count_int_reg_3_	sdcrlq1	2	12	75.00	9	0	0	0	3
count_int_reg_3_1	sdcrlq1	2	12	66.67	8	0	0	0	4
count_int_reg_4_	sdcrlq1	2	12	100.00	12	0	0	0	0
count_int_reg_4_1	sdcrlq1	2	12	91.67	11	0	0	0	1
count_int_reg_5_	sdcrlq1	2	12	75.00	9	0	0	0	3
count_int_reg_5_1	sdcrlq1	2	12	91.67	11	0	0	0	1
count_int_reg_6_	sdcrlq1	2	12	75.00	9	0	0	0	3
count_int_reg_6_1	sdcrlq1	2	12	91.67	11	0	0	0	1
DFTC_1	aor22d1	3	10	50.00	5	0	0	4	1
empty_int_reg1	sdcrlm1	2	12	91.67	11	0	0	0	1
full_int_reg	sdcrlq1	2	12	66.67	8	0	0	0	4
LOCKUP	lanlql1	2	6	100.00	6	0	0	0	0
SD_WFIFO_RAM	ram32x64	265	94	0.00	0	0	0	94	0
sync_reg_0_	sdcrlq1	2	12	75.00	9	0	0	0	3
sync_reg_0_1	sdcrlq1	2	12	66.67	8	0	0	0	4
sync_reg_1_	sdcrlq1	2	12	75.00	9	0	0	0	3
sync_reg_1_1	sdcrlq1	2	12	66.67	8	0	0	0	4
sync_reg_2_	sdcrlq1	2	12	75.00	9	0	0	0	3
sync_reg_2_1	sdcrlq1	2	12	66.67	8	0	0	0	4
sync_reg_3_	sdcrlq1	2	12	75.00	9	0	0	0	3
sync_reg_3_1	sdcrlq1	2	12	66.67	8	0	0	0	4
sync_reg_4_	sdcrlq1	2	12	66.67	8	0	0	0	4
sync_reg_4_1	sdcrlq1	2	12	66.67	8	0	0	0	4
sync_reg_5_	sdcrlq1	2	12	75.00	9	0	0	0	3
sync_reg_5_1	sdcrlq1	2	12	66.67	8	0	0	0	4
sync_reg_6_	sdcrlq1	2	12	75.00	9	0	0	0	3
sync_reg_6_1	sdcrlq1	2	12	66.67	8	0	0	0	4

Figure 5: Tree Map View



As shown in [Figure 5](#), the data displayed in the Tree map is color-coded according to the test coverage. Dark green indicates the maximum coverage, light green is slightly lower coverage, yellow is minimal coverage, and dark red is coverage below the minimum threshold.

When you hold your pointer over a particular instance, a pop-up window will display detailed coverage information for that instance.

Additional details on using the Tree map are provided in the following section, “[Performing Fault Coverage Analysis](#).”

Performing Fault Coverage Analysis

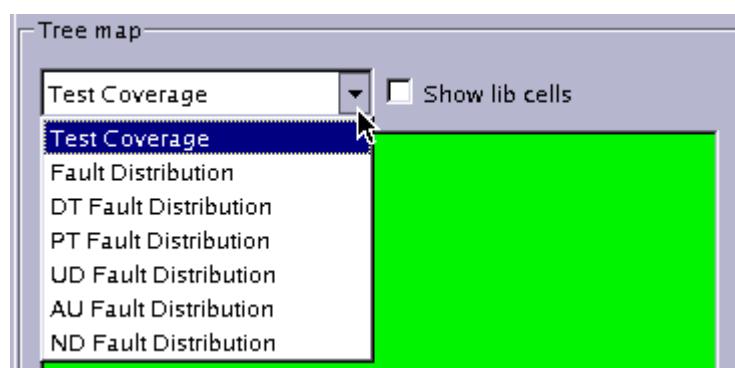
You can access and adjust the display of a variety of interactive test coverage data in the Hierarchical Browser. The following sections show you how to display various types of data that will help you perform fault coverage analysis:

- [Understanding the Types of Coverage Data](#)
 - [Expanding the Design Hierarchy](#)
 - [Viewing Library Cell Data](#)
 - [Adjusting the Threshold Slider Bar](#)
 - [Identifying Fault Causes](#)
 - [Displaying Instance Information in the GSV](#)
-

Understanding the Types of Coverage Data

You can view data in the Tree map based on the overall test coverage, the fault distribution, or the fault class distribution, which includes DT (detected), PT (possibly detected), UD (undetectable), AU (ATPG untestable), or ND (not detected). As shown in [Figure 1](#), the Tree map has a drop-down menu that you can use to select the type of coverage data you want to view.

Figure 1: Selecting the Type of Coverage Data to Display in the Tree Map



The various categories of coverage data include the following information:

- **Test Coverage:** $\text{Test Coverage} = (\text{DT} + \text{PT_CREDIT} * \text{PT}) / \text{Faults}$

Note: UD faults are excluded from the Test Coverage equation, which matches the output of the `report_faults -level` command. However, the output from the

`report_faults -summary` command does not match the Hierarchy Browser because it includes UD faults.

- **Fault Distribution:** <displayed area> = <number of (DT+PT+AU+ND) faults >
- **DT Fault Distribution:** <displayed area> = <number of DT faults>
- **PT Fault Distribution:** <displayed area> = <number of PT faults>
- **UD Fault Distribution:** <displayed area> = <number of UD faults>
- **AU Fault Distribution:** <displayed area> = <number of AU faults>
- **ND Fault Distribution:** <displayed area> = <number of ND faults>

See Also

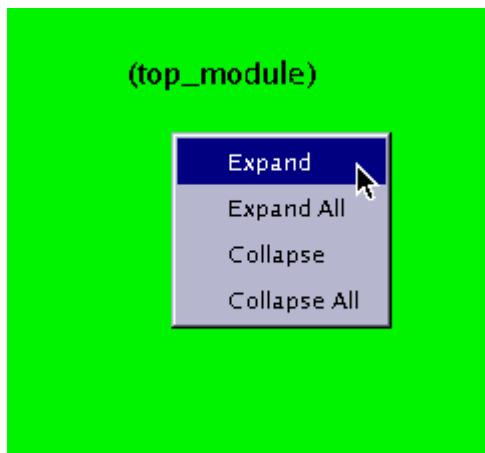
[Fault Categories and Classes](#)

Expanding the Design Hierarchy

When the Hierarchy Browser is initially invoked, the Tree map displays only the top-level instance in the design. The following steps show you how to expand the display of the design hierarchy:

1. Right-click your mouse in the Tree map, then select Expand to expand the display of one level of the design hierarchy.

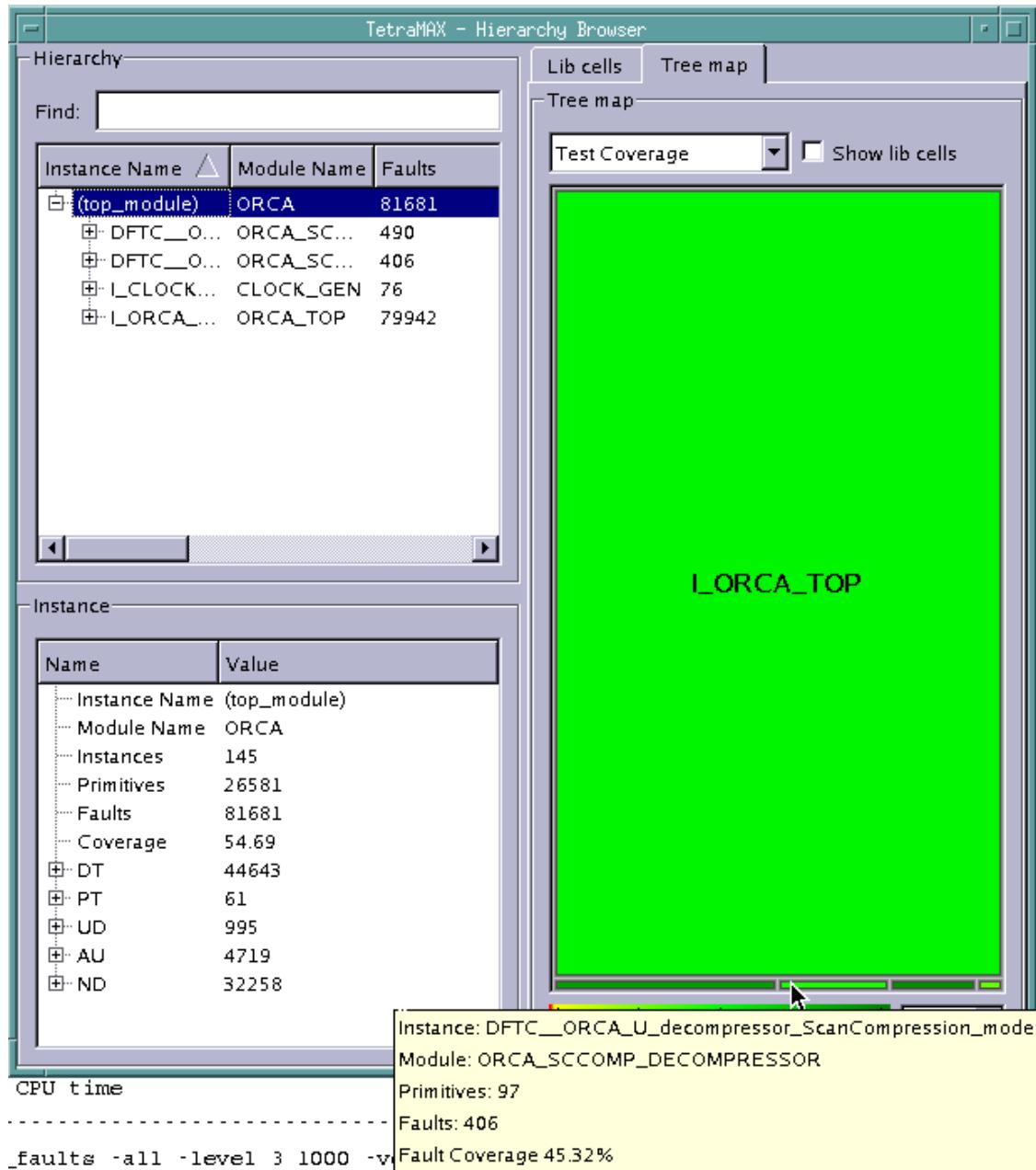
Figure 1: Expanding the Design Hierarchy



After selecting Expand, the next level of hierarchy is displayed in the Tree map, as shown in [Figure 2](#).

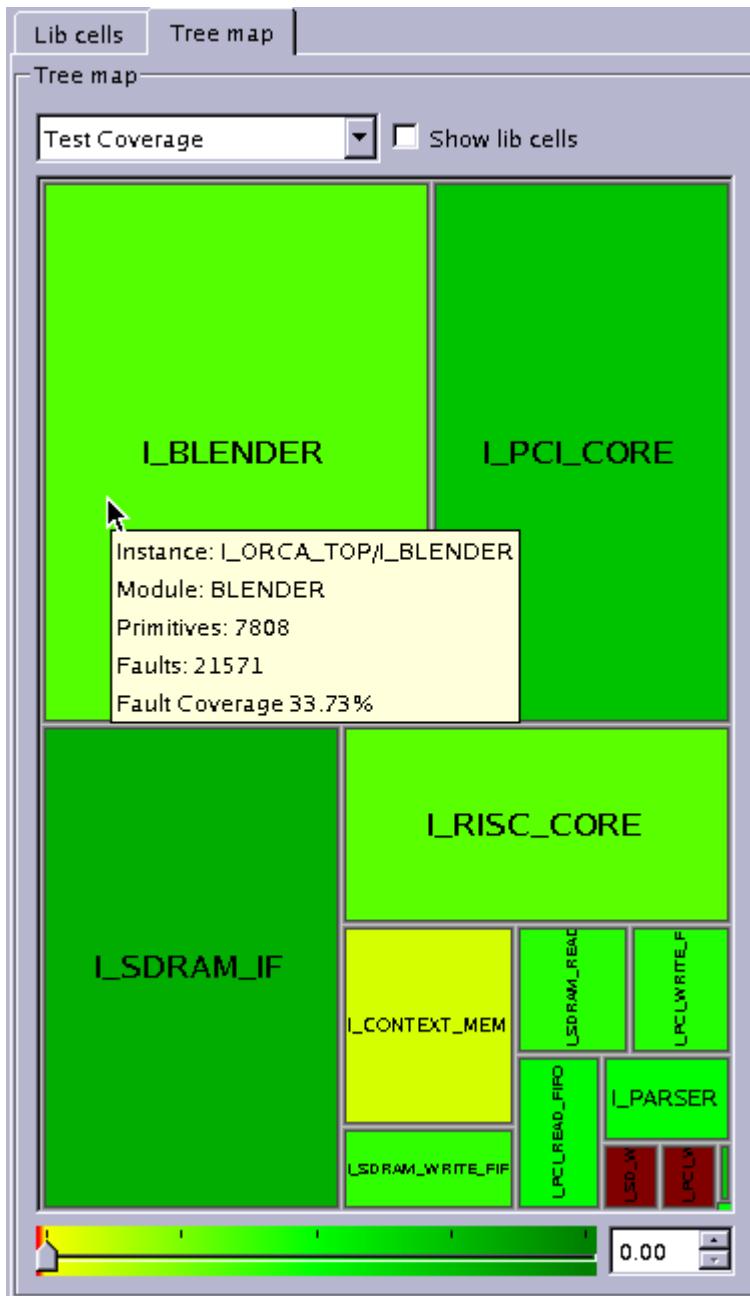
Note: The Hierarchy Browser is not a layout viewer. The size of each graphically represented instance is based on the number of primitives for that instance in proportion to the other instances. In [Figure 2](#), the largest displayed instance is I_ORCA_TOP. Below that instance are four smaller instances. The pointer at the bottom of the window is highlighting the DFTC_ORCA_U instance. Also note that the data in the Hierarchy pane, in the upper left portion of the window, expands to coincide with the Tree map view.

Figure 2: Display of First Level of Hierarchy



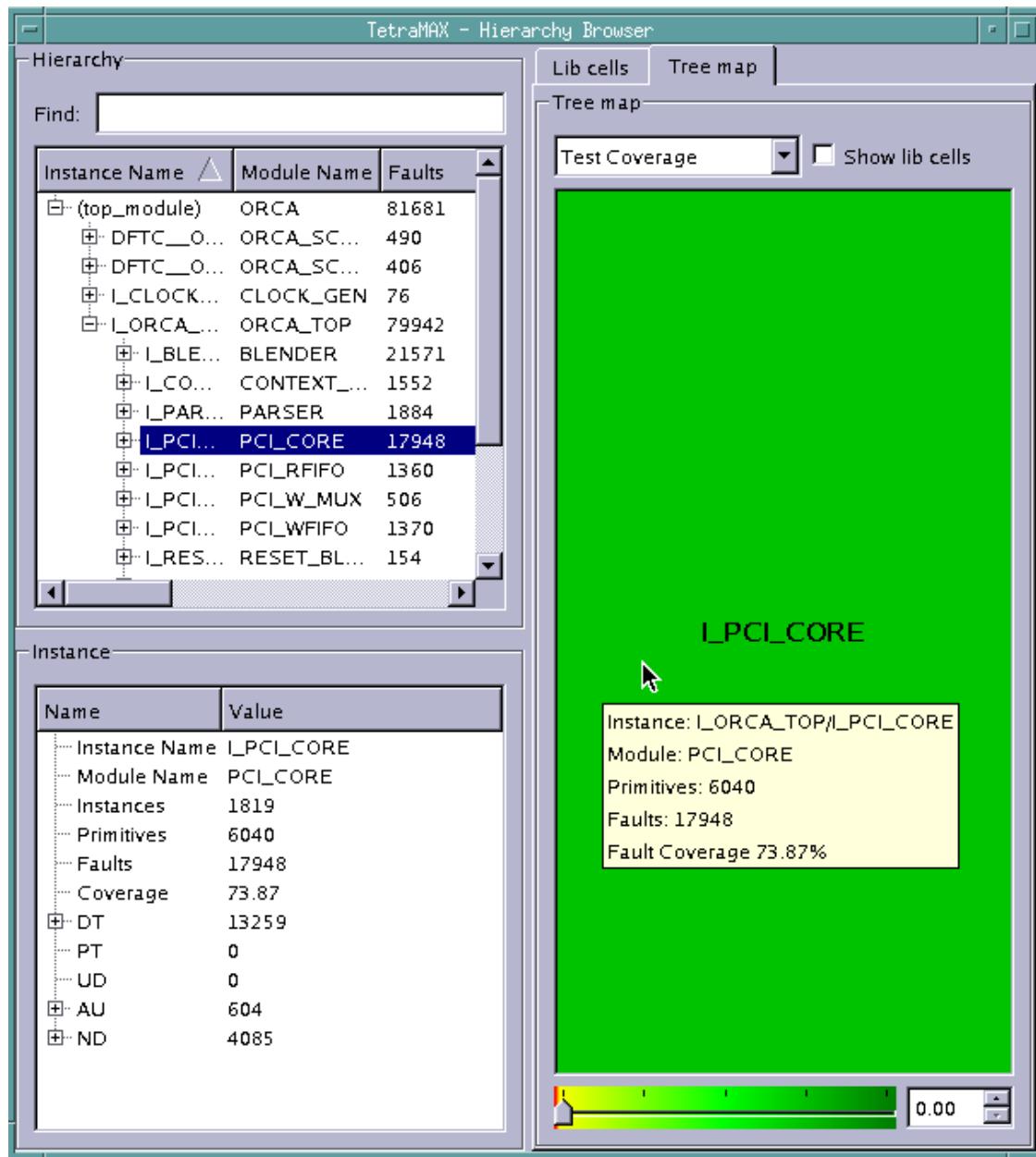
2. You can continue to expand the design hierarchy one level at a time by right-clicking and selecting Expand, or by selecting Expand All to expand the entire design hierarchy. [Figure 3](#) shows the full display of a design's hierarchy.

Figure 3: Display of a Design's Full Hierarchy



3. You can further focus the display of data for a particular instance by clicking on the instance in the Tree map or in the Hierarchy pane, as shown in [Figure 4](#).

Figure 4: Focusing the Tree Map Display on a Single Instance

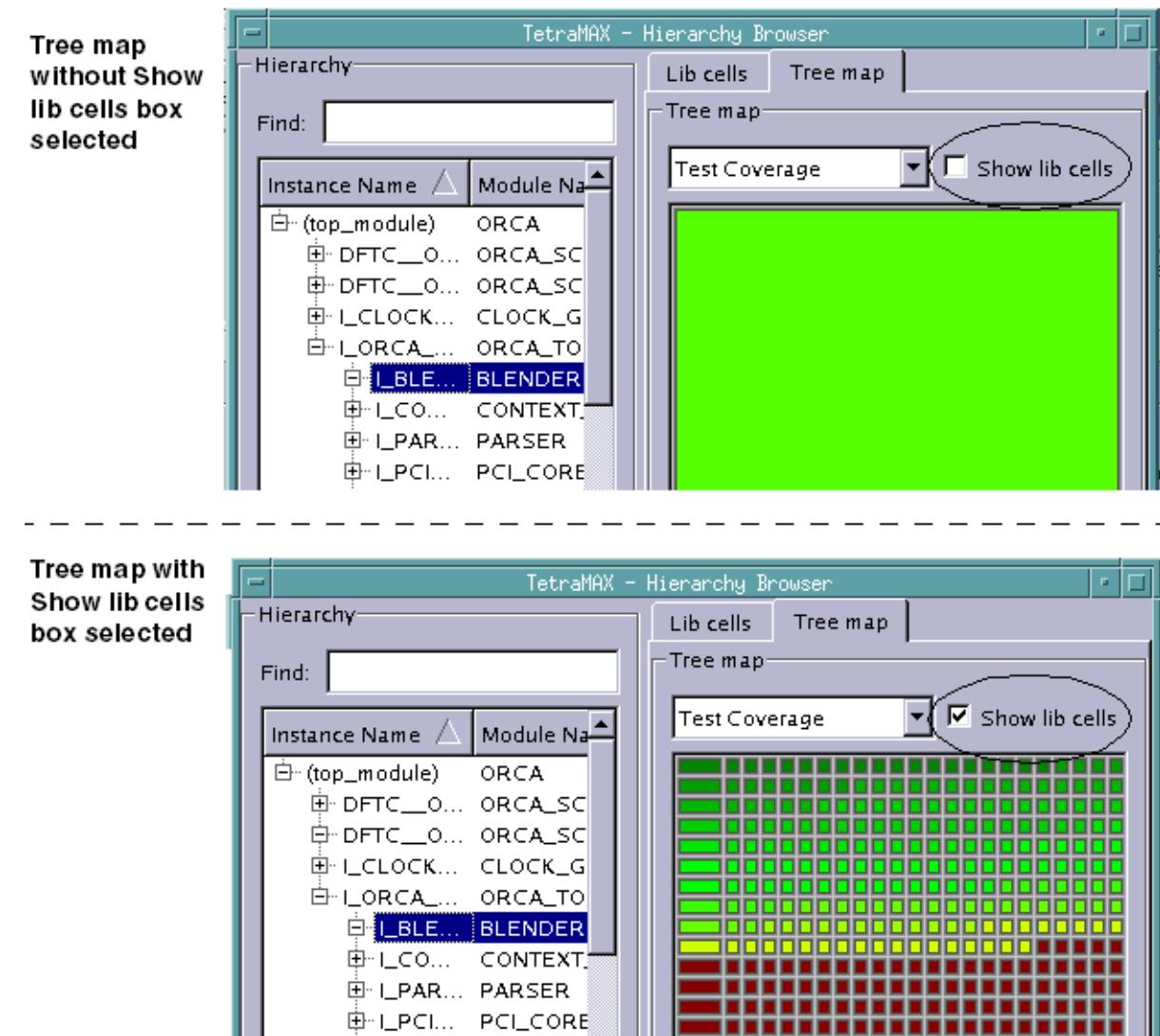


4. To collapse the display of the design hierarchy, right-click anywhere in the Tree map and select Collapse or Collapse All.

Viewing Library Cell Data

You can view a graphical representation of library cells associated with a particular instance by clicking the Show lib cells check box in the Tree map. Figure 1 shows how selecting this check box affects the display of data in an example instance.

Figure 1: How Selecting the Show Lib Cells Box Affects the Tree Map Display

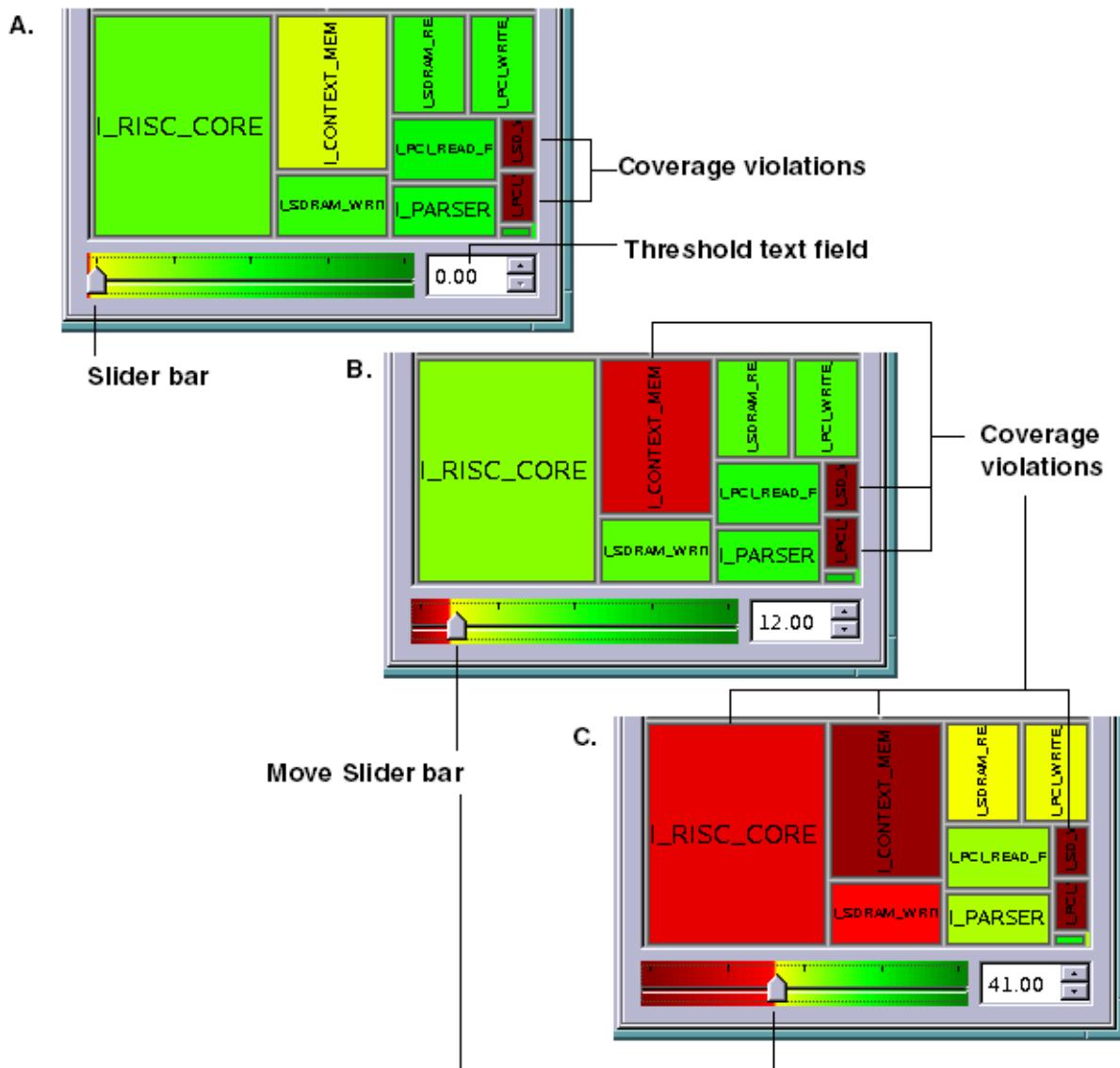


Adjusting the Threshold Slider Bar

The threshold slider bar is located at the bottom of the Tree map. You can use this bar to change the threshold for the color spectrum display of fault coverage. By default, the threshold is set to 0% coverage, which means that any instances with 0% coverage are displayed in red.

To change the threshold, either move the slider bar or enter a different value in the threshold text field. Figure 1 shows the comparative effect of moving the threshold slider bar.

Figure 1: Effect of Moving Threshold from A (0%) to B (12%) to C (41%)



Identifying Fault Causes

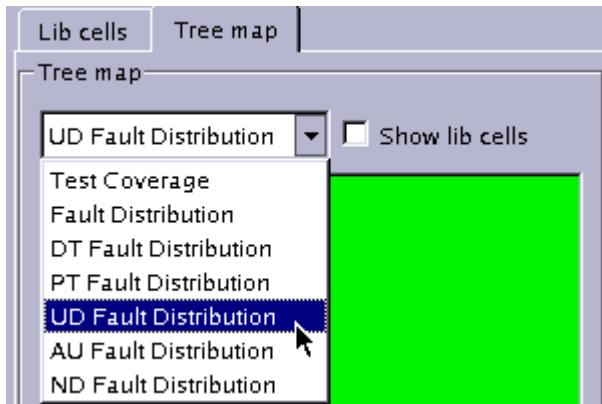
The Hierarchy Browser enables you to identify causes of various faults. The four basic fault causes are as follows:

- Constrain Values
- Constrain Value Blockage
- Connected to <value>
- Connected from <value>

The following steps show you how to identify fault causes for a specific fault class in a specific instance:

1. In the drop-down menu located the top of the Tree map, select the type of coverage you want to display. For example, select UD Fault Distribution.

Figure 1: Drop-Down Menu



2. If required, click the Show lib cells check box to view all the cells in the instance.
3. Expand the display of the design's hierarchy, as needed.
4. Click an instance of interest in the Tree map or select an instance in the Hierarchy pane.

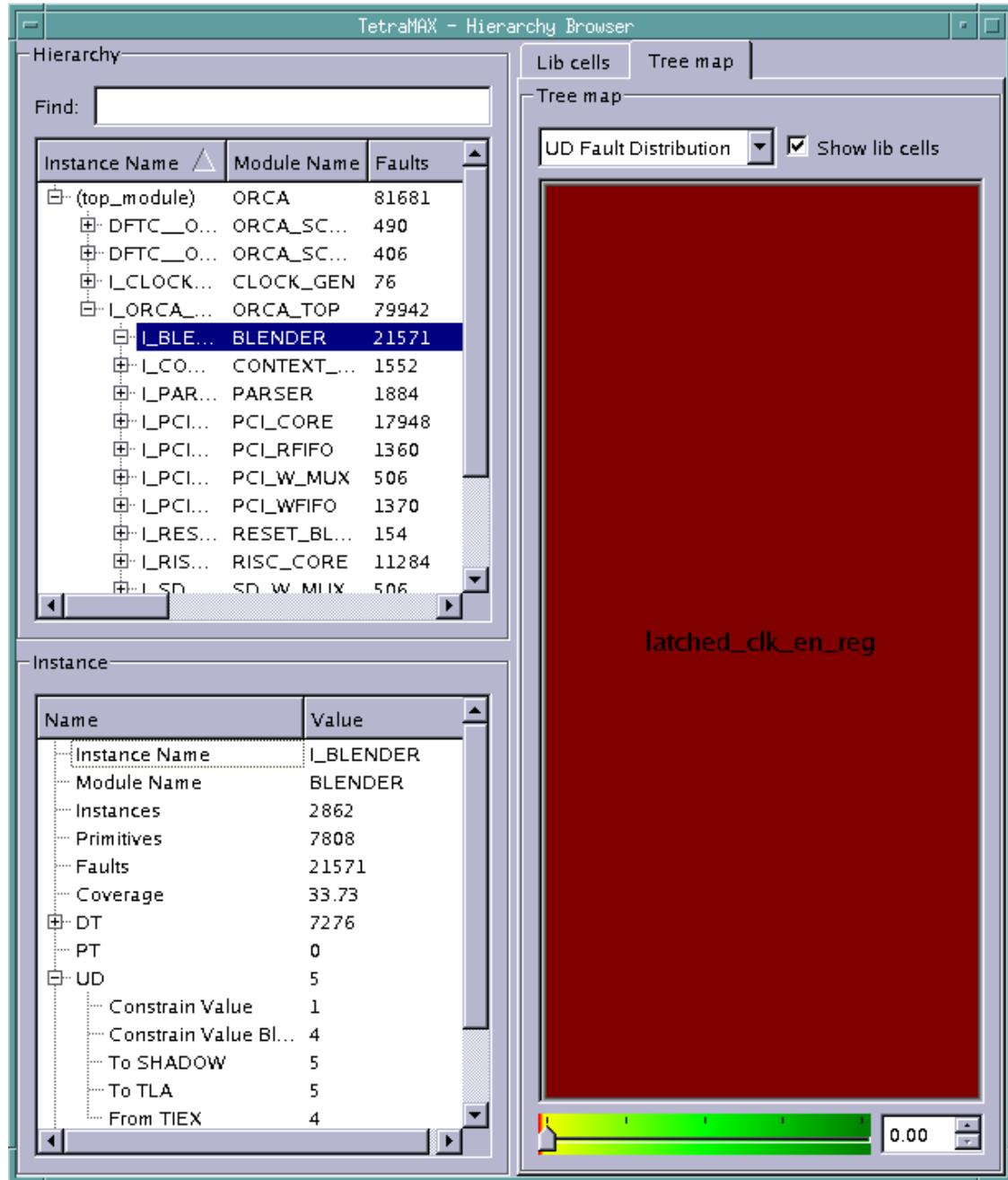
The Instance pane displays the name of the selected instance and its related coverage data, as shown in Figure 2.

Figure 2: Display of Coverage Information for Selected Instance in Instance Pane

Name	Value
Instance Name	I_BLENDER
Module Name	BLENDER
Instances	2862
Primitives	7808
Faults	21571
Coverage	33.73
DT	7276
PT	0
UD	5
AU	798
ND	13497

5. Expand the display of the fault class of interest in the Instance pane. Figure 3 shows the expansion of the UD fault class and display of the related fault causes.

Figure 3: Display of Fault Class and Related Fault Causes



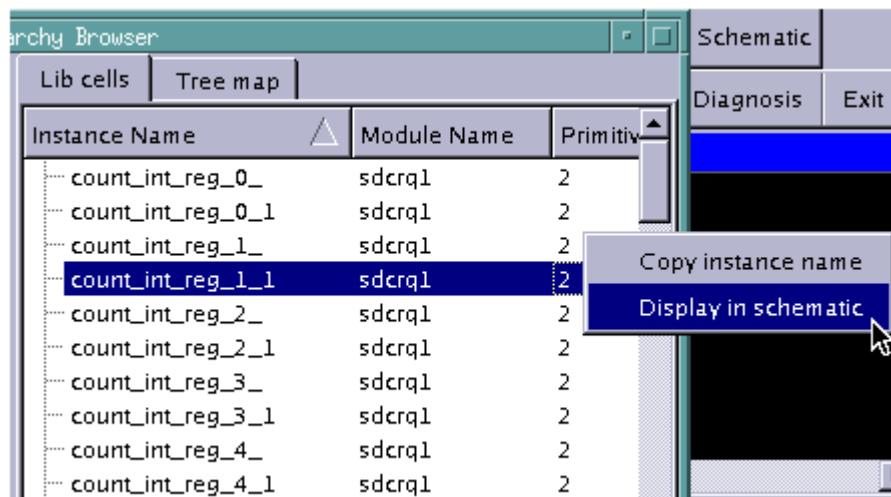
Displaying Instance Information in the GSV

You can select an instance name anywhere in the Hierarchy Browser and display it in the graphical schematic viewer (GSV).

To display a selected instance from the Hierarchy Browser in the GSV:

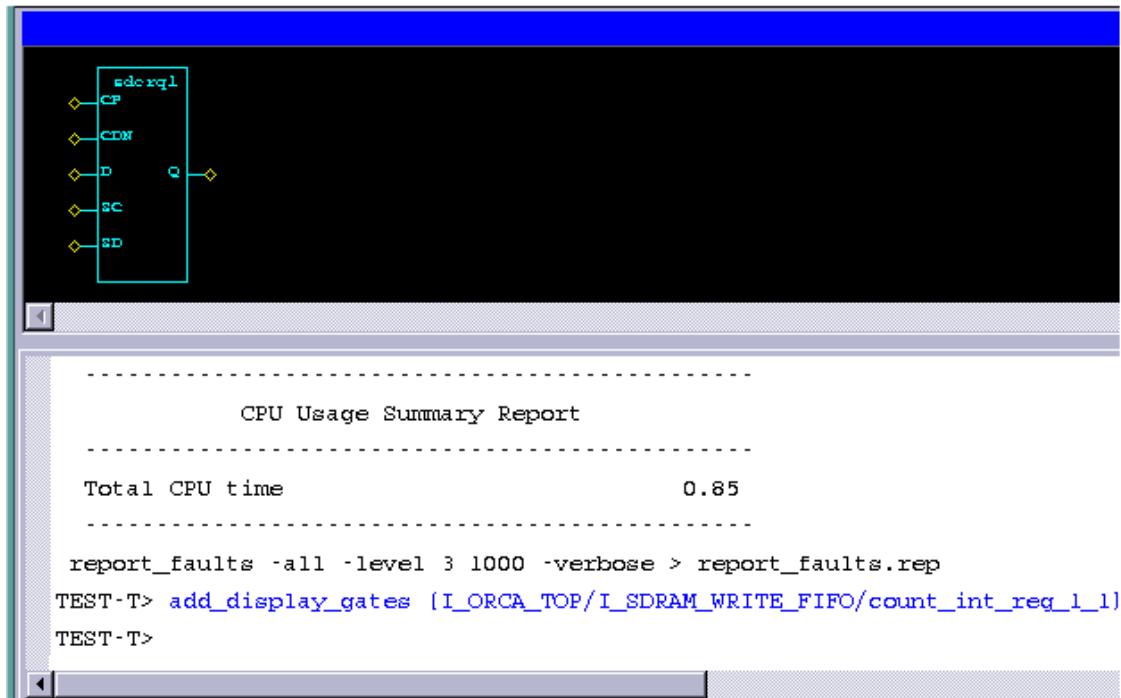
- Right-click an instance name in the Hierarchy Browser, and select Display in schematic, as shown in Figure 1.

Figure 1: Selecting an Instance Name



The selected instance will display in the GSV, as shown in Figure 2.

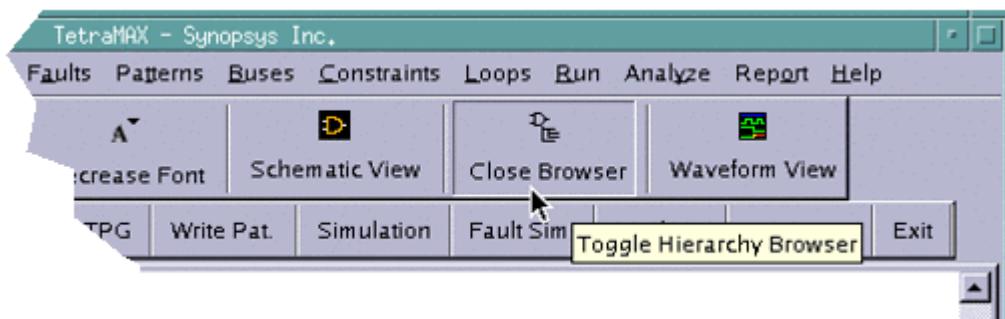
Figure 2: Display of Selected Instance in GSV



Exiting the Hierarchy Browser

To exit the Hierarchy Browser, click the Close Browser button in the TetraMAX GUI.

Figure 1: Exiting the Hierarchy Browser



See Also

[Launching the Hierarchy Browser](#)

7

Using the Simulation Waveform Viewer

You can use the TetraMAX Simulation Waveform Viewer (SWV) to debug internal, external, and imported functional pattern mismatches by displaying the failing simulation values and TetraMAX ATPG simulated values of the test_setup procedure.

The following topics describe how to use the SWV:

- [Getting Started With the SWV](#)
- [Supported Pin Data Types and Definitions](#)
- [Invoking the SWV](#)
- [Using the SWV Interface](#)

Getting Started With the SWV

Before you start using the Simulation Waveform Viewer (SWV), you should familiarize yourself with the graphical schematic viewer (GSV). For more information, see “[Using the GSV for Review and Analysis](#).”

The GSV graphically displays design information in schematic form for review and analysis. It selectively displays a portion of the design related to a test design rule violation so that you can debug a test setup, and or debug internal, external pattern mismatches. You use the GSV to find out how to correct violations and debug the design.

The SWV is intended to add a third level of dimension to DRC debugging. The following methods are currently used with DRC:

- Create or parse the STL procedure file, edit the STL procedure file, and rerun DRC
- Use the GSV to identify and resolve shift errors, and also to view test_setup
- Use the SWV when you want to view large amounts of net instance data in the GSV, such as large test_setup (item 2 above) or run_simulation data

Note that the SWV is only a viewer. Its primary purpose is to enhance what you see in the GSV.

In the GSV, you can view simulation values (or pin data values) on the nets of the design. By default, these values are 10 data bits, although this is user-configurable. Simulation values displayed in the GSV are a subset of values, followed by an ellipsis. You can change this display by changing the default setting, or by moving the data display within the GSV cone of logic. This data synchronizes with the SWV.

When the simulation string becomes more than 20 characters, the space required to display such a long string makes the GSV display impractical. In the SWV, the simulation strings do not need to be displayed in full, because you can look up the transition in the waveforms. When tracing between the GSV cone of logic, the SWV is dynamically updated with the data from the GSV. When you select and move your pointer, the SWV highlights the corresponding bit in the GSV. You can change the default display of simulation values using the following command:

```
set_environment viewer -max_pindata_length d
```

Supported Pin Data Types and Definitions

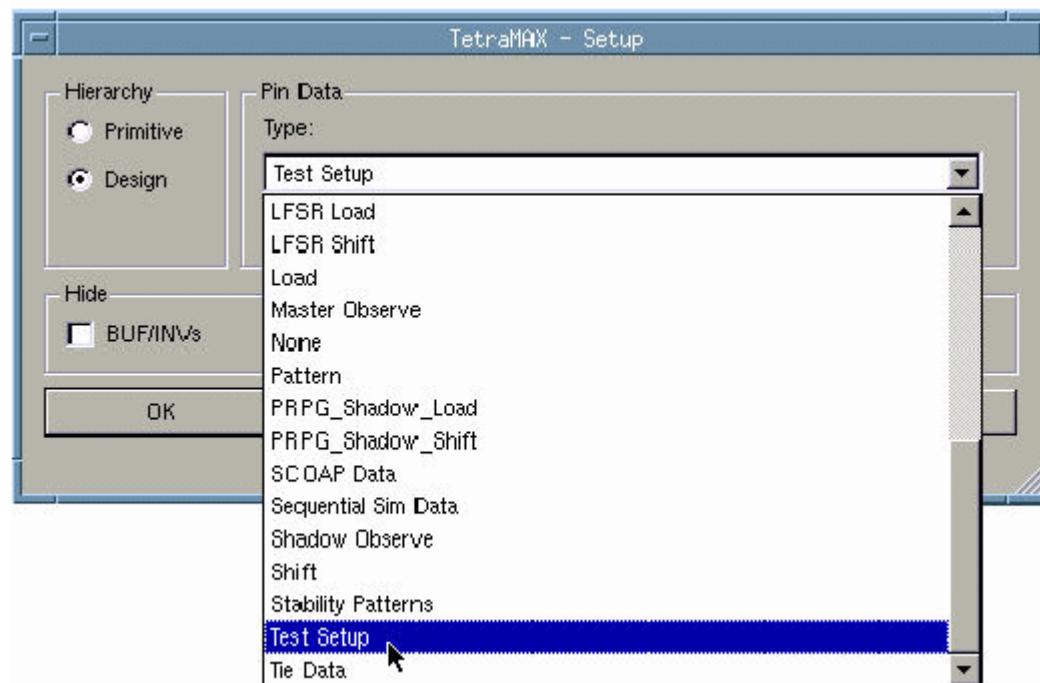
The following pin data types are supported by the Simulation Waveform Viewer (SWV):

- **Test Setup** (test_setup) — Displays simulated values for the test_setup macro displaying debugging problems in a STIL test_setup macro.
- **Debug Sim Data** (debug_sim_data) — Displays imported external simulator values used for debugging golden simulation vector mismatches
- **Sequential Sim Data** (seq_sim_data) — Displays currently stored sequential simulation data used for displaying results of sequential fault simulation (for advanced users of fault simulation)

Note that the SWV does not support all pin data types upon initialization; it supports only test_setup, debug_sim_data, and sequential_sim_data. Several other pin data types are supported after starting the SWV in one of the initial three pin data types. You can choose test_setup after SWV is opened, and then change to any pin data type, such as shift.

[Figure 1](#) shows the TetraMAX pin data type setup menu.

Figure 1: Setting the Pin Data Type



Two of the pin data types require data to be stored internally in TetraMAX ATPG.

By default, only a single logic value is shown, which corresponds to the final logic value at the exit of the test_setup macro. To show all logic values of the test_setup macro, you must change the DRC setting using the `set_drc` command, then rerun the DRC analysis as follows:

```
TEST-T> drc
DRC-T> set_drc -store_setup
DRC-T> run_drc
```

The test_setup pin data type requires the `set_drc -store` command.

Sequential simulation data typically comes from functional patterns. This type of data is stored in the external pattern buffer. When the simulation type in the Run Simulation dialog box is set to Full Sequential, you can select a range of patterns to be stored.

After the simulation is completed, you can display selected data from this range of patterns using the pin data type seq_sim_data, as shown in the following example:

```
TEST-T> set_simulation -data 85 89
TEST-T> run_simulation -sequential
```

The seq_sim_data pin data type requires the output of the `set_simulation -data` command.

See Also

[Defining the test_setup Macro](#)

Invoking the SWV

You can specify commands, select buttons, or use your right mouse button to open menus that cause TetraMAX ATPG to launch the SWV either directly from the GSV or without the GSV.

[Figure 1](#) shows the SWV menu that appears when you right-click after selecting the nets and/or gates. You can add signals, gates, and nets to the waveform using this menu. [Figure 2](#) shows the three ways to invoke the SWV from the GSV.

Figure 1: Opening the SWV Using Your Right Mouse Button

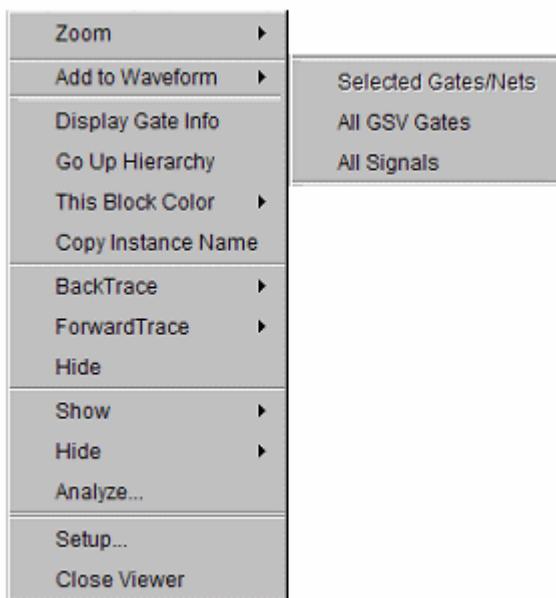
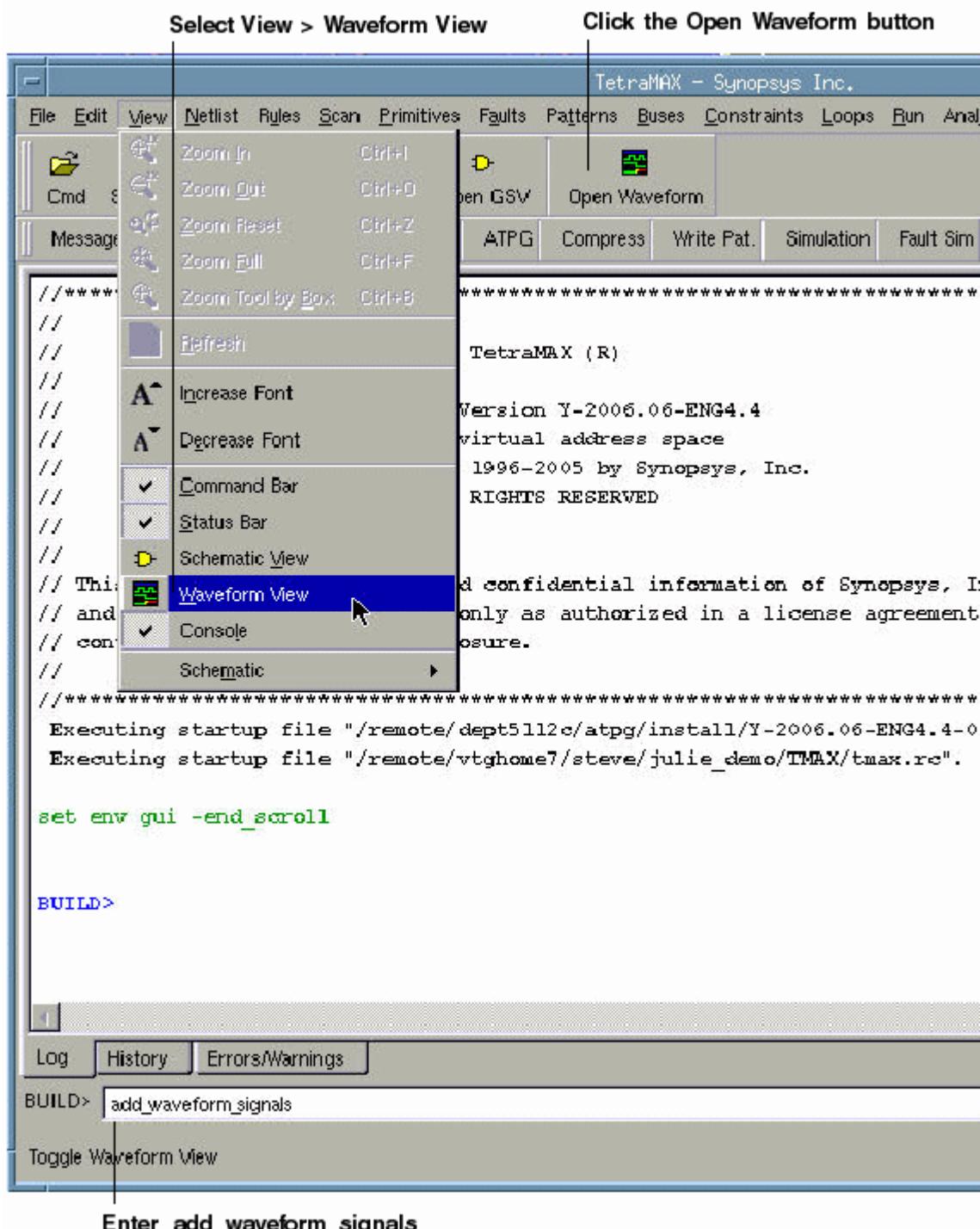


Figure 2: Three Ways to Open the SWV



Using the SWV Interface

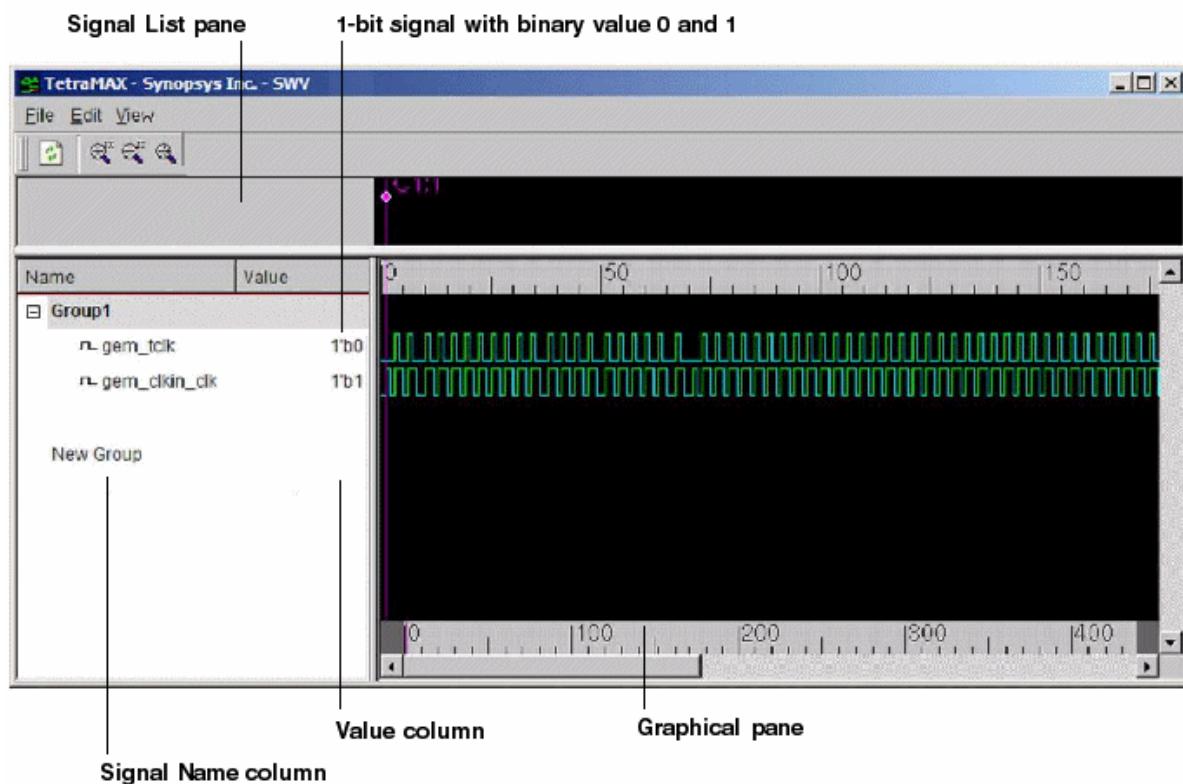
The following topics describe the basic features of the SWV interface:

- [Understanding the SWV Layout](#)
 - [Manipulating Signals](#)
 - [Identifying Signal Types in the Graphical Pane](#)
 - [Using the Time Scales](#)
 - [Using the Marker Header Area](#)
 - [Using the SWV with the GSV](#)
 - [SWV Inputs and Outputs](#)
 - [Analyzing Violations](#)
-

Understanding the SWV Layout

The layout of the SWV is shown in Figure 1.

Figure 1: SWV Layout



Note that the SWV contains a scrollable list view (the Signal List pane) and a corresponding graphical pane (the Graphical pane). The Signal List pane contains two columns: the first

column is the Signal Group tree view with the signal/bus names, and the second column is the value according to the reference cursor.

The Graphical pane consists of equivalent rows of signal in graphical drawing. Also, the reference cursor and marker can be manipulated in the Graphic pane to perform measurement between events. There are two timescales (upper and lower). The upper timescale denote the current view port time range and the lower timescale represent the global (full) time range with data.

Refreshing the View

To refresh the view (similar to the GSV), click the Refresh button or select Edit > Refresh View.

Manipulating Signals

The following sections show you how to manipulate signals:

- [Using the Signal List Pane](#)
- [Adding Signals](#)
- [Deleting Signals](#)
- [Inserting Signals](#)

Using the Signal List Pane

You can manipulate signals using the Signal List pane, which is located on the left side of the SWV. This pane is organized into the following three-level tree view:

- The root node is the group name
- The second level is the signal or bused signal name
- The third level is the individual bit of the bused signal (if applicable)

Signals are grouped together according to the target to which it is added to. New groups can be created with a signal dropped to the (default) new group tree node.

Signal groups provide a logical way to organize your signals. For example, you can keep all input signals in one group and output signals in another group. You can expand or collapse the signal list by clicking the + sign to the left of the group name. The sign changes to - when you expand it.

You can edit group names, but you cannot edit signal names. The SWV enables you only to view the design; you cannot edit or make any changes to the design.

Adding Signals

You can add any number of signals to the SWV base at the current insertion point. By default, the insertion point is to create a new signal group. After a signal is added, the insertion point is advanced to the most recently visited group.

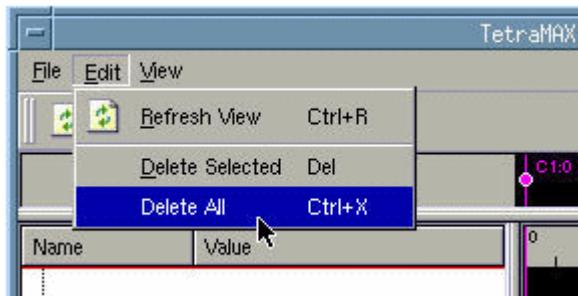
To add a signal, middle-click the signal from the waveform to select it. The red insertion line appears around the signal. Then drag it to the required group.

To add a range of signals, press Shift and click the signals to select the range. A red rubber band box appears around the range. Then drag the box into a group.

Deleting Signals

To delete a signal, or multiple signals, and groups, select the signal (s) to be deleted, and press the Delete button or choose Edit > Delete Selected. To delete multiple signals or groups, choose Edit > Delete All. Figure 1 shows the Edit menu.

Figure 1: Selecting Delete All in the Edit Menu



Inserting Signals

An insertion point is denoted by a red line. There might be times when you need to copy or duplicate a signal (shift + left-click) and move it to other groups. To do this, you can drag the insertion point into the required group.

The target of the insertion point can be specified to be a “New Group” or any group that already exists.

When an insertion point is applied to a group, the signal is added to the bottom of the list. When the insertion point is at a particular position, the signal is added below the position of the insertion point in the signal list view. If the insertion point is in the new group item view, it creates a new group. If the insertion point is in the list view, it just adds the signal to the group.

[Figure 2](#) shows an empty waveform table, and [Figure 3](#) illustrates signal insertion.

Figure 2: Empty Waveform Table

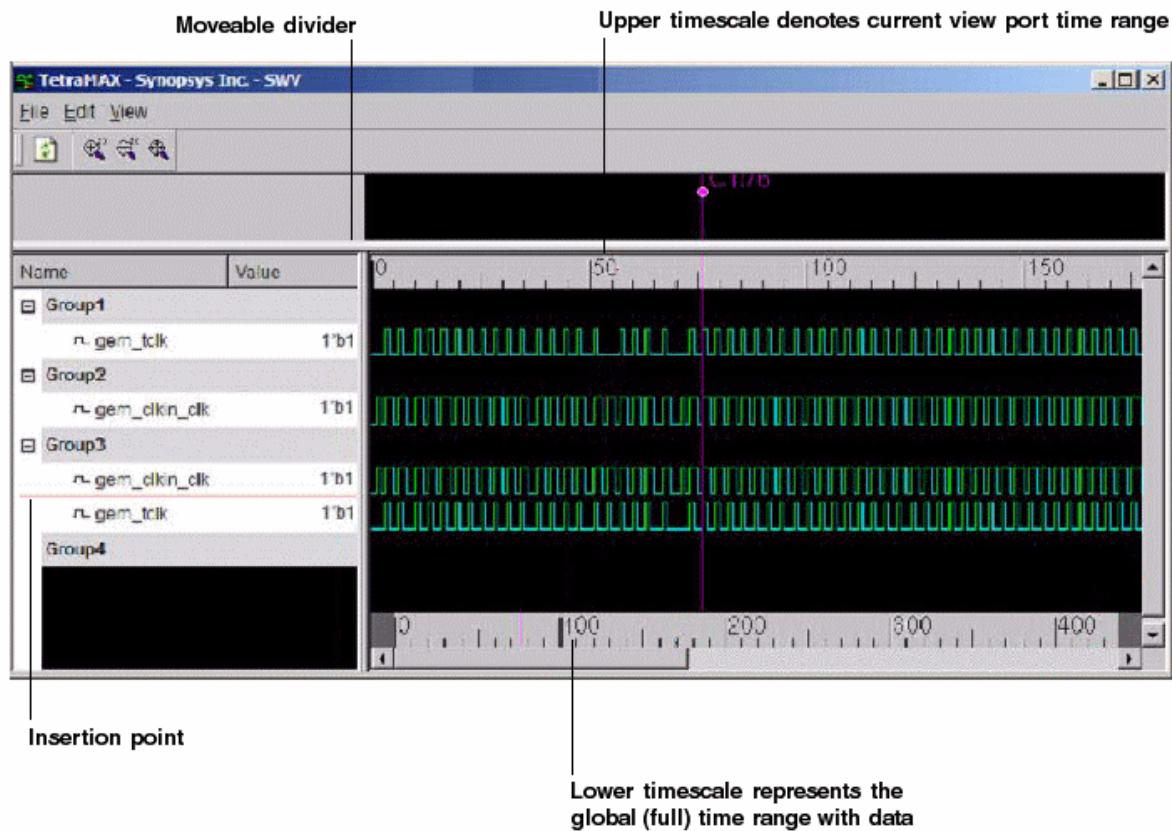
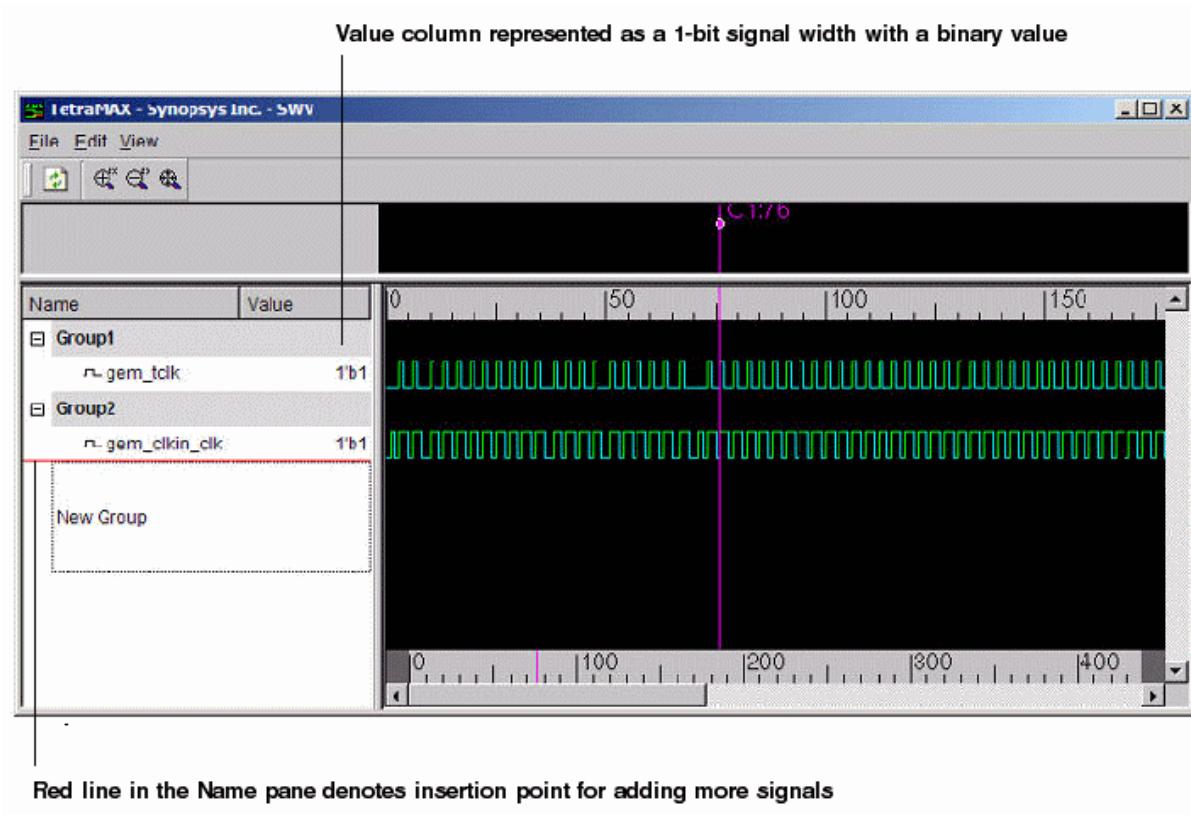


Figure 3: Inserting Signals

Identifying Signal Types in the Graphical Pane

Most signals contain events, and each event change is represented by a transition in the drawing. The viewer signals are classified into scalar type. A scalar signal carries a single bit transition between the values 0, 1, Z, X. A signal band is divided into vertical subsections to draw the values. A line drawn at the bottom of a signal band refers to event 0, while a line drawn on the top of the band refers to event 1. A Z value is drawn in the middle of the band and a filled band denotes an unknown X value.

When a vector contains an X value, it is drawn in the red event (default) color. When the vector contains some Z value, it is drawn in yellow. When all values of the transition vector are unknown, a filled red rectangle is used, and if all are Z, a horizontal yellow line is drawn in the middle of the signal band.

Using the Time Scales

The SWV displays two types of time scales:

- **Upper Time Scale**

This area displays the current viewing time range in x10ps. You can drag markers or cursors visible in the upper time scale to other locations in the view. In addition, you can perform zoom operations in the upper time scale area by clicking your left mouse button

and horizontally dragging to specify a horizontal zoom area. When you release the left mouse button, the current view refreshes with the zoomed in view in the current wave list. You won't need to further adjust the vertical alignment. When in full zoom view, the upper time scale will display the same value and range as the lower time scale.

- **Lower Time Scale**

This area shows the full time range the data occupies. You can control zoom operation using your left mouse button, which causes an adjustment in the current view time range. The width of the scroll thumb in the horizontal scroll bar shows the approximate view area in proportion to the full data time range. Reference and marker cursors are shown in the lower timescale for easy identification of marked location and to maintain the context for navigation.

Using the Marker Header Area

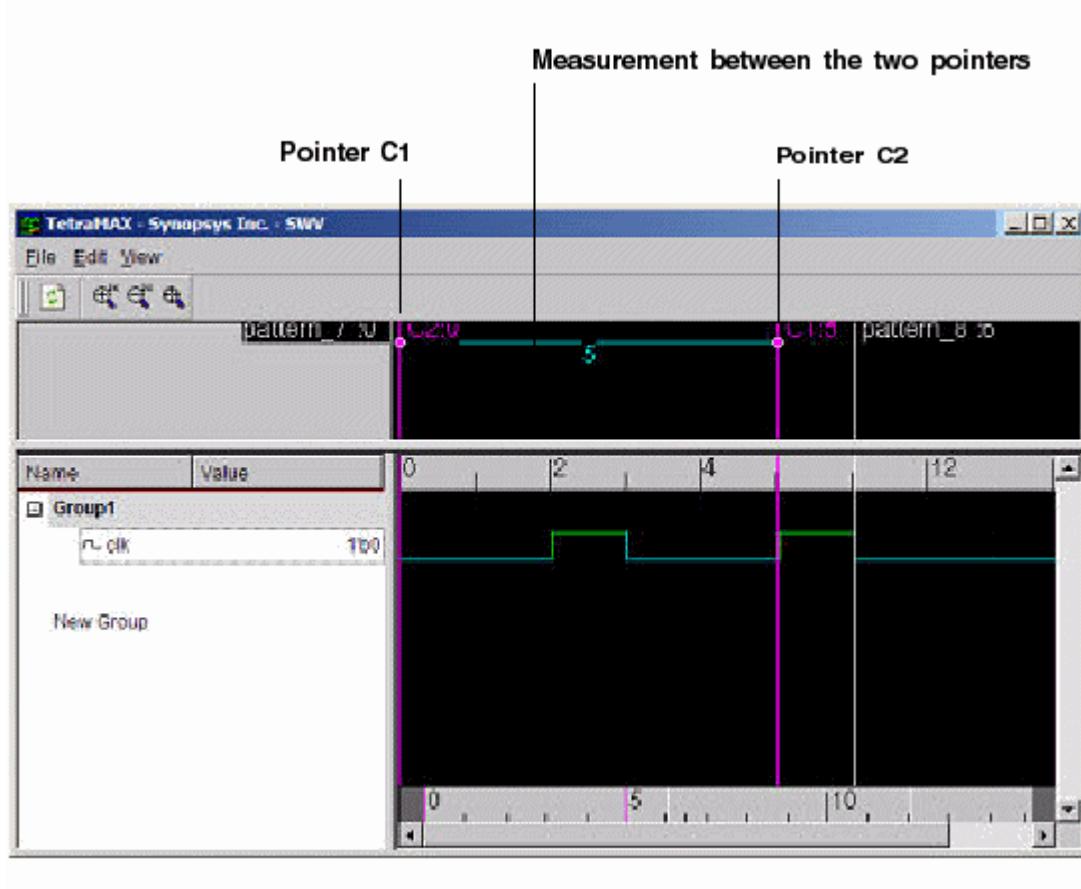
The SWV provides two reference pointers: C1 and C2. These pointers are drawn in magenta, whereas other marker cursors are in white. A marker identifier (a circle) in the marker header area is used for marker selection by the pointer.

The graphical pane shows a graphical representation of equivalent rows of signals. You can manipulate the reference cursors and markers in the graphical pane to measure between events (as shown in Figure 1.)

The following sections show you how to use the marker header area:

- [Adding and Deleting Pointers](#)
- [Moving a Pointer](#)
- [Measuring Between Two Pointers](#)

Figure 1: Reference Pointers



Adding and Deleting Pointers

To add the default reference pointer C1 or C2, you can drag the C1 pointer to the clicked location, or you can use the middle mouse button to drag the C2 reference pointer.

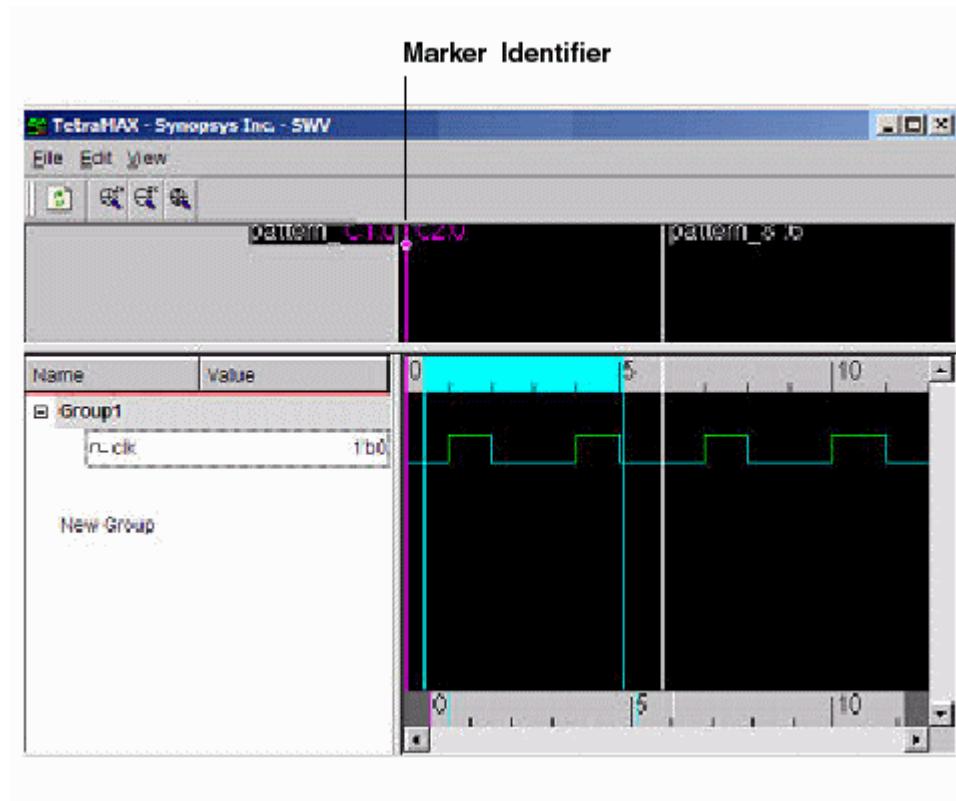
You can delete all markers by first selecting the markers, and then choosing the Delete Selected command (or the Delete This Marker command if you selected only one marker).

Moving a Marker Pointer

There are two methods you can use to move a marker pointer:

- Drag the marker identifier (a circle) in the marker header area to the new location. This method is limited to relocating the marker identifier to a region in the current viewable time range (See Figure 2).
- Drag the left marker, then click to release it.

Figure 2: Moving a marker cursor



Measuring Between Two Pointers

As shown in [Figure 1](#), you can use any pointer as a reference point for measurement. The other pointer value will change according to the currently selected reference cursor.

Using the SWV With the GSV

The primary component of the TetraMAX GUI is the graphical schematic viewer (GSV), which displays annotated simulation values during DRC (for details see "[Using the Graphical Schematic Viewer](#)"). You can expand the GSV to ease DRC debugging. Patterns are displayed in a logic cone view: i.e., logic from each design derived by tracing back from a pair of matched points. Logic cones appear when there are DRC warnings and error messages.

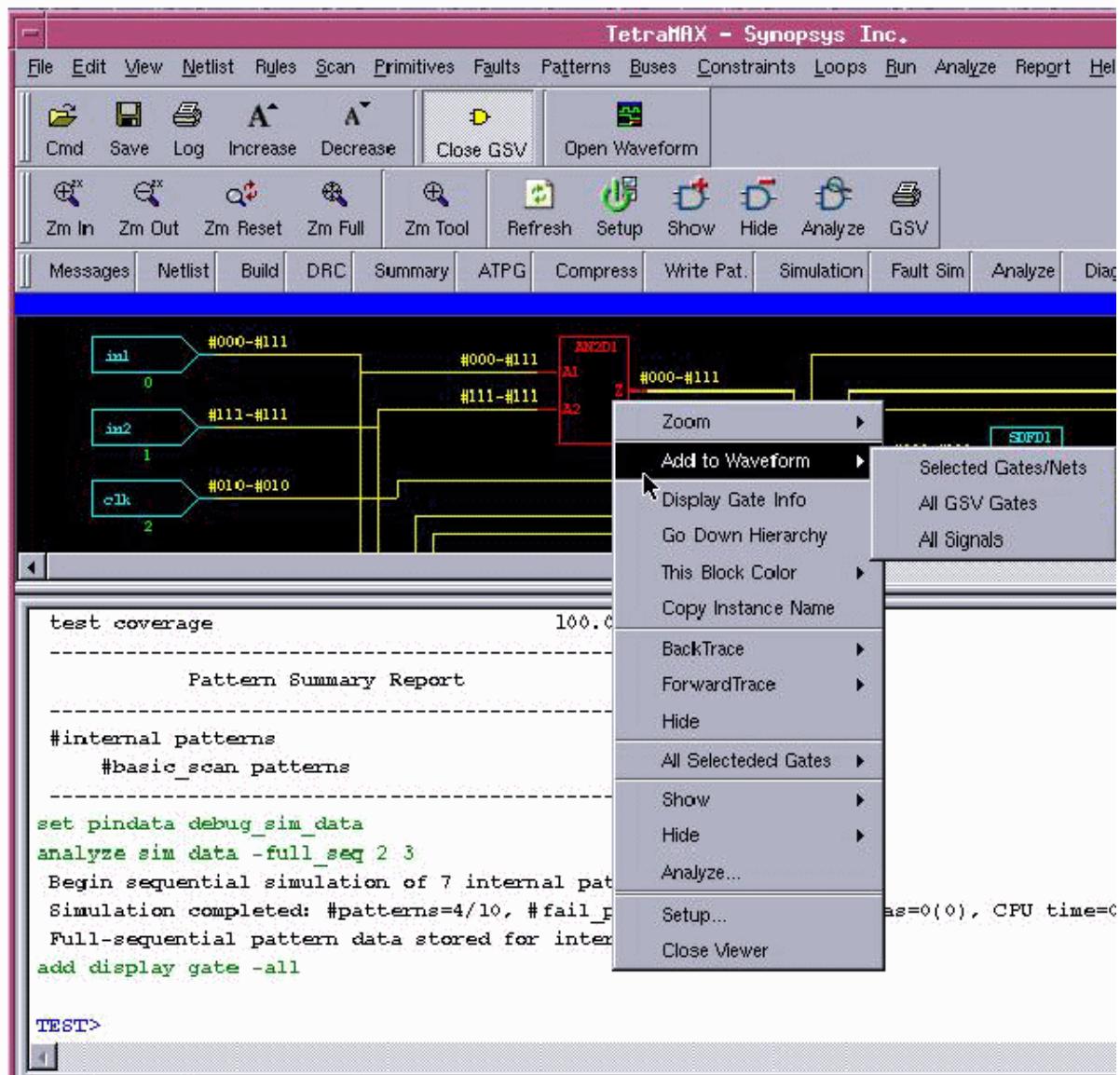
The following steps show you how to launch the SWV window from a selected logic cone view:

1. Select View > Waveform View > Setup.
2. Select "Pin Data Type" as "Test Setup."
3. Click your right mouse button and select Add to Waveform > All GSV Gates.

The simulation waveform is initialized with pattern data associated with the cone view from which it was created during DRC. There is a one-to-one correspondence between GSV and SWV when a DRC violation is used. If the GSV is closed, its corresponding waveform view is not

closed. If the SWV is closed, its corresponding GSV is not closed, and the pattern annotations on it are not cleared.

Figure 1: Using the SWV With the GSV

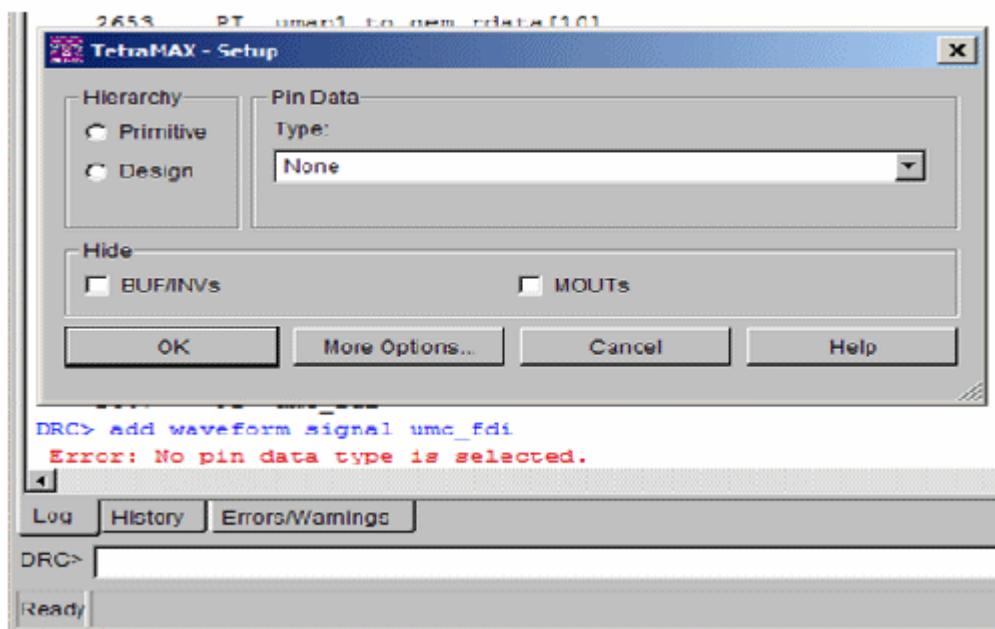


The data values displayed are generated either by DRC or by ATPG. Data values generated by DRC correspond to the simulation values used by DRC in simulating the STIL test_setup protocol to check conformance to the test protocol. Data values generated by ATPG are the actual logic values resulting from a specific ATPG pattern.

When you analyze a rule violation or a fault, TetraMAX ATPG automatically selects and displays the appropriate type of pin data in the GSV. You can also manually select the type of pin data to be displayed by using the SETUP button in the GSV toolbar, or you can use the `set_pindata` command on the command line.

The SWV can use only the pin data types listed in the [Supported Pin Data Types and Definitions](#) section. Figure 2 shows an error caused when you do not select a valid pin data type that is supported by the SWV.

Figure 2: Example of Selecting an Invalid Pin Data Type



Using the SWV Without the GSV

You might need to launch the SWV without the GSV when you have failing external patterns (read externally into TetraMAX) and you want to see the patterns for an overall evaluation of how TetraMAX interprets them. You can view the values of gates and nodes of a design for a particular pattern, or you can just view a waveform if you are already familiar with the circuit nets and nodes and you are running iterative loops in TetraMAX.

The following examples show some sample flows using the SWV. Enter these commands in a command file.

Example Flow

```
set_pindata -test_setup # test_setup is one of the many pin_data_
types
add_display_gates -all # this invokes the GSV containing the gates
of
interest.
set_pindata -test_setup # test_setup is one of the many
# pin_data_types required for SWV
add_waveform_signals < > # this invokes the SWV containing the
# waveforms for the gates of interest. The user might know the
# gates from a previous run in the GSV.
```

Example 2

```
set_simulation -data {85 89} # specify the values to store by
# patterns start/end run_simulation

run_simulation -sequential # execute a sequential simulation

set_pindata -seq_sim_data # required for SWV

add_waveform_signals <> # this invokes the SWV containing the
# waveforms for the I/Os of the patterns 85 through 89
```

Example 3

```
set_patterns -external patterns.stil
analyze_simulation_data pats1.vcd -fast 1
add_display_gates <>
set_pindata -debug_sim_data # should be the default setting
```

SWV Inputs and Outputs

The SWV has two input flows:

- The streaming pin_pathname | gate_id from the GSV to the SWV
- Streaming the externally read pattern data to the SWV displaying all I/Os

The output includes messages, warnings, and errors.

Analyzing Violations

The various TetraMAX ATPG error messages related to the SWV are described as follows:

Error: No pin data type is selected

You cannot select any nets or gates because the pin_data types require data to be stored internally to TetraMAX ATPG using the set_drc -store_setup or the set_simulation -data command. See the [Supported Pin Data Types and Definitions](#) section.

Error: Invalid argument "TOP_template_DW_tap_inst/U34/QN". <M1>

This message means that a gate was selected and added to the SWV but the QN pin is not valid or not used due to no net attached.

TOP_template_DW_tap_inst/U10_1/CP (Gate 41) is already in waveform list as TOP_template_DW_tap_inst/U34/CP.

This message appears when you select two gates that have the same clock, and add them to the SWV. The GSV picks one name and displays a message that the other pin has the same name.

/U1_out (Gate 5) is already in waveform list as U1/Z

This message appears when you select two gates that have the same clock and add them to the SWV. The GSV picks one name and displays a message that the other pin has the same name.

8

Using Tcl With TetraMAX

The following sections describe how to use the TetraMAX Tcl command interface:

- [Converting TetraMAX Command Files to Tcl](#)
- [Converting a Collection to a List](#)
- [Tcl Syntax and TetraMAX Commands](#)
- [Redirecting Output](#)
- [Using Command Aliases](#)
- [Interrupting Commands](#)
- [Using Command Files](#)

For a general guide on how to use Tcl with Synopsys tools, see *Using Tcl With Synopsys Tools*, available through SolvNet at the following URL:

https://solvnet.synopsys.com/dow_retrieve/latest/tclug/tclug.html

Note: In Tcl Mode, it is possible to use Tcl API commands to access, and then manipulate TetraMAX data. For a complete description, see “An Introduction to the TetraMAX Tcl API” in TetraMAX Online Help.

Converting TetraMAX Command Files to Tcl Mode

You can use the `native2tcl.pl` translation script to convert existing native mode TetraMAX command files to Tcl mode TetraMAX command files. This script is in the installation tree at the following location:

```
$SYNOPSYS/auxx/syn/tmax/native2tcl.pl
```

Two database files are provided with the `tmax_cmd.perl` script: `tmax_cmd.grm` and `tmax_cmd.db`.

Usage:

```
native2tcl.pl [-t ext] [- | -r dir]
```

Argument	Description
<code>[-t ext]</code>	Identifies the file extension to assign the converted files; for example, <code>TCL</code> .
<code>[- -r dir]</code>	Accepts input from <code>STDIN</code> or from the specified directory path.

For example, assuming that the native mode script to be converted is located under `/user/TMAX`, the command-line entry would appear as follows:

```
native2tcl.pl -t .TCL -r /user/TMAX
```

Converting a Collection to a List in Tcl Mode

TetraMAX Tcl API netlist query commands, such as `get_clocks` and `get_ports`, return a collection of design objects, but not a Tcl list of named objects. You can use the `get_object_name` procedure to convert a collection to a Tcl list. For example, you can convert a collection of ports to a list of port names.

You can define the `get_object_name` procedure using the following command:

```
source [getenv SYNOPSYS]/auxx/syn/tmax/get_object_name.tcl
```

After the `get_object_name` procedure is sourced within the Tcl environment, it is available for use with various TetraMAX collections. An example is as follows:

```
TEST-T> set coll [get_ports test_si*]
{test_si1 test_si2 test_si3 test_si4 test_si5 test_si6 test_si7}

TEST-T> echo $coll
_se12

TEST-T> set tcllist [get_object_name $coll]
test_si1 test_si2 test_si3 test_si4 test_si5 test_si6 test_si7
```

Tcl Syntax and TetraMAX Commands

The TetraMAX user interface is based on Tcl version 8.4. Using Tcl, you can extend the TetraMAX command language by writing reusable procedures.

The Tcl language has a straightforward syntax. Every Tcl script is viewed as a series of commands, separated by a new-line character or semicolon. Each command consists of a command name and a series of arguments.

There are two types of TetraMAX commands:

- Application commands
- Built-in commands

Each type is described in the following sections. Other aspects of Tcl version 8.4 are also described.

If you need more information about the Tcl language, consult books on the subject in the engineering section of your local bookstore or library.

The following sections describe Tcl syntax and TetraMAX Commands:

- [Specifying Lists in Tcl Mode](#)
- [Abbreviating Commands and Options in Tcl Mode](#)
- [Using Tcl Special Characters](#)
- [Using the Result of a Tcl Command](#)
- [Using Built-In Tcl Commands](#)
- [TetraMAX Extensions and Restrictions in Tcl Mode](#)

Specifying Lists in Tcl Mode

In Tcl mode, you can specify lists in commands within curly braces ({}), or within brackets ([]) if preceded by the `list` keyword.

In the following example, curly braces are used in the `add_pi_constraints` command to specify a list of ports:

```
DRC-T> add_pi_constraints 1 {TEST_MODE TICK CLK}
DRC-T> report_pi_constraints
port_name constrain_value
-----
/TEST_MODE 1
/TICK 1
/CLK 1
```

Alternatively, you can specify a list of ports in the `add_pi_constraints` command using the keyword `list` and brackets:

```
DRC-T> add_pi_constraints 1 [list TEST_MODE TICK CLK]
DRC-T> report_pi_constraints
port_name constrain_value
```

```
-----  
/TEST_MODE 1  
/TICK 1  
/CLK 1
```

Note: In Tcl mode, a list format is required when multiple arguments follow an option. For example:

```
set_build -instance_modify {specbuffer TIEX}
```

Tcl Mode and Backslashes

In Tcl mode, a backslash character (\) specified at the end of a line represents a line continuation. Any single backslash specified in the middle of a word escapes the character following it. The following examples show how to overcome this situation when you want to specify a backslash within a Tcl list:

Use a double-backslash, for example:

```
add_clocks 0 {\\"A[0] \\"B[0]}
```

Use two levels of curly braces, for example:

```
add_clocks 0 {{\A[0]} {\B[0]}}
```

As an alternative, you can remove backslashes entirely. In this case, TetraMAX commands automatically match specified identifiers that have no backslashes to identifiers in the database that have backslashes.

The following examples show various methods for specifying escaped names for a list argument:

```
add_faults {{\abccdef/hij/U1/A}}  
add_faults {\\"abccdef/hij/U1/A}  
add_faults {abccdef/hij/U1/A}  
add_faults [list {\abccdef/hij/U1/A} ]  
add_faults [list \\abccdef/hij/U1/A ]  
add_faults [list abccdef/hij/U1/A ]
```

Using Positional Arguments

Positional arguments must be specified within a Tcl list using curly braces. For example:

```
run_simulation -pin { ucore/freg/u540 01 }
```

However, if multiple specifications of the same argument are required, you must use a separate set of lists, as shown in the following example:

```
run_simulation -pin { ucore/freg/u540 0 } -pin { ucore/alu/u27 1 }
```

Abbreviating Commands and Options in Tcl Mode

Application commands are specific to TetraMAX ATPG. You can abbreviate application command names and options to the shortest unambiguous (unique) string. For example, you can abbreviate the `add_pi_constraints` command to `add_pi_c` or the `report_faults` command option `-collapsed` to `-co`. Conversely, you cannot abbreviate most built-in commands.

Command abbreviation is meant as an interactive convenience. You should not use command or option abbreviations in script files, however, because script files are then susceptible to command changes in subsequent versions of the application. Such changes can make abbreviations ambiguous.

The variable `sh_command_abbrev_mode` determines where and whether command abbreviation is enabled. Although the default is Anywhere, in the site setup file for the application, you can set this variable to Command-Line-Only. To disable abbreviation, set `sh_command_abbrev_mode` to None.

If you enter an ambiguous command, TetraMAX ATPG attempts to help you find the correct command.

For example, the following command is ambiguous:

```
> report_scan_c
Error: ambiguous command 'report_scan_c' matched 2 commands:
(report_scan_cells, report_scan_chains) (CMD-006).
```

TetraMAX ATPG lists up to three of the ambiguous commands in its error message. To list all the commands that match the ambiguous abbreviation, use the help function with a wildcard pattern. For example,

```
> help report_scan_c_*
report_scan_cells # Reports scan cell information for selected
scan cells
report_scan_chains # Reports scan chain information.
```

Using Tcl Special Characters

The characters listed in Table 1 have special meaning for Tcl in certain contexts.

Table 1: Special Characters

Character	Description
\$	Dereferences a variable.
()	Used for grouping expressions.
[]	Denotes a nested command.
\	Used for escape quoting.
""	Denotes weak quoting. Nested commands and variable substitutions still occur.
{ }	Denotes rigid quoting. There are no substitutions.
;	Ends a command.
#	Begins a comment.

Using the Result of a Tcl Command

TetraMAX commands return a result, which is interpreted by other commands as strings, Boolean values, integers, and so forth. With nested commands, the result can be used as

- A conditional statement in a control structure
- An argument to a procedure
- A value to which a variable is set

The following example uses a result:

```
if {[expr $a + 11] <= $b} {  
    echo "Done"  
    return $b  
}
```

Using Built-In Tcl Commands

Most built-in commands are intrinsic to Tcl. Their arguments do not necessarily conform to the TetraMAX argument syntax. For example, many Tcl commands have options that do not begin with a dash, but do have a value argument.

For example, the Tcl string command has a compare option that you use as follows:

```
string compare string1 string2
```

A log file of the TetraMAX session can be created using the `set_messages -log <file>` command, as with native mode. However, some Tcl built-in commands might not be able to write to the log file. For example, the `puts` command cannot write to the TetraMAX log file; use the `echo` command instead.

TetraMAX Extensions and Restrictions in Tcl Mode

Generally, TetraMAX ATPG implements all the Tcl built-in commands. However, TetraMAX ATPG adds semantics to some Tcl built-in commands and imposes restrictions on some elements of the language. The differences are as follows:

- The Tcl `rename` command is limited to procedures you have created.
- The Tcl `load` command is not supported.
- You cannot create a command called `unknown`.
- The auto exec feature found in tclsh is not supported. However, autoload is supported.
- The Tcl `source` command has additional options: `-echo` and `-verbose`, which are non-standard to Tcl.
- The `history` command has additional options, `-h` and `-r`, nonstandard to Tcl, and the form `history <n>`. For example, `history 5` lists the last five commands.
- The TetraMAX command processor processes words that look like bus (array) notation (words that have square brackets, such as `a[0]`), so that Tcl does not try to execute the

index as a nested command. Without this processing, you would need to rigidly quote such array references, as in `{a[0]}`.

- Always use braces (`{ }`) around all control structures and procedure argument lists. For example, quote the `if` condition as follows:

```
if { ! ($a > 2) } {
echo "hello world"
}
```

Redirecting Output in Tcl Mode

You can direct the output of a command, procedure, or a script to a specified file using the `redirect` command or by using the traditional UNIX redirection operators (`>` and `>>`)

The UNIX style redirection operators cannot be used with built-in commands. You must use the `redirect` command when using built-in commands.

You can use either of the following two commands to redirect command output to a file:

```
redirect temp.out {report_nets n56}
report_nets n56 > temp.out
```

You can use either of the following two commands to append command output to a file:

```
redirect -append temp.out {report_nets n56}
report_nets n56 >> temp.out
```

Note: The Tcl built-in command `puts` does not respond to redirection of any kind. Instead, use the TetraMAX command `echo`, which responds to redirection.

The following sections describe in detail how to redirect output:

- [Using the redirect Command in Tcl Mode](#)
 - [Getting the Result of Redirected Tcl Commands](#)
 - [Using Redirection Operators in Tcl Mode](#)
-

Using the `redirect` Command in Tcl Mode

In an interactive session, the result of a redirected command that does not generate a Tcl error is an empty string, as shown in the following example:

```
> redirect -append temp.out { history -h }
> set value [redirect blk.out {plus 12 34}]
> echo "Value is <$value>"
Value is <>
```

Screen output from a redirected command occurs only when there is an error, as shown in the following example:

```
> redirect t.out { report_commands -history 5.0 }
Error: Errors detected during redirect
Use error_info for more info. (CMD-013)
```

This command had a syntax error because 5.0 is not an integer. The error is in the redirect file.

```
> exec cat t.out
Error: value '5.0' for option '-history' not of type
'integer'
(CMD-009)
```

The `redirect` command is more flexible than traditional UNIX redirection operators. The UNIX style redirect operators `>` and `>>` are not part of Tcl and cannot be used with built-in commands. You must use the `redirect` command with built-in commands.

For example, you can redirect `expr $a > 0` only with the following command:

```
redirect file {expr $a > 0}
```

With `redirect` you can redirect multiple commands or an entire script. As a simple example, you can redirect multiple echo commands:

```
redirect e.out {
    echo -n "Hello"
    echo "world"
}
```

Getting the Result of Redirected Tcl Commands

Although the result of a successful redirect command is an empty string, you can get and use the result of the command you redirected. You do this by constructing a set command in which you set a variable to the result of your command, and then redirecting the set command. The variable holds the result of your command. You can then use that variable in a conditional expression.

An example is as follows:

```
redirect p.out {
    set rnet [catch {read_netlist h4c.lib}]
}
if {$rnet == 1} {
    echo "read_netlist failed! Returning..."
    return
}
```

Using Redirection Operators in Tcl Mode

Because Tcl is a command-driven language, traditional operators usually have no special meaning unless a particular command (such as `expr`) imposes some meaning. TetraMAX commands respond to `>` and `>>` but, unlike UNIX, TetraMAX ATPG treats the `>` and `>>` as arguments to the command. Therefore, you must use white space to separate these arguments from the command and the redirected file name, as shown in the following example:

```
echo $spec_variable >> file.out; # Right
echo $spec_variable>>file.out; # Wrong!
```

Keep in mind that the result of a command that does not generate a Tcl error is an empty string. To use the result of commands you are redirecting, you must use the `redirect` command.

The UNIX style redirect operators > and >> are not part of Tcl and cannot be used with built-in commands. You must use the redirect command with built-in commands.

Using Command Aliases in Tcl Mode

You can use aliases to create short forms for the commands you commonly use. For example, the following command duplicates the function of the dc_shell include command when using TetraMAX ATPG:

```
> alias include "source -echo -verbose"
```

After creating the alias in the previous example, you can use it by entering the following command:

```
> include commands.cmd
```

When you use aliases, keep the following points in mind:

- TetraMAX ATPG recognizes an alias only when it is the first word of a command.
- An alias definition takes effect immediately, but only lasts until you exit the TetraMAX session.
- You cannot use an existing command name as an alias name; however, aliases can refer to other aliases.
- Aliases cannot be syntax checked. They look like undefined procedures.

Interrupting Tcl Commands

If you enter the wrong options for a command or enter the wrong command, you can usually interrupt command processing by pressing Control-c.

The time the command takes to respond to an interrupt (to stop what it is doing and return to the prompt) depends on the size of the design and the function of the command being interrupted.

Some commands might take awhile before responding to an interrupt request, but TetraMAX commands will eventually respond to the interruption.

If TetraMAX ATPG is processing a command file (see “[Using Command Files](#)”), and you interrupt one of the file’s commands, script processing is interrupted and TetraMAX ATPG does not process any more commands in the file.

If you press Control-c three times before a command responds to your interrupt, TetraMAX ATPG is interrupted and exits with the following message:

```
Information: Process terminated by interrupt.
```

There are a few exceptions to this behavior, which are documented with the applicable commands.

Using Command Files in Tcl Mode

You can use the source command to execute scripts in TetraMAX ATPG. A script file, also called a command file, is a sequence of commands in a text file.

The syntax is as follows:

```
> source [-echo] [-verbose] cmd_file_name
```

By default, the source command executes the specified command file without showing the commands or the system response to the commands. The -echo option causes each command in the file to be displayed as it is executed. The -verbose option causes the system response to each command to be displayed.

Within a command file you can execute any TetraMAX command. The file can be simple ASCII or gzip compressed.

The following sections describe how to use command files:

- [Adding Comments](#)
 - [Controlling Command Processing When Errors Occur](#)
 - [Using a Setup Command File](#)
-

Adding Comments

You can add block comments to command files by beginning comment lines with the pound sign (#).

Add inline comments using a semicolon to end the command, followed by the pound sign to begin the comment, as shown in the following example:

```
#  
# Set the new string  
#  
set newstr "New"; # This is a comment.
```

Controlling Command Processing When Errors Occur

By default, when a syntax or semantic error occurs while executing a command in a command file, TetraMAX ATPG discontinues processing the file. There are two variables you can use to change the default behavior: `sh_continue_on_error` and `sh_script_stop_severity`.

To force TetraMAX ATPG to continue processing the command file no matter what, set `sh_continue_on_error` to true. This is usually not recommended, because the remainder of the file might not perform as expected if a command fails due to syntax or semantic errors (for example, an invalid option).

Note: The `sh_script_stop_severity` variable has no effect if the `sh_continue_on_error` variable is set to true.

To get TetraMAX ATPG to stop the command file when certain kinds of messages are issued, use the `sh_script_stop_severity` variable. This is set to `none` by default. Set it to `E` to get the file to stop on any message with error severity. Set it to `W` to get the file to stop on any message with warning severity.

Using a Setup Command File

You can use a command file as a setup file so that TetraMAX ATPG will automatically execute it at startup. The default setup file is located in the following directory:

```
$SYNOPSYS_TMAX/admin/setup/tmaxtcl.rc
```

To use a setup command file in the Tcl interface, you must name it either `.tmaxtclrc` or `tmaxtcl.rc`, and place it in the directory where TetraMAX ATPG was started or in your home directory.

9

Design Netlists and Library Models

The "[Preparing a Netlist](#)," "[Reading a Netlist](#)," and "[Reading Library Models](#)" sections provide specific information on how to specify netlists and library models. The following sections provide additional information on reading and processing design netlists and library models:

- [Netlist Format Requirements](#)
- [About Reading a Netlist](#)
- [Using Wildcards to Read Netlists](#)
- [About Reading Library Models](#)
- [Controlling Case-Sensitivity](#)
- [Processes That Occur When Building the ATPG Model](#)
- [Flattening Optimization for Hierarchical Designs](#)
- [Identifying Missing Modules](#)
- [Removing Unused Logic](#)
- [Using Black Box and Empty Box Models](#)
- [Handling Duplicate Module Definitions](#)
- [Memory Modeling](#)
- [Creating Custom ATPG Models](#)
- [Condensing ATPG Libraries](#)

Netlist Format Requirements

TetraMAX ATPG can read netlists in Electronic Design Interchange Format (EDIF), Verilog, and VHDL formats. Some minimal preprocessing might be necessary to make the netlist compatible with TetraMAX ATPG.

The following sections describe the netlist requirements for TetraMAX ATPG:

- [EDIF Netlist Requirements](#)
 - [Verilog Netlist Requirements](#)
 - [VHDL Netlist Requirements](#)
-

EDIF Netlist Requirements

To ensure EDIF netlists are compatible with TetraMAX ATPG, you must review all power and ground logic connections. The following sections describe how to handle these situations:

- [Logic 1/0 Using Global Nets](#)
- [Logic 1/0 by Special Library Cell](#)

Logic 1/0 Using Global Nets

In EDIF, a design can reference two or more global nets, which represent the tie to logic 1 or logic 0 connections. Because there is no driver for these nets, TetraMAX ATPG issues warnings, such as “floating internal net,” as it analyzes the design.

If your design uses this global net approach and you are using Synopsys tools to create your netlist, set the EDIF environment variables as shown in Example 1 before writing the EDIF netlist. The global net names used in the example are logic0 and logic1, but you can use any legal net names.

Example 1: EDIF Variable Settings for Global Logic 1/0 Nets

```
edifout_netlist_only = true
edifout_power_and_ground_representation = net
edifout_ground_net_name = "logic0"
edifout_power_net_name = "logic1"
write options
```

Logic 1/0 by Special Library Cell

The EDIF library can contain special tie_to_low and tie_to_high cells. Every logic connection to power or ground is then connected by a net to one of these cells. If your design uses this library cell approach, you must define an ATPG model for each cell to supply the proper function of TIE1 or TIE0; otherwise, the missing model definition is translated to a TIEX primitive, and the logic 1/0 connections are all tied to X instead of to the required logic value.

If you do not yet have models describing the logic functions of the special cells, you might have to add some module definitions to your library. Normally, your ASIC vendor provides these; if not, see Example 2, which shows a Verilog module description for modules called POWER and

GROUND. You can use this module description by changing the name of the module to match the library cell names referenced by your EDIF design netlist.

Example 2: ATPG Model Definition for Logic 1/0 Library Cells

```
module POWER (pin);
output pin;
_TIE1(pin);
endmodule

module GROUND (pin);
output pin;
_TIE0(pin);
endmodule
```

To provide an ATPG functional model for each EDIF cell description, place the module definition in a separate file to be referenced during the process flow when it is time to read in library definitions.

Verilog Netlist Requirements

Verilog netlist style, syntax, and instance and net naming conventions vary greatly. Use the following guidelines to ensure that your Verilog netlist is compatible with TetraMAX ATPG:

- Do not use a period (.) within the name of any net, instance, pin, port, or module without enclosing it with the standard Verilog backslash mechanism.
- Verify that your Verilog modules are structural and not behavioral, except for modules used to define ATPG RAM/ROM functions.
- Verilog is case-sensitive, although many tools ignore case and treat “specNet” and “specnet” as the same item.
- If you are using Synopsys tools to create your Verilog netlist, review the `define_name_rules` command to find options for adjusting the naming conventions used in your design.

See Also

[ATPG Modeling Primitive Summary](#)

VHDL Netlist Requirements

The following guidelines apply when using a VHDL netlist with TetraMAX ATPG:

- VHDL designs must be completely structural in nature.
- Bits and vectors must only use std_logic types. Other types, such as SIGNED, are not supported.
- Conversion functions are not supported.

About Reading a Netlist

TetraMAX ATPG automatically determines the format of the referenced netlist and reads the file in hierarchical order, starting with the library leaf cells and ending with the top-level module.

You can specify the following options when reading a netlist:

- By default, TetraMAX ATPG treats [Verilog netlists](#) as case-sensitive, and [EDIF](#) and [VHDL](#) netlists as case-insensitive. You can override the default by running the `-sensitive` or `-insensitive` option of the `read_netlist` command or by selecting Sensitive or Insensitive in the Case sensitivity drop-down menu in the Read Netlist dialog box.
- TetraMAX ATPG issues a warning if a module is defined more than one time, and uses the module definition from the last netlist read. You can identify a module as a master module and it will not be replaced if another module is encountered with the same name.
- You can define any variable function, or expression. This option is equivalent to the 'define Verilog statement.
- You can delete previously read netlists currently stored in memory.
- You can prevent TetraMAX ATPG from terminating if it encounters an error when reading multiple netlists.

See Also

[How to Read a Netlist](#)

Using Wildcards to Read Netlists

If your library cells are stored in multiple individual files, you can read them all using wildcards. TetraMAX ATPG supports the asterisk (*) to match occurrences of any character, and the question mark (?) to match any single character.

To read in all files in the directory `speclib` that have the extension `.v`, use the following `read_netlist` command:

```
BUILD-T> read_netlist speclib/*.v
```

To read in all files in `speclib`, enter the following command:

```
BUILD-T> read_netlist speclib/*
```

To read in all files that begin with `DF` and end with `.udp`, in all subdirectories in `speclib` that end in `_lib`, enter the following command:

```
BUILD-T> read_netlist speclib/*_lib/DF*.udp
```

To read in all files that begin with `DFF`, end in `.V`, and have any two characters in between, enter the following command:

```
BUILD-T> read_netlist DFF???.v
```

You can also use wildcards in the Read Netlist dialog box. Use the Browse button to select any file from the directory of interest and click OK. Then replace the file name with an asterisk.

When you use wildcards, you might find it convenient to use the following options:

- **Verbose:** Produces a message for each file rather than the default message for the sum of all files.
- **Abort on error:** Determines whether TetraMAX ATPG stops reading files when it encounters an error with an individual file.

About Reading Library Models

TetraMAX ATPG creates ATPG models based on the functional portion of Verilog simulation models. These models include user-defined primitives (UDPs), which are essential for describing particular library cells. Behavioral models are not recognized.

TetraMAX ATPG recognizes the following Verilog language attributes:

```
`define  
`ifdef  
`include  
`celldefine  
`suppress_faults  
`enable_portfaults
```

You must read in all library models referenced by your design. You can read in one model at a time, or you can read in the entire library with a single command. If your design already contains a module that has the same name as one of the library modules, when you read in the library, the library model overwrites your module.

See Also

[Reading Library Models](#)

[Reading a Netlist](#)

Controlling Case-Sensitivity

Netlist formats differ in whether or not the instance, pin, net, and module names are case-sensitive. When TetraMAX ATPG reads a netlist, it chooses case-sensitive or case-insensitive based on the type of netlist by default as follows:

- Verilog Netlists: case-sensitive
- EDIF Netlists: case-insensitive
- VHDL Netlists: case-insensitive

You can override the defaults by using the `-sensitive` or `-insensitive` option of the `read_netlist` command. For example, to read in all files ending in `.V` in directory `speclib`, using case-insensitive rules, use the following command:

```
BUILD-T> read_netlist speclib/*.* -insensitive
```

Setting Parameters for Learning

When TetraMAX ATPG builds an ATPG model, it also performs a circuit learning process to determine information useful for performing simulation and test generation.

This learning process performs the following tasks:

- Identifies feedback paths
- Orders gates and feedback networks by rank
- Identifies easiest-to-control input and easiest-to-observe fanout for all gates
- Identifies equivalence relationships between gates
- Identifies the potential functional behavior of circuit fragments
- Identifies tied value gates and fault blockages that result from tied gates
- Identifies tied gates and blockages that result from gates whose inputs come from a common or equivalent source
- Identifies equivalent DFF and DLAT devices (those with identical inputs)
- Identifies implication relationships between gates

Learned Behavior Types

During the learning process, each gate is assigned a learned behavior. The possible types of learned behavior are as follows:

Blocked - A gate whose fault effects are blocked from detection by tied circuitry.

Common Input - A gate that has a common source for two or more of its inputs.

Common Tied Input - A gate that is equivalent to a tied gate due to some logical relationship between its inputs. For example, an XOR gate with both inputs attached to the same net is equivalent to a tied-to-0; or an AND gate with a net and its inverted value as inputs will also be equivalent to a tied-to-0.

Constrained - A gate with an input constraint resulting in an output that can never achieve a 0, 1, or Z, or some combinations of these logic values.

Constrained Blocked - A gate whose fault effects are blocked from detection by constraints.

Equivalence - Two gates whose outputs are equivalent or complementary to each other at all times. For example, a NAND and an AND gate with the same input connections always have opposite values on their outputs.

Implications - Two gates whose behavior has been learned to have an implied relationship, such as "gate A at value J implies gate B at value K".

Inverted Inputs - A gate that has an inverted input function. In other words, an inverter has been merged into the input of an otherwise standard gate such as AND, NAND, OR, or NOR.

Learn BUF - A gate whose function is equivalent to a BUF, such as an AND gate with its inputs tied together.

Learn INV - A gate whose function is equivalent to an INV, such as a NAND gate with its inputs tied together.

Learn Tied Gate - A gate whose function is equivalent to a tied 0/1/Z/X gate.

Tied - Any gate learned to be always tied to 0/1/Z/X.

Weak - Any gate with a WEAK input. This is generally a BUS device.

You can view most learned data for a given gate by setting the `-verbose` option for the `set_pindata` command and running the `report_primitives` command for the selected gate.

Controlling the ATPG Learning Algorithm

You can control the ATPG learning algorithms using the `set_learning` command or the Run Build Model dialog box.

The following example uses the `set_learning` command to specify the ATPG equivalence algorithm for learning:

```
BUILD-T> set_learning -atpg_equivalence
```

To use the Run Build Model dialog box to control the learning algorithms:

1. From the command toolbar, click the Build button.
The Run Build Model dialog box appears.
2. Select or enter the appropriate options in the Set Learning section. For descriptions of these controls, see the description of the `set_learning` command in TetraMAX Help.
3. Click OK.

About Building the ATPG Model

To build the ATPG model, TetraMAX ATPG compiles a set of netlist and library models into a single in-memory image. For more information on reading netlists and library models, see "[Reading a Netlist](#)" and "[Reading a Library Model](#)".

When you build an ATPG model of a hierarchical design, TetraMAX ATPG flattens the hierarchy to make a single-level, in-memory model of the design. Several different optimization methods are used to reduce the number of gates and simplify the design. You can control many of these processes using the `set_build` command, as described in "[Controlling the Build Process](#)".

During the process of building a model, TetraMAX ATPG also performs a circuit learning process to determine information useful for performing simulation and test generation. The parameters you can set for this learning process are described in "[Setting Parameters for Learning](#)".

You can specify several parameters that control and optimize the process of building an ATPG model, including:

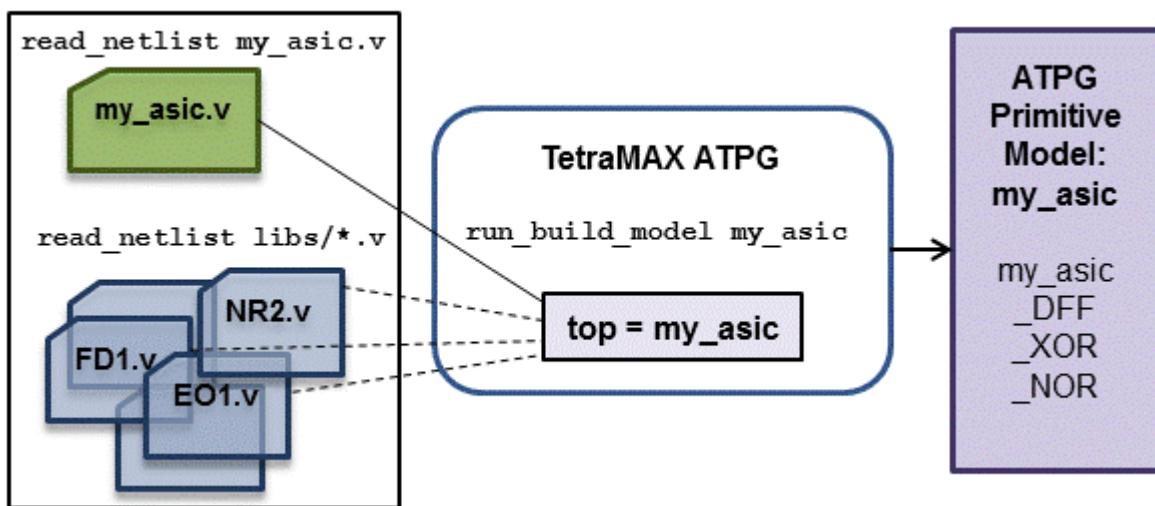
- Add a buffer gate between any latch or flip-flop gate directly connected to another latch or flip-flop gate
- Specify and remove modules designated as black boxes, empty boxes, design library cells
- Preserve pin names of models that would normally be flattened
- Add pulldown and pullup drivers and bus keepers to BUS gates
- Keep or delete unused gates
- Specify an alternate hierarchical delimiter

- Modify or replace selected instances
- Define certain input signals in the top-level module as bidirectional signals
- Limit the number of fanouts for gates
- Specify various model flattening optimization algorithms
- Specify how net connections affect the flattened ATPG model
- Specify parameters for modeling undriven bidirectional nets

For details on these options, see the description of the `set_build` command in TetraMAX Help.

[Figure 1](#) shows the process for building the ATPG model:

Figure 1 Building the ATPG Design Model



See Also

[Building the ATPG Model](#)

Processes That Occur When Building the ATPG Model

During the execution of the `run_build_model` command, the following processes occur:

- The targeted top module for build, usually the top module, is used to form an in-memory image. Each instance in the top level is replaced by the gate-level representation of that instance; this process is repeated recursively until all hierarchical instantiations have been replaced by references to ATPG simulation primitives.
- Special ATPG simulation primitives are inserted for inputs, outputs, and bidirectional ports.

- Special ATPG simulation primitives are inserted to resolve BUS and WIRE nets. Unused gates are deleted based on the last setting of the `set_build -delete_unused_gates` command.
- Each primitive is assigned a unique ID number.
- Some BUF devices are inserted at top-level ports that have direct connections to sequential devices. No fault sites are added by these buffers.
- Various design and module-level rule checks (the “B” series) are performed to determine the following:
 - Missing module definitions
 - Floating nets internal to modules
 - Module ports defined as bidirectional with no internal drivers (These could have been input ports.)
 - Module ports defined as outputs with no internal drivers (These possibly should have been inputs.)
 - Module input ports that are not connected to any gates within the module (These might be extraneous ports.)
 - Instances that have undriven input pins (These might be floating-gate inputs.)
- TIE0, TIE1, TIEZ, and TIEX primitives are inserted into the design where appropriate as a result of determining floating inputs or pins tied to a constant logic level.
- Statistics on the number of ATPG simulation primitives as well as the types of ATPG primitives are collected.

[Example 1](#) shows an example transcript of the `run_build_model` command.

Example 1: Transcript of run_build_model Command Output

```
BUILD-T> run_build_model asic_top
-----
Begin build model for topcut = asic_top ...
-----
Warning: Rule B7 (undriven module output pin) failed 178 times.
Warning: Rule B8 (unconnected module input pin) failed 923 times.
Warning: Rule B10 (unconnected module internal net) failed 32
times.
Warning: Rule B13 (undriven instance pin) failed 2 times.
End build model: #primitives=101071, CPU_time=3.00 sec,
Memory=34529279
-----
```

Flattening Optimization for Hierarchical Designs

When you build a model of a hierarchical design (using the `run_build_model` command), TetraMAX ATPG flattens the hierarchy to make a single-level, in-memory model of the design. Several different optimization methods are used to reduce the number of gates and simplify the

design. Some of these methods are always performed, while others may be enabled or disabled with the `set_build` command.

TetraMAX ATPG can perform 16 different types of optimization. Each optimization method is described, including the user commands for enabling or disabling the method, where applicable. The default configuration (either enabled or disabled) is marked with an asterisk in each such description.

1. BUF elimination

Always enabled; no user control

During the flattening process, buffers are eliminated wherever this is possible without eliminating any fault sites.

2. INV elimination

Always enabled; no user control

During the flattening process, inverters are eliminated wherever this is possible without eliminating any fault sites.

3. Switches(SW) as BUFs or BUFZs

Always enabled; no user control

During the flattening process, each SW primitive found that has its control gate held constantly on is replaced with a BUFZ device; or if the propagation of a Z value is not needed, it is replaced with a BUF device. These BUF/BUFZ device may be removed later by the BUF elimination method (#1 above). This optimization can cause fault sites to be dropped. If this happens, the dropped faults are reported as B22 violations.

4. DLATs as BUFs

Always enabled; no user control

During the flattening process, for each DLAT primitive found that has its gate/clock input held on and its set and reset lines held off so that the latch is always transparent, the DLAT is replaced with a BUF device. This optimization can cause fault sites to be dropped. If this happens, the dropped faults are reported as B22 violations.

5. DFFs as DLATs

Always enabled; no user control

During the flattening process, each DFF primitive found that has its clock permanently off, but able to use its asynchronous set or reset input, is replaced with a latch device.

6. Unused Gates

To enable: `set_build -delete_unused_gates (default)`

To disable: `set_build -nodelete_unused_gates`

An unused gate is one which has no output connections to other gates, including black box or empty box gates. When this optimization method is enabled, unused gates are removed during the flattening process. It is possible for fault sites to be dropped as these gates are removed.

7. TIE propagation

To enable: `set_build -merge_tied_gates_with_pin_loss`

To disable: `set_build -merge_notied_gates_with_pin_loss (default)`

TIE propagation optimization identifies nets and pins tied high or low and attempts to propagate this constant value through logic to reduce the number of gates. When disabled, TIE

propagation is still performed, but only where it does not cause testable fault sites to be dropped. When enabled, TIE propagation occurs even where it causes testable fault sites to be dropped. If any fault sites are dropped, they are reported in a summary message; for example:

There were 38240 primitives and 6318 faultable pins removed during model optimizations

8. Cascaded Gates

To enable: `set_build -merge cascaded_gates_with_pin_loss`

To disable: `set_build -merge noscascaded_gates_with_pin_loss (default)`

Cascaded gate optimization is performed by identifying two gates in series that can be logically merged into a single gate. An example of this is two 2-input AND gates in series, which can be replaced by a single 3-input AND gate. When disabled, cascaded gate optimization is still performed, but only where it does not cause fault sites to be dropped. When enabled, cascade optimization is performed even where it causes fault sites to be dropped. If any fault sites are dropped, they are reported as B22 violations.

9. Bus Keepers

To enable: `set_build -merge Bus_keepers (default)`

To disable: `set_build -merge NOBus_keepers`

Bus keeper recognition is performed by searching for small, constantly enabled combinational loops with a weak driver. This recognition considers paths including BUF and INV, as well as SW and TSD devices used to form bus keepers that hold only one state. After identified, these loops are replaced with BUSK ATPG primitives. When enabled, bus keeper optimization can result in the dropping of faults sites. If any fault sites are dropped, they are reported as B22 violations.

10. Feedback Paths

To enable: `set_build -merge feedback_paths (default)`

To disable: `set_build -merge nofeedback_paths`

Feedback path optimization is done by searching for combinational loops that do not perform any testable function. One of example is a loop involving a BUS with a weak driver and at least one strong, non-three-state driver. The loop through the weak driver may be removed. Another example is a three-state net where all the potential drivers come from top-level primary inputs (strong drivers) and the feedback path is again through a weak driver. Elimination of these feedback paths can cause fault sites to be dropped. If this happens, the dropped faults are reported as B22 violations.

11. MUX Recognition

To enable: `set_build -merge mux_from_gates`

To enable: `set_build -merge muxpins_from_gates (default)`

To enable: `set_build -merge muxx_from_gates`

To disable: `set_build -merge nomux_from_gates`

The MUX recognition optimization method is done by searching for discrete gates that may be combined to create MUX behavior. The most common form is two 2-input AND gates followed by an OR gate. Additional variants are also recognized, such as pass-transistor MUXes. There are three variations of this optimization method:

`Mux_from_gates` - When enabled, discrete-gate forms of MUX behavior are replaced with TetraMAX ATPG MUX primitives. During this optimization, it is possible that fault sites are dropped. If any fault sites are dropped, they are reported as B22 violations.

`Muxpins_from_gates` - When enabled, discrete-gate forms of MUXes are replaced, but only if no fault sites are dropped as a result.

`Muxx_from_gates` - When enabled, discrete-gate forms of MUXes are replaced, but only if no fault sites are dropped as a result, and only for "optimistic MUX" behavior. Gates which form the "pessimistic MUX" behavior are left unchanged.

An "optimistic MUX" produces an output equal to the data inputs when the select line is X and both inputs are identical. The "pessimistic MUX" produces an output of X when the select line is X, even when the data inputs are identical. The TetraMAX MUX primitive implements the "optimistic MUX" behavior.

12. XOR/XNOR Recognition

To enable: `set_build -merge Xor_from_gates`

To enable: `set_build -merge XORPins_from_gates (default)`

To disable: `set_build -merge NOXor_from_gates`

XOR/XNOR recognition optimization is done by searching for discrete gates that form either the XOR or XNOR function. There are two variations of this optimization:

`Xor_from_gates` - When enabled, discrete-gate forms of XOR/XNOR are replaced with TetraMAX XOR/XNOR primitives. This optimization can cause fault sites to be dropped. If this occurs, the dropped fault sites are reported as B22 violations.

`Xorpins_from_gates` - When enabled, discrete-gate forms of XOR/XNOR are replaced with TetraMAX XOR/XNOR primitives, but only where no fault sites are dropped as a result.

13. Equivalent DLAT/DFF

To enable: `set_build -merge equivalent_dlat_dff (default)`

To enable: `set_build -merge equivalent_initialized_dlat_dff`

To disable: `set_build -merge noequivalent_dlat_dff`

This optimization method identifies equivalent DLAT and DFF devices, and merges the equivalent functions into a single device. The `equivalent_initialized_dlat_dff` setting, if enabled, will assume the devices are initialized to their steady state values before determining if they can be merged into a single device. Two DLAT or two DFF devices are equivalent if they share common input connections, including all clock, set, reset, and data inputs. The outputs of the two devices may be identical or complementary to each other. This optimization method replaces one equivalent device with a BUF or INV connected to the output of the other equivalent device. During this process, fault sites might be dropped, in which case they are reported as B22 violations.

14. DLAT pairs as DFF

To enable: `set_build -merge flipflop_from_dlat (default)`

To enable: `set_build -merge flipflop_cell_from_dlat`

To disable: `set_build -merge noflipflop_from_dlat`

This optimization method finds each pair of D-latches that operate together as a D flip-flop, and replaces them with a DFF primitive. This occurs when two DLAT devices are connected serially from the Q output of one to the D input of the other, share common set, reset, and

complementary clocks from the same source. When this optimization occurs, the two DLAT devices are replaced with a DFF primitive, possibly causing fault sites to be dropped. If any fault sites are dropped, they are reported as B22 violations.

`flipflop_cell_from_dlat`- When enabled, merging master-slave latches into a flip-flop is limited to those latch pairs that are part of the same design-level cell. This option should be used by DFT Compiler and when necessary to avoid pin loss due to merging latches to flip-flops.

15. WIRE and BUS gates

To enable: `set_build -merge Wire_to_buffer (default)`

To disable: `set_build -merge NOWire_to_buffer`

This optimization identifies and optimizes WIRE and BUS gates having common-source inputs with buffer gates. Such WIRE and BUS gates are like those found common in clock and scan-enable repowering networks. The buffer gates thus created can be further removed by other optimizations. This optimization can result in faultable pin losses; however, the lost faults are untestable anyway. In many designs, this optimization results in fewer primitives in the final model, particularly fewer WIRE gates; in many cases the number of WIRE gates is reduced to 0, which also results in faster DRC. The default is `-wire_to_buffer` (optimization is enabled).

16 Tied inputs and MUX gates

To enable: `set_build -merge Global_tie_propagate (default)`

To disable: `set_build -merge NOGlobal_tie_propagate`

This optimization identifies and optimizes global tie 0/1 value propagations and replaces certain [N]AND, [N]OR and MUX gates with buffers/inverters or eliminates them completely. The analysis ensures that all faults eliminated are either undetectable-redundant (UR) or equivalent to other faults that are preserved. The memory and CPU time required by the flattening process are not measurably affected by this analysis.

This optimizations might change the reported test coverage, because:

- Eliminated UR faults could have been classified as ATPG-untestable (AU).
- Eliminated equivalent faults might change equivalence classes size and affect uncollapsed coverage.

The following optimizations are performed during analysis:

- [N]AND, [N]OR gate with controlling tied inputs (T0/T1): replaced with T0/T1 if no output fault and no faults on the tied inputs. All faults lost are classified UR.
- [N]AND, [N]OR gate with non-controlling tied inputs (T1/T0): replaced with buffer/inverter if only one input is not tied. Faults lost are either classified UR or equivalent to the corresponding output fault.
- MUX gate with tied select input: replaced with buffer/inverter from selected data input if no fault on the select input or data lines cannot have complementary values. All faults lost are classified UR.
- MUX gate with data inputs driven by common gate, with same inversion: replaced with buffer/inverter from a data input if no faults on data inputs. All faults lost are classified UR.
- MUX gate with data inputs driven by T0/T1: replaced with buffer/inverter from select line. Faults lost are either classified UR or equivalent to the corresponding output fault.

Disabling this optimization could be desirable in the following cases:

- If pins targeted by an `add_net_connections` command or by reading in external fault files are eliminated by the new analysis.
- If design-level viewing is limited because instances of interest have been "flattened-down" (these are tracked by B22 violations).

The current value of all optimization settings can be reviewed by using the `report_settings build` command. An example of a report is as follows:

```
BUILD> report settings build
build = add_buffer=yes, delete_unused_gates=yes, fault_
boundary=lowest,
          hierachal_delimiter='/', pin_assign=256, undriven_
bidi=PIO,
          net_connections_change_netlist=yes,
merge: bus_keepers=yes
          cascaded_gate_with_pin_loss=no
          equivalent_dlat_dff=on
          feedback_paths=yes
          flipflop_from_dlat=on
          mux_from_gates=pin-preserve
          tied_gates_with_pin_loss=no
          global_tie_propagate=yes
          wire_to_buffer=yes
          xor_from_gates=pin-preserve
```

During the flattening process, gate optimization details are reported if expert-level messages have been enabled with the `set_messages -level expert` command. In addition, a summary of optimization results is available at any time after the build process is completed by using the `report_summaries optimizations` command, as shown in the following example.

```
TEST> report_summaries optimizations
          Optimizations Report
-----
optimization #occurrences #primitives #pins      #modules
type      eliminated    lost    optimized
-----
unused gates 15905        15905     2552      133
tied gates    0           42        0        0
buffers       44152        44152     0        313
inverters     10601        10601     0        100
cascaded gates 529        529        0        2
SWs as BUFS   42          0         0        1
DLATs as BUFS 0           0         0        0
MUXs          3261         16242     8        19
XORs          0           0         0        0
equiv. DLAT/DFF 1831      0         0        8
DLATs as DFFs 0           0         0        0
DFFs as DLATs 0           0         0        0
BUS keepers   60          0         0        1
```

feedback paths	18	36	18	1
<hr/>				
total	76399	87507	2638	322
<hr/>				

If, during the optimization process, faults sites are eliminated as gates are removed, those faults sites are identified as B22 violations. Use the `report_violations b22` command to get a detailed list of fault sites removed during optimization or the `report_rules b22` command to get a summary count.

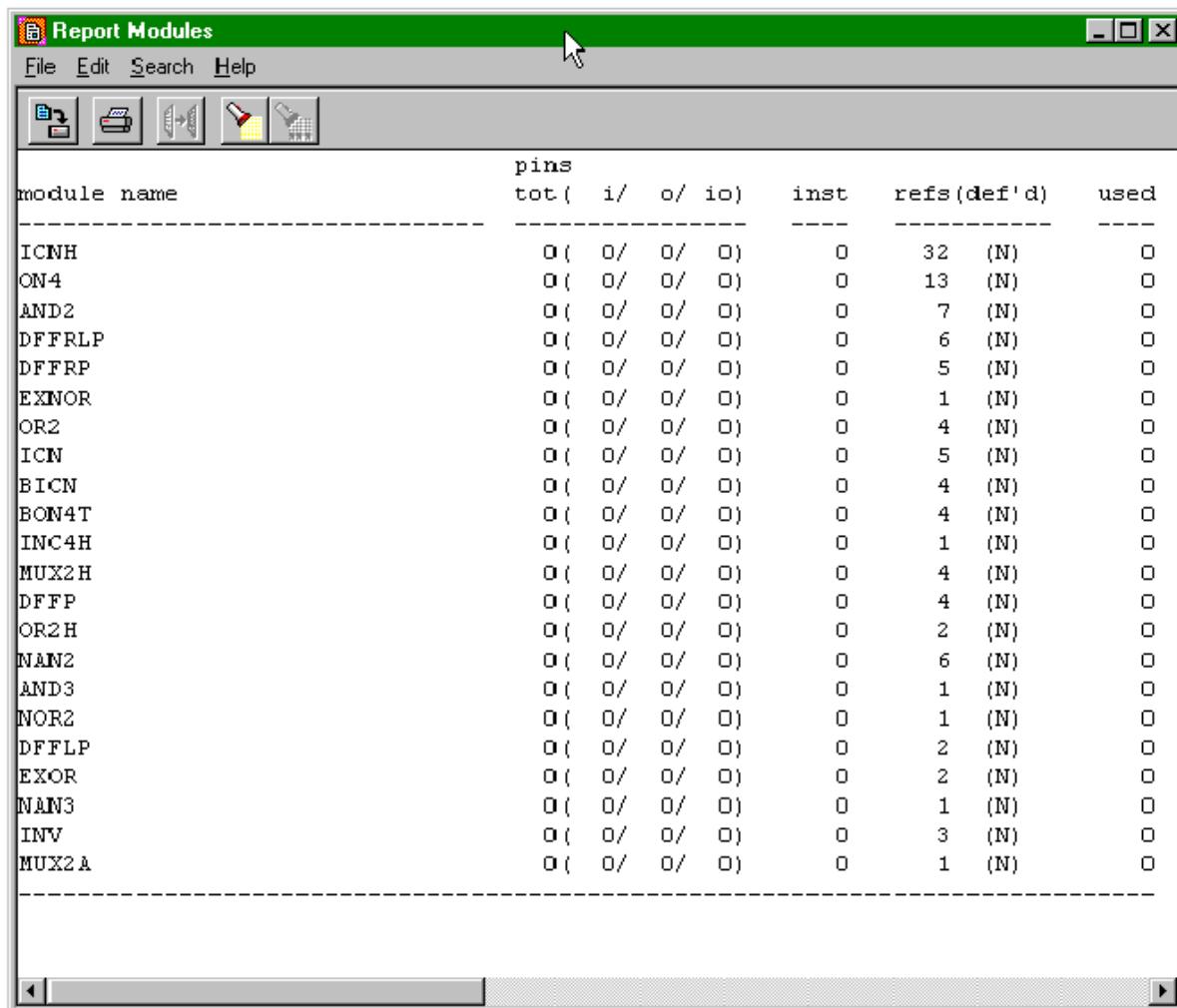
Identifying Missing Modules

If your design references undefined modules, TetraMAX ATPG sends you error messages during execution of the `run_build_model` command. To identify all currently referenced undefined modules, you can use the Netlist > Report Modules menu command, or you can enter the `report_modules -undefined` command at the command line, for example:

```
BUILD-T> report_modules -undefined
```

An example of such a report is shown in [Figure 1](#).

Figure 1: Report Modules Window Listing Undefined Modules



The screenshot shows a Windows application window titled "Report Modules". The menu bar includes "File", "Edit", "Search", and "Help". Below the menu is a toolbar with icons for file operations. The main area is a table with the following columns: module name, pins (tot, i/o, o/ io), inst, refs (def'd), and used. The table lists various undefined logic modules such as ICNH, ON4, AND2, DFFRLP, DFFRP, EXNOR, OR2, ICN, BICN, BON4T, INC4H, MUX2H, DFFP, OR2H, NAN2, AND3, NOR2, DFFLP, EXOR, NAN3, INV, and MUX2A. All entries show 0 for total pins and instances, and (N) for references.

module name	pins				refs (def'd)	used
	tot	i/	o/	io		
ICNH	0(0/	0/	0)	0	(N)
ON4	0(0/	0/	0)	0	(N)
AND2	0(0/	0/	0)	0	(N)
DFFRLP	0(0/	0/	0)	0	(N)
DFFRP	0(0/	0/	0)	0	(N)
EXNOR	0(0/	0/	0)	0	(N)
OR2	0(0/	0/	0)	0	(N)
ICN	0(0/	0/	0)	0	(N)
BICN	0(0/	0/	0)	0	(N)
BON4T	0(0/	0/	0)	0	(N)
INC4H	0(0/	0/	0)	0	(N)
MUX2H	0(0/	0/	0)	0	(N)
DFFP	0(0/	0/	0)	0	(N)
OR2H	0(0/	0/	0)	0	(N)
NAN2	0(0/	0/	0)	0	(N)
AND3	0(0/	0/	0)	0	(N)
NOR2	0(0/	0/	0)	0	(N)
DFFLP	0(0/	0/	0)	0	(N)
EXOR	0(0/	0/	0)	0	(N)
NAN3	0(0/	0/	0)	0	(N)
INV	0(0/	0/	0)	0	(N)
MUX2A	0(0/	0/	0)	0	(N)

In the report, the columns for the total number of pins, input pins, output pins, I/O pins, and number of instances all contain 0. Because the corresponding modules are undefined, this information is unknown. In the “refs (def'd)” column, the first number indicates the number of times the module is referenced by the design, and (N) indicates that the module has not yet been defined.

For additional variations of the `report_modules` command, see TetraMAX Online Help.

Any undefined module referenced by the design causes a B5 rule violation when you attempt to use the `run_build_model` command. The default severity of rule B5 is error, so the build process stops.

If you set the B5 rule severity to warning, TetraMAX ATPG automatically inserts a black box model for each missing module when you build the design. In a black box model, the inputs are terminated and the outputs are tied to X. For more information, see “Using Black Box and Empty Box Models.”

To change the B5 rule severity to warning, use the following command:

```
BUILD-T> set_rules B5 warning
```

With this severity setting, when you use the `run_build_model` command, missing modules do not cause the build process to stop. Instead, TetraMAX ATPG converts each missing module into a black box. After this process, use the `report_violations` command to view an explicit list of the missing modules:

```
DRC-T> report_violations B5
```

Leaving the B5 rule severity set to warning might cause you to miss true missing module errors later. To be safe, you should set the rule severity back to error. Before you do this, use the `set_build` command to explicitly declare the black box modules in the design, as explained in the next section. Then you can set the B5 rule severity back to error and still build your design successfully.

Removing Unused Logic

Designs can contain unused logic for several reasons:

- Existing modules are reused and some sections of the original module are not used in the new design.
- Synthesis optimization has not yet been performed to remove unused logic.
- Gates are created as a side effect to support timing checks in the defining modules.

[Example 1](#) shows a module definition for a scan D flip-flop with asynchronous reset. Because of timing check side effects, the module contains extra gates, with instance names `timing_check_1`, `timing_check_2`, and so on. These gates form outputs that are referenced exclusively in the `specify` section. This is a common technique for developing logic terms used in timing checks, such as setup and hold.

Example 1: Example Module With Extra Logic

```
module sdffr (Q, D, CLK, SDI, SE, RN);
    input D, CLK, SDI, SE, RN;
    output Q;
    reg notify;

    // input mux
    not mux_u1 (ckb, CLK);
    and mux_u2 (n1, ckb, D);
    and mux_u3 (n2, CLK, SDI);
    or mux_u4 (data, n1, n2);

    // D-flop
    DFF_UDP dff (Q, data, CLK, RN, notify);

    // timing checks
    not timing_check_1 (seb, SE);
    and timing_check_2 (rn_and_SE, RN, SE);
    and timing_check_3 (rn_and_seb, RN, seb);

    specify
        if (RN && !SE) (posedge CLK => (Q +: D)) = (1, 1);

```

```

if (RN && SE) (posedge CLK => (Q +: SDI)) = (1, 1);
(negedge RN => (Q +: 1'b0)) = (1, 1);
$setup (D, posedge CLK && rn_and_seb, 0, notify);
$hold (posedge CLK,D && rn_and_seb, 0, notify);
$setup (SDI, posedge CLK && rn_and_SE, 0, notify);
$hold (posedge CLK,SDI && rn_and_SE, 0, notify);
$setup (SE, posedge CLK && RN, 0, notify);
$hold (posedge CLK,SE && RN, 0, notify);
endspecify
endmodule

```

When this module is converted into a gate-level representation, the timing check gates in the internal module representation are retained. The output of the `report_modules -verbose` command for module `sdffr` in [Example 2](#) shows each primitive in the TetraMAX model, with the timing check gates present.

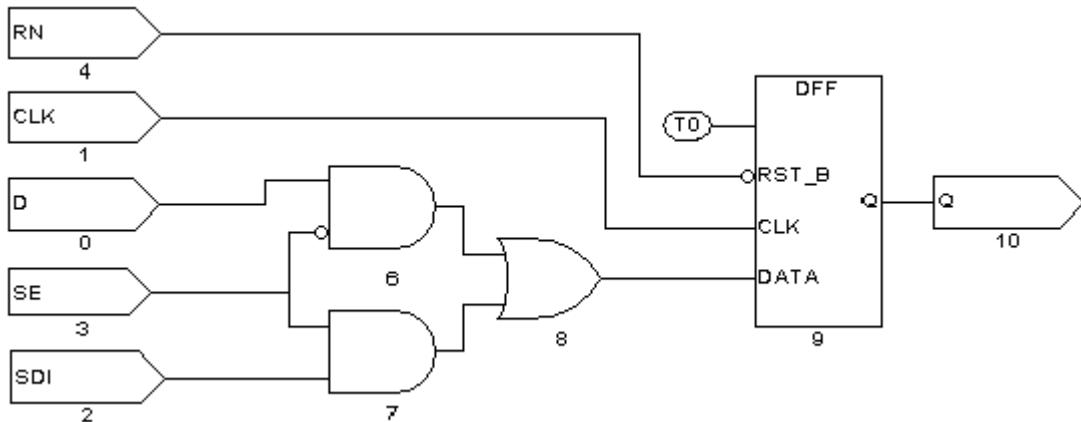
Example 2: Module Report Showing Unused Gates

```
BUILD-T> report_modules sdffr -verbose
          pins
module name      tot( i/ o/ io) inst refs(def'd) used
-----
sdffr           6( 5/ 1/ 0)     8    0 (Y)      1
  Inputs : D ( ) CLK ( ) SDI ( ) SE ( ) RN ( )
  Outputs : Q ( )
  mux_u1 : not conn=( O:ckb I:CLK )
  mux_u2 : and conn=( O:n1 I:ckb I:D )
  mux_u3 : and conn=( O:n2 I:CLK I:SDI )
  mux_u4 : or conn=( O:data I:n1 I:n2 )
  dff : DFF_UDP conn=( O:Q I:data I:CLK I:RN I:notify )
  timing_check_1: not conn=( O:seb I:SE )
  timing_check_2: and conn=( O:rn_and_SE I:RN I:SE )
  timing_check_3: and conn=( O:rn_and_seb I:RN I:seb )
-----
```

By default, TetraMAX ATPG deletes unused gates when it builds the design. To specify whether unused gates are to be deleted or kept, choose Netlist > Set Build Options, which displays the Set Build dialog box. Notice that, in this case, the “Delete unused gates” box is checked, meaning that the deletion of unused gates is selected. To keep the extra gates, deselect the “Delete unused gates” box.

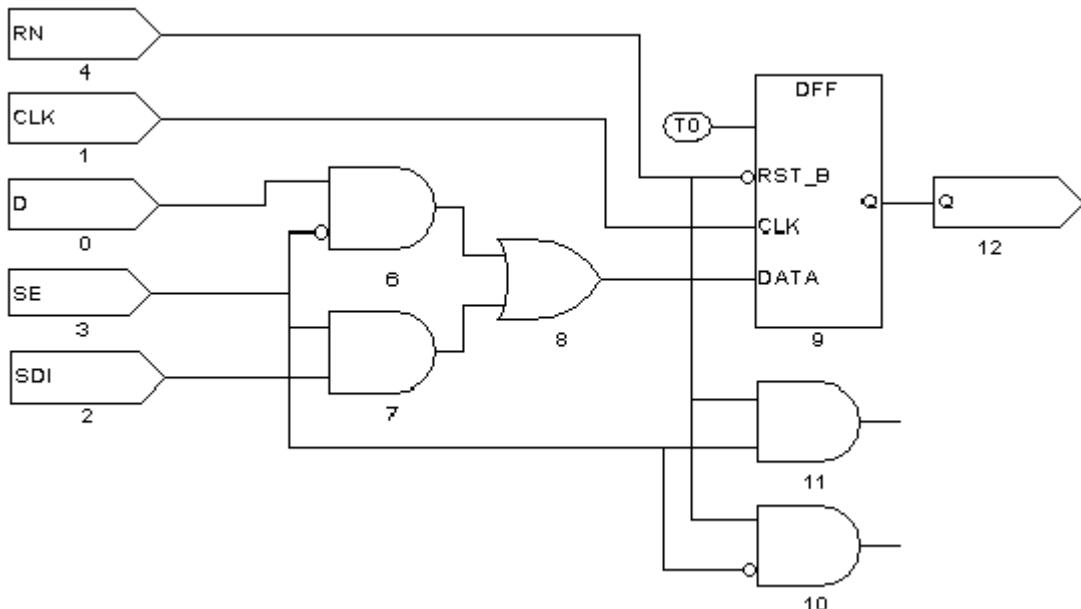
[Figure 1](#) shows the GSV display of the schematic created when the “Delete unused gates” option is selected. The extra gates do not appear in the schematic.

Figure 1: Design Schematic With Delete Unused Gates On



To keep the extra gates, deselect the “Delete unused gates” option of the Set Build dialog box. [Figure 2](#) shows the resulting schematic. The design retains the three extra timing check gates logically as two additional primitives with unused output pins. These extra gates can produce extra fault-site locations, increasing the total number of faults in the design and therefore increasing the processing time. Any faults on these gates are categorized as UU (undetectable, unused). Although these UU faults do not lower the test coverage, they still cause an increase in memory usage and processing time.

Figure 2: Design Schematic With Delete Unused Gates Off



If you want to change the “Delete unused gates” setting, you must do so before executing the `run_build_model` command on your design. If you build your design and then change the setting, you must return to build mode and rerun the `run_build_model` command.

You can also change the unused gate deletion setting by using the `set_build` command with the `-delete_unused_gates` or `-nodelete_unused_gates` option. The following command overrides the default and keeps unused gates:

```
BUILD-T> set_build -nodelete_unused_gates
```

Using Black Box and Empty Box Models

You might prefer not to perform ATPG on some blocks in a design – referred to as *black boxes* or *empty boxes*.

You can declare any block in the design to be a black box or an empty box, including phase-locked loop block, an analog block, a block that is bypassed during test, or a block that is tested separately, such as a RAM block.

The following sections describe how to use of black box and empty box models:

- [Declaring Black Boxes and Empty Boxes](#)
- [Behavior of RAM Black Boxes](#)

See Also

[Binary Image Files](#)

[Excluding Vectors from Simulation](#)

Declaring Black Boxes and Empty Boxes

You can declare black box or any empty box by using one of the following commands:

```
set_build -black_box module_name  
set_build -empty_box module_name
```

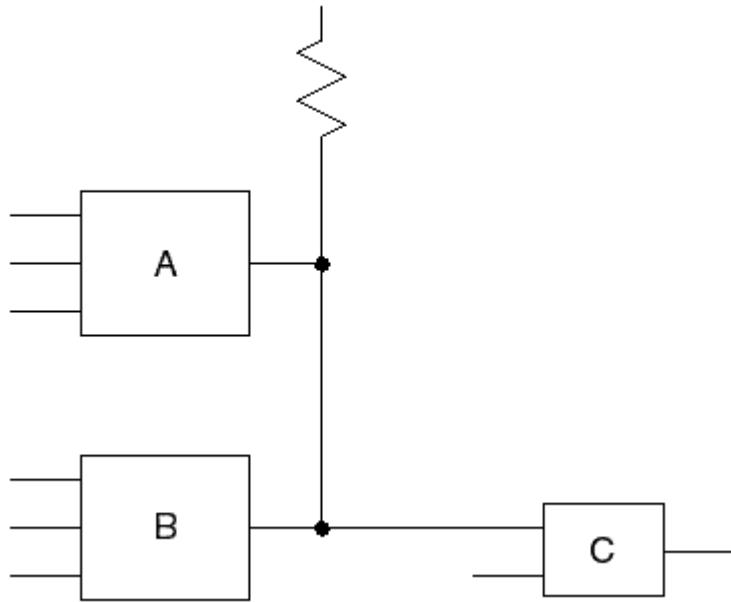
If you declare a block to be a black box, TetraMAX ignores the contents of the block when you build the model with the `run_build_model` command. Instead, it terminates the block inputs and connects TIEX primitives to the outputs. Thus, the block outputs are unknown (X) for ATPG.

An empty box is the same as a black box, except that the outputs are connected to TIEZ rather than TIEX primitives. Thus, the block outputs are assumed to be in the high-impedance (Z) state for ATPG.

The black box model is the usual and more conservative model for any block that is to be removed from consideration for ATPG. In certain cases, however, this model can cause contention, thereby preventing patterns from being generated for logic outside of the black box. In these cases, the empty box model is a better choice.

For example, suppose that you have two RAM blocks called A and B, both with three-state outputs. The block outputs are tied together and connected to a pullup resistor, as shown in [Figure 1](#). If the enabling logic is working properly, no more than one RAM is enabled at any given time, thus preventing contention at the outputs.

Figure 1: RAM Blocks Modeled As Empty Boxes



If you declare blocks A and B to be black boxes, their outputs are unknown (X), resulting in a contention condition that could prevent pattern generation for logic downstream from the outputs. However, if you are sure that both block A and block B is disabled during test, you can declare these two blocks to be empty boxes. In that case, their outputs is Z, and the pullup will pull the output node to 1 for ATPG.

Be careful when you use an empty box declaration. The pattern generator cannot determine whether the outputs are really in the Z state during test. If they are not really in the Z state, the generated patterns might result in contention at the empty box outputs.

You can build your own black box and empty box models if you prefer to do so. Here is an example of a model that works just like a black box declaration:

```

module BLACK (i1,i2, o1, o2, bidil, bidi2);
input i1, i2;
output o1, o2;
inout bidil, bidi2;
    _TIEX (i1, i2, o1); // terminate inputs & drive
    output
        _TIEX (o2);
        _TIEX (bidil);
        _TIEX (bidi2);
endmodule

```

Here is an example of a model that works just like an empty box declaration:

```

module EMPTY (i1,i2, o1, o2, bidil, bidi2);
input i1, i2;
output o1, o2;
inout bidil, bidi2;
    _TIEZ (i1, i2, o1);
    _TIEZ (o2);
    _TIEZ (bidil);

```

```
_TIEZ (bidi2);
endmodule
```

Note that an empty box is not the same as a model without any internal components or connections, such as the following example:

```
module NO_GOOD (i1,i2, o1, o2, bidil, bidi2);
input i1, i2;
output o1, o2;
inout bidil, bidi2;
endmodule
```

If you use such a model, TetraMAX interprets it literally, resulting in multiple design rule violations (unconnected module inputs and undriven module outputs). The unconnected inputs are considered “unused,” so the gates that drive these inputs might be removed by the ATPG optimization algorithm, thus affecting the gate count and fault list. Each unconnected output triggers a design rule violation and is connected to a TIEZ primitive, which becomes an X on most downstream gate inputs.

To avoid these problems, create a model like one of the earlier examples, or use the `set_build` command to declare the block to be a black box or empty box.

Behavior of RAM Black Boxes

When the behavior of your RAM black box is not what you expected, you should consider how the memory itself was modeled. The following six cases revolve around how the memory module is or is not in the netlist, and how TetraMAX treats that memory device. Additionally, pros and cons are provided for each case.

Case 1

Netlist Contains: No module definition for memory

TetraMAX Session: Defines memory as an EMPTY BOX

In this case, because you do not have a module definition for the RAM, use the `set_build -empty_box specRAM` command to tell TetraMAX to treat the module as an empty box.

Pros: No modeling required.

Cons: If the memory has an output enable that is not held off, then this model is not accurate. TetraMAX will have a false environment where it sees no contention but there could really be contention occurring.

Case 2

Netlist Contains: No module definition for memory

TetraMAX Session: Defines memory as a BLACK BOX

In this case, because you do not have a module definition for the RAM, use the `set_build -black_box specRAM` command to instruct TetraMAX to treat the module as a black box.

Pros: No modeling required.

Cons: If multiple black box or empty box devices are connected together, then TetraMAX ATPG might not be able to determine if a pin is an input or an output. An output pin that is mistakenly

considered an input means a TIEX that might have exposed a contention problem will go unnoticed.

Case 3

Netlist Contains: Null module definition for memory

TetraMAX Session: Defines memory as an EMPTY BOX

In this case, you take the memory module port definition from your simulation model and delete the behavioral or gate level description, leaving only the input/output definition list. This is known as a "null" module, because it has no gates within it. You then optionally use the `set_build -empty_box specRAM` command to explicitly document that this module is an empty box. The `set_build -empty_box` command in this particular case is actually not needed, but it is good practice to record in the log file that the model is intentionally and explicitly to be an empty box. Without this, someone reviewing your work at a later time would have to know what was in the RAM ATPG model definition to know what type of model was chosen.

Pros: Modeling takes just a few minutes if you already have a simulation model.

There is no ambiguity within TetraMAX as to which pins are inputs or outputs as in Case 2.

Cons: If the memory has an output enable that is not held off, then this model is not accurate. TetraMAX will have a false environment where it sees no contention, but there could really be contention occurring.

Here's an example null module:

```
module specRAM (read, write, cs, oe,
data_in, data_out, read_addr, write_addr );
input read, write, cs, oe;
input [7:0] data_in;
input [3:0] read_addr;
input [3:0] write_addr;
output [7:0] data_out;
// all core gates deleted to form NULL module
endmodule
```

Note: Null module definitions generate numerous Nxx warnings about unconnected inputs. These can be eliminated by adding a TIEZ gate and connecting all input pins to this gate so that they are terminated and connecting the output to a dumspec net.

Case 4

Netlist Contains: Null module definition

TetraMAX Session: Defines memory as a BLACK BOX

In this case, you create a null module as in Case 3, but you use the `set_build -black_box specRAM` command to instruct TetraMAX that the outputs of the module should be connected to TIEX drivers. The `set_build` command is not optional for this case, or you would have an empty box instead of a black box.

Pros: Modeling takes just a few minutes if you already have a simulation model.

There is no ambiguity within TetraMAX as to which pins are inputs or outputs as in Case 2.

There is no danger of creating a false environment where potential contention is masked by the model as in Cases 1, 2, or 3.

Cons: If the RAM has tristate outputs considered constantly TIEX, then an overly pessimistic environment is created. When a design has multiple RAMs whose outputs are tied together, this pessimistic model will produce contention that cannot be avoided. Depending on the contention settings chosen for ATPG pattern generation, TetraMAX ATPG might discard all the patterns produced.

Case 5

Output enable modeling

In this case, you start with a null module definition and add only enough gates to properly model the tristate output of the device. This is usually a few AND/OR gates and BUFIF gates enabled by some sort of chip select or output enable.

Pros: Modeling effort is light to medium. Most models can be created in less than half an hour with experience.

There is no ambiguity within TetraMAX as to which pins are inputs or outputs as in Case 2.

There is no danger of creating a false environment where potential contention is masked by the model as in Cases 1, 2, or 3.

There is no danger of an overly pessimistic output that introduces contention problems as in Case 4.

Cons: Although this model solves most problems, it does not let the TetraMAX generate patterns that would use the RAM to control and observe circuitry around the RAM, thereby leaving faults in the "shadow" of the RAM undetected.

The following example is a memory module with OEN modeling:

```
module specRAM (read, write, cs, oe,
data_in, data_out, read_addr, write_addr );
input read, write, cs, oe;
input [7:0] data_in;
input [3:0] read_addr;
input [3:0] write_addr;
output [7:0] data_out;
and u1 (OEN, cs, oe); // form output enable
buf u2 (TX, 1'bx);
bufif1 do_0 (data_out[0], TX, OEN);
bufif1 do_1 (data_out[1], TX, OEN);
bufif1 do_2 (data_out[2], TX, OEN);
bufif1 do_3 (data_out[3], TX, OEN);
bufif1 do_4 (data_out[4], TX, OEN);
bufif1 do_5 (data_out[5], TX, OEN);
bufif1 do_6 (data_out[6], TX, OEN);
bufif1 do_7 (data_out[7], TX, OEN);
endmodule
```

Case 6

Full functional modeling

In this case, you create a functional RAM model for ATPG using the limited Verilog syntax supported by TetraMAX.

Pros: Eliminates all problems of Cases 1 through 5.

Cons: Most time consuming. Can be as quick as an hour or if multiple days to construct, test, and verify an ATPG model for a memory.

The following example shows a memory module with full functional modeling (see "[Memory Modeling](#)" for additional examples):

```

// --- level sensitive RAM with active high chip select, read,
// write, and output enable controls.
//
module specRAM (read, write, cs, oe,
data_in, data_out, read_addr, write_addr );
input read, write, cs, oe;
input [7:0] data_in;
input [3:0] read_addr;
input [3:0] write_addr;
output [7:0] data_out;
reg [7:0] memory [0:15];
reg [7:0] DO_reg, data_out;
event WRITE_OP;
and u1 (REN, cs, read); // form read enable
and u2 (WEN, cs, write); // form write enable
and u3 (OEN, cs, oe); // form output enable
always @ (WEN or write_addr or data_in) if (WEN) begin
memory[write_addr] = data_in;
#0; ->WRITE_OP;
end
always @ (REN or read_addr or WRITE_OP)
if (REN) DO_reg = memory[read_addr];
always @ (OEN or DO_reg)
if (OEN) data_out = DO_reg;
else data_out = 8'bZZZZZZZZ;
endmodule

```

Troubleshooting Unexplained Behavior

You should double-check the following specific items when you see unexplained behavior from your RAM block box are described next:

1. Did you follow the guidelines in the previous cases in terms of how the memory module was (or was not) defined in the netlist, as well as what command was issued in TetraMAX?
2. Was the `set_build -black_box` command used properly? That is, in particular, the target of this command must be the module name of the RAM, and not a particular instance of the RAM; for example:

```
set_build -black_box spec_ram1024x8
# spec_RAM1 is the module name
```

If you're still not sure, consider the commands:

```
# report on as yet undefined modules;
```

```
# A black box showing up in the rightmost column of this
report
# indicates that the module is recognized as a black box:
report_modules -undefined

OR

# report on what TetraMAX thinks are memories:
report_memory -all -verbose
```

If you have properly performed steps 1 and 2 listed earlier, but are still seeing unexplained behavior, determine if your RAM has bidirectional (inout, tristate) ports. If this is the case, then perform the following steps:

1. Determine why you've opted for a black box instead of an empty box. The black box model uses TIEX to drive outputs, whereas the empty box model uses TIEZ. When RAM or ROM devices have inout/tristate ports used as outputs, they drive "Z" (not "X") when disabled. Therefore, an empty box model would be more appropriate here.
2. If you determine that a black box is still required for a RAM having inout ports used as outputs, then you have some choices to make because there is no way that TetraMAX can determine whether a particular inout should be an "in" or an "out" given only the null module declaration in the netlist:
 - Make a TetraMAX ATPG model for the black box RAM using TIEX ATPG primitives inside the model to force the inout ports to TIEX, and read this in as yet another source file (for example, spec_RAM1_BBmodel.v, which will in essence redefine the module spec_RAM1 to now have these TIEX primitives on its inout ports). The RAM inouts will now act as outputs driving out 'X' values. The TetraMAX graphical schematic viewer (GSV) will show you only the TIEXs representing the RAM at this point, not the RAM itself.
 - Make a TetraMAX ATPG mode similar to the previous example, but instead of placing TIEX ATPG primitives in the model, use actual tristate driver ATPG models (TSD) to drive the inout ports being used as outputs. Also, tie the TSD enable and input pins to TIEX primitives, and the result is not only a RAM whose inout ports now drive out "X", but also a RAM that is visible in the GSV.

Handling Duplicate Module Definitions

You can read a module definition more than one time. By default, TetraMAX ATPG uses the most recently read module definition and issues an N5 rule violation warning for any subsequent module definitions that have the same name.

You can change this default behavior so that the first module defined is always kept, using the `-redefined_module` option of the `set_netlist` command. Alternatively, you can choose Netlist > Set Netlist Options and use the Set Netlist dialog box or click the Netlist button on the command toolbar and use the Read Netlist dialog box.

If you are certain that there are no module name conflicts, you can change the severity of rule N5 from warning to error:

```
BUILD-T> set_rules n5 error
```

With a severity setting of error, the process stops when TetraMAX ATPG encounters the error, thus preventing redefinition of an existing module by another module with the same name.

When you use the `read_netlist` command, you can use the `-master_modules` option to mark all modules defined by the file being read as “master modules.” A master module is not replaced when other modules with the same name are encountered. This mechanism can be useful for reading specific modules that are intended as module replacements, independent of the reading order. Note that a master module can be replaced by a module with the same name if the `-master_modules` switch is again used.

Memory Modeling

You can define RAM and ROM models using a simple Verilog behavioral description. The following sections describe memory modeling:

- [Memory Model Functions](#)
 - [Basic Memory Modeling Template](#)
 - [Initializing RAM and ROM Contents](#)
 - [Improving Test Coverage for RAMs](#)
-

Memory Model Functions

Memory models can have the following functions:

- Multiple read and write ports
- Common or separate address bus
- Common or separate data bus
- Edge-sensitive or level-sensitive read and write controls
- One qualifier on the write control
- One qualifier on the read control
- A read off state that can hold or return data to 0/1/X/Z
- Asynchronous set and reset capability
- Memory initialization files

You create a ROM by defining a RAM that has an initialization file and no write port.

Note: Write ports cannot simultaneously be both level-sensitive and edge-sensitive. However, the read ports can be mixed edge-sensitive and level-sensitive, and can be different from the write ports.

TetraMAX ATPG uses a limited Verilog behavioral syntax to define RAM and ROM models for ATPG use. In cone timept, this is equivalent to defining some simple RAM/ROM functional models.

For detailed information on RAM and ROM modeling, see the “TetraMAX Memory Modeling” topic in Online Help.” The topics covered in Online Help include defining write ports and read

ports, read off behavior, memory address range, multiple read/write ports, contention behavior, memory initialization, and memory model debugging.

Basic Memory Modeling Template

[Example 1](#) is a basic template for a 16-word by 8-bit RAM that can be applied to a ROM.

Example 1: Basic Memory Modeling Template

```
module spec_ATPG_RAM ( read, write, data_in, data_out,
read_addr,write_addr );
    input read, write;
    input [7:0] data_in; // 8 bit data width
    input [3:0] read_addr; // 16 words
    input [3:0] write_addr; // 16 words
    output [7:0] data_out; // 8 bit data width
    reg [7:0] data_out; // output holding register
    reg [7:0] memory [0:15] ; // memory storage

    event WRITE_OP; // declare event for write-through
    ...memory port definitions...
endmodule
```

The template consists of a Verilog module definition in which you make the following definitions:

- The inputs and outputs (in any order and with any legal port name)
- The output holding register, “`data_out`” in this example
- The memory storage array, “`memory`” in this example

This basic structure changes very little from RAM to ROM. The port list might vary for more complicated RAMs or ROMs with multiple ports, but the template is essentially the same. Note that the ATPG modeling of RAMs requires that bused ports be used.

Initializing RAM and ROM Contents

If a RAM is to be initialized, you must provide the vectors that initialize it.

If your design contains ROMs, you must initialize the ROM image by loading data into it from a memory initialization file. You create a default initialization file and reference it in the ROM’s module definition.

If you want to use a different memory initialization file for a specific instance, use the `read_memory_file` command to refer to the new memory initialization file. In TetraMAX ATPG, ROMs and RAMs are identical in all respects except that the ROM does not have write data ports. Thus, the following discussion about ROMs also applies to RAMs.

The Memory Initialization File

ROM memory is initialized by a hexadecimal or binary ASCII file called a memory initialization file. [Example 2](#) shows a sample hexadecimal memory initialization file.

Example 2: Memory Initialization File

```
// 16x16 memory file in hex
0002
0004
0008
0010
0020
0040
0080
0100
0200
0400
0800
1000
2000
4000
8000
```

For additional examples of Memory Initialization Files, see the “TetraMAX Memory Modeling” topic in Online Help.

Default Initialization

To establish the default memory initialization file, specify its file name in the module definition of the ROM. [Example 3](#) defines a Verilog module for a ROM that has 16 words of 16 data bits.

Example 3: 16x16 ROM Model

```
module rom16x16 (ren, a, dout);
parameter addrbits = 4;
parameter addrmax = 15;
parameter databits = 16;
input ren;
input [addrbits-1:0] a;
output [databits-1:0] dout;
reg [databits-1:0] specmem [0:addrmax];
reg [databits-1:0] dout ;
initial $readmemh("rom_init.dat", specmem);
always @ ren if (ren) dout <= specmem[a] ;
endmodule
```

The initial \$readmemh statement in this example indicates that the data in the rom_init.dat file is used to initialize the memory core specmem. The \$readmemh() function is for hexadecimal data; there is a similar function, \$readmemb(), for binary data.

Verilog defines the order in which data is loaded into the specmem core. This order is based on how you define the specmem index, as follows:

- The format specmem[0:15] indicates that the first data word in the file is to be loaded into address 0 and the last data word into address 15.
- The format specmem[15:0] indicates that the first data word in the file is to be loaded into address 15 and the last data word into address 0.

In [Example 3](#), the following line indicates that the first data word is loaded into address 0 and the last data word is loaded into the address specified by `addrmax`:

```
reg [databits-1:0] specmem [0:addrmax];
```

Instance-Specific Initialization

If you use more than one ROM instance in your design, you might not want to initialize all the ROMs from the same memory initialization file.

For each specific ROM instance, you can override the memory initialization file specification in the module definition using the Read Memory File dialog box, or you can enter the `read_memory_file` command at the command line.

1. To use the Read Memory File dialog box to override the memory initialization file specification in the module definition for a specific ROM instance, perform the following steps:
 2. From the menu bar, choose the Primitives > Read Memory File. The Read Memory File dialog box appears.
 3. Enter the instance and then enter or browse to the memory initialization file. For more information about the controls in this dialog box, see Online Help for the `read_memory_file` command.
 4. Click OK.

You can also override the memory initialization file specification in the module definition using the `read_memory_file` command. For example:

```
DRC-T> read_memory_file i007/u1/mem/rom1/rom_core i007.d3 -hex
```

The following example indicates that the instance `/TOP/BLK1/rom1/rom_core` is to be initialized using the hexadecimal file `U1_ROM1.dat`.

```
DRC-T> read_memory_file /BLK1/rom1/rom_core U1_ROM1.dat -hex
```

Note: In responding to the `read_memory_file` command, TetraMAX ATPG always loads the first word in the data file into memory address 0, the second word into address 1, and so on, regardless of how the memory index is defined in the Verilog module.

Improving Test Coverage for RAMs

Test patterns for RAMs intrinsically require more clock cycles than most other types of tests. Also, a RAM usually requires the justification of considerably more values (all address bus bits, data bus bits, and enable signals) than most combinational gates. In addition, the behavior of RAMs is more complex than the behavior of other circuit elements, which may increase the difficulty of getting tests for these faults.

TetraMAX ATPG minimizes the complexity of memory test generation by separating the various memory operations into different scan chain loads. For example, if a test for a RAM fault involves two write operations and one read operation, TetraMAX ATPG will generally do the following:

1. Scan chain load 1
2. Write operation 1
3. Scan chain load 2
4. Write operation 2
5. Scan chain load 3
6. Read operation

A RAM must be load stable to make use of multiple scan chain loads. This means all RAM operations must be disabled during the scan chain load-unload procedure. You can do this by gating the RAM clock with the scan-enable signal or by turning off the RAM enable signals (including Chip Select, if such a signal exists) during the scan chain load. A load-stable RAM enables TetraMAX ATPG to maximize its efficiency when generating tests for RAM faults, however the tool still cannot generate tests for all RAM faults.

Creating Custom ATPG Models

You can create custom models specifically for ATPG use by constructing a Verilog gate-level representation of the logic function using a combination of Verilog primitives, TetraMAX primitives, and other defined modules. For a list of TetraMAX primitives, see “ATPG Modeling Primitives Summary” in TetraMAX Online Help.

Use only Verilog primitives or instances of other Verilog modules when possible. Because Verilog understands these devices, you can simulate these modules to validate that they function as expected.

[Example 1](#) uses TetraMAX primitives to model the test mode of a particular device. The model provides a constant 1 on the output lock, and a constant 0 on the outputs `ref_out`, `div2`, and `div4` when test is asserted. Otherwise, these outputs are X.

Example 1: Custom ATPG Model Using ATPG Primitives

```
module phase_lock1 (test, ref_in, delayed_in, ref_out, div2, div4,
lock);
input test, ref_in, delayed_in;
output ref_out, div2, div4, lock;
wire xval;

_TIEX u1 (delayed_in, xval);
_MUX u2 (test, ref_in, 1'b0, ref_out);
_MUX u3 (test, xval, 1'b0, div2);
_MUX u4 (test, xval, 1'b0, div4);
_MUX u5 (test, xval, 1'b1, lock);
endmodule
```

[Example 2](#) uses Verilog primitives to implement the same functions.

Example 2: Custom ATPG Model Using Verilog Primitives

```
module mux (sel,d0,d1, out);
```

```

input d0,d1,sel;
output out;
wire n1,n2,n3;
not u1 (selb, sel);
and u2 (n2, d1,sel);
and u3 (n3, d0,selb);
or u4 (out, n1,n2);
endmodule
module phase_lock2 (test, ref_in, delayed_in, \
ref_out, div2, div4, lock);
input test, ref_in, delayed_in;
output ref_out, div2, div4, lock;
wire xval;

buf u1 (xval, 1'bx);
mux u2 (test, ref_in, 1'b0, ref_out);
mux u3 (test, xval, 1'b0, div2);
mux u4 (test, xval, 1'b0, div4);
mux u5 (test, xval, 1'b1, lock);
endmodule

```

[Example 3](#) shows a custom ATPG model of a D flip-flop with a rising-edge clock, asynchronous active-high `set`, asynchronous active-low `resetn`, and scan input `sdi` enabled when scan is asserted. The flip-flop has true and complementary outputs `q` and `qn`, and an output `sdo`, a buffered replica of output `q` used for scan.

Example 3: Custom ATPG Model of a D Flip-Flop

```

module DFWSRB (clk, data, sdi, scan, set, resetn, q, qn, sdo);
input clk, data, sdi, scan, set, resetn;
output q, qn, sdo; wire din;
_MUX u1 (scan, data, sdi, din);
_DFF u2 (set, !resetn, clock, din, q);
_INV u3 (q, qb);
_BUF u3 (q, sdo);
endmodule

```

Condensing ATPG Libraries

TetraMAX ATPG attempts to condense each module's functionality in a netlist into a gate-level representation using TetraMAX simulation primitives. This condensation task can be considerable and can produce some warning messages, which are typically unimportant and can be ignored.

You can create a file that has already been condensed into TetraMAX description form. Creating a condensed form of the library modules has the following benefits:

- Space econospec. The modules are stripped of timing and other non-ATPG related information. In addition, the file can be created in compressed form.

- No error or warning messages. The modules are preprocessed and written using either ATPG modeling primitives or simple netlists instantiating other modules.
- Faster module reading. The modules require less time during analysis and are processed faster.
- Information protection. The file can be created in a compressed binary form that is unreadable by any other tool and partially protects the library information within. When you read in the library and write it out again, you see only a stripped-down functional gate version of the original module; no timing or other information remains.

The transcript in [Example 1](#) illustrates the creation of a condensed library file, which is a two-step process:

1. Read in all required modules.
In [Example 1](#), 1,436 modules are initially found in 1,430 separate files. The read process took 21.5 seconds and reported 16 warnings.
2. Write out the modules as a single file in your choice of formats.
In the example, the modules are written out as a single GZIP compressed file.

Example 1: Creating a Condensed Library File

```
BUILD-T> read_netlist lib/*.v
Begin reading netlists ( lib/*.v )...
Warning: Rule N12 (invalid UDP entry) failed 8 times.
Warning: Rule N13 (X_DETECTOR found) failed 8 times.
End reading netlists: #files=1430, #errors=0, #modules=1436,
#lines=157516,
CPU_time=21.5 sec

BUILD-T> write_netlist parts_lib.gz -compress gzip
End writing Verilog netlist, CPU_time = 1.13 sec, \
File_size = 47571

BUILD-T> read_netlist parts_lib.gz -delete
Warning: All netlist and library module data are now deleted.
(M41)
Begin reading netlist ( parts.lib )...
End parsing Verilog file parts.lib with 0 errors;
End reading netlist: #modules=1436, #lines=18929, CPU_time=0.84
sec
```

The next `read_netlist` command processed the data in less than 1 second and produced the same 1,436 modules, this time without rule violation warnings.

10

STIL Procedures

The STIL language describes scan-shifting protocol, test procedures, and ATPG signal, timing, and data information. STIL procedures provide information TetraMAX ATPG uses as a basis to perform design rule checking (DRC).

You can provide a set of STIL procedures to TetraMAX ATPG through a file, called STIL procedure file (SPF). You can use an existing SPF written by a tool, such as DFT Compiler, or you can create a new SPF. TetraMAX ATPG supports a subset of STIL syntax for input to describe scan chains, clocks, constrained ports, and pattern/response data as part of the STL procedure file definitions. If you use an existing SPF, make sure it meets the parameters recognized by TetraMAX, as described in "[STIL Language Support](#)."

If you create an SPF, you can initially define the minimum information needed by TetraMAX ATPG to run DRC. If you are using the TetraMAX GUI, you can provide this information via the QuickSTIL tab in the DRC dialog box.

The following sections describe the guidelines for using STIL procedures:

- [STIL Procedure File Guidelines](#)
- [Creating a New STIL Procedure File](#)
- [Defining STIL Procedures](#)
- [Specifying Synchronized Multi Frequency Internal Clocks for an OCC Controller](#)
- [Specifying Internal Clocking Procedures](#)
- [JTAG/TAP Controller Variations for the load_unload Procedure](#)
- [Multiple Scan Groups](#)
- [Limiting Clock Usage](#)
- [DFTMAX Adaptive Scan with Serializer](#)

STIL Procedure File Guidelines

TetraMAX ATPG can read and write a properly formatted SPF. Any STIL files written by TetraMAX contain an expanded form of the minimum information and may also contain pattern and response data produced by the ATPG process. After an SPF is generated for a design, TetraMAX ATPG can read it again at a later time to recover the clock, constraint, and chain data, and the pattern and response data, or both. You can also use several TetraMAX commands to supplement or provide the same or similar information as STIL procedures.

The following general guidelines, tips, and shortcuts help you efficiently and accurately work with STIL procedure files:

- To save time and avoid typing errors, use the `write_drc_file` command to create the STIL template. The more information that you provide to TetraMAX ATPG before the `write_drc_file` command, the more TetraMAX ATPG will provide in the template. If possible, build your design model and define all clock and constrained inputs before you create the STIL template.
- STIL keywords are case-sensitive. All keywords start with an uppercase character, and many contain more than one uppercase character.
- Use `SignalGroups` to define groups of ports so that you can easily assign values and timing.
- At the beginning of the `load_unload` procedure, always place the ports declared as clocks in their off states.
- Except for the `test_setup` and `Shift` procedures, every procedure should include initializing all clocks to their off state and all PI constraints and PI equivalences to their proper values at the beginning of the procedure.
- If you have constrained ports or bidirectional ports, define a `test_setup` macro and initialize the ports.
- A `test_setup` procedure must initialize all clocks to their off states, and all PI constraints and PI equivalences to their proper values by the end of the procedure. Note that it is not necessary to stop Reference clocks, including what DFT Compiler refers to as ATE clocks. All other clocks still must be stopped.
- Bidirectional ports should be forced to Z within a `test_setup` macro and forced to Z at the beginning of the `load_unload` procedure.
- For non-JTAG designs, it is usually not necessary to apply a reset to the design within a `test_setup` macro.
- When defining pulsed ports, define the 0/1/Z mapping for cycles when the clock is inactive, as in the following example:
`CLOCK { 01Z { '0ns' D/U/Z; } }`

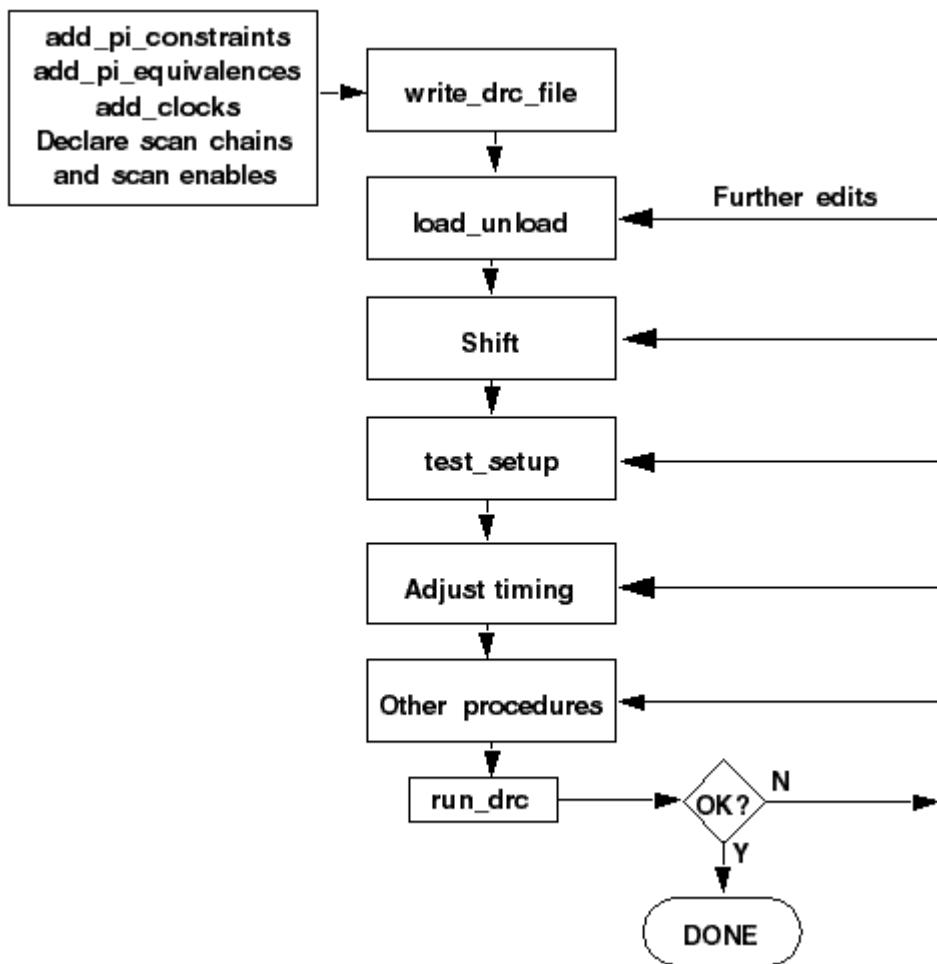
Creating a New STIL Procedure File

To create a new STIL procedure file , you first need to define the primary input (PI) constraints, clocks, and scan chain information using a series of TetraMAX commands. You can then use the `write_drc_file` command to create a STIL template file, and edit this file to define the required STIL procedures and port timing.

The following sections describe how to create an STL procedure file with no prior input:

- [Declaring Primary Input Constraints](#)
- [Declaring Clocks](#)
- [Declaring Scan Chains and Scan Enables](#)
- [Writing the Initial STIL Template](#)

Figure 1 Flow for Creating an Initial STL Procedure File With No Prior Input



See Also

[Defining STIL Procedures](#)

Declaring Primary Input Constraints

In most design-for-test (DFT) scenarios, a design shifts into ATPG mode based on the top-level ports. The success of the ATPG algorithm usually requires that these ports are held to a constant state.

You can use STIL procedures to force a constrained port to a state other than the requested constrained value for a limited number of tester cycles, and then return the port to its constrained value. For example, you might want to hold a global reset port to an off state for general ATPG patterns, but then allow it to be asserted to initialize the design (for more information, see "[Defining the test_setup Macro](#)").

You can declare a port using the Add PI Constraints dialog box in the TetraMAX GUI, the `add_pi_constraints` command, or by defining it in the STIL procedure file. For more information on using the STIL procedure file to define PI constraints, see "[Defining Constrained Primary Inputs](#)".

The following sections show you how to use TetraMAX ATPG to declare primary input constraints:

- [Using the Add PI Constraints Dialog Box](#)
- [Using the add_pi_constraints Command](#)

Note: A port that enables a test mode for a design is different from the `scan_enable` port and other ports that change state during the shift and capture operations.

Using the Add PI Constraints Dialog Box

To use the Add PI Constraints dialog box to declare a PI constraint:

1. From the menu bar, choose Constraints > PI Constraints > Add PI Constraints.
The Add PI Constraints dialog box appears.
2. In the Port Name field, enter the name of the port you want to constrain. To select from a list of ports, click the down-arrow button at the end of the Port Name field.
In this case, a port named TEST_MODE must be held to a constant state of logic 1 for all patterns generated by the ATPG algorithm.
3. From the Value list, choose the value to which you want to constrain the port.
4. Click Add.
The dialog box remains open so that you can add more constraints if needed.
5. Click OK.

Using the add_pi_constraints Command

You can use the `add_pi_constraints` command to declare a PI constraint. For example:

```
DRC-T> add_pi_constraints 1 TEST_MODE
```

Declaring Clocks

You can declare a port as a clock only if the port affects the state of flip-flops and latches or controls RAM/ROM read or write ports. You declare a clock in terms of its natural off state. An active-high clock has an off state of 0, and an active-low clock has an off state of 1.

You can declare a clock in the TetraMAX GUI using the Add Clocks dialog box, the Edit Clocks dialog box, or the DRC dialog box. You can also use the `add_clocks` command to declare a clock, or edit the timing block in the STL procedure file. For more information on declaring clocks in the timing block of the STL procedure file, see "[Defining Basic Signal Timing](#)."

The following sections show you how to declare clocks:

- [Using the Edit Clocks Dialog Box](#)
- [Using the add_clocks Command](#)
- [Asynchronous Set and Reset Ports](#)

Using the Edit Clocks Dialog Box

To declare clocks using the Edit Clocks dialog box:

1. From the menu bar, choose Scan > Clocks > Edit Clocks.
The Edit Clocks dialog box appears.
2. To declare a clock, select the port name from the Port Name list, specify the off state (0 or 1), and specify whether the clock is used for scan shifting. Clock signals used for asynchronous set/reset or RAM/ROM control are not used for scan shifting and are not pulsed during shift procedures.
3. If you want to specify the test cycle period, leading edge time, trailing edge time, and measure time of the clock, fill in the corresponding fields (Period, T1, T2, and Measure) and set the time units (ns or ps). In the absence of explicit timing specifications, the defaults are: Period=100, T1=50, T2=70, Measure=40, and Unit=ns.
The same period, measure time, and units apply to all clocks in the system, but each clock can have its own leading and trailing edge times (T1 and T2). A measure time less than T1 implies a preclock measure protocol, whereas a measure time greater than T2 implies an end-of-cycle measure protocol.
4. Click Add.
The clock declaration is added to the list box.
5. Repeat steps 2 to 4 for each clock input in the design. You can also remove, copy, or modify an existing clock definition.
6. Click OK to implement the changes you have made in the dialog box.

Using the add_clocks Command

You can also declare clocks, and define the test cycle period and timing parameters by using the `add_clocks` command. For example:

```
DRC-T> add_clocks 0 CLK1 -timing 200 50 80 40 -unit ns -shift
```

Asynchronous Set and Reset Ports

By default, latches and flip-flops whose set and reset lines are not off when all clocks are at their off state are treated as unstable cells. Because they are unstable, their output values are unknown and they cannot be used during test pattern generation.

One way to make these elements stable is to declare their asynchronous set/reset input signals to be clocks. During ATPG, TetraMAX ATPG holds these inputs inactive while other clocks are being used. However, test coverage surrounding the elements might still be limited.

To have these latches and flip-flops treated as stable cells without declaring their set/reset inputs to be clocks, use the `set_drc -allow_unstable_set_resets` command. See. "[Cells With Asynchronous Set/Reset Inputs](#)" for details.

Declaring Scan Chains and Scan Enables

You can use the DRC dialog box in the TetraMAX GUI or enter a command at the command line to declare the scan chains and scan enable inputs. You can also declare scan chains in the STIL Procedure file, as described in "[Defining Scan Chains](#)".

The following sections describe how to declare scan chains and scan enables in TetraMAX ATPG:

- [Using the DRC Dialog Box](#)
- [Declaring Scan Chains at the Command Line](#)

Using the DRC Dialog Box

To use the DRC dialog box to declare scan chains:

1. Click the DRC button in the command toolbar at the top of the TetraMAX ATPG main window.
The DRC dialog box appears.
2. Click the Quick STIL tab if it is not already selected. Under the tab, select the Scan Chains/Scan Enables view if it is not already selected.
Note: If you select the Clocks view, the Edit Clocks dialog box appears, as described in "[Declaring Clocks](#)".
3. To specify a scan chain, enter a name for the scan chain in the Name field. Specify the Scan In and Scan Out ports by selecting the port names from the pull-down lists.
4. Click Add.
The scan chain definition is added to the list.
5. To define a scan enable input, select the port name from the Port Name pull-down list. In the Value field, specify the port value during scan shifting.
6. Click Add.
The scan enable port definition is added to the list.
7. When you finish running the scan chain and scan enable information, click OK.

Declaring Scan Chains at the Command Line

You can use the following commands to declare, report, and remove scan chains and scan enables at the command line:

- `add_scan_chains`
- `add_scan_enables`
- `report_scan_chains`
- `report_scan_enables`
- `remove_scan_chains`
- `remove_scan_enables`

Writing the SPF Template

You can create an SPF template file after executing the `run_build_model` command. This template includes all clocks, PI equivalences, PI constraints, or scan chain information you have previously specified.

To create an SPF template from the TetraMAX GUI:

1. Click the DRC button in the command toolbar at the top of the TetraMAX GUI main window.
The DRC dialog box appears.
2. Click the Write tab in the DRC dialog box.
3. In the Name field, enter the name of the STIL procedure file you want to create.
4. Click the Write button.

The following example shows how to create a STIL template using the `write_drc_file` command:

```
write_drc_file template.spf
```

Example SPF Template File

The following example shows an STIL procedure file template file created from the `write_drc_file` command:

```
STIL 1.0 {
    Extension Design P2011;
}
Header {
    Title " TetraMAX 2010.06-i000622_173054 STIL output";
    Date "Wed Dec 31 17:21:05 2011";
    History {}
}
Signals {
    CLK In; RSTB In; SDI2 In; SDI1 In; INC In; SCAN In; HACKIN In;
    si4 In;
    six In; D0 InOut; D1 InOut; D2 InOut; D3 InOut; SDO2 Out; COUT
```

```

Out;
    HACKOUT Out; so4 Out; sox Out;
}
SignalGroups {
    _pi = 'D0 + D1 + D2 + D3 + CLK + RSTB + SDI2 + SDI1 + INC +
SCAN + HACKIN + si4 + six';
    _default_Clk1_Timing_ = 'RSTB';
    _io = 'D0 + D1 + D2 + D3' { WFCMap 0X->0; WFCMap 1X->1; WFCMap
ZX->Z; WFCMap NX->N; }
    _po = 'SDO2 + COUT + D0 + D1 + D2 + D3 + HACKOUT + so4 + sox';

    _default_In_Timing_ = 'D0 + D1 + D2 + D3 + CLK + RSTB + SDI2 +
SDI1 + INC + SCAN + HACKIN + si4 + six';
    _default_Out_Timing_ = 'SDO2 + COUT + D0 + D1 + D2 + D3 +
HACKOUT
    + so4 + sox';
    _default_Clk0_Timing_ = 'CLK';
}
ScanStructures {
    # Uncomment and modify the following to suit your design
    # ScanChain chain_name { ScanIn chain_input_name; ScanOut
chain_output_name; }
}
Timing {
    WaveformTable _default_WFT_ {
        Period '100ns';
        Waveforms {
            _default_In_Timing_ { 0 { '0ns' D; } }
            _default_In_Timing_ { 1 { '0ns' U; } }
            _default_In_Timing_ { Z { '0ns' Z; } }
            _default_In_Timing_ { N { '0ns' N; } }
            _default_Clk0_Timing_ { P { '0ns' D; '50ns' U; '80ns' D;
} }
            _default_Clk1_Timing_ { P { '0ns' U; '50ns' D; '80ns' U;
} }
            _default_Out_Timing_ { X { '0ns' X; } }
            _default_Out_Timing_ { H { '0ns' X; '40ns' H; } }
            _default_Out_Timing_ { T { '0ns' X; '40ns' T; } }
            _default_Out_Timing_ { L { '0ns' X; '40ns' L; } }
        }
    }
}
PatternBurst _burst_ { PatList {
    _pattern_
}
}
}

```

```

PatternExec {
    PatternBurst _burst_;
}

Procedures {
    capture_CLK {
        W _default_WFT_;
        forcePI: V { _pi=\r13 # ; _po=\j \r9 X ; }
        measurePO: V { _po=\r9 # ; }
        pulse: V { CLK=P; _po=\j \r9 X ; }
    }
    capture_RSTB {
        W _default_WFT_;
        forcePI: V { _pi=\r13 # ; _po=\j \r9 X ; }
        measurePO: V { _po=\r9 # ; }
        pulse: V { RSTB=P; _po=\j \r9 X ; }
    }
    capture {
        W _default_WFT_;
        forcePI: V { _pi=\r13 # ; _po=\j \r9 X ; }
        measurePO: V { _po=\r9 # ; }
    }

    # Uncomment and modify the following to suit your design
    # PRE_CLOCK_MEASURE Procedures {
    # load_unload {
        # W _default_WFT_;
        # C { test_so=X; test_si=0; test_si2=0; test_so2=X; clk=0;
tclk=0; reset=1; test_se=1; }
        # Shift { W _default_WFT_;
        # V { _si=#; _so=#; CLK = P; }
        # }
    }
    # TMAX GENERATED POST_CLOCK_MEASURE (Closer to DFTCompiler
Procedures {
    # load_unload {
        # W _default_WFT_;
        # C { test_si=0; test_si2=0; clk=0; tclk=0; reset=1; test_
se=1; }
        # V { _so=#; }
        # Shift { W _default_WFT_;
        # V { _si=#; _so=#; clk=P; }
        # }
    }
    MacroDefs {
        test_setup {
            W _default_WFT_;
            V { CLK=0; RSTB=1; }
        }
    }
}

```

{}

Defining STIL Procedures

There are a variety of STIL procedures you can specify in the SPF, including the load_unload, shift, and test_setup procedures, and capture, system capture, generic capture, and sequential capture procedures. You can also define signal timing and signal groups, scan chains, primary input parameters, PO masks, and many other parameters. Some of these settings can be specified using TetraMAX commands.

If you don't have an existing SPF, see "[Creating a New STIL Procedure File](#)."

The following sections describe how to define STIL procedures:

- [Defining Scan Chains](#)
- [Defining the load_unload Procedure](#)
- [Defining the test_setup Procedure](#)
- [Predefined Signal Groups](#)
- [Defining Basic Signal Timing](#)
- [Defining Capture Procedures](#)
- [Defining System Capture Procedures](#)
- [Creating Generic Capture Procedures](#)
- [Defining a Sequential Capture Procedure](#)
- [Defining Constrained Primary Inputs](#)
- [Defining Equivalent Primary Inputs](#)
- [Defining PO Masks](#)
- [Defining Reflective I/O Capture Procedures](#)
- [Using the master_observe Procedure](#)
- [Using the shadow_observe Procedure](#)
- [Using the delay_capture_start Procedure](#)
- [Using the delay_capture_end Procedure](#)
- [Using the test_end Procedure](#)
- [Scan Padding Behavior](#)
- [Using the Condition Statement](#)
- [Excluding Vectors from Simulation](#)
- [Defining Internal Clocks for PLL Support](#)
- [Specifying an On-Chip Clock Controller Inserted by DFT Compiler](#)

Note that STIL keywords are case-sensitive. When you enter a keyword in an STL procedure file, ensure that you use uppercase and lowercase letters correctly (for example, ScanStructures, ScanChain, ScanIn, ScanOut). Incorrect case is a common cause of syntax errors.

Throughout the STIL examples in the following sections, text strings are sometimes enclosed in quotation marks. The general rule in STIL procedure files is that quotation marks are optional unless the text string contains parentheses “()”, braces “[]”, or spaces.

Defining Scan Chains

You define scan chains in the ScanStructures block of the STL procedure file. In the following example, the text in bold type illustrates four scan chains. The labels "c1", "c2", etc., are the symbolic names assigned to the scan chains. The STIL specification indicates a length, but this item is optional for TetraMAX input.

The following example also represents the minimum STL procedure file needed by TetraMAX ATPG as it defines the scan chains, the load_unload procedure, and the Shift procedure.

```
STIL;

ScanStructures {
    ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
    ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
    ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
    ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}

Procedures {
    "load_unload" {
        V {
            CLOCK = 0; RESETB = 1;
            SCAN_ENABLE = 1;
        }
        Shift {
            V { _si #####; _so #####; CLOCK=P; }
        }
    }
}
```

Defining the load_unload Procedure

The `load_unload` procedure describes how to place a design into a state in which the scan chains can be loaded and unloaded. This typically involves asserting a `SCAN_ENABLE` port, or other control line, and placing bidirectionals into a Z state. Standard DRC rules also require that ports defined as clocks must be placed in their off states at the start of the scan chain load/unload process if they are not initialized to an off state in the `test_setup` procedure.

The `load_unload` procedure is required by TetraMAX ATPG. If you define the scan enable information before you write the STIL file, TetraMAX ATPG automatically creates the `load_unload` procedure.

The scan chain length is required in standard STIL syntax, but is optional for STIL input files used by TetraMAX ATPG. When writing a STIL pattern file, TetraMAX ATPG determines the scan chain lengths and defines the correct length of each scan chain while writing STIL output.

[Example 1](#) shows the syntax used to define scan chains. This example consists of the STIL header followed by the `ScanStructures` keyword and four scan chains. In this example, the scan chains are named `c1` through `c4`. The `Procedures` section defines a procedure called `load_unload`, which consists of one test cycle (a "V {...}" vector statement). In the test cycle, the `CLOCK` and `RESETB` clocks are set to their off states and the `SCAN_ENABLE` port is driven high to enable the scan chain shift paths.

Example 1: Defining Scan Chain Loading and Unloading in the STL procedure file

```
STIL;
ScanStructures {
    ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
    ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
    ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
    ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}
Procedures {
    "load_unload" {
        V { CLOCK=0; RESETB=1; SCAN_ENABLE=1; }
    }
}
```

See Also

[JTAG/TAP Controller Variations for the load_unload Procedure](#)

Controlling Bidirectional Ports

During scan chain shifting defined by the `load_unload` procedure, the control logic for bidirectional ports sometimes operates at random states. This condition causes Z class DRC violations. You can prevent these violations by doing the following:

- Place a Z value on the bidirectional port, which turns off the ATE tester drive
- Enable a top-level control port, applied only for test mode, to globally disable all bidirectional drivers

Example 1 illustrates a design with a top-level bidirectional control port called `BIDI_DISABLE` (shown in bold). This example uses the `SignalGroups` section to define an ordered grouping of ports referenced by the label `bidi_ports`, thus facilitating assignment to multiple ports.

Example 1 Controlling Bidirectional Ports in the STL Procedure File

```
STIL;
SignalGroups {
    bidi_ports = '"D[0]" + "D[1]" + "D[2]" + "D[3]" + "D[4]" + "D[5]" + "D[6]" +
                 + "D[7]" + "D[8]" + "D[9]" + "D[10]" + "D[11]" + "D[12]" +
                 + "D[13]" + "D[14]" + "D[15]";
}
```

```

ScanStructures {
    ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
    ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
    ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
    ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}
Procedures {
    "load_unload" {
        V {
            CLOCK=0; RESETB=1; SCAN_ENABLE = 1;
            BIDI_DISABLE = 1;
            bidi_ports = ZZZZ ZZZZ ZZZZ ZZZZ;
        }
        V {}
        V { bidi_ports = \r4 1010 ; }
        Shift {
            V { _si#####; _so#####; CLOCK=P; }
        }
        V { CLOCK=0; RESETB=1; SCAN_ENABLE = 0; }
    }
}
MacroDefs {
    test_setup {
        V {TEST_MODE = 1; PLL_TEST_MODE = 1; PLL_RESET = 1;
           BIDI_DISABLE = 1; bidi_ports = ZZZZZZZZZZZZZZZZ; }
        V {PLL_RESET = 0; }
        V {PLL_RESET = 1; }
    }
}

```

You can use both the `load_unload` procedure and the `test_setup` procedure for bidirectional control. The control mechanisms for the `load_unload` procedure are as follows:

- You can add the following lines to the first test cycle:

```
BIDI_DISABLE = 1;
bidi_ports = ZZZZ ZZZZ ZZZZ ZZZZ;
```

Setting the `BIDI_DISABLE` port to 1 disables all bidirectional drivers in the design. Assigning Z states to the `bidi_ports` ensures that the ATE tester does not try to drive the bidirectional ports.

- You can also use an empty test cycle:

```
V{ }
```

The empty braces indicate that no signals are changing. This provides a cycle of delay between turning off bidirectional drivers with `BIDI_DISABLE=1` and forcing the bidirectional ports as inputs in the third cycle. This is not usually necessary, but illustrates one technique for adding delay using an empty test cycle.

- A third test cycle:

```
V{ bidi_ports = \r4 1010 ; }
```

In this case, the `bidi_ports` are driven to a non-Z state so that they do not float while the drivers are disabled. The `\r4` syntax indicates that the following string is to be repeated four times. In other words, the pattern applied to the `bidi_ports` group is 10101010101010.

In the `test_setup` procedure, the following line can be added to the first test cycle:

```
BIDI_DISABLE = 1; bidi_ports = ZZZZZZZZZZZZZZZ;
```

In this case, the `BIDI_DISABLE` port is forced high and the `bidi_ports` are set to a Z state.

Defining the Shift Procedure

The `Shift` procedure specifies how to shift the scan chains. It is placed within the `load_unload` procedure.

`Shift` is a recognized keyword to the STIL language and is not enclosed in quotation marks. The `_si` and `_so` names are predefined symbolic names used by TetraMAX ATPG to represent the list of scan inputs and scan outputs. "CLOCK" is the name of a clock port that affects scan chains. More than one clock port is often required.

The bold text shown in Example 1 defines the `Shift` procedure.

Example 1: STIL procedure file: Defining the Scan Chain Shift Procedure

```
STIL;
ScanStructures {
    ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
    ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
    ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
    ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}
Procedures {
    "load_unload" {
        V { CLOCK=0; RESETB=1; SCAN_ENABLE = 1; }
        Shift {
            V { _si=####; _so=####; CLOCK=P; }
        }
        V { CLOCK=0; RESETB=1; SCAN_ENABLE = 0; }
    }
}
```

The `Shift` procedure consists of a test cycle (`V`) in which the scan inputs `_si` are set from the next available stimulus data (`#`); the scan outputs `_so` are measured from the next available expected data (`#`); and the port `CLOCK` is pulsed (`P`). There are four `#` symbols, one for each scan chain defined.

When the `load_unload` procedure is applied, the `Shift` procedure is applied repeatedly as required to shift as many bits as are in the longest scan chain.

A test cycle is added after the `Shift` procedure to ensure that the clocks and asynchronous reset/set ports are at their off states. This is an optional cycle if all procedures start out by ensuring that the clocks and asynchronous set/reset ports are at the off state.

The `_si` and `_so` grouping names are expected by TetraMAX. They refer to the scan inputs and scan outputs. The STIL file output generated by TetraMAX completely describes the port names and ordering contained in the groupings `_si` and `_so`; you do not have to enter this information.

Defining the test_setup Procedure

The `test_setup` procedure defines all initialization sequences that a design needs for test mode or to ensure that the device is in a known state.

In Example 1, the `test_setup` procedure is highlighted in bold text. This example procedure consists of three test cycles:

- The first cycle sets the inputs `TEST_MODE`, `PLL_TEST_MODE`, and `PLL_RESET` to 1
- The second cycle changes `PLL_RESET` to 0
- The third cycle returns `PLL_RESET` to 1.

Example 1: Defining the test_setup Macro in the STL procedure file

```
STIL;
ScanStructures {
    ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
    ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
    ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
    ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}
Procedures {
    "load_unload" {
        V { CLOCK=0; RESETB=1; SCAN_ENABLE = 1; }
        Shift {
            V { _si=####; _so=####; CLOCK=P; }
        }
        V { CLOCK=0; RESETB=1; SCAN_ENABLE = 0; }
    }
}
MacroDefs {
    test_setup {
        V {TEST_MODE = 1; PLL_TEST_MODE = 1; PLL_RESET = 1; }
        V {PLL_RESET = 0; }
        V {PLL_RESET = 1; }
    }
}
```

If you need to initialize a port to X in the `test_setup` procedure, use the "N" STIL assignment character. An "X" character indicates that an output measure is performed and the result is masked.

You can use the `test_setup` procedure to perform several other tasks, including:

- Place a device in ATPG test mode
- Place clocks in their off states

- Initialize constrained ports
- Initialize bidirectional ports to Z
- Initialize JTAG TAP controllers
- Implement Loop statements (see the "Loop Statements" section).

Using Loop Statements

You can use loops in `test_setup` procedures, however you should limit their usage. If you use too many loops:

- The size of the `test_setup` procedure dramatically increases
- The time to simulate the clock pulses dramatically increases

You should represent only the necessary events required to initialize the device for ATPG efforts in the `test_setup` procedure. Loops that represent device test (for example, one million vectors to lock a PLL clock at test) are not appropriate or necessary in the ATPG environment when a PLL clock is a black box.

Vectors may be extracted before the Loop and after the Loop, and the Loop count decremented as appropriate for each extracted vector.

Each extracted vector must contain the exact same sequence of clocks as specified in the vector inside the Loop statement. Empty vectors (no events) may appear between the vectors that contain clock pulses — but it is critical that any vector that contains a signal assignment, match in order with the signal assignments for the vector inside the Loop. Otherwise, this extracted vector will not be recognized as consistent with the internal vector, and the extracted vector will not be "re-rolled" into the Loop count, causing DRC analysis errors.

The only supported contents of a Loop in a *setup procedure are C {} condition statements, V {} vectors, or WaveformTable "W" statements.

Example 1:

```
MacroDefs {
    "test_setup" {
        W "_default_WFT_";
        C {"all_inputs" = NNN; "all_outputs" = \r6 X; }
        Loop 10 { V { "s_in"=0; "clk"=P; } }
    }
}
```

Example 2:

```
MacroDefs {
    "test_setup" {
        W "_default_WFT_";
        V { "CK"=0; }
        Loop 4 { V { "s_in"=0; "clk"=P; } }
    }
}
```

Loops in STIL may contain other references (for example, calls to other macros and procedures). These constructs are not supported within the setup environment.

Predefined Signal Groups in STIL

A SignalGroup is a method in STIL that describes a list of pins using a symbolic label. You can use symbolic labels to reference a large number of pins without excessive typing.

TetraMAX ATPG accepts the following predefined SignalGroups:

- `_in` = input pins
- `_out` = output pins
- `_io` = bidirectional pins
- `_pi` = inputs + bidirectional pins
- `_po` = outputs + bidirectional pins
- `_si` = scan chain inputs
- `_so` = scan chain outputs

If your STIL DRC description defines a symbolic group with the same name as the predefined TetraMAX groups, your definition supersedes the predefined definition.

Defining Basic Signal Timing

You can define clocks and other pulsed ports, such as asynchronous sets and resets, in the STL procedure file. This is an alternative method to using the Edit Clocks dialog box in the TetraMAX GUI or the `add_clocks` command (for more information, see "[Declaring Clocks](#)."

You do not need to define signal timing to perform DRC or to generate patterns. However, timing definition is necessary for writing patterns that require meaningful timing. If you do not explicitly define the signal timing, TetraMAX ATPG uses a set of default values.

You should avoid editing signal timing values in ATPG-generated pattern files because it causes simulation mismatches or ATE mismatches. Make sure you define signal timing in the STL procedure file and run DRC with the same STL procedure file before generating hand-off patterns with ATPG.

Example 1: STL procedure file: Defining Timing

```

1. STIL;
2. UserKeywords PinConstraints;
3. PinConstraints { "TEST_MODE" 1; "PLL_TEST_MODE" 1; }
4. SignalGroups {
5.   bidi_ports '"D[0]" + "D[1]" + "D[2]" + "D[3]" + "D[4]" + "D[5]" +
+ "D[6]" + "D[7]" + "D[8]" + "D[9]" + "D[10]" + "D[11]" + "D[12]" +
"D[13]" + "D[14]" + "D[15]" ';
6.   input_grp1 'SCAN_ENABLE + BIDI_DISABLE + TEST_MODE + PLL_TEST_-
MODE' ;
7.   input_grp2 'SDI1 + SDI2 + DIN + "IRQ[4]"' ;
8.   in_ports 'input_grp1 + input_grp2';
9.   out_ports 'SDO2 + D1 + YABX + XYZ';
10. }
11. Timing {
12.   WaveformTable "BROADSIDE_TIMING" {

```

```

13. Period '1000ns';
14.     Waveforms {
15.         CLOCK { P { '0ns' D; '500ns' U; '600ns' D; } } // clock
16.         CLOCK { 01Z { '0ns' D/U/Z; } }
17.     RESETB { P { '0ns' U; '400ns' D; '800ns' U; } } /
18.         / async reset
19.     RESETB { 01Z { '0ns' D/U/Z; } }
20.     input_grp1 { 01Z { '0ns' D/U/Z; } }
21.     input_grp2 { 01Z { '10ns' D/U/Z; } }
22.         // outputs are to be measured at t=350
23.     out_ports { HLT { '0ns' X; '350ns' H/L/T/X; } }
24.         // bidirectional ports as inputs are forced at t=20
25.     bidi_ports { 01Z { '0ns' Z; '20ns' D/U/Z; } }
26.         // bidirectional ports as outputs are measured at t=350
27.     bidi_ports { X { '0ns' X; } }
28.     bidi_ports { HLT { '0ns' X; '350ns' H/L/T; } }
29. }
30. } // end BROADSIDE_TIMING
31. WaveformTable "SHIFT_TIMING" {
32.     Period '200ns';
33.     Waveforms {
34.         CLOCK { P { '0ns' D; '100ns' U; '150ns' D; } }
35.         CLOCK { 01Z { '0ns' D/U/Z; } }
36.         RESETB { P { '0ns' U; '20ns' D; '180ns' U; } }
37.         RESETB { 01Z { '0ns' D/U/Z; } }
38.         in_ports { 01Z { '0ns' D/U/Z; } }
39.         out_ports { X { '0ns' X; } }
40.         out_ports { HLT { '0ns' X; '150ns' H/L/T; } }
41.         bidi_ports { 01Z { '0ns' Z; '20ns' D/U/Z; } }
42.         bidi_ports { X { '0ns' X; } }
43.         bidi_ports { HLT { '0ns' X; '100ns' H/L/T; } }
44.     }
45.     } // end SHIFT_TIMING
46. }
47. ScanStructures {
48.     ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
49.     ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
50.     ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
51.     ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
52. } // end scan structures
53. Procedures {
54.     "load_unload" {
55.         W "BROADSIDE_TIMING" ;
56.         V {CLOCK=0; RESETB=1; SCAN_ENABLE=1; BIDI_DISABLE=1;
57.             bidi_ports = \r16 Z;}
58.         V {} bidi_ports = \r4 1010 ;
59.         Shift {
60.             W "SHIFT_TIMING" ;
61.             V { _si=###; _so=###; CLOCK=P; }
62.         }
63.     }
64. }

```

```

59. W "BROADSIDE_TIMING" ;
60. V { CLOCK=0; RESETB=1; SCAN_ENABLE=0; }
61. } // end load_unload
62. } //end procedures
63. MacroDef {
64.     "test_setup" {
65.         W "BROADSIDE_TIMING" ;
66. V {TEST_MODE = 1; PLL_TEST_MODE = 1; PLL_RESET = 1;
67. BIDI_DISABLE = 1; bidi_ports = ZZZZZZZZZZZZZZ; }
68. V {PLL_RESET = 0; }
69. V {PLL_RESET = 1; }
70. } // end test_setup
71. } //end procedures

```

Lines were added for the following purposes:

- Lines 6–9: Defines some additional signal groups so that timing for all inputs or outputs can be defined in just a few lines, instead of explicitly naming each port and its timing.
- Lines 12–27: This is a waveform table with a period of 1000 ns that defines the timing to be used during nonshift cycles.
- Lines 28–42: This is another waveform table, with a period of 200 ns, that defines the timing to be used during shift cycles.
- Line 52: Addition of the W statement ensures that the BROADSIDE_TIMING is used for V cycles during the load_unload procedure.
- Line 57: Addition of the W statement ensures that the SHIFT_TIMING is used during application of scan chain shifting.
- Line 65: Causes the test_setup macro to use BROADSIDE_TIMING.

Defining Pulsed Ports

You can define pulsed ports for clocks and asynchronous sets and resets using the Add Clocks dialog box in the TetraMAX GUI, the add_clocks command, or by specifying an optional section in the STL procedure file.

The bold text in Example 1 defines two pulsed ports, **CLOCK** and **RESETB** in the STL procedure file. This specification adds a **Timing { . . . }** section and a **WaveformTable** definition with the special-purpose name recognized by TetraMAX, **_default_WFT_**.

Example 1: STL procedure file: Defining Pulsed Ports

STIL;

```

Timing {
  WaveformTable "_default_WFT_" {
    Period '100ns';
    Waveforms {
      CLOCK { P { '0ns' D; '50ns' U; '80ns' D; } }
      CLOCK { 01Z { '0ns' D/U/Z; } }
      RESETB { P { '0ns' U; '10ns' D; '90ns' U; } }
      RESETB { 01Z { '0ns' D/U/Z; } }
    }

```

```

        }
    }

ScanStructures {
    ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
    ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
    ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
    ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}

Procedures {
    "load_unload" {
        V {
            CLOCK=0; RESETB=1; SCAN_ENABLE = 1;
            BIDI_DISABLE = 1;
            bidi_ports = ZZZZ ZZZZ ZZZZ ZZZZ;
        }
        V {}
        V { bidi_ports = \r4 1010 ; }
        Shift {
            V { _si####; _so####; CLOCK=P; }
        }
        V { CLOCK=0; RESETB=1; SCAN_ENABLE = 0; }
    }
}

MacroDefs {
    test_setup {
        V {TEST_MODE = 1; PLL_TEST_MODE = 1; PLL_RESET = 1;
           BIDI_DISABLE = 1; bidi_ports = ZZZZZZZZZZZZZZZ; }
        V {PLL_RESET = 0; }
        V {PLL_RESET = 1; }
    }
}
}

```

This timing definition has the following features:

- The period of the test cycle is 100 ns.
- The following line defines the port `CLOCK` as a positive-going pulse that starts each cycle at a low value (`D` = force down), transitions up (`U` = force up) at an offset of 50 ns into the cycle, then transitions down at an offset of 80 ns:

```
CLOCK { P { '0ns' D; '50ns' U; '80ns' D; } }
```

- The next line indicates that for test cycles in which the `CLOCK` port has a constant value, the change to that value occurs at an offset of 0 ns into the test cycle:

```
CLOCK { 01Z { '0ns' D/U/Z; } }
```

- The following lines define the port `RESETB` as a negative-going pulse:

```
RESETB { P { '0ns' U; '10ns' D; '90ns' U; } }
RESETB { 01Z { '0ns' D/U/Z; } }
```

Note that `RESETB` is defined nearly identically to `CLOCK`, with two exceptions:

- First, it starts each pulse cycle in the U (force up) position, transitions to D (force down), and then to U again.
- Second, the timing is slightly different, with the first transition at an offset of 10 ns into the cycle and the last transition at an offset of 90 ns.

Selecting Strobed or Windowed Measures in STIL

Some testers and vendors prefer a windowed measure for selecting timing in STIL. For this approach, the outputs are compared continuously for a window of time against the expected values instead of at a single time.

STIL supports the definition of windowed measures by using some slightly different syntax involving lowercase Waveform Events. The first following example illustrates strobed comparisons that occur at an offset of 450ns into each cycle.

```
Timing {
    WaveformTable "STROBED_COMPARE" {
        Period '1000ns';
        Waveforms {
            clocks { P { '0ns' D; '500ns' U; '600ns' D; } }
            input_ports { 01Z { '0ns' D/U/Z; } }
            out_ports { X { '0ns' X; '450ns' X; } }
            out_ports { HLT { '0ns' X; '450ns' H/L/T; } }
            bidi_ports { X { '0ns' X; } }
            bidi_ports { 01Z { '0ns' D/U/Z; } }
            bidi_ports { HLT { '0ns' X; '450ns' H/L/T; } }
        }
    }
}
```

This second example uses a windowed comparison for the group "out_ports" that compares the outputs between the offsets of 450 nS and 490 nS into each test cycle. Notice how the standard STIL WaveformChars of "H/L/T" have been replaced by lowercase STIL Waveform Events of "h", "l", "t". This indicates to TetraMAX ATPG that windowed measures are required.

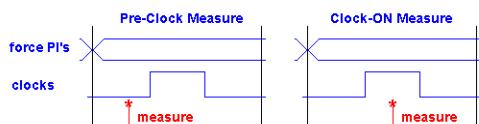
```
Timing {
    WaveformTable "WINDOW_COMPARE" {
        Period '1000ns';
        Waveforms {
            clocks { P { '0ns' D; '500ns' U; '600ns' D; } }
            input_ports { 01Z { '0ns' D/U/Z; } }
            out_ports { X { '0ns' X; } }
            out_ports { HLT { '0ns' X; '450ns' h/l/t; '490ns' X; } }
            bidi_ports { X { '0ns' X; } }
            bidi_ports { 01Z { '0ns' D/U/Z; } }
            bidi_ports { HLT { '0ns' X; '450ns' H/L/T; } }
        }
    }
}
```

{}

TetraMAX ATPG supports the definition of windowed measure in the STIL timing block and if STIL or WGL output patterns are written, then this timing definition is carried into the output patterns. However, when writing Verilog or VHDL patterns, the patterns will contain a strobed measure and the call for a windowed measure is not supported and is ignored.

Supporting Clock ON Patterns in STIL

The default patterns generated by TetraMAX ATPG use a preclock measure. Certain types of faults on combinational paths involving clock pins and primary outputs require a different style of pattern, called a "Clock ON" pattern, where the measure is performed during the interval in which the clock is asserted. This difference is shown graphically below and these types of faults in the design are signaled by the presence of C17 DRC violations.



TetraMAX ATPG does not generate this additional style of patterns by default, because it is not supported by all testers. Your target tester must either support multiple waveform timings and dynamically switching between them on a pattern by pattern basis, or you must be willing to create patterns that contain clock-on measure and preclock measure and write them into separate pattern blocks before use (the `-type` option of the `write_patterns` command is handy for this).

- The first step necessary to support the generation of Clock-On patterns is to edit your DRC file and to create a unique timing definition to be used for the clock-on measure patterns. This is usually accomplished by copying the existing or default waveform timing and adjusting the measure time of outputs to occur within the time interval where the clock is asserted.
- After creating a unique waveform timing definition modify or create the non-clocking capture procedure named "capture" and have it reference the clock-on waveform timing.
- Next enable the generation of clock-on measure patterns by use of the `-allow_clockon_measures` option of the `set_atpg` command.
- Finally, use the modified DRC file in your `run_drc` command.

When the ATPG algorithm generates patterns, it will reference the defined waveform timing of the non-clocking capture procedure for any patterns created that require the clock-on measure. These patterns have a recognizable label when reported with the `-types` option of the `report_patterns` command.

Note: It is also possible for the ATPG algorithm to create regular patterns that do not require a clock. If this occurs, these patterns will also reference the defined timing of the "capture" procedure. Usually only a few patterns are generated for any particular design that do not require a clock be used. These patterns should work but can increase the amount of dynamic timing switches in your tester. If this is a cone timern, then explore the `-clock -one_hot` option of the `set_drc` command as a way to inhibit the generation of non-clocking patterns.

The following example defines a unique timing set for use by the clock-on patterns. The timing of CLOCK_ON is identical to PRE_CLOCK, except that the measure time has been moved from 40ns (preclock) to 60ns (clock asserted).

```
:
:

Timing {
    WaveformTable "PRE_CLOCK" {
        Period '100ns';
        Waveforms {
            clocks { P { '0ns' D; '50ns' U; '80ns' D; } }
            clocks { 01Z { '0ns' D/U/Z; } }
            _in { 01ZN { '0ns' X; '40ns' L/H/T/X; } }
            _out { LHZX { '0ns' X; '40ns' L/H/T/X; } }
            _io { LHZX { '0ns' X; '40ns' L/H/T/X; } }
            _io { 01ZN { '0ns' D/U/Z/N; } }
        }
    }
    WaveformTable "CLOCK_ON" {
        Period '100ns';
        Waveforms {
            clocks { P { '0ns' D; '50ns' U; '80ns' D; } }
            clocks { 01Z { '0ns' D/U/Z; } }
            _in { 01ZN { '0ns' D/U/Z/N; } }
            _out { LHZX { '0ns' X; '60ns' L/H/T/X; } }
            _io { LHZX { '0ns' X; '60ns' L/H/T/X; } }
            _io { 01ZN { '0ns' D/U/Z/N; } }
        }
    }
}

:
:

capture_CLK {
    W PRE_CLOCK;
    V { _pi=\r13 # ; _po=\j \r9 X ; }
    V { _po=\r9 # ; }
    V { CLK=P; _po=\j \r9 X ; }
}

capture {
    W CLOCK_ON; // reference the alternate timing definition
    V { _pi=\r13 # ; _po=\j \r9 X ; }
    V { _po=\r9 # ; }
}
:
:\line
```

Defining the End-of-Cycle Measure

The preferred ATPG cycle has the measure point coming before any clock events in the cycle. However, an end-of-cycle measure is possible with a few minor adjustments to the STL procedure file.

The STL procedure file in [Example 1](#) illustrates the two changes that allow TetraMAX ATPG to accommodate an end-of-cycle measure:

- The timing of the measure points defined in the `Waveforms` section is adjusted to occur after any clock pulses.
- A measure scan out ("`_so"="###`") is placed within the `load_unload` procedure and before the `Shift` procedure.

In addition, the capture procedures must be either the default of three cycles or a two-cycle procedure where the force/measure events occur in the first cycle and the clock pulse occurs in the second.

Example 1: End-of-Cycle Measure

```

Timing {
    WaveformTable "BROADSIDE_TIMING" {
        Period '1000ns';
        Waveforms {
            measures { X { '0ns' X; } }
            CLOCK { P { '0ns' D; '500ns' U; '600ns' D; } }
            CLOCK { 01Z { '0ns' D/U/Z; } }
            RESETB { P { '0ns' U; '400ns' D; '800ns' U; } }
            RESETB { 01Z { '0ns' D/U/Z; } }
            input_grp1 { 01Z { '0ns' D/U/Z; } }
            input_grp2 { 01Z { '10ns' D/U/Z; } }
            bidi_ports { 01Z { '0ns' Z; '20ns' D/U/Z; } }
            measures { HLT { '0ns' X; '950ns' H/L/T; } }
        }
    }
    WaveformTable "SHIFT_TIMING" {
        Period '200ns';
        Waveforms {
            measures { X { '0ns' X; } }
            CLOCK { P { '0ns' D; '100ns' U; '150ns' D; } }
            CLOCK { 01Z { '0ns' D/U/Z; } }
            RESETB { P { '0ns' U; '20ns' D; '180ns' U; } }
            RESETB { 01Z { '0ns' D/U/Z; } }
            in_ports { 01Z { '0ns' D/U/Z; } }
            bidi_ports { 01Z { '0ns' Z; '20ns' D/U/Z; } }
            measures { HLT { '0ns' X; '190ns' H/L/T; } }
        }
    }
    ScanStructures {
        ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
    }
}

```

```

    ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
    ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
    ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}
Procedures {
    "load_unload" {
        W "BROADSIDE_TIMING";
        V {CLOCK=0; RESETB=1; SCAN_ENABLE=1;
            BIDI_DISABLE=1; bidi_ports = \r16 Z;}
        V { _so = ##### ; }
        V { bidi_ports = \r4 1010 ; }
        Shift {
            W "SHIFT_TIMING";
            V { _si=#####; _so=#####; CLOCK=P; }
        }
    }
}

```

Defining Capture Procedures in STIL

Capture procedures offer the flexibility to control the timing of the force primary inputs and bidis, measure primary outputs and bidis, and, optionally, a capture operation with a functional (nonscan) clock. These three events must be in the order shown, and may be arranged in three, two, or one tester cycles (Vectors). Different capture procedures are used for each capture clock, as well as a non-clock capture procedure.

NOTE: Each port can only be forced one time among all vectors in the capture procedure.

The following examples are from a design with CLK and RSTB defined as clock ports. The "capture_CLK" procedure illustrates forcing PI's, measuring PO's, and pulsing the clock in the same cycle. The "capture_RSTB" illustrates using two cycles.

```

"capture_CLK" {
    W "_default_WFT_";
    V { "_pi"=\r 12 #; "_po"=\r 8 #; "CLK"=P; }
}

"capture_RSTB" {
    W "_default_WFT_";
    "force_and_measure": V { "_pi"=\r 12 #; "_po"=\r 8 #; }
    "pulse": V { "RSTB"=P; }
}

"capture" {
    W "_default_WFT_";
    "forcePI": V { "_pi"=\r 12 #; }
    "measurePO": V { "_po"=\r 8 #; }
}

```

The default algorithm for combinational ATPG produces an event order of: force inputs, measure outputs, pulse clocks (optional). If you should need to produce an end-of-cycle

measure or postclock measure instead of this preclock measure, you will need to use a specific 2-cycle capture procedure with the following event order: cycle 1 {force inputs, measure outputs}, cycle 2 {pulse clocks}. You will also need to adjust the defined timing on the ports. An example of this style follows:

```
"capture_CLK" {
    W "spec_timing_set";
    V { "_pi"=\r 12 # ; "_po"=\r 8 #; }
    V { "_po"=\r 8 X ; CLK=P; }
}
```

TetraMAX ATPG defaults to the first WaveformTable encountered in the file if it is not specified in the sequential_capture procedure (if present) or defined in a capture procedure in the DRC file. This WaveformTable can be, but does not need to be named "_default_WFT_". In other words, if your STL procedure file had two waveform tables, say "_first_WFT_" followed later by "_default_WFT_", and you did not list your capture clocks in the STL procedure file, then TetraMAX ATPG would use "_first_WFT" for waveform timing information.

Limiting Clock Usage

You might need to limit the clocks used by the ATPG algorithm during the capture procedures. For example, sometimes only the TCK clock should be used or the TAP controller state machine will get out of step. If you need to restrict usage of defined clocks to a single clock, use the `-clock` option of the `set_drc` command:

```
DRC-T> set_drc -clock TCK
```

This option restricts the ATPG algorithm to use only the specified clock for capture.

Defining Constrained Primary Inputs

You can use the STIL Procedure file to define constraints on ports. This is an alternative method to using the `add_pi_constraints` command or the Constraints menu in the TetraMAX GUI.

The following example is a fragment of a STIL file in which the "F{...}" or Fixed construct is used to define a fixed port condition. The STIL specification defines that this Fixed relationship persists only within the procedure in which it occurs. A TetraMAX PI constraint applies to every capture procedure. Because of this difference, you should repeat the Fixed relationship in every capture procedure. If you don't, TetraMAX ATPG issues V12 warnings and continues as if the missing Fixed statements are present.

TetraMAX ATPG does not support use of the "F{...}" statement in the `test_setup` or `load_unload/shift` or other procedures. You must explicitly set any ports you want held at fixed values in these procedures. In the following example, the ports `TEST_MODE` and `PLL_TEST_MODE` are explicitly set in both the `load_unload` procedure and the `test_setup` procedure.

```
STIL;

ScanStructures {
    ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
    ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
    ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
    ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
```

```

}

Procedures {
    "load_unload" {
        V {
            CLOCK = 0; RESETB = 1;
            SCAN_ENABLE = 1;
            TEST_MODE=1; PLL_TEST_MODE=0;
        }
        Shift {
            V { _si=####; _so=####; CLOCK=P; }
        }
    }
    "capture_CLOCK" {
        F { TEST_MODE=1; PLL_TEST_MODE=0; }
        V { "_pi"=\r 12 # ; "_po"=\r 8 #; "CLOCK"=P; }
    }
    "capture_RESETB" {
        F { TEST_MODE=1; PLL_TEST_MODE=0; }
        V { "_pi"=\r 12 # ; "_po"=\r 8 #; "RESETB"=P; }
    }
    "capture" {
        F { TEST_MODE=1; PLL_TEST_MODE=0; }
        V { "_pi"=\r 12 # ; "_po"=\r 8 #; }
    }
}

MacroDefs {
    test_setup {
        V { TEST_MODE=1; PLL_TEST_MODE=0; CLOCK=0; }
    }
}

```

Defining Equivalent Primary Inputs

Primary inputs that need to be held at the same values or at complementary values can be defined in the STL procedure file as an alternative to using the `add_pi_equivalences` command. Example 1 shows how to define equivalent primary inputs in the STL procedure file.

Example 1: STL procedure file: Defining Equivalent Ports

```

Procedures {
    "capture" {
        W "_default_WFT_";
        E "ck1" "ck2";
        C { "all_inputs"=\r30 N; "all_outputs"=\r30 X ; } }
        V { "_pi"=\r35 # ; "_po"=\r30 # ; } }
}

```

```

"capture_ck1" {
    W "_default_WFT_";
    E "ck1" "ck2";
    C { "all_inputs"=\r30 N; "all_outputs"=\r30 X ; }
    "measurePO": V { "_pi"=\r35 #; "_po"=\r30 # ; }
    C { "InOut1"=X; "PA1"=X; "DOA"=X; "NA1"=X; "NA2"=X; }
    "pulse": V { "ck1"=P; }
}
"load_unload" {

```

Defining PO Masks

You can use the STIL Procedure file to define masks on output port measures. The following example shows a fragment from a STIL file in which the "F{...}" or Fixed construct is used to define a masked output condition by setting the expect value to X. This Fixed relationship definition persists only within the procedure in which it occurs, so it must be repeated in all capture procedures to properly define a PO Mask to TetraMAX ATPG.

TetraMAX ATPG does not support use of the "F{...}" statement in the test_setup or load_unload/shift or other procedures.

The "F{...}" statement is also used for defining PI Constraints.

```

STIL;

ScanStructures {
    ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
    ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
    ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
    ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}

Procedures {
    "load_unload" {
        V {
            CLOCK = 0; RESETB = 1;
            SCAN_ENABLE = 1;
            TEST_MODE=1;
        }
        Shift {
            V { _si####; _so####; CLOCK=P; }
        }
    }
    "capture_CLOCK" {
        F { YOUT = X; }
        V { "_pi"=\r 12 # ; "_po"=\r 8 #; "CLOCK"=P; }
    }
    "capture_RESETB" {
        F { YOUT = X; }
        V { "_pi"=\r 12 # ; "_po"=\r 8 #; "RESETB"=P; }
    }
}
```

```

"capture" {
    F { YOUT = X; }
    V { "_pi"=\r 12 # ; "_po"=\r 8 #; }
}
}

MacroDefs {
    test_setup {
        V { TEST_MODE=1; PLL_TEST_MODE=0; CLOCK=0; }
    }
}

```

Defining System Capture Procedures

TetraMAX ATPG uses a default capture procedure that defines how a declared clock port is pulsed for a system (nonscan) test cycle. This procedure uses the naming convention `capture_clockname` (where `clockname` is the clock port name).

The default system capture procedure usually contains three test cycles that perform the following tasks:

1. Force inputs
2. Measure outputs
3. Pulse the clock/set/reset port (optional)

If you defined ports named `CLOCK` and `RESETB` to be clocks using the `write_drc_file` command, the output file contains default capture procedures similar to those shown in Example 1.

Example 1: Default Capture Procedures

```

"capture_CLOCK" {
    W "_default_WFT_";
    "forcePI": V { "_pi"=\r10 # ; }
    "measurePO": V { "_po"=#####; }
    "pulse": V { "CLOCK"=P; }
}

"capture_RESETB" {
    W "_default_WFT_";
    "forcePI": V { "_pi"=\r10 # ; }
    "measurePO": V { "_po"=#####; }
    "pulse": V { "RESETB"=P; }
}

"capture" {
    W "_default_WFT_";
    "forcePI": V { "_pi"=\r10 # ; }
    "measurePO": V { "_po"=#####; }
}

```

If you want to use non-default timing or sequencing, copy the definitions for the capture procedures from the default output template into the `Procedures` section of your STL procedure file and edit the procedures.

TetraMAX ATPG defaults to the first WaveformTable it encounters in the file if a WaveformTable is not specified in the `sequential_capture` procedure when present or defined in a capture procedure in the DRC file. This WaveformTable can be named, for example, `"_default_WFT"`. If your STL procedure file has two waveform tables, `"_first_WFT"` and `"_default_WFT"`, and you do not list your capture clocks in the STL procedure file, TetraMAX uses `"_first_WFT"` for waveform timing information.

The bold text in Example 2 shows some typical modifications to the capture procedure files. In this case, the three cycles are merged into a single cycle and the non-default timing is specified using the `BROADSIDE_TIMING` statement.

Example 2: Modified Capture Procedure Examples

```
Procedures {
    "load_unload" {
        W "BROADSIDE_TIMING" ;
        V {CLOCK=0; RESETB=1; SCAN_ENABLE=1;
            BIDI_DISABLE=1; bidi_ports = \r16 Z;}
        V {}
        V { bidi_ports = \r4 1010 ; }
        Shift {
            W "SHIFT_TIMING" ;
            V { _si=####; _so=####; CLOCK=P; }
        }
        W "BROADSIDE_TIMING" ;
    }
    "capture_CLOCK" {
        W "BROADSIDE_TIMING";
        V { "_pi"=\r10 # ; "_po"=#####; "CLOCK"=P; }
    }
    "capture_RESETB" {
        W "BROADSIDE_TIMING";
        V { "_pi"=\r10 # ; "_po"=#####; "RESETB"=P; }
    }
    "capture" {
        W "BROADSIDE_TIMING";
        V { "_pi"=\r10 # ; "_po"=#####; }
    }
}
MacroDefs {
    "test_setup" {
        W "BROADSIDE_TIMING" ;
        V {TEST_MODE = 1; PLL_TEST_MODE = 1; PLL_RESET = 1;
            BIDI_DISABLE = 1; bidi_ports = ZZZZZZZZZZZZZZ; }
        V {PLL_RESET = 0; }
        V {PLL_RESET = 1; }
    }
}
```

Creating Generic Capture Procedures

This section describes how to write a set of single-cycle generic capture procedures. These procedures include: `multiclock_capture()`, `allclock_capture()`, `allclock_launch()`, and `allclock_launch_capture()`.

Generic capture procedures offer the following advantages:

- A single cycle capture procedure is efficient because it matches the event ordering (force PI, measure PO, pulse clock) in TetraMAX ATPG without any manual modifications.
- Stuck-at and at-speed ATPG can use a single common protocol file.
- The `stuck-at_default_WFT_WaveformTable` is used as a template for modifying the timing of the at-speed WaveformTables.

The following topics describe how to create generic capture procedures:

- [Generating Generic Capture Procedures](#)
- [Controlling Multiple Clock Capture](#)
- [Using Allclock Procedures](#)
- [Using load_unload for Last-Shift-Launch Transition](#)
- [Example Post-Scan Protocol](#)
- [Limitations](#)

See Also

[Defining a Sequential Capture Procedure](#)

[Defining a System Capture Procedure](#)

Generating Generic Capture Procedures

Generic capture procedures are generated by default when you specify the `write_drc_file` command (the `-generic_captures` option of the `write_drc_file` command is on by default). This command overrides the default-generated procedures (`capture_clockname`—except for an explicitly defined clocked capture procedure from a prior `run_drc` command. An unclocked capture procedure is not written. Also, the default timing is compatible with single-cycle capture procedures (a Z event is produced by default for the measure events H, L, T, and X, at time zero) when this option is used.

WaveformTables

If the default timing is defined, only one WaveformTable is generated in the output file, and all procedures will reference that same timing. If you want to create multiple WaveformTables ("`_launch_WFT_`", "`_capture_WFT_`", and "`_launch_capture_WFT_`"), use the `set_faults -model transition` command or the `set_faults -model path_delay` command before the `write_drc_file` command. These command specify that the data should be generated to cover this mode of operation.

Note the following requirements and scenarios:

- You must use generic capture procedures for Internal/External Clocking.
- Capture procedures using the internal clocks must use `_multiclock_capture_WFT_` procedures, which is appropriate because the PLL pulse trains are internally generated independently of the external timing. The timings that should be changed to get at-speed transition fault testing on the external clocks are in the `_allclock_WaveformTables` (`launch_WFT`, `capture_WFT`, `launch_capture_WFT`). Be careful not to change the Period or the timings of the Reference Clocks or else the PLLs might lose lock. Only change the rise and fall times of the external clocks. (For more information, see the “Creating Generic Capture Procedures” section in the *DFT Compiler User Guide*.)
- A two-clock transition fault test consists of a launch cycle using `_allclock_launch_WFT_` followed by a capture cycle using `_allclock_capture_WFT_`. The active clock edges of these two cycles should be moved close to each other. Make sure that the clock leading edge comes after the `all_outputs` strobe times, and adjust those times (for all values: L, H, T and X) in the `_allclock_capture_WFT_` if necessary. The remaining Waveform Table, `_allclock_launch_capture_WFT`, is only used when launch and capture are caused by opposite edges of the same clock. Here, the only important timing is from the clock leading edge to the same clock's trailing edge. In practice, this only happens in Full- Sequential ATPG. and in most cases it can be ignored.

Generating QuickSTIL File Flows

There are three scenarios to carefully consider when generating a QuickSTIL file flows:

- Running stuck-at STIL procedure file generation with no generic captures creates a set of default generic captures which use the default WaveformTables. All the generic captures are defined — not just the multiclock WaveformTables. But the `allclock_*` WaveformTables are defined at this time.
- Running transition STIL procedure file generation with no generic captures creates generic captures using all of the transition WaveformTables. All the generic captures are defined — not just the multiclock WaveformTables. But the `allclock_*` WaveformTables are defined at this time. You are responsible to update the timing needed for the at-speed timing WaveformTables.
- Running transition STIL procedure file generation with generic captures already present (for example, from a stuck-at flow), will not change or update the generic captures or WaveformTables already present in the original STIL procedure file. If you want transition timing and full WaveformTables in your STIL procedure file, you need to one of the following:
 - Edit and copy the “`_default_wft_`” multiple times, and change them to the transition WaveformTables and timing needed for their design.
 - Rerun DRC with the deletion of the `allclock_*` procedures, and regenerate default timing in transition mode for these procedures.

For an example that compares the different techniques, see [Figure 1](#) and [Figure 2](#). Note that the **blue** font follows the default WaveformTables, while the **green** font follows the at-speed WaveformTables.

Figure 1: Comparing Generic Captures Flows (Part 1)

SA-Fault Generic Captures Flow	TR-Fault Generic Captures Flow
<pre> Timing { WaveformTable *_default_WFT_* { Period '100ns'; Waveforms { "all inputs" { 0 { '0ns' D; } } "all inputs" { 1 { '0ns' U; } } "all inputs" { Z { '0ns' Z; } } "all outputs" { X { '0ns' Z; } } "all outputs" { H { '0ns' Z; '40ns' H; } } "all outputs" { T { '0ns' Z; '40ns' T; } } "all outputs" { L { '0ns' Z; '40ns' L; } } "CK" { P { '0ns' D; '45ns' U; '55ns' D; } } } } } </pre>	<pre> Timing { WaveformTable *_launch_capture_WFT_* { Period '100ns'; Waveforms { < same contents as in _default_WFT_ below User to modify timing specifics > } } WaveformTable *_launch_WFT_* { Period '100ns'; Waveforms { < same contents as in _default_WFT_ below User to modify timing specifics > } } WaveformTable *_capture_WFT_* { Period '100ns'; Waveforms { < same contents as in _default_WFT_ below User to modify timing specifics > } } WaveformTable *_default_WFT_* { Period '100ns'; Waveforms { "all inputs" { 0 { '0ns' D; } } "all inputs" { 1 { '0ns' U; } } "all inputs" { Z { '0ns' Z; } } "all outputs" { X { '0ns' Z; } } "all outputs" { H { '0ns' Z; '40ns' H; } } "all outputs" { T { '0ns' Z; '40ns' T; } } "all outputs" { L { '0ns' Z; '40ns' L; } } "CK" { P { '0ns' D; '45ns' U; '55ns' D; } } } } } </pre>

Figure 2: Comparing Generic Captures Flows (Part 2)

SA-Fault Generic Captures Flow	TR-Fault Generic Captures Flow
<pre> Procedures { "load unload" { W "default_NFT"; C { "CK"-0; "SO"-X; "SI"-0; "SEN"-0; } Shift { W "default_NFT"; V { "CK"-P; "SEN"-0; "SO"-#; "SI"-#; } } } "multiclock capture" { W "default_NFT"; V { "pi"\j \r7 #; "po"\j \r1 #; } } "allclock capture" { W "default_NFT"; V { "pi"\j \r7 #; "po"\j \r1 #; } } "allclock launch" { W "default_NFT"; V { "pi"\j \r7 #; "po"\j \r1 #; } } "allclock launch capture" { W "default_NFT"; V { "pi"\j \r7 #; "po"\j \r1 #; } } } </pre>	<pre> Procedures { "load unload" { W "default_NFT"; C { "CK"-0; "SO"-X; "SI"-0; "SEN"-0; } Shift { W "default_NFT"; V { "CK"-P; "SEN"-0; "SO"-#; "SI"-#; } } } "multiclock capture" { W "default_NFT"; V { "pi"\j \r7 #; "po"\j \r1 #; } } "allclock capture" { W "capture_NFT"; V { "pi"\j \r7 #; "po"\j \r1 #; } } "allclock launch" { W "launch_NFT"; V { "pi"\j \r7 #; "po"\j \r1 #; } } "allclock launch capture" { W "launch_capture_NFT"; V { "pi"\j \r7 #; "po"\j \r1 #; } } } </pre>

Controlling Multiple Clock Capture

You can control multiple clock capture by specifying a single general capture procedure, called `multiclock_capture`, in an STIL procedure file. This procedure enables you to map all capture behaviors irrespective of the number of clocks present, to use this procedure. In addition to supporting capture operations that contain multiple clocks, this procedure also eliminates the need to manually define a full set of clock-specific capture procedures or allow them to be defined by default.

There are several different methods associated with specifying multiple clock capture:

- [Multiple Clock Capture for a Single Vector](#)
- [Multiple Clock Capture for Multiple Vectors](#)
- [Using Multiple Capture Procedures](#)

Multiple Clock Capture for a Single Vector

The following example shows how to specify `multiclock_capture` for a single vector, which is the simplest form of this procedure:

```

Procedures {
  "multiclock_capture" {
    W "TS1";
    C { "_po"=\r9 X ; }
    V { "_po"=\r9 # ; "_pi"=\r11 # ; }
  }
}

```

Note that the single vector form does not require an explicit parameter to support the clock pulses because the clocks are always listed in the `_pi` arguments, and also in the `_po` arguments for any clocks that are bidirectional. It is strongly recommended that you specify an initial Condition statement to set the `_po` states to an X in this procedure. A default should be present in this procedure because not all calls from the Pattern data provide explicit output states.

As is the case with all capture procedures, the single-vector form of `multiclock_capture` requires the timing in the WaveformTable to follow the TetraMAX ATPG event order for captures. This means that all input transitions must occur first, all output measures must occur next, and all clock pulses must be defined as the last event.

Multiple Clock Capture for Multiple Vectors

As with standard capture procedures, the `multiclock_capture` procedure can consist of multiple vectors. In this case, you need to specify an additional argument to hold the variable clock-pulse information, as shown in the following example:

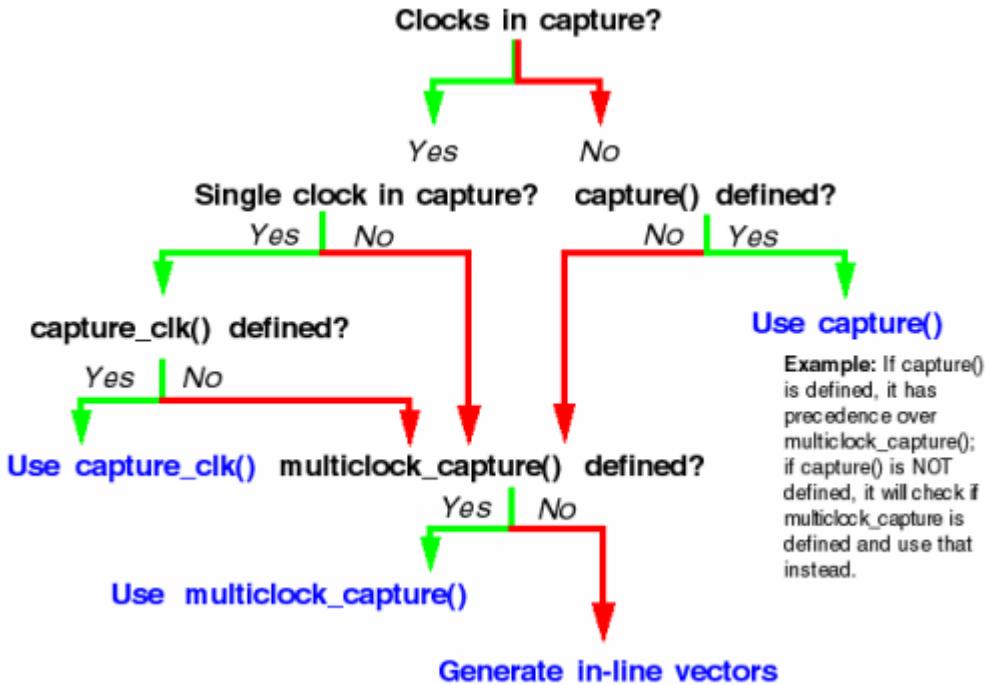
```
Procedures {
    "multiclock_capture" { // 2-cycle
        W "TS1";
        C { "_po"=\r9 X ; }
        V { "_pi"=\r11 # ; "_po"=\r9 # ; }
        C { "_po"=\r9 X ; }
        V {"_clks"= ##; }

    }
    "multiclock_capture" { // 3-cycle
        W "TS1";
        C { "_po"=\r9 X ; }
        V { "_pi"=\r11 # ; }
        V { "_po"=\r9 # ; }
        C { "_po"=\r9 X ; }
        V {"_clks"= ##; }

    }
}
```

Using Multiple Capture Procedures

Figure 3 shows how the `multiclock_capture` procedure is used when other `capture()` or `capture_clk()` procedures are defined.

Figure 3: Using Multiple Capture Procedures

Using Allclock Procedures

Allclock procedures directly replace specifically named WaveformTables (WFTs) by designating launch, capture, and launch_capture-specific timing parameters. This approach replaces an inline vector and WFT switch with a procedure call.

You can specify a set of allclock procedures for use in specific contexts in which a sequence of capture events supports a launch and capture operation. These sequences are generated in system clock-launched transition tests. Full-sequential patterns use inline vectors and not procedure calls. This is because the full-sequential operation has dependencies on the sequential_capture definition, which affects how capture operations will occur. Because inline vectors are used, transition and path delay timing is controlled by using fixed WaveformTable names (and not the allclock capture procedures) for full-sequential patterns.

Last-shift-launch contexts do not identify the launch or the capture operation. This means a last-shift-launch uses a standard capture procedure designation and does not reference allclock procedures even if they are present.

Standard capture procedure designation apply the multiclock_capture procedure in this situation if it is present (based on the presence of other capture procedures as diagrammed in Figure 11-4), and you may define the timing of the transition capture operation from this procedure. The timing of the launch operation is defined by the last vector of the load_unload procedure for a last-shift-launch context.

TetraMAX ATPG supports the following allclock procedures:

- `allclock_capture()` — Applies to tagged capture operations in launch/capture contexts only.
- `allclock_launch()` — Applies to tagged launch operations in launch/capture contexts only
- `allclock_launch_capture()` — Applies to tagged launch-capture operations only.

Specifying a Typical Allclock Procedure

By default, an allclock procedure applies to a single vector, although it doesn't have to carry the redundant clock parameter. An allclock procedure may reference any WFT for each operation. See the following example `allclock_capture()` procedure:

```
Procedures {
    "allclock_capture" {
        W "TS1";
        C { "_po"=\r9 X ; }
        V { "_po"=\r9 # ; "_pi"=\r11 # ; }
    }
}
```

Interaction of the Allclock and Multiple Clock Procedures

A defined `multiclock_capture()` procedure is always used for any capture operation that is not controlled by another defined procedure. This means that if an allclock procedure is not defined, the `multiclock_capture` procedure is applied in its place.

Interaction of Allclock Procedures and Named Waveform Tables

If an allclock procedure is defined, a named WFT is not applied on inline vectors even if it is defined. This is because allclock procedures always replace the generation of inline vectors in pattern data, and WFT names are supported only when inline vectors are generated.

It is strongly recommended that you define a sufficient set of allclock procedures for a particular context, even if the procedures are identical. This preserves pattern operation information that might otherwise be difficult to identify.

Using load_unload for Last Shift-Launch Transition

The `load_unload` procedure supports passing the pi data into the first vector of the `load_unload` operation. This means `load_unload` supports last shift-launch transition tests, presents the leading PI states at the time of the last shift operation (the launch), and supports transitioning those states.

Because of this implementation, it is important to provide sufficient information as part of the `load_unload` definition to permit standalone operation of the `load_unload` procedure. It is important to consider that the `load_unload` procedure is also used to validate scan chain tracing. Required states on inputs necessary to support scan chain tracing must be provided to this routine even if these signals are subsequently presented as parameterized values to the procedure, as shown in the following example:

```
Procedures {
    "load_unload" {
```

```

W "_default_WFT_";
C { "test_se"=1; } // required for scan chain tracing
V { "_pi"=\r34 #; }
Shift {
    V { "_ck"=\r3 P; "_si"=\r8 #; "_so"=\r8 #; }
}
}
}

```

Example Post-Scan Protocol

The following example shows a post-scan protocol containing generic capture procedures:

```

Procedures {
    "multiclock_capture" {
        W "_default_WFT_";
        C {
            "_po" = XXXX;
        }
        V {
            "_po" = #####;
            "_pi" = \r9 #;
        }
    }
    "allclock_capture" {
        W "_default_WFT_";
        C {
            "_po" = XXXX;
        }
        V {
            "_po" = #####;
            "_pi" = \r9 #;
        }
    }
    "allclock_launch" {
        W "_default_WFT_";
        C {
            "_po" = XXXX;
        }
        V {
            "_po" = #####;
            "_pi" = \r9 #;
        }
    }
    "allclock_launch_capture" {
        W "_default_WFT_";
        C {
            "_po" = XXXX;
        }
        V {
            "_po" = #####
        }
    }
}

```

```

        "_pi" = \r9 #;
    }
}

"load_unload" {
    W "_default_WFT_";
    C {
        "all_inputs" = NN0011NN1; // moved scan enable here
        "all_outputs" = XXXX;
    }
}

"Internal_scan_pre_shift" : V { "_pi" = \r9 #; }

Shift {
    V {
        "_clk" = PP11;
        "_si" = ##;
        "_so" = ##;
    }
}
}
}

```

Generic Capture Procedures Limitations

Note the following limitations related to generic capture procedures:

- WGL patterns are not supported if the multiclock_capture is multiple cycle or the clock (_clk) parameter is used; in this case, the WGL will not contain the clock pulses. WGL pattern format is only supported with single-cycle multiclock_captures that do not use a "clock" parameter (_clk).
 - WGL, VHDL, and legacy Verilog formats do not support 3-cycle generic capture procedures.
 - Using the DFT Compiler flow, the timing from the _default_WFT_ waveform table is copied to the allclock waveform tables (launch_WFT, capture_WFT, launch_capture_WFT). You will need to modify these multiple identical copies of this information with the correct timing before running at-speed ATPG.
 - TetraMAX transition-delay ATPG using the command set_delay -launch_cycle last_shift is not supported with the allclock capture procedures, only system_clock launch is supported.
 - MUXclock is not supported (D, E, P waveforms).

Defining Sequential Capture Procedures

A sequential capture procedure lets you customize the capture clock sequence applied to the device during Full-Sequential ATPG. For example, you can define the clocking sequence for a two-phase latch design, where CLKP1 is followed by CLKP2. Using a sequential capture procedure is optional, and it only affects Full-Sequential ATPG. For more information on ATPG modes, see ["ATPG Modes."](#)

With Full-Sequential ATPG and a sequential capture procedure, the relationships between clocks, tester cycles, and capture procedures can be more flexible and more complex. Using

Basic-Scan ATPG results in one clock per cycle, one clock per capture procedure, and one capture procedure per TetraMAX ATPG pattern. Using Full-Sequential ATPG and a sequential capture procedure, a cycle can be defined with one or more clocks, a capture procedure can be defined with any number of cycles, and an ATPG pattern can contain multiple capture procedures.

A sequential capture procedure can pulse multiple clocks, define clocks that overlap, and specify both optional and required clock pulses. A very long or complex sequential capture procedure is more computationally intensive than a simple one, which can affect the Full-Sequential ATPG runtime.

The following sections describe how to define sequential capture procedures:

- [Using Default Capture Procedures](#)
- [Using a Sequential Capture Procedure](#)
- [Sequential Capture Procedure Syntax](#)

Using Default Capture Procedures

By default, all ATPG modes use the same `capture_clockname` procedures described in ["Defining System Capture Procedures."](#) The Full-Sequential algorithm assumes the same order of events for each vector as the other algorithms. Under these default conditions, the Full-Sequential algorithm uses a fixed capture cycle consisting of three time frames, in which the tester does the following:

1. Loads scan cells, changes inputs, and measures outputs (optional)
2. Applies a leading clock edge
3. Applies a trailing clock edge, and optionally unloads scan cells

The Full-Sequential ATPG algorithm can choose any one of the available capture procedures for each vector, including the one that does not pulse any clocks. The algorithm can produce patterns using any sequence of these capture procedures to detect faults.

Using a Sequential Capture Procedure

To use a sequential capture procedure, add the `sequential_capture` procedure to the STIL file, then set the `-clock -seq_capture` option of the `set_drc` command, as shown in the following example:

```
DRC-T> set_drc -clock -seq_capture
```

Using this command option causes the Full-Sequential ATPG algorithm to use only the sequential capture procedure and to ignore the `capture_clockname` procedures defined by the STIL file or the `add_clocks` command. This option has no effect on the Basic-Scan and Fast-Sequential algorithms. [Sequential Capture Procedure in STIL](#) for more information.

Sequential Capture Procedure Syntax

A sequential capture procedure can be composed of one or more vectors. You can specify each vector using any of the following events:

- Force PI (must occur before clock pulses; required for the first vector)
- Measure PO (might occur before, during, or after clock pulses)
- Clock pulse (no more than one per clock input)

Each vector corresponds to a tester cycle. Be sure to consider any hardware limitations of the test equipment when you write the sequential capture procedure.

You can specify an optional clock pulse, which means that the clock is not required to be pulsed in every sequence. The Full-Sequential ATPG algorithm determines when to use or not use the clock. To define such a clock pulse, use the following statement:

```
V {"clock_name"="#; }
```

You can specify a required (mandatory) clock pulse, which means that the clock must be pulsed in every capture sequence. To define such a clock pulse, use the following statement:

```
V {"clock_name"=P; }
```

The following example shows a sequential capture procedure:

```
"sequential_capture"
W "default_WFT";
F {"test_mode"= 1; }
V {"_pi"= \r48 #; "_po"= \r12 X ; }
V {"CLK1"= P; CLK2= #; }
V {"CLK3"= P; }
V {"_po"= \r12 #; }
}
```

A sequential capture procedure can contain multiple tester cycles by supporting one or more vectors (multiple `V` statements), but there can be only one WaveformTable reference (`W` statement).

The procedure can have one force PI event per input per vector. Each force PI event must occur before any clock pulse events in that cycle. All inputs must be forced in the first vector of the sequential capture procedure; each input holds its state in subsequent vectors unless there is an optional change caused by another force PI event.

The procedure can have one required (`=P`) or optional (`=#`) clock pulse event per clock input per vector. Nonequivalent clocks can be pulsed at different times, and these clock pulses can overlap or not overlap.

The procedure can have one measure PO event per output per vector, which can occur anywhere in the cycle. However, no input or clock pulse events can be specified between the earliest and latest output measurements. The procedure also supports equivalence relationships and input constraints (`E` and `F` statements).

Sequential ATPG and simulation can model input changes only in the first time frame of each cycle. TetraMAX adds more time frames only as necessary to model discrete clock pulse events. It strobos outputs in no more than one of the existing time frames for each cycle.

Defining Reflective I/O Capture Procedures

A few ASIC vendors have special requirements for the application of the tester patterns when the design contains bidirectional pins. These vendors require the design to contain a global

disable control, available in ATPG test mode, which is used to turn off all potential bidirectional drivers. Further, the following sequence is required during the application of nonshift clocking and nonclocking capture procedures:

1. Force primary inputs with bidirectional ports enabled
2. Measure values on outputs as well as bidirectional ports
3. Disable bidirectional drivers
4. Use tester to force bidirectional ports with values measured in step 2
5. (Optional) Apply clock pulse

You identify toTetraMAX which port acts as the global bidirectional control using the `-bidi_control_pin` option of the `set_drc` command. For example, to indicate that the value 0 on the port `BIDI_EN` disables all bidirectional drives, enter the following command:

```
DRC-T> set_drc -bidi_control_pin 0 BIDI_EN
```

To define the corresponding reflective I/O capture procedures, you use % characters instead of # as data placeholders. In Example 1, each capture procedure measures primary outputs with the string %%%% instead of the string #####. A few cycles later, the string %% appears in an assignment of the symbolic group "`_io`", which is shorthand for the bidirectional ports.

The number of ports in the "`_po`" symbolic list is usually larger than the set of bidirectional ports referenced by "`_io`", so it is common for the %%%% string for "`_po`" to be longer than the string for the "`_io`" reference where the reflected data is reapplied. TetraMAX understands the correspondence required for proper pattern data.

Example 1: Capture Procedures With Reflective I/O Syntax

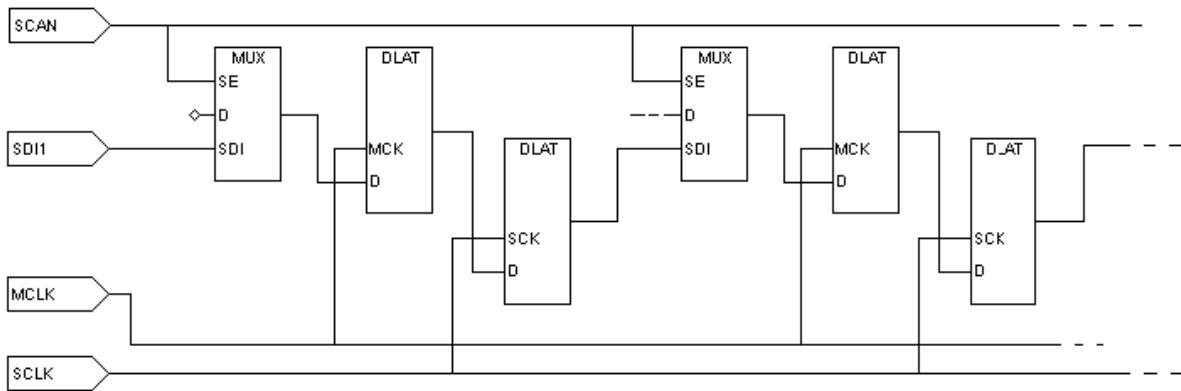
```
"capture_CLOCK" {
    W "_default_WFT_"; // force PI, measure PO, BIDI_EN=1
    V { "_pi"=\r10 # ; "_po"=%%%% ; } // disable bidis, mask PO
measures
    V { BIDI_EN=0; "_po"=XXXXXX; } // reflect bidis, pulse CLOCK
    V { "_io"=%%; CLOCK=P; }

}
capture_RESETB {
    W "_default_WFT_"; // force PI, measure PO, BIDI_EN=1
    V { "_pi"=\r10 # ; "_po"=%%%% ; } // disable bidis, mask
PO measures
    V { BIDI_EN=0; "_po"=XXXXXX; } // reflect bidis, pulse RESETB
    V { "_io"=%%; RESETB=P; }
}
capture {
    W "_default_WFT_";
    V { "_pi"=\r10 # ; "_po"=###### ; } // force PI, measure PO
    V { "_po"=XXXXX; } // mask measures
    V { } // pad procedure to 3 cycles
}
```

Using the master.observe Procedure

Use the `master.observe` procedure if the design has separate master-slave clocks to capture data into scan cells, as shown in [Figure 1](#). In system (nonscan) mode, after applying the `capture_clockname` procedure corresponding to the master clock, you must apply the slave clock to propagate the data value captured from the master latches to the slave latches. In the `master.observe` procedure, you describe how to pulse the slave clock and thereby observe the master.

Figure 1 Master-Slave Scan Chain



[Example 1](#) shows a `master.observe` procedure that uses two tester cycles. In the first cycle, all clocks are off except for the slave clock, which is pulsed. In the second cycle, the slave clock is returned to its off state.

Example 1 Example master.observe Procedure

```

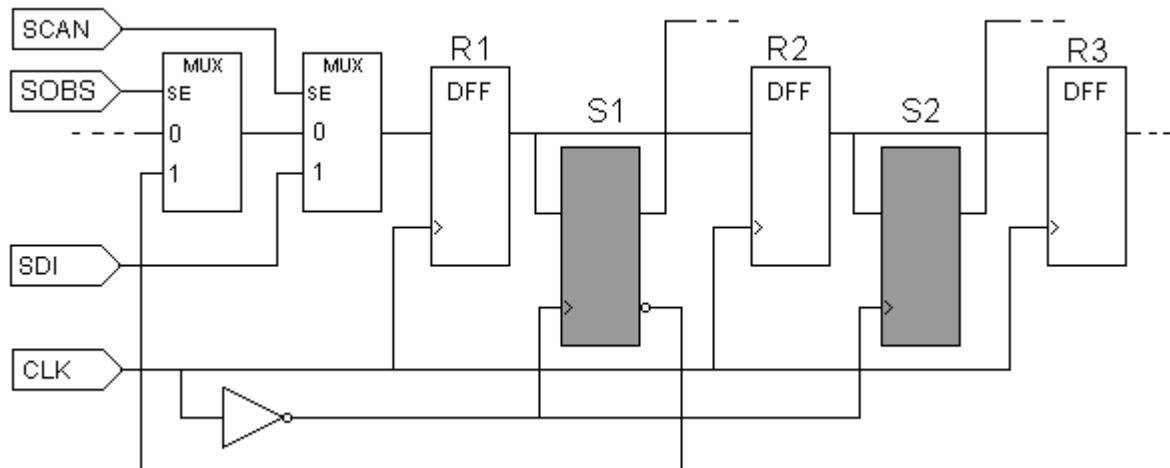
Procedures {
    "load_unload" {
        W "BROADSIDE_TIMING"
        V { MCLK=0; SCLK=0; RESETB=1; SCAN_ENABLE=1; BIDI_DISABLE=1; }
        V { bidi_ports = \r16 Z ; }
        Shift {
            W "SHIFT_TIMING";

            V { _si=##; _so=##; MCLK=P; SCLK=0; }
            V { MCLK=0; SCLK=P; }
        }
        V { SCLK=0; }
    }
    master.observe {
        W "BROADSIDE_TIMING";
        V { MCLK=0; SCLK=P; RESETB=1; }
        V { SCLK=0; }
    }
}
  
```

Using the shadow_observe Procedure

You use a `shadow_observe` procedure when a design has shadow registers and each shadow register output is observable at the scan cell to which it is a shadow. [Figure 2](#) shows two shadow registers, S1 and S2, which are shadows of R1 and R2, respectively. Shadow S1 has a combinational path back to its scan cell (R1) and would benefit from the definition of a `shadow_observe` procedure. Shadow S2 does not have a path back to R2 and would not benefit from a `shadow_observe` procedure.

Figure 2 A Shadow Register



[Example 3](#) shows a `shadow_observe` procedure that corresponds to [Figure 2](#). The first cycle places all clocks at off states and sets up the path from S1 back to R1 by setting `SCAN=0` and `SOBS=1`. The second cycle pulses the `CLK` port, and the third cycle turns off `CLK` and returns `SOBS` to zero.

Example 3 Example shadow_observe Procedure

```
Procedures {
    load_unload {
        V { CLK=0; RSTB=1; SCAN=1; }
        Shift { V { _si=###; _so=###; CLK=P; } }
        V { CLK=0; }
    }

    shadow_observe {
        V { CLK=0; RSTB=1; SCAN=0; SOBS=1; }
        V { CLK=P; }
        V { CLK=0; SOBS=0; }
    }
}
```

Using the `delay_capture_start` Procedure

You can use the `delay_capture_start` procedure to specify a wait period at the start of a capture operation. TetraMAX ATPG inserts calls to this procedure in the patterns at the end of each shift to establish the presence of the delay before the start of the capture operation. The first PI state present in the capture operation is asserted in this procedure, otherwise the transition time of slow-propagating signals will not occur before other capture events.

There are several different methods you can use to specify the parameters of this delay:

- Specify the number of vectors contained in the `delay_capture_start` procedure
- Control the period of the WaveformTable referenced by this procedure
- Control the number of times to insert this procedure at the end of shift using the `-use_delay_capture_start` option of the `write_patterns` command

The `delay_capture_start` procedure has several format requirements to operate as a simple `wait` statement. The default form of this procedure is as follows:

```
"delay_capture_start" {
    W "_default_timing_";
    C { "_po"=\rn X; }
    V { "_pi"=\rm #; }
}
```

Note the following:

- This procedure must contain a `Condition` statement that sets the `_po` to `x` at the start of the procedure. This ensures that any potential measure contexts at the end of the last operation are reset.
- This procedure must contain a call to the `_pi` group, with a parameter assignment of values, to ensure that the `_pi` states are applied at the start of the capture through this wait operation. No other signal assignment should be made in the `V` statement, as this will cause unpredictable results when the STIL patterns are used.
- The default form of this procedure calls the current default WaveformTable defined in the flow, and contains a single `Vector` statement. If this procedure is not defined in the incoming STIL procedure file, this default form is generated with the first use of the `-use_delay_capture_start` option of the `write_patterns` command. In this situation, this procedure is present in the `Procedure` block for all subsequent `write_patterns` or `write_drc_file` commands, although it will only be applied in the patterns if `-use_delay_capture_start` is specified.
- When the pattern set is written for transition patterns, the `set_delay -nopi` changes command inserts one leading delay cycle in the `delay_capture_start` operation and uses the `multiclock_capture` procedure to set the PI states into all transition capture operations. Therefore, the pattern set will already have one delayed capture start event present. The `delay_capture_start` procedure calls are be inserted after the number of requested delays exceed the number already present in the pattern data. If you require additional delays beyond those already present in the patterns, you need to set the `delay_capture_start` calls to a number greater than 1.

- If you define the `delay_capture_start` procedure in your STIL procedure file, it is present or defined in all subsequent STIL and WGL patterns that are written out.
 - If the `delay_capture_start` procedure is not defined in the STIL procedure file, then the first time that `write_patterns -use_delay_capture_start` is specified, it is created and defined in all STIL patterns that are written out. After it is created from the `write_patterns -use_delay_capture_start` command, this procedure will function just as if it were specified in the STL procedure file. However, it will not be called in the patterns unless the `write_patterns -use_delay_capture_start` command is specified.
 - The `delay_capture_start` procedure calls are eliminated when patterns are read back into TetraMAX. Each `write_patterns` command must use the `-use_delay_capture_start` option in order for this procedure present. While this procedure is not present on the internal pattern data, the presence of these function calls, and the generated vectors due to these procedures, are still counted during the pattern read-back operation. This allows cycle-based diagnostic flows to function with no changes. When the patterns are rewritten, you must set the `-use_delay_capture_start` option properly for every pattern write operation.
-

Using the `delay_capture_end` Procedure

You can use the `delay_capture_end` procedure to specify a wait period at the end of a capture operation. You will need to insert calls to this procedure in the patterns at the end of each capture to establish the presence of the delay before the start of the next LOAD operation.

There are several ways you can specify the parameters of this delay:

- Specify the number of vectors contained in the `delay_capture_end` procedure
- Control the period of the WaveformTable referenced by this procedure
- Control the number of times to insert this procedure at the end of capture by using the `-use_delay_capture_end` option of the `write_patterns` command.

The `delay_capture_end` procedure has several format requirements that enable it to operate as a simple wait statement. The default form of this procedure is as follows:

```
"delay_capture_end" {
    W "_default_timing_";
    C {"_po"=\rn X; }
    V{ "_pi"=\rm #; }
}
```

Note the following:

- If you define the `delay_capture_end` procedure in your STIL procedure file, it is present or defined in all subsequent STIL and WGL patterns that are written out.
- If the `delay_capture_end` procedure is not defined in the STIL procedure file, then the first time that `write_patterns -use_delay_capture_end` is specified, it is created and defined in all STIL patterns that are written out. After it is created from the `write_patterns -use_delay_capture_end` command, this procedure will

function just as if it were specified in the STIL procedure file. However, it will not be called in the patterns unless the `write_patterns -use_delay_capture_end` command is specified.

- The `delay_capture_end` procedure calls are eliminated when patterns are read back into TetraMAX. Each `write_patterns` command must use the `-use_delay_capture_end` option in order for this procedure present. While this procedure is not present on the internal pattern data, the PRESENCE of these function calls, and the generated vectors due to these procedures, are still counted during the pattern read-back operation. This allows cycle-based diagnostic flows to function with no changes. When the patterns are rewritten, you must set the `-use_delay_capture_end` option properly for every pattern write operation.
-

Using the `test_end` Procedure

You can define the `test_end` procedure or macro in the `Procedures` or `MacroDefs` sections of the STIL procedure file so that it is called at the end of every pattern block that is written out. This procedure must contain only signal drive assignments; measures are not supported.

When you define the `test_end` procedure or macro, TetraMAX places it at the end of STIL and WGL-formatted patterns only. When STIL or WGL patterns are read back, this procedure is removed. It will not be included in the internal pattern data. If patterns are rewritten, then the `test_end` procedure must be present in the STIL procedure file to place (or replace) it at the end of any new patterns.

Scan Padding Behavior

When scan chains are of unequal lengths and shifted in parallel, the shorter scan data must be padded or extended during the time after the short chain is exhausted but the shift procedure (or pattern operation) continues to complete the shifting of the longest chain.

Some TetraMAX output formats allow you to control the padding state from command-line options. For example, the combination of the `set wgl -pad` and `write_patterns -pad_character` commands control padding values for WGL files.

The STIL environment defines the padding values directly from the procedure definitions. For instance, for the `load_unload` procedure, the last assigned state, even if it was not applied in a Vector (it might have only been defined in a Condition statement) before the Shift block or first statement that contains an assignment of '#' to a scan signal, is the value used to pad that signal. If the scan signals are not assigned values before the Shift block, then when the `load_unload` procedure is written out, the inputs are assigned '0' and the outputs assigned 'X'. These defaults are sufficient for most environments, but sometimes additional data is specified in the procedure that might affect the padding behavior. When this occurs, it might become necessary to assign explicit values to signals in order for the STIL file to have the expected behavior.

Several DRC messages might be generated when STIL padding issues are detected in the patterns. These messages are all warnings, because consistency of the STIL data might not be a concern if your flow uses a different format. These warnings are either V12 (unexpected

item) or V14 (missing state) messages. All of these messages contain the text "STIL scan pad", to indicate they are being generated for STIL scan padding issues.

Certain design situations (for instance, reusing scan signals on multiple scanchains and making use of scan groups) limits the ability of these messages to detect all error conditions. Assigning an X to the scan outputs before the Shift block will define a correct test program, and is the most direct path to fixing padding problems.

TetraMAX tri-state checks might require that bidirectional scan outputs be assigned a Z WaveformCharacter, to trace a scan chain properly. This Z value enforces that the bidirectional output values are visible during the shift operation. However, during the scan operation it is likely that an X WaveformCharacter would be preferable, especially for padding. The Z reference is not wrong, but it is a "drive" waveform being assigned to a bidirectional being used as an output. Some environments might not like to see this drive value on a scan output. One way to define an X for pad operation is to place this X in a Condition statement before the first assignment to a '#'. For example:

```
V { .... sol=Z; ... }
C { sol=X; }
Shift { V { sol=# ... }
```

The DRC messages are generated only when a potential violation is detected. This will only happen for scan chains that are shorter than the longest chain in the shift operation. These checks will not occur on the longest chain in the design, even if the values assigned to the scan signals of that chain are incorrect, because the longest chain will not be padded.

These messages are not generated during the STL procedure file checks, because at this point there is not sufficient analysis of the scan chains to accomplish this checking. Therefore, these DRC messages are generated later in the process. These messages are V warnings but are not generated with the STL procedure file read, so be aware that additional V messages might be generated after the STL procedure file read has completed.

The specific messages, and how to address each, are explained as follows:

- V12, Scan output [name] is assigned a drive [state] before Shift; used as STIL scan pad
In this circumstance, a signal identified as a scan output, has been assigned a value commonly associated with an input signal. The [state] value is one of 0, 1, or N.
In most circumstances this is easily fixed by adding a C {} statement before the Shift block, and assigning the bidirectional scan output to an X value (or other preferred state). This was shown in the previous example.
- V14, Scan input [name] has no assignment before Shift, output [name] is [state]; missing STIL scan pad [input_state]
This warning is generated when a scan-output is assigned a known state, either H or L, before the Shift block, but the scan-input is not specified. The fix is to either assign the scan-output to an X before the Shift block (as done above), or to assign the appropriate input drive value (0 or 1) to the scan-input. Be aware that the appropriate value is a function of the parity of the scan chain, as discussed in the next message.

Note that this warning occurs only when scan outputs have been specified, but scan-inputs have not been assigned. The reverse condition, when scan-inputs are specified but scan-outputs have not been specified, is not a problem because the default handling of

scan-outputs will place an X on the outputs, making the environment insensitive to input states.

- V14, Scan output [name] is assigned [state] before Shift, input [name] is [state]; wrong STIL scan pad

V14, Scan input [name] is assigned [state] before Shift, output [name] is [state]; wrong STIL scan pad

These two warnings indicate the same condition, it just depends on the order of the signals in the design as to which you will see. These warnings are generated when both the scan-input and scan-output signals are assigned known values, but the scan data (and the parity of the scan chain) will cause this data to fail at test. In this circumstance, either the scan-input state must be changed, or the scan-output state, (but, obviously, not both) or the scan-output can be assigned an X value before the Shift.

DRC checks STIL padding to validate special conditions around bidirectional signals used as scan outputs.

- V14, Scan output bidi (signal) has no assignment before Shift; Z added for scan padding

During pattern write (of STIL data), the Z assignment is added to these signals, correcting the situation. You can always override the correction (and eliminate the V14) by specifying an assignment to this scan signal before the first # in the load_unload operation.

Using the Condition Statement in STIL

You can use the Condition statement, C{...}, to define force or measure values for defaults, without immediately resulting in an action. The conditioned values are deferred from being applied until a vector statement, V{...}, is encountered.

The WaveformTable used to translate the conditions is the waveform in effect at the time of the next Vector statement, not the waveform in effect at the time of the Condition statement.

Multiple Condition statements may be defined between vector statements. The last state defined in a Condition or vector statement for each pin is the state applied to that pin on the vector statement.

A vector statement defining a value to be applied to a pin will override any value defined in any preceding Condition statements.

Condition statements are useful when setup information is available; however, if this setup is applied as a vector, then the subsequent data becomes difficult to align. A typical situation in which to use a Condition statement is to enable the scan clocks preceding a Shift operation. Condition statements are also useful at the end of a macro to set up information for the return. Note that Condition statements would not be useful at the end of procedures because procedures return to the state before the procedure call and any condition information would be discarded.

```
STIL;

ScanStructures {
    ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
    ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
```

```

ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}

Procedures {
"load_unload" {
    C { CLOCK = 1; RESETB = 1; SCAN_ENABELE = 1;
    Shift {
        V { _si=####; _so=####; CLOCK=P; } // pulse shift clock
    }
}
}

MacroDefs {
test_setup {
    V { TEST_MODE = 1; CLOCK = 0; RESETB = 1; }
}
}

```

Excluding Vectors From Simulation

Passing a large number of loops through DRC negatively affects the performance of TetraMAX ATPG. You can use the `DontSimulate` statement in the `test_setup` procedure of the STL procedure file to eliminate loops with large count values and reduce activity, such as clock pulses, in the vectors of the loop.

The following sections describe how to use the `DontSimulate` statement:

- [Using the DontSimulate Statement for Loops and Reference Clocks](#)
- [Syntax and Example of the DontSimulate Statement](#)

Using the DontSimulate Statement for Loops and Reference Clocks

The `DontSimulate` statement identifies a PLL initialization or synchronization block of vectors for exclusion from DRC simulation. Because most PLL models are black boxes in TetraMAX ATPG, DRC results are not affected when clock pulses associated with these models are bypassed.

DRC procedures often require excessive memory resources when expanding the events of a loop. Also, the `test_setup` procedure causes excessive runtime when processing loop events, even though they do not affect the simulation. If the number of events in the `test_setup` procedure exceeds a 32-bit time value, the memory and runtime of all subsequent TetraMAX operations are affected.

The `DontSimulate` statement prevents the insertion of loop events into TetraMAX operations, while preserving the loops throughout the flow. This includes writing the loops in the patterns even though the loop events bypassed other TetraMAX operations.

In addition to loops, the `DontSimulate` statement applies to the following clocks:

- Reference clocks that are not identified as free-running clocks (because they affect other logic in the design).
- Reference clocks with a large number of pulsed vectors during setup. These pulses are not necessary for DRC because they are only driving the black box PLL logic.

You must make sure that any other logic driven by a reference clock uses all the necessary pulses.

Syntax and Example for Excluding Vectors

To exclude vectors from simulation:

1. Add the `UserKeywords DontSimulate` construct before the appropriate `MacroDefs` block in the STL procedure file. The syntax for this statement is as follows:

```
UserKeywords DontSimulate;
```

2. Add the `DontSimulate ATPGDRC` construct before the appropriate `Loop` statement in the STL procedure file. The syntax for this construct is as follows:

```
DontSimulate ATPGDRC;
```

The following example from an STL procedure file shows a typical implementation for excluding vectors from simulation:

```
UserKeywords DontSimulate;
MacroDefs {
  "pll_setup" {
    DontSimulate ATPGDRC;
    Loop 1000 {V {clock1=P;}}
  }
  "test_setup" {
    W "default_WFT_";
    C {"all_inputs"= ...; "all_outputs" = ...;}
    ...
  }
  Macro "pll_setup";
}
}
```

See Also

[Using Internal Clocking Procedures](#)

Defining Internal Clocks for PLL Support

TetraMAX ATPG supports two methods for defining internal clocks for PLL support:

- From the command line using the `add_clocks` command
- From an optional `ClockStructures` block of a STL procedure file .

The following is an example defining two internal clocks for PLL support by entering commands from the command line.

```
add_clocks 0 intclk3 -intclk -pll_source pllclk3 \
    -cycle { 0 clock_chain/cell[3]/Q 1 \
    1 clock_chain/cell[4]/Q 1 }
add_clocks 0 intclk1 -intclk -pll_source pllclk3 \
    -cycle { 0 clock_chain/cell[1]/Q 1 clock_chain/cell[0]/Q 0 \
    clock_chain/cell[2]/Q 1 clock_chain/cell[0]/Q 0 }
```

In the preceding example:

- intclk3 as an internal clock with offstate 0; its PLL source is pllclk3; intclk3 is pulsed in cycle 0 when cell 3 of chain clock_chain is 1, and is pulsed in cycle 1 when cell 4 of chain clock_chain is 1.
- intclk1 as an internal clock with offstate 0; its PLL source is pllclk3; intclk1 is pulsed in cycle 0 when cell 1 of chain clock_chain is 1 and cell 0 is 0, and is pulsed in cycle 1 when cell 2 of chain clock_chain is 1 and cell 0 is 0.

The following is an example showing the corresponding definitions in a STIL procedure file ClockStructures block.

```
Signals { "refclk1" In; "refclk2" In;
          "pllclk1" Pseudo; "pllclk2" Pseudo; "pllclk3" Pseudo;

          "intclk1" Pseudo; "intclk2" Pseudo; "intclk3" Pseudo;

}
SignalGroups { "all_inputs" = '... + "refclk1" + "refclk2" + ...
` }

Timing {
    WaveformTable "_default_WFT_" {
        Period '100ns';
        Waveforms {
            "all_inputs" { 01ZN { '0ns' D/U/Z/N; } }
            "refclk1" { P { '0ns' D; '45ns' U; '55ns' D; } }
            "refclk2" { P { '0ns' D; '45ns' U; '55ns' D; } }
        }
    }
}

UserKeywords ClockStructures;
ClockStructures {
    PLLStructures specpll {
        PLLCycles 2;
        Clocks {
            "refclk1" Reference; "refclk2" Reference;
            "pllclk1" PLL { Offstate 0 ; }
            "pllclk2" PLL { Offstate 0 ; }
            "pllclk3" PLL { Offstate 1 ; }
        }
    }
}
```

```

    "intclk1" Internal { Offstate 0; PLLSource "pllclk3";
        Cycle 0 "clock_chain/cell[0]/Q" 0;
        Cycle 0 "clock_chain/cell[1]/Q" 1;
        Cycle 1 "clock_chain/cell[0]/Q" 0;
        Cycle 1 "clock_chain/cell[2]/Q" 1;
    }
    "intclk3" Internal { Offstate 0; PLLSource "pllclk3";
        Cycle 0 "clock_chain/cell[3]/Q" 1;
        Cycle 1 "clock_chain/cell[4]/Q" 1;
    }
}
}
}

```

The following is a template for a generic STL procedure file ClockStructures block.

```

UserKeywords ClockStructures;
ClockStructures {
    (PLLStructures struct_name {
        (PLLCycles integer ;)
        (RefCycles integer ;)
        (Clocks {
            (sig_name <Reference | PLL| Internal> ;)*
            (sig_name <Reference | PLL| Internal> {
                (Offstate <0|1> ;)
                (PLLSource sig_name ;)
                (Cycle integer {AlwaysOn| AlwaysOff} ;)*
                (Cycle integer {net_or_pin_name <0|1>}+ ;)*
            })*
        })*
    })*
}

```

Where:

PLLcycles specifies the number of PLL clock cycles supported per load. This block is required if Cycle constructs are used. The **PLLcycles** block must precede all Cycle constructs.

`RefCycles` specifies the minimum number of system cycles each pattern must have.

`Clocks` defines the clocks in the `PLLStructures` block. The `sig_name` construct identifies the clock name and type. A type is required and must be one of the values shown. The `Offstate` construct is syntactically optional, but semantically required for all but reference clocks, and must be 0 or 1 (the offstate for reference clocks is derived from a `Waveformtable`). The `PLLSource` construct is used for internal clocks and identifies the corresponding `PLL` clock

source. The `Cycle` construct is used for internal clocks and identifies the corresponding control nets and their values.

Specifying an On-Chip Clock Controller Inserted by DFT Compiler

This section describes the process for specifying an OCC controller inserted by the `insert_dft` command in DFT Compiler. For information on signal requirements, see the "On-Chip Clocking Support" chapter in the *DFT Compiler User Guide*.

The following commands are used for specifying a default OCC controller inserted by DFT Compiler (**Note:** For user-defined OCC controllers, the commands are similar but will differ if the OCC controller is controlled differently):

- `add_scan_chains ...`
- `add_scan_enables 1 test_se`
DFT Compiler uses the default pin name `test_se`, if a name is not provided.
- `add_pi_constraints 1 test_mode`
DFT Compiler uses the default pin name `test_mode`, if a name is not provided.
- `add_pi_constraints 0 {test_se pll_reset pll_bypass}`
DFT Compiler uses the default pin names `test_se`, `pll_reset`, and `pll_bypass` if the pin names are not provided.
- `set_drc -num_pll_cycles`
- `add_clocks 0 {port_names} -shift -timing {period LE TE measure_time}`
Use this command to specify external clocks that are controllable by ATPG.
- `add_clocks 0 {port_names} -shift -refclock -timing {period LE TE measure_time}`
Use this command to specify ATE and reference clocks with the same period as the shift clock.
- `add_clocks 0 {port_names} -refclock -ref_timing {period LE TE }`
Use this command to specify reference clocks with different periods than the shift clock.
- `add_clocks 0 {pin_names} -pllclock`
Use this command to specify the PLL clocks.
- `add_clocks 0 pin_name -intclock -pll_source node_name -cycle ...`
Use this command to specify the internal clock and the PLL source clock.
- `write_drc_file file_name`

In addition to using these commands, you will need to make the following changes from the output STIL procedure file created by the `write_drc_file` command to the final protocol file:

1. Copy and paste the entire WaveformTable (WFT) "`_default_WFT_{...}`" block four times.

2. Rename new WFT blocks as follows:

```
"_multiclock_capture_WFT_"
"_allclock_capture_WFT_"
"_allclock_launch_WFT_"
"_allclock_launch_capture_WFT_"
```

3. Change the WFT for each procedure (except load_unload) as follows:

```
"multiclock_capture" { W "_multiclock_capture_WFT_";
"allclock_capture" { W "_allclock_capture_WFT_";
"allclock_launch" { W "_allclock_launch_WFT_";
"allclock_launch_capture" { W "_allclock_launch_capture_WFT_";
```

4. In load_unload, add the following just before Shift loop, and specify only the ATE clocks and reference clocks with the same period:

```
V { "clkate"=P; "clkref0"=P; }
```

5. In test_setup, copy the V statement and do the following:

- Change the polarity of the PLL reset constraint in the first V statement (the PLL reset is the same port identified as `pll_reset` in the previous command list).
- Change 0 to P for all the ATE clocks and synchronous reference clocks in both V statements (these are exactly the same clocks specified in Step 4).

6. Change the timing of the WFTs as required. This can be done in an editor, or you can specify another TetraMAX ATPG run and use the `update_wft` and `update_clock` commands.

Specifying Synchronized Multi Frequency Internal Clocks for an OCC Controller

You can use the `ClockTiming` block to implement synchronized internal clocks at one or multiple frequencies in an OCC Controller. The `ClockTiming` block is placed in the top level of the `ClockStructures` block that already describes other aspects of the internal clocks.

The following sections show you how to specify synchronized internal clocks at one or multiple frequencies in an OCC Controller:

- [ClockTiming Block Syntax](#)
- [Timing and Clock Pulse Overlapping](#)
- [Controlling Latency for the PLLStructures Block](#)

- [ClockTiming Block Selection](#)
- [ClockTiming Block Example](#)

For more information on this feature, see the "[Using Synchronized Multi Frequency Internal Clocks](#)" section.

ClockTiming Block Syntax

The syntax and location of the `ClockTiming` block is as follows, with instance-specific input in **italics**, optional input in **[squarebrackets]** and mutually-exclusive choices separated by a **|** pipe symbol:

```

ClockStructures [name] {
    PLLStructures name {
        // The contents of the PLLStructures block are unchanged,
        // except for the addition of the optional Latency statement.
    }
    [PLLStructures name2 {
        // Multiple PLLStructures blocks are possible, and have a specific
        meaning.
        // See the section PLLStructures Block and Latency.
    }]
    ClockTiming name {
        SynchronizedClocks name {
            Clock name { Location "internal_clock_signal";
Period 'time';
                [Waveform 'rise' 'fall';]
                [Clock name2 { Location "internal_clock_signal2";
Period 'time2';
                    [Waveform 'rise2' 'fall2';]}
            // Multiple Clocks can be defined within a SynchronizedClocks
block.
            // These Clocks are considered to be synchronized to each other.
            // Note that each clock's Location value is used, not its name.
                    [MultiCyclePath number [Start|End] {
                        [From clocklocation;]
                        [To clocklocation2;]
                    }]
            // As many MultiCyclePath blocks as needed might be defined.
            // All clocks inside them must be in the current SynchronizedClocks
group.
                    }
            [SynchronizedClocks name2 {
                // Multiple SynchronizedClocks blocks can be defined within a
ClockTiming block
                // These SynchronizedClocks are considered to be asynchronous to
each other
                // The Clocks defined in each SynchronizedClocks group must be
different.
            }]
        }
    }
}

```

```

        }
[ClockTiming name2 {
// Multiple ClockTiming blocks can be defined, but only one is
used.
// The Clocks must be defined again in each ClockTiming block.
// See the ClockTiming Block Selection section.
    }]
}

```

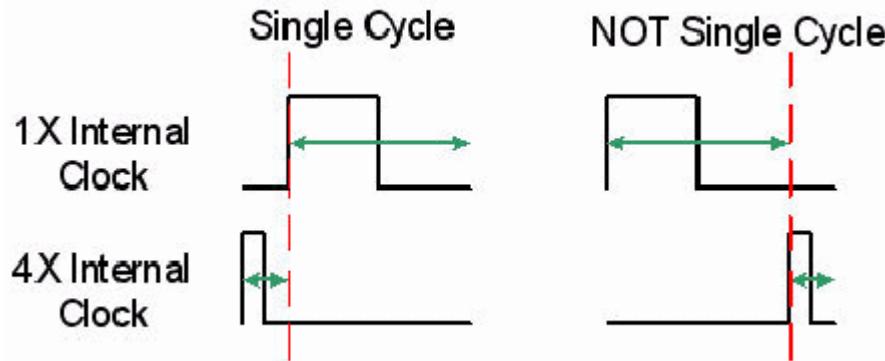
Note the following:

- The `ClockTiming name` is arbitrary and is only used by the `set_drc -internal_clock_timing` option. See “[ClockTiming Block Selection](#)” for details.
- The `SynchronizedClocks name` and `Clock name` are arbitrary.
- The `Location` argument must be identical to the name of the internal clock source defined in one of the `PLLStructures` blocks (see the previous example).
- The `Period` and `Waveform` times are either ns or ps. If they are defined as ps but the rest of the STL procedure file is in ns, they are converted to ns and the fractional part truncated. For example, 1900 ps is converted to 1 ns.

Timing and Clock Pulse Overlapping

The `Waveform` values and `MultiCyclePath` blocks are optional, and ATPG can use different clocks safely for launch and capture without them. However, different frequency clock pulses are not allowed to overlap if they are missing. Figure 1 illustrates synchronized clocking without overlapping pulses.

Figure 1: Non-Overlapping Synchronized Internal Clock Pulses



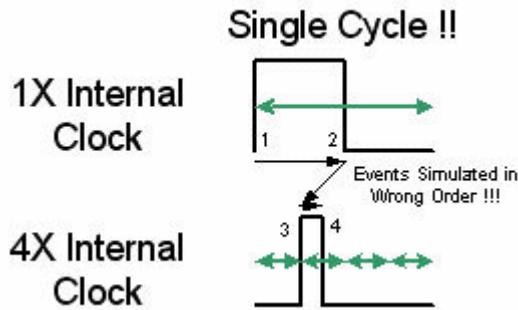
The criteria for allowing synchronized clocks of different frequencies to have overlapping pulses, which in turn allows single-cycle transition fault testing from the slower to the faster clock domain, are as follows:

- The `Waveform` values must be specified for both clocks.
- A `MultiCyclePath 1` block must be specified from the slower clock to the faster clock.

- For non-integer Period ratios, a `MultiCyclePath 1` block is needed for both directions.

Figure 2 illustrates synchronized clocking with overlapping pulses from the slower clock to the faster clock. For the other direction, from faster clock to slower clock, there is no difference between the overlapping and non-overlapping cases.

Figure 2: Overlapping Synchronized Internal Clock Pulses



TetraMAX ATPG generates and simulates patterns with both edges of the first clock pulse preceding either edge of the second clock pulse. Clock pulse overlapping can change this timing relationship. If this situation also changes the behavior of the circuit when simulated on a timing simulator, then the patterns will mismatch. This will occur when trailing-edge or mixed-edge clocking is used, and paths exist from the fast clock to the trailing-edge of the slow clock. If these paths exist, TetraMAX ATPG will prevent overlapping of the clock pair.

Controlling Latency for the `PLLStructures` Block

The number of `PLLStructures` blocks that are specified affects clock latency, which in turn affects the required length of the clock chain. Latency can be controlled using the following top-level statement in the `PLLStructures` block:

`Latency number;`

The default latency is 5. This number refers to the number of pulses of the `PLLClock` that must pulse before the first internal clock pulse is issued by the OCC controller. This number is important when clocks from the same `SynchronizedClocks` group are defined as internal clocks in more than one `PLLStructures` block. In this case, each `PLLStructures` block is interpreted as being a separate OCC controller with its own latency.

If there is more than one clock in a `PLLStructures` block, the latency is in terms of the fastest clock defined in that `PLLStructures` block. (Thus, the same `Latency` number might mean very different latency times in different `PLLStructures` blocks.) The latency time for each clock should be an integer multiple of its period. For example, if a `PLLStructures` block contains synchronized clocks of 10 ns and 20 ns periods, and the latency is allowed to default to 5, then the latency time is $5 * 10$ ns which is not a multiple of the 20 ns clock's period. The 20 ns clock gets a C40 violation and is flagged as restricted.

The latency number is not used if the clocks for each `SynchronizedClocks` group are defined in a single `PLLStructures` group. In that case, it can be set to 0.

ClockTiming Block Selection

By default, the last `ClockTiming` block to be defined in an STL procedure file is used. To use a specific block in a case where multiple `ClockTiming` blocks have been defined, use the following `set_drc` command:

```
set_drc -internal_clock_timing name
```

To ignore all `ClockTiming` blocks and return to legacy non-synchronized internal clocks behavior, use the following setting:

```
set_drc -nointernal_clock_timing
```

ClockTiming Block Example

The following `ClockStructures` block defines three synchronized clocks in one group:

```
ClockStructures Internal_scan {
    PLLStructures "TOTO" {
        PLLCycles 6;
        Latency 4;
        Clocks {
            "clkate" Reference;
            "dut/CLKX4" PLL {
                OffState 0;
            }
            "TOTO/U2/Z" Internal {
                OffState 0;
                PLLSource "dut/CLKX4";
                Cycle 0 "snps_clk_chain_0/U_shftreg_0/ff_38/q_
reg/Q" 1;
                //Note that the rest of the clock chain goes here.
            }
            "dut/CLKX2" PLL {
                OffState 0;
            }
            "TOTO/U5/Z" Internal {
                OffState 0;
                PLLSource "dut/CLKX2";
                Cycle 0 "snps_clk_chain_0/U_shftreg_0/ff_19/q_
reg/Q" 1;
                //The rest of the clock chain goes here.
            }
            "dut/CLKX1" PLL {
                OffState 0;
            }
            "TOTO/U8/Z" Internal {
                OffState 0;
                PLLSource "dut/CLKX1";
            }
        }
    }
}
```

```

        Cycle 0 "snps_clk_chain_0/U_shftreg_0/ff_0/q_
reg/Q" 1;

//The rest of the clock chain goes here.

    }

}

ClockTiming CTiming_2 {
    SynchronizedClocks group0 {
        Clock CLKX4 { Location "TOTO/U2/Z"; Period '10ns';
    }
        Clock CLKX2 { Location "TOTO/U5/Z"; Period '20ns';
    }
        Clock CLKX1 { Location "TOTO/U8/Z"; Period '40ns';
    }
}
}

ClockTiming CTiming_1 {
    SynchronizedClocks group0 {
        Clock CLKX4 { Location "TOTO/U2/Z"; Period '10ns';
            Waveform '0ns' '5ns'; }
        Clock CLKX2 { Location "TOTO/U5/Z"; Period '30ns';
            Waveform '0ns' '15ns'; }
        MultiCyclePath 1 { From "TOTO/U5/Z"; To
"TOTO/U2/Z"; }
    }
    SynchronizedClocks group1 {
        Clock CLKX1 { Location "TOTO/U8/Z"; Period '40ns';
    }
}
}
}

```

This example shows two `ClockTiming` blocks. The one labeled `CTtiming_1` is the default because it is the last one to be defined.

In the first `ClockTiming` block, the three clocks can be used as a single synchronization group. However, clock pulse overlapping is not possible because there are no `Waveform` statements.

In the second `ClockTiming` block, the same three clocks are defined but in a different relationship. The first two clocks are in one `SynchronizedClocks` group and, because their `Waveforms` and a `MultiCyclePath 1` relationship is defined, clock pulse overlapping can be done. The third clock is defined separately, so it is considered to be asynchronous to the others. For this purpose, it could also have been omitted since any clock that is not assigned to a `SynchronizedClocks` group is considered to be asynchronous to all other clocks.

Note that when `ClockTiming` blocks are used, the lengths of the clock chains might be different for different internal clocks. (This is still an error when there is no `ClockTiming` block.)

Specifying Internal Clocking Procedures

Internal clocking procedures are comprised of combinations of internal clock pulses. The following sections describe the syntax for specifying internal clocking procedures in the STIL procedures file (SPF):

- [ClockConstraints and ClockTiming Block Syntax](#)
- [Specifying the Clock Instruction Register](#)
- [Specifying External Clocks](#)
- [Example 1](#)
- [Example 2](#)

For more information on using internal clocking procedures in the ATPG flow, see the "[Using Internal Clocking Procedures](#)" section.

ClockConstraints and ClockTiming Block Syntax

You can use either the `ClockConstraints` block or the `ClockTiming` block in the top level of the `ClockStructures` block to specify internal clocking procedures. However, you cannot combine the `ClockConstraints` and `ClockTiming` blocks. External clocks specified in the `ClockStructures` block must be specified as separate entities in the SPF.

The following syntax is used for specifying internal clocking procedures:

```
ClockStructures (name) {
    (ClockController name { // alias of PLLStructures - either may
    be used
        (PLLCycles number;)
        (MinSysCycles number;) // equivalent to set_atpg -min_
        ateclock_cycles
        (Clocks {
            (location <External|Internal|PLL> {
                (OffState <0|1>)
                (Name name;)

                ...
            }) *
            (name <External|Internal|PLL> {
                (OffState <0|1>)
                (Location location (location)+;

                ...
            }) *
        } ) *
    } ) *
    ...
    (InstructionRegister name { (signame;)+}) *
} ) *
(ClockTiming (name) { ... }) *
(ClockConstraints (name) {
    (UnspecifiedClockValue <Off|On|0|1>;)
```

```

(ClockingProcedure name {
    (UnspecifiedClockValue <Off|On|0|1> ;)
    (clk=<0|1|P>+;) * // Clock assignments
    (clkIR=<0|1>+;) * // Corresponding Clock Instruction
                         // Register assignments
} ) *
} ) *
}

```

Note the following when specifying internal clocking procedures:

- The `MinSysCycles` keyword specifies the number of external ATE cycles required for the clock controller to return to its initial state after being enabled. This statement works for any type of on-chip clocking. It specifies the minimum number of ATE cycles of the capture operation. Extra cycles are appended to the end of the capture sequence if necessary. This keyword can be used instead of `set_atpg -min_ateclock_cycles` command, which has the same behavior in both cases.
- You can define external clocks in the `Clocks` block. You can also define aliases between the location of the clock (which must be the pin pathname to its driving cell) and a name that can be used in the constraint definitions. A clock name can be defined with multiple locations, which allows multiple clocks to be defined with one statement in the constraint definitions.
- The `InstructionRegister` block is a construct in the `ClockController` block. It consists of a sequence of locations that must be set externally to control the specifics of a clocking sequence. The `InstructionRegister` is associated with the controller – not with individual clocks. The `InstructionRegister` construct subsumes the clock chains as specified by using the `Cycle` statements. When constraints are used, the `Cycle` statements are ignored if present.
- Only one `ClockTiming` or `ClockConstraints` block can be used at the same time. If a `ClockTiming` block exists, you must specify the `set_drc -nointernal_clock_timing` command to use internal clocking procedures.
- The `ClockConstraints` block describes a set of clock constraints. The `ClockingProcedure` block describes a set of clock pulse sequences intended to be used jointly. The `ClockingProcedure` specifications satisfies the `ClockConstraints` specifications.
- A `ClockingProcedure` block consists of two types of assignments:
 - The first set of assignments correspond to the clocks, which constitute the actual clock constraints:
`clk=<0|1|P>+;`
 In this case, `clk` is the name of an internal clock, as defined in the `ClockController` block. The nth bit at the end of the line is the constrained assignment to the clock in the nth capture time frame of the pattern in which it is used. The clock-off value (from the `Clocks` definition) indicates no pulse; the `P` value indicates a pulse. The non clock-off value is synonymous to `P`.
 - The second set of assignments are for the `InstructionRegister` contents. These assignments specify the externally assignable values required to realize the

clock assignments. In this case, the nth bit at the end of the line is the value used to set the nth bit of the `InstructionRegister` in the same order that the bits were defined in the `ClockController` block.

The components of the clock controller hardware must be apparent to validate that the specified assignments to the `InstructionRegister` contents actually cause the expected clock pulses. This typically requires functional validation techniques and is beyond the scope of test DRC. Therefore, functional validation is your responsibility. As a last resort, a full timing simulation of the generated patterns should detect any issue.

- The `UnspecifiedClockValue` statement defines the behavior of clocks that are not defined in a particular `ClockingProcedure` block. An `UnspecifiedClockValue` statement outside of all of the `ClockingProcedure` blocks globally specifies the behavior of all unspecified clocks. However, an `UnspecifiedClockValue` statement inside a `ClockingProcedure` block overrides the global value for that block only. The values that can be specified are `Off` (the clocks do not pulse), `On` (the clocks do pulse), `0` or `1`. The default is that the clocks are unspecified. Incomplete clocking procedures are not recognized, so either the `On` or `Off` value should be used or all clock values should be specified.

Specifying the Clock Instruction Register

The `InstructionRegister` block can comprise any of the following defined signals:

- Outputs of scan cells
- Primary inputs
- Outputs of nonscan cells

You can define a combination of any of these signals. Nonscan cells in the clock instruction register must be constant-value C0 or C1 cells and are not allowed to change during the test.

You can apply the `add_cell_constraints` or `add_pi_constraints` command to the clock instruction register members if you want to limit the number of usable clocking procedures.

Specifying External Clocks

External clocks are clocks that are controlled from top-level design ports. They are generally incompatible with internal clocking. When internal clocking procedures are used, unspecified external clocks should be disabled using the `add_pi_constraints` command.

External clocks can be specified inside internal clocking procedures. Although they are already defined as clocks elsewhere in the STL procedure file, they must be redefined in the `Clocks` block of the `ClockController` block if they are specified in the `ClockConstraints` block.

The external clocks are defined just like other clocks in the `ClockingProcedure` blocks. The external clock pulses can be used to control the pulsing of internal clock pulse and are considered as part of the clock instruction register.

You can define the external clocks in a way that they do not affect the internal clocking and are allowed to pulse. In this case, you should define multiple `ClockingProcedure` blocks. These

blocks are identical except for the external clock definitions. As a result, the external clocks are not part of the conditioning to specify the pulses of the internal clocks.

Example 1

```
ClockStructures {
    ClockController controller1 {
        PLLCycles 2;
        Clocks {
            // All clocks are defined by their instance/pin names as usual.
            // The Cycle statements are not needed, so they can be omitted.
            "U1/U2/U_CLK_A_CNTL/Y" Internal {OffState 0;};
            "U1/U2/U_CLK_B_CNTL/Y" Internal {OffState 0;};
            "U1/U2/U_CLK_C_CNTL/Y" Internal {OffState 0;};
            // The next two clocks are equivalent inside the clocking
            // procedures.
            "ClkDE" Internal {
                Offstate 0;
                Location "U1/U2/U_CLK_D_CNTL/Y" "U1/U2/U_CLK_E_CNTL/Y";
            }
        }
        InstructionRegister CLKIR {
            "U1/U3/U_CLK_REG/clk_reg_2/Q";
            "U1/U3/U_CLK_REG/clk_reg_1/Q";
            "U1/U3/U_CLK_REG/clk_reg_0/Q";
        }
    }
    ClockConstraints constraints1 {
        UnspecifiedClockValue Off;
        ClockingProcedure one { // A launches, B captures, C & DE
        default (off)
        // Clocks are still defined by their instance/pin names.
        "U1/U2/U_CLK_A_CNTL/Y"=P0;
        "U1/U2/U_CLK_B_CNTL/Y"=0P;
        CLKIR=001;
    }
        ClockingProcedure three { // B & C both launch & capture, A
        & DE are off
        UnspecifiedClockValue On;
        "U1/U2/U_CLK_A_CNTL/Y"=00;
        "ClkDE"=00;
        CLKIR=010;
    }
        ClockingProcedure four { // DE launches & captures, C also
        captures
        "ClkDE"=PP;
        "U1/U2/U_CLK_C_CNTL/Y"=0P;
        CLKIR=100;
    }
}
```

```

    ClockingProcedure ClockOff { // All clocks off to prevent a
C37 error
        "U1/U2/U_CLK_A_CNTL/Y"=00;
        "U1/U2/U_CLK_B_CNTL/Y"=00;
        "U1/U2/U_CLK_C_CNTL/Y"=00;
        CLKIR=011;
    }
// These are all that are defined, so no other clock pulse
combinations
// or CLKIR values are allowed in the ATPG patterns.
}
}

```

Example 2

```

ClockStructures Internal_scan {
    ClockController "PLL_STRUCT_0" {
        PLLCycles 2;
        Clocks {
            "dutm/clk1" Internal { OffState 0; }
            "dutm/clk2" Internal { OffState 0; }
            "clkref" Reference;
        // "clkext0" is used to control clocking
            "clkext0" External;
        // "clkext1" is allowed to pulse in some procedures
            "clkext1" External;
        }
        InstructionRegister CLKIR {
            "dutcl/FF_0_reg/Q";
            "dutcl/FF_1_reg/Q";
        }
    }
    ClockConstraints constraints1 {
// Force external clocks off when they're unspecified
        UnspecifiedClockValue Off;
        ClockingProcedure intraU1 {
            "dutm/clk1"=PP;
            "dutm/clk2"=00;
            CLKIR=11;
            "clkext0"=00;
        }
        ClockingProcedure extraU1 {
            "dutm/clk1"=PP;
            "dutm/clk2"=P0;
            CLKIR=11;
            "clkext0"=P0;
        }
        ClockingProcedure intraU0 {
            "dutm/clk1"=00;
            "dutm/clk2"=PP;

```

```

        CLKIR=01;
        "clkext1"=PP;
    }
    ClockingProcedure ClockOff {
        "dutm/clk1"=00;
        "dutm/clk2"=00;
        CLKIR=00;
    }
}
}
}

```

See Also

[Using Internal Clocking Procedures](#)

JTAG/TAP Controller Variations for the load_unload Procedure

The `load_unload` procedure defines how to place the design into a state in which the scan chains can be loaded and unloaded. This typically involves asserting a scan-enable input or other control line and possibly placing bidirectional ports into the Z state. Standard DRC rules also require that ports defined as clocks be placed in their off states at the start of the scan chain load/unload process.

In designs that use the test access port (TAP) controller to set up internal scan chain access or boundary scan access, it is very common to need to perform the very last scan shift with the test mode select (TMS) port asserted. This is accomplished by placing as many scan chain force and measure events outside of the `Shift` procedure as necessary. Usually only one final force/measure event is needed.

The bold text in [Example 1](#) shows one additional scan chain force and measure placed outside of the `Shift` procedure. For a scan chain length of N , TetraMAX ATPG performs $N - 1$ shifts using the vector inside the `Shift` procedure, and the final shift using the vector which follows, where `TMS=1`.

Example 1: JTAG/TAP Controller Adjustments to load_unload

```

Procedures {
    "load_unload" {
        V { TMS=0; TCK=0; CLOCK=0; RESETB=1; SCAN_ENABLE = 1; }
        Shift {
            V { _si####; _so####; TCK=P; }
        }
        V { TMS=1; _si####; _so####; TCK=P; }
    }
}

```

For more examples of `load_unload` procedures, see “[JTAG Support](#).”

Multiple Scan Groups

You should set up TetraMAX ATPG for multiple scan-group support if your design has multiple scan chains that cannot be accessed simultaneously (for example, they share the same I/O pins). TetraMAX ATPG supports designs that have multiple scan groups by using IEEE Std. 1450.1 extensions to STIL.

If you have a design with multiple scan groups that must be accessed in serial, not in parallel, during the `load_unload` process, perform the following steps.

1. Define multiple `ScanStructure` blocks.

Each `ScanStructure` block defines one scan chain group. Use a unique label for each scan chain group. In [Example 1](#), the `ScanStructure` labels are `g1`, `g2`, `g3`, and `g4`. TetraMAX ATPG also requires each `ScanChain` label to be unique across all scan chain definitions.

2. Add `ScanStructure` statements to the `load_unload` procedure.

Within the `load_unload` procedure, add the `ScanStructure` statement ahead of any scan input or scan output references. The `ScanStructure` statement identifies the scan group label that is active for any lines that follow.

3. Reference scan inputs and outputs with symbolic labels.

Within the `load_unload` and `Shift` procedures, reference the appropriate set of scan inputs and scan outputs with symbolic labels: `_si1`, `_so1`, `_si2`, `_so2`, and so on.

TetraMAX ATPG associates these symbolic labels with the scan inputs and scan outputs of the appropriate scan group. You are not required to use the `_so` prefix on scan output symbolic labels, but if you use the `_so` prefix, you must also use the `_si` prefix on symbolic labels for the scan input.

If STIL patterns are written out from this data, then each scan signal in each Shift Vector needs a unique symbolic label. A V14 warning is generated when this constraint is not followed, identifying the signal that needs a unique symbolic label. In most other situations, all scan signals can be referenced with a single symbolic label, such as `Shift { V { _si=#; _so=#; ... } }`.

If you are using named `ScanStructure` blocks, they must to be specified as part of the `PatternBurst` block. However, if STIL patterns are written out, then each scan signal used across more than one scan group will require a separate symbolic label to associate scan data with this specific scan block. [The example](#) shown in the "[JTAG/TAP Controller Variations for the load_unload Procedure](#)" section does not follow this constraint (and would generate V14 warnings which may be ignored if STIL patterns are not generated), however [Example 1](#) below (with only one scan chain per Shift) does. [Example 1](#) demonstrates a multiple scan chain per Shift implemented with this restriction.

When symbolic labels must be associated with individual scan signals, it is necessary to define a `SignalGroups` block to establish these associations and define the symbolic labels. [Example 1](#) identifies the necessary `SignalGroup` definitions that must be part of this context.

Example 1: Four Scan Groups Structured for STIL Pattern Generation

STIL 1.0;

```

SignalGroups {
    _si11="SDI[1]" {ScanIn;} _si12="SDI[2]" {ScanIn;} _si13="SDI[3]"
    {ScanIn;}
    _so11="SDO[1]" {ScanOut;} _so12="SDO[2]" {ScanOut;} _so13="SDO[3]"
    {ScanOut;}
    _si21="SDI[1]" {ScanIn;} _si22="SDI[2]" {ScanIn;} _si23="SDI[3]"
    {ScanIn;}
    _so21="SDO[1]" {ScanOut;} _so22="SDO[2]" {ScanOut;} _so23="SDO[3]"
    {ScanOut;}
    _si31="SDI[1]" {ScanIn;} _si32="SDI[2]" {ScanIn;} _si33="SDI[3]"
    {ScanIn;}
    _so31="SDO[1]" {ScanOut;} _so32="SDO[2]" {ScanOut;} _so33="SDO[3]"
    {ScanOut;}
    _si41="SDI[1]" {ScanIn;} _si42="SDI[2]" {ScanIn;} _si43="SDI[3]"
    {ScanIn;}
    _so41="SDO[1]" {ScanOut;} _so42="SDO[2]" {ScanOut;} _so43="SDO[3]"
    {ScanOut;}
}

PatternBurst "_burst_{
    ScanStructures g1;ScanStructures g2;ScanStructures g3;
    ScanStructures g4;
    PatList {"_pattern_{
    }

ScanStructures g1 {
    ScanChain g1_0 { ScanIn "SDI[1]"; ScanOut "SDO[1]"; }
    ScanChain g1_1 { ScanIn "SDI[2]"; ScanOut "SDO[2]"; }
    ScanChain g1_2 { ScanIn "SDI[3]"; ScanOut "SDO[3]"; }
}

ScanStructures g2 {
    ScanChain g2_0 { ScanIn "SDI[2]"; ScanOut "SDO[1]"; }
    ScanChain g2_1 { ScanIn "SDI[3]"; ScanOut "SDO[2]"; }
    ScanChain g2_2 { ScanIn "SDI[1]"; ScanOut "SDO[3]"; }
}

ScanStructures g4 {
    ScanChain g4_0 { ScanIn "SDI[3]"; ScanOut "SDO[1]"; }
    ScanChain g4_1 { ScanIn "SDI[2]"; ScanOut "SDO[2]"; }
    ScanChain g4_2 { ScanIn "SDI[1]"; ScanOut "SDO[3]"; }
}

ScanStructures g3 {
    ScanChain g3_0 { ScanIn "SDI[3]"; ScanOut "SDO[1]"; }
    ScanChain g3_1 { ScanIn "SDI[1]"; ScanOut "SDO[2]"; }
    ScanChain g3_2 { ScanIn "SDI[2]"; ScanOut "SDO[3]"; }
}

Procedures {
}

```

```

load_unload {

V { mclk=0; clk=0; rst=1; scan_en=1; inc=0; }
V { mode=0; }
V { chain_sel = 0; mclk=P; }
V { }

ScanStructures g1;
"single_shift0:" V { _si11=#; _si12=#; _si13=#; _so11=#; _so12=#; _so13=#; clk=P; mclk=0; }
Shift { ScanStructures g1; V { _si11=#; _si12=#; _si13=#; _so11=#; _so12=#; _so13=#; clk=P; } }

ScanStructures g4;
Shift { V { _si41=#; _si42=#; _si43=#; _so41=#; _so42=#; _so43=#; clk=P; mclk=0; } }
V { clk=0; mclk=0; mode=1; }
"single_shift1:" V { _si11=#; _si12=#; _si13=#; _so11=#; _so12=#; _so13=#; clk=P; }
V { chain_sel = 0; mclk=P; clk=0; }
V { chain_sel = 1; }
V { mclk=0; }

ScanStructures g2;
Shift { V { _si21=#; _si22=#; _si23=#; _so21=#; _so22=#; _so23=#; clk=P; } }
"single_shift2:" V { _si21=#; _si22=#; _si23=#; _so21=#; _so22=#; _so23=#; clk=P; }
V { chain_sel = 1; mclk=P; clk=0; }

ScanStructures g3;
V { chain_sel = 0; mclk=P; }
Shift { V { _si31=#; _si32=#; _si33=#; _so31=#; _so32=#; _so33=#; clk=P; mclk=0; } }
V { clk=0; }
V { chain_sel = 1; mclk=P; }
V { }

}

}

MacroDefs {
"test_setup" {

V { "mclk"=0; "clk"=0; "rst"=1; scan_en=0; inc=0; mode=1; }
}
}

```

[Example 1](#) identifies the necessary expansion to the symbolic references, to support proper STIL pattern generation of a design containing scan groups that are sequentially shifted. This example shows,

- The SignalGroups definitions necessary to support association of the individual signals in the `load_unload` procedure.
- The use of the symbolic references in the `load_unload` procedure to reference individual scan signals.
- The presence of pre-shift and post-shift vectors that also consume scan data. Look for the labels `single_shift0`, `single_shift1`, and `single_shift2` in the [Example 1](#).

Note: It is a DFT requirement that the scan cells of one scan group not be disturbed during the scan shifting of other scan groups. You must consider this restriction when you plan to use multiple scan groups.

[Example 2](#) illustrates syntax for a design with four different groups of scan chains that must be accessed serially during the `load_unload` process.

Example 2: Four Scan-Chain Groups Loaded Serially

```

ScanStructures g1 {
    ScanChain g1_0 { ScanIn "SDI[1]"; ScanOut "SDO[1]"; }
    ScanChain g1_1 { ScanIn "SDI[2]"; ScanOut "SDO[2]"; }
    ScanChain g1_2 { ScanIn "SDI[3]"; ScanOut "yama"; }
}
ScanStructures g2 {
    // STIL allows same chain name in another group,
    // but TMAX does not
    ScanChain GROUP2_0 { ScanIn "SDI[2]"; ScanOut "data23"; }
    ScanChain GROUP2_1 { ScanIn "SDI[3]"; ScanOut "SDO[2]"; }
}
ScanStructures g4 {
    ScanChain "g4_0" { ScanIn "SDI[3]"; ScanOut "SDO[1]"; }
    ScanChain "g4_1" { ScanIn "SDI[2]"; ScanOut "SDO[2]"; }
    ScanChain "g4_2" { ScanIn "SDI[1]"; ScanOut "SDO[3]"; }
}
ScanStructures g3 {
    ScanChain g3_0 { ScanIn "SDI[3]"; ScanOut "SDO[1]"; }
    ScanChain g3_1 { ScanIn "SDI[1]"; ScanOut "SDO[2]"; }
    ScanChain g3_2 { ScanIn "SDI[2]"; ScanOut "SDO[3]"; }
}
Procedures {
    load_unload {
        V { mclk=0; clk=0; rst=1; scan_en=1; inc=0; }
        V { mode=0; }
        ScanStructures g1;
            V { chain_sel = 0; mclk=P; }
            V { chain_sel = 0; mclk=P; }
            Shift {
                V { _sil=###; _sol=###; clk=P; mclk=0; }
            }
        }
        ScanStructures g2;
            V { chain_sel = 0; mclk=P; clk=0; }
            V { chain_sel = 1; mclk=P; }
            V { mclk=0; }
    }
}

```

```

        Shift {
            V { _si2=##; _so2=##; clk=P; }
        }
    ScanStructures g3;
        V { chain_sel = 1; mclk=P; clk=0; }
        V { chain_sel = 0; mclk=P; }
        Shift {
            V { _si3=###; _so3=###; clk=P; mclk=0; }
        }
    ScanStructures g4;
        V { clk=0; }
        V { chain_sel = 1; mclk=P; }
        V { chain_sel = 1; mclk=P; }
        Shift {
            V { _si4=###; _so4=###; clk=P; mclk=0; }
        }
    }
    V { clk=0; mclk=0; mode=1; }
}
}

```

The design for the multiple scan group protocol in [Example 2](#) has the following elements:

- Four scan chain groups. Three groups have three scan chains and the fourth has two. A simple MUX control selects the active scan group by marching a 2-bit code into the `chain_sel` port using the `mclk` clock.
- The `load_unload` procedure begins with two `V{...}` statements to place the design into a shift mode.
- The first `ScanStructures` statement makes group `g1` active for the lines that follow.
- A `Shift{...}` procedure uses the symbolic label `_si1`. This symbolic label is associated with the scan input pins defined in the `ScanStructures g1` block.
- Following the first scan group are three additional sequences of `ScanStructures`, followed by `V{...}` statements that select the appropriate chain group, and a `Shift{...}` procedure.

As you saw in [Example 2](#), a simple MUX control accomplished the sharing of similar I/O pins across four scan groups. But some boundary-scan designs that need to support multiple scan chains can have more complicated control sequences. For example, it is not uncommon to require the final shift of the TAP-controlled scan chain to be done outside of the `Shift` procedure.

The cone timepts and rules for supporting multiple scan groups are the same for a design with boundary scan as for a design without boundary scan.

[Example 3](#) shows a more complicated sequence for a design with three scan groups of one scan chain each. In this design, to load an instruction that accesses each internal scan chain through its test data in (TDI) and test data out (TDO) pins, the TAP controller must be stepped through each of its various states.

Example 3: Design With Three Scan Groups

```

STIL 1.0;
ScanStructures A { ScanChain "A1" { ScanIn "tdi"; ScanOut "tdo"; }
}

```

```

ScanStructures B { ScanChain "B1" { ScanIn "tdi"; ScanOut "tdo"; }
}
ScanStructures C { ScanChain "C1" { ScanIn "tdi"; ScanOut "tdo"; }
}
// Instructions to enable scanning of each of the previous 3
groups:
// -----
// Group Tap instruction
// -----
// 1 SCAN_MODULE_A 7'b00011
// 2 SCAN_MODULE_B 7'b00101
// 3 SCAN_MODULE_C 7'b00111
//
Procedures {
    load_unload {

        V { clock=0; test_enab=1; scan_enab=1; _io=Z ;
            tms=0; tck=0; resetN=1; TBC=0; }
        ScanStructures A;
        V { tms=1; tdi=0; tck=P; clock=0; } // move to SELECT-DR
        V { tms=1; tdi=0; tck=P; } // move to SELECT-IR
        V { tms=0; tdi=0; tck=P; } // move to CAPTURE-IR
        V { tms=0; tdi=0; tck=P; } // move to SHIFT-IR
        V { tms=0; tdi=1; tck=P; } // shift IR, inst=1xxxx
        V { tms=0; tdi=1; tck=P; } // shift IR, inst=11xxx
        V { tms=0; tdi=0; tck=P; } // shift IR, inst=011xx
        V { tms=0; tdi=0; tck=P; } // shift IR, inst=0011x
        V { tms=1; tdi=0; tck=P; } // shift IR, inst=00011, mv to
EXIT1-IR
        V { tms=1; tdi=0; tck=P; } // move to UPDATE-IR
        V { tms=0; tdi=0; tck=P; } // move to IDLE
        V { tms=0; tdi=0; tck=0; } // clocks off
        Shift { V { _si1=# ; _so1=# ; clock=P; } }
        ScanStructures B;
        V { tms=1; tdi=0; tck=P; clock=0; } // move to SELECT-DR
        V { tms=1; tdi=0; tck=P; } // move to SELECT-IR
        V { tms=0; tdi=0; tck=P; } // move to CAPTURE-IR
        V { tms=0; tdi=0; tck=P; } // move to SHIFT-IR
        V { tms=0; tdi=1; tck=P; } // shift IR, inst=00101
        V { tms=0; tdi=0; tck=P; } // shift IR
        V { tms=0; tdi=1; tck=P; } // shift IR
        V { tms=0; tdi=0; tck=P; } // shift IR
        V { tms=1; tdi=0; tck=P; } // shift IR, move to EXIT1-IR
        V { tms=1; tdi=0; tck=P; } // move to UPDATE-IR
        V { tms=0; tdi=0; tck=P; } // move to IDLE
        V { tms=0; tdi=0; tck=0; } // clocks off
        Shift { V { _si2=# ; _so2=# ; clock=P; } }
        ScanStructures C;
        V { tms=1; tdi=0; tck=P; clock=0; } // move to SELECT-DR
        V { tms=1; tdi=0; tck=P; } // move to SELECT-IR
    }
}

```

```

V { tms=0; tdi=0; tck=P; } // move to CAPTURE-IR
V { tms=0; tdi=0; tck=P; } // move to SHIFT-IR
V { tms=0; tdi=1; tck=P; } // shift IR, inst=00111
V { tms=0; tdi=1; tck=P; } // shift IR
V { tms=0; tdi=1; tck=P; } // shift IR
V { tms=0; tdi=0; tck=P; } // shift IR
V { tms=1; tdi=0; tck=P; } // shift IR, move to EXIT1-IR
V { tms=1; tdi=0; tck=P; } // move to UPDATE-IR
V { tms=0; tdi=0; tck=P; } // move to IDLE
V { tms=0; tdi=0; tck=0; } // clocks off
Shift {
V { _si3=# ; _so3=# ; clock=P; }
}
V { tms=1; tdi=#; tck=#; } // move to EXIT1-DR
V { tms=1; tdi=0; tck=0; } // move to UPDATE-DR
V { tms=1; tdi=0; tck=0; } // move to SELECT-DR
V { tms=0; tdi=0; tck=0; } // move to CAPTURE-DR
} // end load_unload
capture_tck {
V { _pi=# ; _po=# ; tck=P; }
}
capture_clock {
V { _pi=# ; _po=#; clock=P; }
}
capture_resetN {
V { _pi=# ; _po=# ; resetN=P; }
}
capture {
V { _pi=# ; _po=# ; }
}
}
MacroDefs {
test_setup {
V { _io=Z ; tms=1; tdi=0; tck=0; resetN=1; test_enab=1;
scan_enab=0; clock=0; TBC=0; }
V { tms=1; tdi=0; tck=0; resetN=P; clock=0; } // move to
RESET
V { tms=1; tdi=0; tck=P; resetN=1; clock=P; } // stay in
RESET
V { tms=0; tdi=0; tck=P; resetN=1; clock=P; } // move to
IDLE
V { tms=0; tdi=0; tck=0; } // clocks off
}
}

```

DFTMAX Compression with Serializer

The DFTMAX compression scan architecture creates by default a combinational connection between the input and output of the compressor/decompressor (“CODEC”) to the top-level ports or pins. To improve the ATPG quality of results (QOR) for designs or blocks with a limited

number of top-level ports, DFTMAX compression also supports an optional serial connection between the CODEC and the top-level ports, called "serializer."

You should refer to the "DFTMAX with Serializer" chapter in the *DFTMAX Compression User Guide* to see an example STIL procedure file specifically used with serializer. This chapter includes a description of the `SerializerStructures` statement, which is specific to serializer.

Also note that the `report_serializers` command in TetraMAX ATPG generates a report containing data for the specified serializers.

11

Design Rule Checking

The DRC process verifies that the physical layout of a design satisfies a series of parameters or rules required by semiconductor manufacturers. By performing DRC, you can verify that a design will function properly when it is fabricated.

You can refer to "[Performing Test Design Rule Checking](#)" for a basic guide on how to specify basic DRC settings, run DRC, and review DRC results.

The following sections describe the various settings you make when performing DRC:

- [Understanding the DRC Process](#)
- [Contention Analysis](#)
- [Scan Chain Tracing](#)
- [Clock Grouping](#)
- [Declaring Equivalent and Differential Input Ports](#)
- [Cells With Asynchronous Set/Reset Inputs](#)
- [Masking Input and Output Ports](#)
- [Masking Scan Cell Inputs and Outputs](#)
- [Previewing Potential Scan Cells](#)
- [Transparent Latches](#)
- [Shadow Register Analysis](#)
- [Feedback Paths Analysis](#)
- [Procedure Simulation](#)
- [Changing the Design Rule Severity](#)
- [Understanding the DRC Summary Report](#)
- [Binary Image Files](#)
- [Save/Restore in TEST Mode](#)

Understanding the DRC Process

When performing design rule checking, TetraMAX ATPG takes the following actions:

1. Reads the STIL procedures to gather information and to check for syntax and consistency errors. For more information, see "[STIL Procedures](#)."
2. Performs contention ability checks on buses and wired logic. This step identifies drivers that could potentially be placed in a conflicting state and cause internal device contention. For more information, see "[Contention Analysis](#)."
3. Simulates the test procedures in the STL procedure file to determine whether certain conditions have been met involving the state of clocks and the sequencing of procedural events.
4. Simulates each scan chain under the direction of the defined test procedures, to guarantee that the scan path is operational and complies with all scan chain rules. For more information, see "[Scan Chain Tracing](#)."
5. Analyzes all clocks and clocked devices against the ATPG rules for clock usage. For more information, see C Rules.
6. Analyzes all nonscan devices, including latches, RAMs, ROMs, and bus keepers (S Rules). Nonscan devices that hold state are identified and used for ATPG purposes. Latches that can be made transparent are identified, and latches that cannot be made transparent are replaced with TIEX logic.
7. Analyzes the multi driver nets identified in step 2 as potentially causing conflict to determine which drivers actually cause conflict.
8. Performs some additional circuit learning that depends on the results of the previous steps. After identifying scan, nonscan, transparent and nontransparent devices, and sequential devices at a constant state, TetraMAX ATPG propagates the effects of PI constraints, ATPG constraints, and TIEX effects throughout the design.
9. Produces a summary report listing the types and totals of DRC violations encountered. For more information, see "[Understanding the DRC Summary Report](#)."

Contention Analysis

Three-state circuitry is characterized by its ability to use the high impedance state (Z state). The supported gate types that model the logical behavior of three-state circuitry and use the Z state include the BUS, BUSK, TSD, SW, PI, PO, PIO, and TIEZ gates.

Most three-state activity occurs on a BUS gate, which is primarily used to resolve the net value from a net with multiple drivers. A BUS gate can have bidirectional connections to external pins (PIO) or bus keepers (BUSK). All inputs and bidirectional connections may be strong or weak.

A *contention condition* occurs when a BUS gate has two strong drivers of opposing values. This condition can damage the chip, so extensive contention checking is required to prevent its occurrence.

The following sections describe contention checking:

- [BUS Contention Ability Checking](#)
- [BUS Z State Ability Checking](#)
- [Contention Prevention Checking](#)
- [Simulation Contention Detection](#)
- [ATPG Contention Prevention](#)
- [Post-Capture Contention Checking](#)

For information on settings you can make for contention checking, see "[Settings for Contention Checking](#)."

BUS Contention Ability Checking

During DRC, the Z1 rule checks BUS gates with circuitry that could potentially cause BUS contention. This check eliminates false contention reporting when multiple inputs to a BUS are at X. The BUS contention ability analysis searches for two strong three-state drivers on a BUS gate that can simultaneously have their enable lines active. Unless the `-nomultiple_on` option of the `set_contention` command is set for contention checking, the data lines must be at different values to fail the check. After BUS contention ability checking is performed, a summary message shows the number of buses falling into each of the following contention ability categories:

- Pass - The BUS gate cannot satisfy contention conditions and may be ignored for contention checking.
- Bidi - The BUS has an external bidirectional connection. Except for this connection, it passes contention ability checking. To control contention, it need only be controlled by the value placed on the bidirectional port.
- Fail - The BUS is capable of contention and must be checked and controlled.
- Abort - Contention ability checking of the BUS was aborted. It is uncertain whether the BUS is capable of contention, so it must be checked and controlled.

The BUS contention ability analysis is performed only when required. If the analysis was previously performed and nothing has changed that could affect the results for a BUS, it is not checked again.

BUS Z State Ability Checking

During DRC, the Z2 rule identifies BUSes with circuitry that could potentially cause a Z state. This check attempts to satisfy the conditions necessary to justify a Z state on a BUS gate. After this check is performed, a summary message shows the number of BUSes falling into each Z state ability category:

- Pass - The BUS gate cannot satisfy Z-state conditions.
- Bidi - The BUS has an external bidirectional connection. Except for this connection, it passes Z-state checking.

- Fail - The BUS is capable of holding a Z state.
- Abort - Z-state ability checking of the BUS was aborted. It is uncertain whether the BUS is capable of holding a Z state.

The BUS Z state ability analysis is performed only when required. If the analysis was previously performed and nothing has changed that could affect the results for a BUS, it is not checked again.

Contention Prevention Checking

For BUSes that fail or abort the Z1 rule, an ATPG analysis is performed by the Z7 rule to determine if it is possible to simultaneously satisfy the conditions necessary to prevent contention on these buses. A Z7 failure indicates that ATPG is unlikely to be successful in avoiding bus contention. See the description of the Z7 rule in TetraMAX Help for a complete description of how to properly analyze a Z7 failure.

Simulation Contention Detection

BUS gates that fail contention ability checking during simulation are checked to determine if there are in a potential contention condition. A violation of BUS contention during fault simulation causes the pattern to be rejected and disallowed any detection credit. A message is issued for each simulation pass indicating the number of patterns rejected due to contention and the site of the first contention. You can turn off contention checking during simulation using the nobus option of the `set_contention` command.

ATPG Contention Prevention

BUS gates that fail contention ability checking during test generation are forced to satisfy a contention-free state. If the process of satisfying contention prevention causes an abort condition, a special message reports the number of faults per simulation pass (32 patterns) that were aborted due to this condition. You can turn off ATPG contention prevention using the –noatpg switch of the `set_contention` command.

Post-Capture Contention Checking

Normal scan-based simulation only considers the effect of values loaded into scan cells and not the effect of values that may be captured. If the enable lines of three-state drivers of bus gates that are not contention-free are connected to scan cells, it is possible for these BUS gates to go into contention after the capture clock, even if they were contention-free before the capture clock. These conditions are checked by the Z9 and Z10 rules.

You can configure the simulation process to simulate the captured values using the –capture option of the `set_contention` command. As a result, the simulation checks for contention that could occur at capture time, and rejects and reports patterns that fail the contention check.

Settings for Contention Checking

When TetraMAX ATPG checks bus contention, it discards patterns that can potentially cause contention and generates additional patterns to avoid contention. You can select optional bus contention checks using the Set Contention dialog box, or you can enter the `set_contention` command from the command line.

The following sections show you how to choose settings for contention checking:

- [Using the Set Contention Dialog Box](#)
- [Using the set_contention Command](#)

For more information on contention checking, see "[Contention Analysis](#)."

Using the Set Contention Dialog Box

To use the Set Contention dialog box to set contention options,

1. From the menu bar, choose Buses > Set Contention Options.
The Set Contention dialog box appears.
2. If you require settings other than the defaults, choose them now.
The default settings are the ones used most often. For details about these and other settings, see Online Help for the `set_contention` command.
3. Click OK.

Using the set_contention Command

You can also select bus contention options using the `set_contention` command. For example, the following command is conservative without being overly restrictive to TetraMAX ATPG:

```
DRC-T> set_contention bidi bus ram -capture -atpg -multiple_on
```

For the complete syntax and option descriptions, see Online Help for the `set_contention` command.

See Also

[Contention Analysis](#)

Scan Chain Tracing

When performing scan chain tracing, TetraMAX ATPG takes the following actions:

1. Initializes constrained ports to their constrained states.
2. Simulates the events in the `test_setup` macro.
3. Simulates the events in the `load_unload` procedure.

4. Simulates the events in the Shift procedure, and monitors the elements in the scan chain to ensure that the scan data path is valid, the scan cells are clocked, and any asynchronous set/clear pins are stable in their off positions.

To see a verbose report on the scan chain tracing, execute the following command:

```
BUILD-T> set_drc -trace
```

The default is to not show the verbose tracing of scan chains.

See Also

[Performing Scan Chain Diagnostics](#)

Clock Grouping

TetraMAX ATPG applies dynamic clocking grouping by default. This enables basic scan ATPG to simultaneously pulse clocks and detect clocks that can be serially pulsed during the same capture cycle. Clocks with a small amount of sequential effects can also be detected and grouped. In this case, TetraMAX ATPG sets up the pattern generation environment to avoid generating vectors that would fail simulation.

Clock grouping can potentially reduce pattern count since ungrouped clocks require separate scan loads and patterns to test faults in each clock domain for basic-scan patterns. Grouped clocks can be pulsed in a single pattern. The clocks pulsed for a given vector are selected dynamically during pattern generation, maximizing the fault detection and minimizing the pattern count.

In addition to dynamic clocking, TetraMAX ATPG can use disturbed clocking to group some clocks with a limited number of cells containing sequential effects. In this case, even if there are sequential effects, grouping these clock can further reduce pattern count. TetraMAX ATPG then masks any disturbed cells to avoid sequential effects. Potential disturbed grouping is done during DRC analysis.

During the DRC process, TetraMAX ATPG automatically performs clock grouping analysis and reports the results in the transcript. All PI equivalences are removed, except for differential inputs.

The following sections describe how to work with clock groups:

- [Reducing the Pattern Count Through Clock Grouping](#)
- [Clock Grouping Analysis](#)
- [Generating a Clock Group Report](#)
- [Clock Grouping Limitations](#)

Reducing the Pattern Count Through Clock Grouping

When you generate combinational vectors in basic-scan ATPG, TetraMAX ATPG normally uses only one clock pulse per pattern. However, it is sometimes possible to pulse several clocks in the

same vector, which enables you to observe more logic and reduces the need for additional patterns.

If your design has two independent clocks (for example, when you pulse one clock, no logic driven by the other clock is affected), then you need two patterns to exercise the logic in the two clock domains. However, because the clocks are independent, you can pulse them at the same time, which saves one test vector. When you use static parallel clock grouping, the grouped clocks must always be pulsed together. None of the clocks in the group may be pulsed alone.

Dynamic clock grouping selects the clocks pulsed for a given vector during pattern generation, which maximizes the fault detection and minimizes the pattern count.

The disturbed clocking scheme allows TetraMAX ATPG to group some clocks with a limited number of cells having sequential effects. In this case, even if there are sequential effects, it can be useful to group those clocks to further reduce pattern count. TetraMAX ATPG cannot use the disturbed cells. To manually group clocks, use the `add_pi_equivalences` command. After you have defined a group, any clock that belongs to this group cannot be pulsed alone.

To use clock grouping to reduce the pattern count:

1. Read your netlist and library model files, and build your design in TetraMAX ATPG. For details, see "[Setting Up and Building the ATPG Model](#)."
2. Choose your criteria for clock grouping. The `set_drc` command has several options that affect clock grouping, including the `-allow_unstable_set_resets`, the `-blockage_aware_clock_grouping`, the `-clock_dynamic`, the `-disturb_clock_grouping`, and the `-dynamic_clock_equivalencing` options.
3. Run DRC. For details, see "[Performing Test Design Rule Checking](#)."
DRC performs an analysis for clock grouping. For details, see "[Clock Grouping Analysis](#)."
4. Generate the basic-scan test vectors, for example:

```
run_atpg -auto_compression
```

Clock Grouping Analysis

During the DRC process, clock grouping analysis is automatically performed and the results are reported in the transcript, as shown in the following example:

```
Clocks C1 (8) and C2 (13) were identified as potentially
groupable.
Clocks C1 (8) and C3 (17) were identified as potentially
groupable.
Clocks C1 (8) and C4 (19) were identified as potentially
groupable.
Clocks C1 (8) and W4 (20) were identified as potentially
groupable.
Clocks C2 (13) and C3 (17) were identified as potentially
groupable.
Clocks C2 (13) and C4 (19) were identified as potentially
groupable.
Clocks C2 (13) and W4 (20) were identified as potentially
groupable.
Clocks C3 (17) and C4 (19) were identified as potentially
```

```
groupable.  
Clocks C3 (17) and W4 (20) were identified as potentially  
groupable.  
Clock grouping analysis completed, #clock_groups_identified=9
```

The lines in the example indicate that clock 'C1', with gate ID 8, can be grouped with clocks 'C2', 'C3', 'C4', and 'W4'.

In addition, clock 'C2' is groupable with {C3,C4,W4} and 'C3' is groupable with {C4,W4}. Clock grouping might be affected by the order in which the clock list is processed. It is suggested that if you define clocks using `add_clocks` commands, that you define the clock with the highest fanout first, and all asynchronous set/resets last.

The clock grouping algorithm considers clocks as groupable if all of the following conditions are true:

- The clocks do not connect to a common clock-off stable state element.
- There are no level sensitive (LS) or trailing edge (TE) ports where one clock is connected to the clock or write port input and the other clock has a clock-effect connection to the port data input with any of the following conditions:
 - LS/LE connection
 - TE connection
 - where the off-time of the first clock occurs later than the off-time of the other clock.
- There are no LE ports where one clock is connected to the clock or write port input and the other clock has a clock-effect connection to the port data input with any of the following condition:
 - LS/LE connection
 - where the on-time of the first clock occurs later than the on-time of the other clock.
 - There are no LS or TE ports where one clock is connected to the clock or write port input and the other clock has a clock-effect connection to the port clock/write input with any of the following conditions:
 - LS/LE connection
 - TE connection
 - where the off-time of the first clock occurs later than the off-time of the other clock.
 - There are no LE ports where one clock is connected to the clock or write port input and the other clock has a clock-effect connection to the port clock or write input with any of the following condition:
 - LS/LE connection
 - where the on-time of the first clock occurs later than the on-time of the other clock.
- If the design has two state elements A and B such that:
 - The output of A is connected to the input of B.
 - A and B are clocked by different clocks that have nearly identical timing

Then, the two clocks have a parallel grouping relationship only if the capture edge of clock B occurs at or before the capture edge of clock A minus the skew value.

Otherwise, the clocks are ungrouped, or have a disturbed grouping. The default for the skew is 1 time unit, which eliminates clocks with exactly the same timing from being grouped in this type of design connectivity.

Note that the unstable state elements (including transparent latches) are ignored for this clock grouping analysis.

The clock grouping analysis is always performed at the end of the clock rules checking during DRC with all grouped clocks reported in the transcript.

Generating a Clock Group Report

To report results of clock grouping analysis, use the following command:

```
report_clocks -matrix -verbose
```

The `-matrix` option of the `report_clocks` command displays a matrix of clock pairs that can be grouped together. In the clock matrix, each row indicates the potential grouping relationships of a candidate clock with all of the other candidate clocks.

For example:

id#	clock_name	type	0	1	2	3	4	5	6	7	8	9
0	clk	C	---	--A	--A	--A	--A	--A	--A	--A	--A	---
1	iopclk11	C	B--	---	--A	BPA	BPA	BPA	BPA	BPA	BPA	B--
2	iopclk12	C	B--	B--	---	--A	BPA	BPA	BPA	BPA	BPA	B--
3	iopclk21	C	B--	BPA	B--	---	--A	BPA	BPA	BPA	BPA	B--
4	iopclk22	C	B--	BPA	BPA	B--	---	BPA	BPA	BPA	BPA	B--
5	iopclk31	C	B--	BPA	BPA	BPA	BPA	---	--A	BPA	BPA	B--
6	iopclk32	C	B--	BPA	BPA	BPA	BPA	B--	---	BPA	BPA	B--
7	iopclk41	C	B--	BPA	BPA	BPA	BPA	BPA	BPA	---	--A	B--
8	iopclk42	C	B--	BPA	BPA	BPA	BPA	BPA	BPA	B--	---	B--
9	tx_intf1_clk	C	---	--A	--A	--A	--A	--A	--A	--A	--A	---
10	tx_intf2_clk	C	---	--A	BPA	--A	---	--A	BPA	--A	BPA	B--
11	tx_intf3_clk	C	---	--A	BPA	--A	BPA	--A	---	--A	BPA	B--
12	tx_intf4_clk	C	-D-	BPA	BPA	--A	BPA	--A	BPA	--A	---	BP-
13	por	R	---	--A	--A	--A	--A	--A	--A	--A	--A	--A
14	rst	SR	---	---	---	---	---	---	---	---	---	---
id1	id2	C1	#masks	C2	masked	gates						

Clock Grouping Limitations

Clock grouping has the following limitations:

- Dynamic and disturbed clocking are not used by Full-Sequential ATPG.
- Disturbed clocking can result in a slightly lower test coverage because of disturbed cell masking.

Declaring Equivalent and Differential Input Ports

You can declare two primary input ports to be equivalent or differential. During ATPG, equivalent ports are always driven with the same values and differential ports are always driven with complementary values.

You can use the Add PI Equivalences dialog box to make this kind of declaration, or you can enter the `add_pi_equivalences` command at the command line.

The following sections describe how to declare equivalent and differential input ports:

- [Using the Add PI Equivalences Dialog Box](#)
- [Using the add_pi_equivalences Command](#)

Using the Add PI Equivalences Dialog Box

The following steps describe how to use the Add PI Equivalences dialog box to make two primary input ports to be equivalent or differential:

1. From the menu bar, choose Constraints > PI Equivalences > Add PI Equivalences. The Add PI Equivalences dialog box appears.
2. Select the ports and logic relationships.
For additional information about the available options, see the description of the `add_pi_equivalences` command in TetraMAX Help.
3. Click OK.

Using the `add_pi_equivalences` Command

You can also declare equivalent or differential input ports by using the `add_pi_equivalences` command, as shown in the following example:

```
DRC-T> add_pi_equivalences ENA_P -inv ENA_N
```

For the complete syntax and option descriptions, see the description of the `add_pi_equivalences` command in TetraMAX Help..

In the following example, the first line defines the two input ports `spec_port1` and `spec_port2` as equivalent; the second line defines that the following ports should be constrained to be at an inverted value relative to the first port in the list.

```
DRC-T> add_pi_equivalences {spec_port1 spec_port2}
DRC-T> add_pi_equivalences spec_port1 -invert spec_port2
```

When differential inputs are also clocks, you must first define each port as a clock and then define the equivalence relationship, as in the following example:

```
DRC-T> add_clocks 0 clock_pos
DRC-T> add_clocks 1 clock_neg
DRC-T> add_pi_equivalences clock_pos -differential clock_neg
```

The third line defines them as differential. This is similar in function to the `-invert` option with two differences. The first difference is that only two pins are accepted. The second difference is

that pins declared as having a `-differential` relationship that are also clocks retain that relationship when clock grouping is enabled. A differential clock relationship formed with the `-invert` option may be ignored by clock grouping. Pins declared as having a differential relationship are driven to opposite values by generated patterns.

See Also

[Understanding Flattening Optimization](#)

PI Equivalences Report in TetraMAX Help

Differential Input Models in TetraMAX Help

Cells With Asynchronous Set/Reset Inputs

You can use the `set_drc` command to specify the treatment of latches and flip-flops whose set and reset lines are not off when all clocks are at their off state. By default, these latches and flip-flops are treated as unstable cells, which prevents them from being used during test pattern generation.

To have these latches and flip-flops treated as stable cells, use the `set_drc -allow_unstable_set_resets` command. Then the ATPG algorithm can use the cells with unstable set/reset inputs to improve test coverage. In that case, it is not necessary to define the set/reset inputs as clocks.

In certain cases, the `-remove_false_clocks` option of the `set_drc` command automatically invokes the “allow unstable set/reset” behavior. When a primary input port has been defined as a clock and a DRC analysis determines that the port cannot capture data into a sequential device, the input port is determined to be a “false clock.” In the default DRC configuration, the result is a C4 violation. However, using the `set_drc -remove_false_clocks` command causes automatic removal of the clock declaration for each false clock, instead of a C4 violation.

If a primary input port declared to be a clock is connected to the set/reset inputs of sequential gates, and also to the D inputs of other sequential gates, it is considered a false clock. As a result, the algorithm removes the clock declaration for that port and then enables unstable set/reset cells, just like executing the `set_drc -allow_unstable_set_resets` command.

The `-allow_unstable_set_resets` option can be useful if a scan-enable signal is used to disable the set/reset inputs of scan cells during load. Using this option means that the scan-enable signal does not have to be defined as a clock, which can greatly improve test coverage.

See Also

[Declaring Clocks](#)

[Power Aware Testing with Asynchronous Primary Inputs](#)

Masking Input and Output Ports

You can mask an input port or output port to isolate it from the design during debugging. For example, if a lower-level module you are testing appears to have full controllability and observability of all of its input and output ports in standalone configuration but loses this control when placed in the higher-level module, you might want to mask those inputs and outputs that are not controllable or observable.

You mask an input port by defining a primary input constraint in which the input port is held to an X value. You can define the constraint by using the Add PI Constraints dialog box (see the “Declaring Primary Input Constraints” section) or by using the `add_pi_constraints` command:

```
DRC-T> add_pi_constraints X port_name
```

You mask an output port by listing it in the Add PO Masks dialog box (opened by choosing Constraints > PO Masks > Add PO Masks menu command) or by using the `add_po_masks` command:

```
DRC-T> add_po_masks port_name
```

Masking Scan Cell Inputs and Outputs

TetraMAX ATPG supports a number of scan cell controls. You can define these controls by using the Add Cell Constraints dialog box, or you can enter the `add_cell_constraints` command at the command line.

The following sections describe how to mask scan cell inputs and outputs:

- [Specifying Cell Constraints Locations and Scan Cell Controls](#)
- [Using the Add Cell Constraints Dialog Box](#)
- [Using the add_cell_constraints Command](#)

Specifying Cell Constraints Locations and Scan Cell Controls

You specify the location of the cell constraint using either of the following techniques:

- Use the name of the scan chain and the bit position, with bit 0 as the bit closest to the scan chain output
- Use an instance path name to the scan chain element

You can use any of the following five scan cell controls:

- 0 –The scan cell is always loaded with a 0 during the scan chain load.
- 1 –The scan cell is always loaded with a 1.
- x –The scan cell is always loaded with an X.

- **OX** – No restrictions exist on the loaded value, but any data captured by the regular system clock is considered to be observed as X. That is, the scan cell can be loaded to control logic connected to its outputs, but its data input is always considered X.
- **XX** – The load is always X, and the observe is always X.

Note: The loading of a scan cell with an X value for the X or XX cell constraint provides an X for simulation. However, on a device tester, the X is translated into a 0 or a 1 because you cannot drive an X on a tester.

Using the Add Cell Constraints Dialog Box

The following steps describe how to use the Add Cell Constraints dialog box to define scan cell controls:

1. From the menu bar, choose Constraints > Cell Constraints > Add Cell Constraints. The Add Cell Constraints dialog box appears.
2. Specify the location of the cell constraint by entering the name of a scan chain or instance.
3. Enter a bit position for the scan chain and scan cell control values for the scan chain and instance.
For additional information about the available options, see description of the `add_cell_constraints` command in TetraMAX Help.
4. Click OK.

Using the `add_cell_constraints` Command

You can also define scan cell controls using the `add_cell_constraints` command, as shown in the following example:

```
DRC-T> add_cell_constraints 0 /TOP/U1/sifter/reg42
```

For the complete syntax and option descriptions, see the description of the `add_cell_constraints` command in TetraMAX Help.

Previewing Potential Scan Cells

You can preview the effect on your design of changing flip-flops and latches from nonscan elements to scan elements in scan chains without actually changing your design. To do this, you place one or more nonscan sequential devices in a virtual scan chain. TetraMAX ATPG treats the virtual scan chain as a true scan chain. Remember to set up the clocks, and when you run ATPG, you see the potential effect on test coverage.

Sequential devices in the Set Scan Ability list must meet all DRC rule checks for scan chain elements. Some of the devices might fail DRC because of uncontrolled asynchronous set/reset connections. (TetraMAX ATPG converts the devices into a scan chain but does not change set/reset pins.)

The following sections describe how to preview potential scan cells:

- [Using the Set Scan Ability Dialog Box](#)
 - [Using the set_scan_ability Command](#)
-

Using the Set Scan Ability Dialog Box

You can place the nonscan devices in a virtual scan chain by listing them in the Set Scan Ability dialog box. The following steps describe how to use the Set Scan Ability dialog box to list the nonscan devices in a virtual scan chain:

1. From the menu bar, choose Scan > Set Scan Ability. The Set Scan Ability dialog box appears.
 2. Select the method and add DLAT/DFF gates from the list.
For more information about the controls in this dialog box, see Online Help for the `set_scan_ability` command.
 3. Click OK.
-

Using the `set_scan_ability` Command

You can also place nonscan sequential devices in a virtual scan chain using the `set_scan_ability` command, as shown in the following example:

```
DRC-T> set_scan_ability on core/host/status
```

For the complete syntax and option descriptions, see the description of the `set_scan_ability` command in TetraMAX Help.

The following example adds four devices to the virtual scan chains:

```
DRC-T> set_scan_ability on /top/U1/U2/reg1
DRC-T> set_scan_ability on /top/U1/U2/reg2
DRC-T> set_scan_ability on /top/U1/U2/reg3
DRC-T> set_scan_ability on /top/U1/U2/reg4
```

When you use a list format in the `set_scan_ability` command, you might not be able to write patterns because the patterns include the virtual scan chain. Any patterns that are written will fail simulation unless the design is modified to convert the virtual scan chain into a real scan chain.

Note that the `set_scan_ability` command is not compatible with any type of scan compression. DRC will fail if the STIL procedure file contains a CompressorStructures block.

Transparent Latches

A transparent latch is a latch in which the enable line can be asserted so that data passes through it without activating any of the design's defined clocks. During the rule checking process, TetraMAX ATPG automatically determines the location of all latches in the design and checks to see whether the latches can be made transparent. For ATPG, you must be able to disconnect the latch control from any clock ports.

When latches are transparent, it is easier for TetraMAX ATPG to detect faults around those latches. When latches are not transparent, you might need to use a Full-Sequential ATPG run to get good fault coverage around those latches.

Shadow Register Analysis

A shadow register is not in the scan chain, but is loaded when its master register in the scan chain is loaded, by the same clock or by a separate clock. A shadow register is considered a control point but not an observe point. During the DRC analysis, TetraMAX ATPG searches for nonscan cells that can be considered shadow registers.

If the shadow register's state can be observed at the shadow's master, TetraMAX ATPG classifies the register as an observable shadow. This usually requires defining a shadow.observe procedure in the STL procedure file.

The default is to search for shadow registers. You can disable the default by executing the following command:

```
BUILD-T> set_drc -noshadow
```

Feedback Paths Analysis

During initial processing, TetraMAX ATPG identifies feedback paths within the design and assigns each path a unique feedback path ID.

During DRC, TetraMAX ATPG analyzes the feedback paths to ensure that the loop of logic gates can be broken at some combinational gate within the loop. If the logic loop does not have a blocking point, simulations performed during ATPG will oscillate without resolving to a final value. If DRC analysis cannot find a set of inputs and scan chain load values that can break the loop and still maintain any other constraints in effect, TetraMAX ATPG issues an X1 rule violation.

See Also

[Analyzing a Feedback Path](#)

Procedure Simulation

In addition to the test_setup, load_unload, and Shift procedures, there are other procedures in the STL procedure file or implied by the definition of clock ports. TetraMAX ATPG simulates all of these procedures as part of the design rule checking process to guarantee that they accomplish their intended purposes. For details on the running the various procedures, see "[STIL Procedure Files](#)."

Changing the Design Rule Severity

Each design rule is assigned a severity level that determines the action taken if a rule violation occurs. A design rule violation has possible four severity levels:

- Ignore - The rule is not checked and no messages are issued.
- Warning - Violation of the rule produces a warning message, and the current process continues.
- Error - Violation of the rule produces an error message, and the current processing step is terminated. Before continuing, you must either correct the problem or change the rule severity level.
- Fatal - Violation of the rule produces an error message, and the current processing step is terminated. the severity level cannot be changed. Before continuing, you must correct the problem.

You can change the rule severity level by using the Set Rules dialog box, or by running the `set_rules` command from the command line.

You can determine the severity level setting of a particular rule and the number of violations that have occurred by selecting Rules > Report Rules in the TetraMAX GUI or by running the `report_rules` command.

Using the Set Rules Dialog Box

To change the rule severity by using the Set Rules dialog box:

1. From the menu bar in the TetraMAX GUI, choose Rules > Set Rule Options. The Set Rules dialog box appears.
2. Enter a rule ID and select a severity level.
For additional information about the available options, see the description of the `set_rules` command in TetraMAX Help.
3. Click OK.

Using the `set_rules` Command

You can change the rule severity level of any rule (except those that are Fatal) by using the `set_rules` command, as shown in the following example:

```
BUILD-T> set_rules B5 warning
```

When running DRC (before ATPG) on circuits which include blocks that have both default and high X-tolerant architectures, specify the following command:

```
set_rules R22 warning
```

This will downgrade a check done for fully X-tolerant designs built by DFTMAX compression which were built with blocks that include both default X-tolerant architectures and high X-tolerant architecture.

Understanding the DRC Summary Report

The `run_drc` command performs design rule checking (DRC), and produces a DRC summary report, as shown in the following example:

```
DRC> run_drc top.spf
-----
Begin scan design rule checking...
-----
Begin reading test protocol file top.spf...
End parsing STIL file slo_gin.spf with 0 errors.
Test protocol file reading completed, CPU time=0.08 sec.
-----
Begin Bus/Wire contention ability checking...
Bus summary: #bus_gates=40, #bidi=40, #weak=0, #pull=0,
#keepers=0
    Contention status: #pass=0, #bidi=40, #fail=0, #abort=0, #not_
analyzed=0
    Z-state status : #pass=0, #bidi=40, #fail=0, #abort=0, #not_
analyzed=0
    Bus/Wire contention ability checking completed, CPU time=0.04
sec.
-----
Begin simulating test protocol procedures...
Nonscan cell constant value results: #constant0 = 4, #constant1 =
7
Nonscan cell load value results : #load0 = 4, #load1 = 7
Warning: Rule Z4 (bus contention in test procedure) was violated
12 times.
Test protocol simulation completed, CPU time=0.15 sec.
-----
Begin scan chain operation checking...
Chain c1 successfully traced with 31 scan_cells.
Chain c2 successfully traced with 31 scan_cells.
Chain c3 successfully traced with 31 scan_cells.
Chain c4 successfully traced with 31 scan_cells.
Chain c5 successfully traced with 31 scan_cells.
: : : : :
Chain c44 successfully traced with 30 scan_cells.
Chain c45 successfully traced with 30 scan_cells.
Chain c46 successfully traced with 30 scan_cells.
Scan chain operation checking completed, CPU time=0.47 sec.
-----
Begin clock rules checking...
Warning: Rule C17 (clock connected to PO) was violated 16 times.
Warning: Rule C19 (clock connected to non-contention-free BUS)
```

```
was violated 1 times.  
Clock rules checking completed, CPU time=0.15 sec.  
-----  
Begin nonscan rules checking...  
Nonscan cell summary: #DFF=201 #DLAT=0 tla_usage_type=none  
Nonscan behavior: #C0=4 #C1=7 #LE=11 #TE=179  
Nonscan rules checking completed, CPU time=0.04 sec.  
-----  
Begin DRC dependent learning...  
DRC dependent learning completed, CPU time=1.01 sec.  
-----  
Begin contention prevention rules checking...  
26 scan cells are connected to bidirectional BUS gates.  
Warning: Rule Z9 (bidi bus driver enable affected by scan cell)  
was violated 24 times.  
Contention prevention checking completed, CPU time=0.03 sec.  
-----  
DRC Summary Report  
-----  
Warning: Rule C17 (clock connected to PO) was violated 16 times.  
Warning: Rule C19 (clock connected to non-contention-free BUS)  
was violated 1 times.  
Warning: Rule Z4 (bus contention in test procedure) was violated  
12 times.  
Warning: Rule Z9 (bidi bus driver enable affected by scan cell)  
was violated 24 times.  
There were 54 violations that occurred during DRC process.  
Design rules checking was successful, total CPU time=2.27 sec.
```

scan design rule checking

This indicates the beginning of the scan design rule checking process.

reading test protocol file

The first message indicates the beginning of the reading of the test protocol file. The second message indicates the parsing of the file was successful with no errors. The last message indicates the process is completed and the CPU time in seconds that was used for the process.

Bus/Wire contention ability checking

The first message indicates the beginning of the bus and wire contention checking rules.

The second message summarizes the types of bus gates that are used in the circuit. This includes the total number of bus gates, the number of bidirectional bus gates, the number of weak bus gates (only weak drivers), the number of pull bus gates (a mixture of strong and weak drivers), and the number of bus gates which have a bus keeper.

The next message gives a summary of contention ability status of the bus gates after the analysis is completed. This includes the number of buses which pass (proven contention free), are bidirectional (contention free except for the bidi input), fail (proven contention sensitive), and abort (aborted during analysis).

The next message gives a summary of Z-state ability status of the bus gates after the analysis is completed. This includes the number of buses which pass (proven incapable of attaining a Z state), are bidirectional, fail (proven capable of attaining a Z state), and abort (aborted during analysis).

The last message indicates the process is completed and the CPU time in seconds that was used for the process.

simulating test protocol procedures

The first message indicates the beginning of the simulation of the test protocol procedures. The results of the simulation is used to first determine state elements that have a constant state behavior and those that attain a set value after the scan chain load.

The second message in the example indicate 4 state elements have a constant 0 behavior and 7 state elements have a constant 1 behavior.

The third message indicate 4 state elements are set to 0 and 7 state elements are set to 1 at the end of the scan chain load. During simulation, certain rules are checked. In this case, the rule checking for bus contention during the test procedures was violated 12 times and a warning message is given.

The last message indicates the process is completed and the CPU time in seconds that was used for the process.

scan chain operation checking

The first message indicates the beginning of the scan chain operation checking. The results of the previous simulation are used to verify the operation of the scan chains and identify the associated scan cells. As each scan chain is successfully verified, a message is given indicating its completion with its name and length. The last message indicates the process is completed and the CPU time in seconds that was used for the process.

clock rules checking

The first message indicates the beginning of the clock rules checking. During this process many clock rules are checked and messages are given when violations occur. In this case, two messages are given indicating there were 16 violations of rule C16 and 1 violation of rule C19. The last message indicates the process is completed and the CPU time in seconds that was used for the process.

nonscan rules checking

The first message indicates the beginning of the nonscan rules checking. The objective of this checking is to determine the appropriate behavior for all non scan state elements.

The second message gives a summary of the nonscan state elements. This includes the nonscan DFFs, nonscan DLATs, and the transparent latch usage. In this case, there are no transparent latches.

The next message gives a summary of the calculated nonscan behaviors. This includes C0 (constant 0), C1 (constant 1), LE (edge sensitive state elements that capture on the leading edge of a pulse on the clock pin), and TE (edge sensitive state elements that capture on the trailing edge of a pulse on the clock pin).

The last message indicates the process is completed and the CPU time in seconds that was used for the process.

DRC dependent learning

The first message indicates the beginning of the DRC dependent learning process. Using the behaviors learned during DRC, analyses are performed to determine control ability, observe ability, constraint effects, and blockages due to constraint effects for all gates in the circuit. The last message indicates the process is completed and the CPU time in seconds that was used for the process.

contention prevention rules checking

The first message indicates the beginning of the contention prevention rules checking.

The second message indicates that there were 26 scan cells which had connectivity to bidirectional bus gates. This normally indicates a potential problem that will cause some rule violations.

The next message indicates the rule violations that was the result of that connectivity.

The last message indicates the process is completed and the CPU time in seconds that was used for the process.

DRC Summary Report

The first message indicates the beginning of the summary report. For each rule that had at least one violation, a summary message for that rule is given indicating the number of times it was violated.

The next message indicates the total number of rule violations that occurred during the DRC process.

The last message indicates the process is completed and the CPU time in seconds that was used for the process.

Binary Image Files

A binary image file is a data file that stores design information in an efficient and proprietary format for reading by TetraMAX ATPG. It contains a flattened version of the design, along with some selected TetraMAX settings.

Using an image file provides several key benefits:

- **Simplifies file management**

Because an image file stores all netlist, library, and STL procedure file details in a single file, it is easy to archive and share design data.

- **Avoids repetitive tasks**

When TetraMAX ATPG reads an image file, you do not need to repeat the entire build and DRC phases, since this data is already stored in the file. This results in significant time savings when using large designs.

- **Restricts command usage**

You can create secure image files that allow only a restricted set of commands. These commands are stored in the encrypted image file. You can also control whether schematic viewing is allowed.

When a secure image file is read, the TetraMAX session switches to a secure state in which only the allowed commands can be executed. If you specify a disallowed command, TetraMAX ATPG does not execute it and issues a warning message.

- **Provides intellectual property protection**

TetraMAX ATPG can obfuscate instance, net, and module names when creating a binary image. This provides an additional level of security by hiding design context.

- **Stores context-sensitive design data**

An image file stores different types of design information, depending on what mode is active when you create it:

- When the Test mode is active, both build and DRC data is stored in the image file.
- When the DRC mode is active, only the build data is stored in the image file.

Creating and Reading Image Files

You use the `write_image` command to create an image file and the `read_image` command to read it.

You can also create and read secure image files. This functionality is implemented through the following commands:

- `set_commands [-secure command | -all] [-nosecure <command | -all>]`
- `report_commands [-secure]`
- `write_image file_name [-password string] [-schematic_view]`
- `read_image file_name [-password string]`

Note that TetraMAX ATPG can obfuscate instance, net and module names. This provides an additional level of security by hiding design context. The names are changed to the following format (where "###" is an integer number of any length):

- Instance names use the format u###
- Net names use the format n###

- Module names use the format m###

The `-garble` option of the `write_image` command modifies the names in the output image. You can send this secure image to a third party with controlled data access. You can also translate the modified instance and net names back to the original names using the `-ungarble` option of the `report_nets` command and the `report_instances` command, if the original design database or the unmodified image file is accessible.

After TetraMAX ATPG reads the image, it remains in the same mode in which the image was created (DRC or Test). An image file does not create an identical session as when it was originally created. Some settings and data, such as net names and intermediate levels of hierarchy, are not in the image file. Thus, TetraMAX ATPG can only operate in primitive view and not design view.

You can use the `report_settings -all -command_report` command to view the stored settings.

For details on how to create non-secure and secure image files, see the following sections:

- [Creating a Non-Secure Image File](#)
- [Creating a Secure Image File](#)

Creating a Non-Secure Image File

To create a non-secure image file:

1. Read the netlist file, as shown in the following example:

```
read_netlist top.v
```

2. Read the library models.

```
read_netlist spec_lib.v -library
```

3. Create the design model.

```
run_build_model spec_chip
```

4. Perform design rule checking.

```
run_drc spec_chip.spf
```

5. Write the image file.

```
write_image spec_chip_post_drc.img -violations -replace
```

To read the image during a subsequent run, use the `read_image` command, as shown in the following example:

```
read_image spec_chip_post_drc.img
```

Creating a Secure Image File

You can use a combination of `set_commands` and `write_image` commands to create a secure image file.

Note: When using a secure image file, the following neutral commands are always allowed: `exit`, `alias`, `unalias`, `help`, `source`, `c`, and `pwd`.

To create a secure image file:

1. Read the netlist file, as shown in the following example:

```
read_netlist top.v
```

2. Read the library models.

```
read_netlist spec_lib.v -library
```

3. Create the design model.

```
run_build_model spec_chip
```

4. Perform design rule checking.

```
run_drc spec_chip.spf
```

5. Use the `set_commands` command to specify all commands you want to allow in the secure image. For example:

```
set_commands -secure add_equivalent_nofaults
set_commands -secure add_nofaults
set_commands -secure source
set_commands -secure help
set_commands -secure read_faults
set_commands -secure read_nofaults
set_commands -secure remove_nofaults
set_commands -secure report_licenses
set_commands -secure report_nofaults
set_commands -secure report_patterns
set_commands -secure report_version
set_commands -secure set_simulation
set_commands -secure run_diagnosis
set_commands -secure run_simulation
set_commands -secure set_patterns
set_commands -secure set_diagnosis
```

6. Use the `write_image` command to create the secure image. For example:

```
write_image image_enc.gz -password top_secret \
-schematic_view -replace -garble
```

7. Generate the ATPG patterns.

```
run_atpg -auto
```

8. Write the ATPG patterns to a binary file.

```
write_patterns pat.bin -format binary
```

To read the secure image:

1. Force TetraMAX ATPG to BUILD mode, as shown in the following example:

```
build -force
```

2. Read the image.

```
read_image image_enc.gz -password top_secret
```

3. Read the binary pattern format.

```
set_patterns -external pat.bin
```

4. Create the STIL/WGL patterns to be used with garbled image.

```
write_patterns pat_garbled.wgl -format wgl -external  
write_patterns pat_garbled.stil -format stil -external
```

To translate a garbled name back to the original name:

1. Read the original design database, as shown in the following example:

```
read_netlist specnetlist.v
```

2. Create the design mode.

```
run_build_model ...
```

3. Perform design rule checking.

```
run_drc ...
```

4. Supply the garbled name as argument to get ungarbled name in output.

```
report_instances u43259 -ungarble
```

Save/Restore in TEST Mode

You can use the save/restore feature to reduce the time needed to read the netlists, build the design, and run the design rule checker (DRC) for subsequent ATPG runs. This feature is implemented through the `write_image` and `read_image` commands.

After a successful DRC run, use the `write_image` command while in TEST mode to save the in-memory TetraMAX database (gates) to a file. You can optionally save the DRC violations for the C, D, L, S, X, and Z rules with the `-violations` option. When you later decide to do more runs, issue a `read_image` command to read the database file and proceed.

Optimizing ATPG

You can specify a variety of parameters to control the ATPG process. The "[Running ATPG](#)" section describes some of the basic ATPG settings you can make, including running the ATPG mode and setting a test coverage target value. You can further optimize the ATPG process by setting ATPG constraints and test points, limiting the number of patterns and aborted decisions, applying pattern masking, and running multicore ATPG.

The following sections describe the various settings you can make to optimize ATPG:

- [Using ATPG Constraints](#)
- [Using the Random Decision Option](#)
- [Obtaining Target Test Coverage Using Fewer Patterns](#)
- [Maximizing Test Coverage Using Fewer Patterns](#)
- [Improving Test Coverage With Test Points](#)
- [Optimizing Basic Scan Patterns](#)
- [Limiting the Number of Patterns](#)
- [Limiting the Number of Aborted Decisions](#)
- [Creating Test Patterns for Diagnosing Scan Chain Failures](#)
- [Performing Scan Chain Diagnostics](#)
- [Creating End-of-Cycle Measures in ATPG Patterns](#)
- [Per-Cycle Pattern Masking](#)
- [Deleting Top-Level Ports From Output Patterns](#)
- [Detecting Faults Multiple Times](#)
- [WGL Pattern Generation Options](#)
- [Running Multicore ATPG](#)
- [Running Logic Simulation](#)
- [Data Volume and Test Application Time Reduction Calculations](#)

Using ATPG Constraints

You can use ATPG constraints to define internal restrictions that you cannot define with the `add_pi_constraints` command. ATPG constraints are in effect during ATPG and optionally during test design rule checking (DRC). The use of ATPG constraints is illustrated in the following examples:

- [Usage Example 1](#)
- [Usage Example 2](#)

Usage Example 1

In this example, a library module called FIFO has two control inputs, push and pop. Under normal operation, the control logic for push and pop ensures that both are never asserted at the same time. However, under the random conditions of ATPG, this control is not guaranteed. To ensure that push and pop are never asserted at the same time, you can define an ATPG constraint at the module level by first adding a temporary gate to facilitate the ATPG constraint and then defining the constraint itself.

You want a logic function with a single output that can be monitored to determine that the push and pop pins are at the required logic states. The following steps describe how to define an ATPG primitive to implement this logic function:

1. Choose Constraints > ATPG Primitives > Add ATPG Primitives. The Add ATPG Primitives dialog box appears.
2. In the Type list, select the ATPG primitive `SEL01`. (For a list of all available ATPG primitives, see the online reference for the `add_atpg_primitives` command.) The `SEL01` function produces a 1 as its output if all inputs are 0 or if only one input is 1 and the rest are 0. For the example two-input implementation, `SEL01` produces a 0 only if both inputs are 1.
3. In the ATPG Primitive Name field, type the name you want to give this primitive.
4. In the Module field, type the name of the module in which you want this primitive to be.
5. In the Input Constraints field, enter the inputs that are to be constrained (in this case, push and pop). Click Add after each entry. The inputs are added to the list in the Input Constraints window
6. Click OK.

Alternatively, you can add the primitive using the `add_atpg_primitives` command, as shown in the following example:

```
DRC-T> add_atpg_primitives FIFO_CTRL sel01 -module FIFO push pop
```

The new gate, `FIFO_CTRL`, is added to the module `FIFO` and uses the module-level pins named `push` and `pop` as input to the `SEL01` function. The output pin of the function is referenced by the name `FIFO_CTRL`.

If necessary, you can add more primitives and cascade the logic to build more complex logic functions.

To apply a constraint to the output of the newly added primitive, you can use the `add_pi_constraints` command, as shown in the following example:

```
DRC-T> add_atpg_constraints spec_LABEL 1 -module FIFO FIFO_CTRL
```

This command defines a constraint, referenced by `spec_LABEL`, that holds the output `FIFO_CTRL` to a 1 value. `SEL01` cannot have an output of 1 if both of its inputs are 1, so this constraint ensures that the pins `push` and `pop` are never asserted at the same time.

Usage Example 2

In this example, a combinational gate is buried within the design hierarchy. Under random conditions, there is a timing-sensitive path causing attempts to generate ATPG patterns to fail simulation. Your analysis concludes that if you could hold two of the pins of a four-input NAND gate at a high value, you could block the use of this timing-sensitive path.

The instance path name of the NAND gate is `asic_top/BRL/regbank2/u1`, and the input pins you want to control are A and C.

The following steps describe how to add the required constraints:

1. Choose Constraints > ATPG Constraints > Add ATPG Constraints. The Add ATPG Constraints dialog box appears.
2. For each constraint, you specify a constraint name, the constraint site, and value.
3. You can apply the constraint to a single site or to selected pins of all instances of a module.
4. Click OK.

You can also add the constraints using the `add_pi_constraints` command, as shown in the following example:

```
BUILD-T> add_atpg_constraints NAND_BLK2 1 /asic_top/BRL/u1/C
```

Using the Random Decision Option

You can use the Random Decision check box in the Run ATPG dialog box to specify how TetraMAX ATPG makes the initial choice for any algorithm decision cone during ATPG pattern generation. By default, Random Decision is off and the initial choice is made based on controllability criteria. Checking Random Decision for ATPG pattern compression can result in a smaller number of patterns.

The following `set_atpg` command is equivalent to checking the Random Decision check box:

```
TEST-T> set_atpg -decision random
```

See Also

[Specifying ATPG Settings](#)

Obtaining Target Test Coverage Using Fewer Patterns

To obtain a target test coverage value while minimizing the number of patterns, follow the procedure for obtaining maximum test coverage and set the coverage percentage (`-coverage` option) to a number between 1 and 99 that represents your target test coverage.

Note: TetraMAX ATPG creates patterns in groups of 32 and checks this limit at each 32-pattern boundary, so the patterns generated might exceed the target test coverage.

Review the transcript. If you find that your target is met with the first few patterns of the last group of 32 and you do not want to include all of the last group of patterns, use the `write_patterns -last` command to truncate the patterns written as output at the point at which the target was met.

The target coverage is affected by your use of the `set_faults -report` command. If fault reporting is set to collapsed, the target percentage is in collapsed fault numbers. If fault reporting is set to uncollapsed, the target percentage is in uncollapsed numbers. The test coverage obtained through the uncollapsed fault list is usually higher and within a few percentage points of the test coverage obtained through the collapsed fault list (note, however, test coverage can slightly more with the fault report set to collapsed compared to the test coverage with fault coverage set to uncollapsed). To be conservative, set fault reporting to collapsed before you generate patterns for a specific target coverage. When you have finished, display the test coverage using the uncollapsed fault list numbers. Often, the actual test coverage achieved is higher than your target.

Maximizing Test Coverage Using Fewer Patterns

To obtain the maximum test coverage while minimizing the number of patterns:

1. Obtain an estimate of test coverage using the Quick Test Coverage technique. For details, see "[Quickly Estimating Test Coverage](#)." If you are not satisfied with the estimate, determine the cause of the problem and obtain satisfactory test coverage before you attempt to achieve minimum patterns.
2. Set the abort limit to 100–300.
3. Set the merge effort to High.
4. Execute `run_atpg -auto_compression`.
5. Examine the results. If there are still too many NC or NO faults remaining, increase the Abort Limit by a factor of 2 and execute `run_atpg` again.

Improving Test Coverage With Test Points

You can improve TetraMAX test coverage by adding control and observation points to specific areas with known low controllability and observability. TetraMAX ATPG then generates

additional patterns for faults that are controlled or fed into these points. This process is particularly useful if you want to achieve very high test coverage targets — usually in the 99 percent range.

You can use TetraMAX ATPG to further improve test coverage by performing an analysis to determine the optimal placement of test points.

The following sections describe how to improve test coverage with test points:

- [Test Points Analysis Options](#)
 - [Running the Test Points Analysis Flow](#)
 - [Limitation](#)
-

Test Points Analysis Options

You can use the `analyze_test_points` command to select a particular type of analysis:

```
analyze_test_points -target <pattern_reduction | testability | fault_class>
```

The analysis options are described as follows:

- `pattern_reduction` — Uses static analysis with SCoAP (Sandia Controllability and Observability Analysis Program) numbers to target reduced pattern size with observe points (does not require prior ATPG).
 - `testability` — Uses iterative static analysis with random patterns to target improved test coverage with control and observe points (does not require prior ATPG).
 - `fault_class` — Uses dynamic analysis with fault cone topology to target improved test coverage with observe points for fault classes (requires initial ATPG for analysis of fault cones).
-

Running the Test Points Analysis Flow

The following steps describe the flow for running test-point insertion:

1. Run the `run_atpg -auto` command or use any other method for generating patterns.

Note: If you do not perform ATPG before running the `analyze_test_points` command, all undetected faults are analyzed, which might result in very long runtimes.

2. Run the `analyze_test_points` command to generate a list of test points. For example:

```
analyze_test_points -target fault_class -test_points_file tp_file_a
```

Note: You can run the `analyze_test_points -target testability` or the `analyze_test_points -target pattern_reduction` commands before or after ATPG to obtain a list of test points. A previous ATPG run is only required when you use the `-target fault_class` option with the `analyze_test_points` command.

3. Use the `run_atpg -auto` command to launch another ATPG run. TetraMAX ATPG estimates the test coverage improvement by reading in the generated test points file. For example:

```
run_atpg -auto -observe_file tp_file_a
```

Note: The total number of faults reported after running ATPG will not include the faults from the additional test points.

4. Use DFT Compiler to insert the test points by reading in the file generated by the `analyze_test_points` command, then rerun TetraMAX ATPG on the new netlist to generate the final ATPG patterns and coverage.

Limitation

Note the following limitation associated with test points analysis:

- If you have a LSSD design, you can use the `analyze_test_points` command in TetraMAX ATPG. However, DFT Compiler does not support the insertion of observe and control test points on this style of scan. In this case, a TESTXG-61 message is issued in DFT Compiler.

Optimizing Basic Scan Patterns

You can use the `-optimize_patterns` option of the `run_atpg` command to produce a compact set of patterns with high test coverage. This option enables you to use a single `run_atpg` command instead of iterating multiple `run_atpg` commands and manually adjusting various parameters.

When the `-optimize_patterns` option is set, TetraMAX ATPG monitors the ATPG process and dynamically adjusts the internal algorithms to generate a compact pattern set. The trade-off is a longer runtime. All manually specified `run_atpg` settings, such as abort limits, minimum detects, and merge limits, are ignored during this operation. However, these settings are restored after pattern optimization is completed.

Note that the `-optimize_patterns` option generates two-clock ATPG patterns as basic scan patterns. But they are stored, read, and simulated as fast-sequential patterns. As a result, a fault simulation that uses two-clock ATPG patterns usually takes longer than the original ATPG run.

The `-optimize_patterns` option of the `run_atpg` command will work with the `-chain_test`, `-coverage`, and `-patterns` options of the `set_atpg` command. This option also works with all power aware options of the `set_atpg` command. However, the power aware options might impact the effectiveness of the pattern optimization process.

The `-optimize_patterns` option is useful during a final TetraMAX ATPG run when you want to optimize the pattern count. It generates a lower number of patterns and produces similar test coverage compared to a single `run_atpg -auto_compression` command. You cannot use the `-optimize_patterns` option with any additional `run_atpg` options.

You should use the `run_atpg -auto_compression` command for general pattern generation purposes, such as initial test coverage estimates, writing patterns for verification, analyzing the effects of various options, and obtaining good test coverage and pattern count without increased runtimes. For details on using the `-auto_compression` option, see [Using Automatic Mode to Generate Optimized Patterns](#).

Note the following limitations when using the `-optimize_patterns` option:

- Multiple `run_atpg` commands are supported, but pattern optimization can only be specified one time.
- A learned recipe is not saved.
- Fast-Sequential and Full-Sequential ATPG modes are not supported.
- Be aware that unlike the `run_atpg -auto_compression` command, specifying `set_atpg -capture_cycle number` will not enable Fast-Sequential ATPG during the pattern optimization process. To run Fast-Sequential top-off ATPG, it must be done as an extra step. For example:

```
run_atpg -optimize_patterns  
set_atpg -capture 4  
run_atpg -auto fast_sequential
```

- Only stuck-at and transition fault models are supported.
- Distributed ATPG is not supported.

Limiting the Number of Patterns

By default, the number of ATPG patterns TetraMAX ATPG produces is limited only by the RAM and disk space of your computer or workstation. You can specify a limit on the number of patterns by entering an integer value in the Max Patterns field of the Run ATPG dialog box, or by issuing a command similar to the following example:

```
TEST-T> set_atpg -patterns 1234
```

If there is a pattern limit in effect, you can turn it off by running the value 0 as the pattern limit.

Limiting the Number of Aborted Decisions

The search for a pattern by the ATPG algorithm involves making a decision and certain assumptions, setting inputs and scan chain values, and determining whether controllability and observability can be attained. When an assumption is proved false or some restriction or blockage is encountered, the algorithm backs up, remakes the decision, and proceeds until the abort limit is reached or a pattern is found to detect the fault.

To control the level of effort used in searching for a pattern to detect a specific fault, use the `-abort_limit` option of the `set_atpg` command or enter a number in the Abort Limit field of the Run ATPG dialog box. The default limit is 10. Higher numbers indicate higher levels of effort.

The default of 10 has been found to return reasonable results for most designs. Some possible reasons for adjusting the abort limit are,

- You want a quick estimate of total coverage (see [“Quickly Estimating Test Coverage”](#)).
- You find that after performing pattern generation, you have ND (not detected) faults remaining. See [“Analyzing the Cause of Low Test Coverage”](#).
- You have aborted buses reported during design rule checking (DRC). See [“Analyzing Buses”](#).
- You are using a high compression effort and you want to generate enough patterns to ensure that the CPU time spent merging patterns is worthwhile.

Creating Test Patterns for Diagnosing Scan Chain Failures

By default, the `run_atpg` command creates an initial pattern, called a chain test, to test scan cells, scan clocking, and scan enable signals. This pattern does not pulse capture clocks or asynchronous set and reset signals. It only loads and unloads the repeating pattern of 0 and 1 signals. If the chain test pattern fails, TetraMAX ATPG assumes that all failures are caused by scan chain defects.

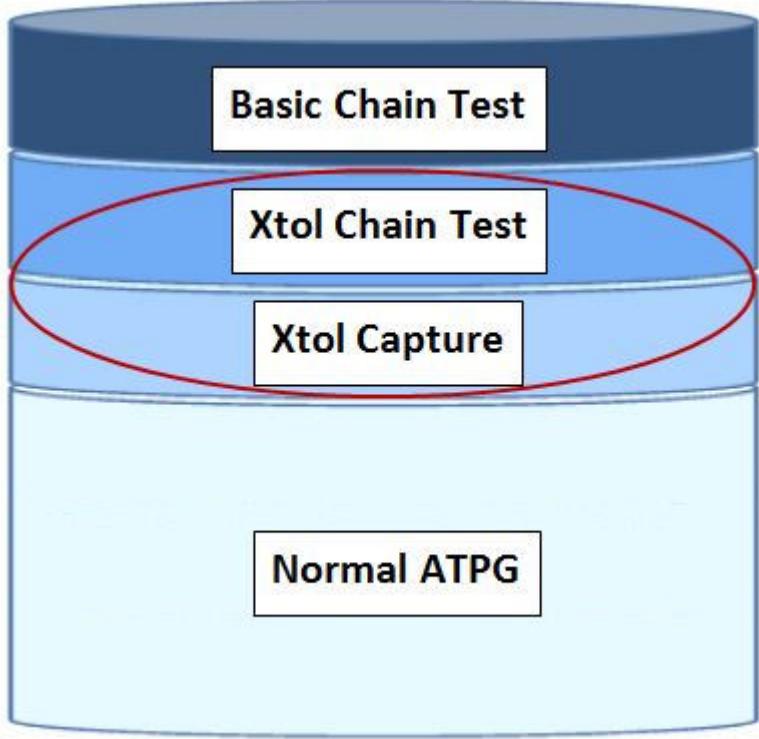
If your design uses DFTMAX compression with high X-tolerance, you can use the `-xtol_chain_diagnosis` option of the `set_atpg` command to create additional patterns that improve the identification of failing scan chains, failing scan cells, and multiple chain defects. When you specify the `-xtol_chain_diagnosis` option, the `run_atpg` command creates two additional sets of patterns:

- X-tolerant chain tests (also known as augmented chain test patterns)
- X-tolerant capture patterns

After generating these additional pattern sets, the `run_atpg` command continues generating normal capture patterns and targets the remaining undetected faults.

Figure 1 shows the components of the final generated pattern set when the `-xtol_chain_diganosis` option is enabled.

Figure 1: Pattern Set Generated Using the `-xtol_chain_diganosis` Option



The following sections explain the process for generating test patterns for diagnosing scan chain failures:

- [Understanding DFTMAX Unload Modes and Chain Diagnosis Patterns](#)
- [Generating Pattern Sets](#)

For more information on DFTMAX high X-tolerance scan compression, see the "Managing X Values in Scan Compression" chapter in the *DFTMAX User Guide*.

Understanding DFTMAX Unload Modes and Chain Diagnosis Patterns

The X-tolerant chain tests and X-tolerant capture patterns use additional unload modes from the high X-tolerance DFTMAX architecture. These modes dynamically configure the compressor so each internal scan chain is observed on no more than one scan output pin of the device under test. These modes are classified as either N:1 or 1:1 modes.

The N:1 modes observe multiple internal scan chains on each scan output pin. The 1:1 modes observe a single internal scan chain on each scan output pin, which is optimal for mapping tester failures back to a failing scan cell. Each 1:1 mode can observe only a subset of all internal scan chains, inversely proportional to the compression ratio. As a result, TetraMAX ATPG implements several 1:1 modes so it can directly observe all internal scan chains. You can use the

`report_compressors -unload` command to report the number of unload modes and list the 1:1 modes.

The X-tolerant chain tests use the 1:1 X-tolerant modes to observe individual chains. Each 1:1 mode is enabled for 20 shift cycles, and each X-tolerant chain test pattern has an additional padding pattern. To calculate the number of X tolerant chain test patterns, multiply the compression ratio by 20 and divide by the number of shift cycles. You should double this number to account for padding patterns. For example, a design with 32 scan I/Os, 1600 internal chains, and a maximum chain length of 250 requires 8 ($=2*1600*20/(32*250)$) additional patterns.

The X-tolerant capture patterns use all available N:1 or 1:1 X-tolerant modes. These patterns pulse capture clocks and target primary and secondary faults similar to patterns produced from standard ATPG. However, X-tolerant capture patterns are optimized to improve the diagnosis of failing scan cells, while standard ATPG maximizes the number of faults detected per pattern. When you specify the `-xtol_chain_diagnosis low` option of the `set_atpg` command, 32 capture patterns and 32 padding patterns are generated that use only N:1 X-tolerant modes. These modes provide a limited improvement for identifying failing scan cells. When the `high` option is specified, TetraMAX ATPG generates 10 capture patterns for each available 1:1 X-tolerant mode. Specifying the `high` option creates additional patterns which provide a significant improvement for diagnosing chain defects.

Generating Pattern Sets

To generate pattern sets for diagnosing scan chain failures, specify the `-xtol_chain_diagnosis` option of the `set_atpg` command, followed by the `run_atpg` command.

The following example creates a single pattern set for both accurate diagnosis of scan chain defects and high manufacturing test coverage:

```
TEST-T> set_atpg -xtol_chain_diagnosis high
TEST-T> run_atpg -auto
```

You might need two separate patterns sets: one for accurate diagnosis of scan chain defects and another for high manufacturing test coverage. You can use the `run_atpg -only_chain_diagnosis` command to terminate ATPG after generating the X-tolerant chain tests and capture patterns. The following example shows how to generate a separate pattern set for diagnosis of chain defects. This pattern set includes an additional 100 standard ATPG patterns to further improve diagnosis resolution.

```
TEST-T> set_atpg -xtol_chain_diagnosis high
TEST-T> run_atpg -only_chain_diagnosis -auto
TEST-T> set_atpg -xtol_chain_diagnosis off
TEST-T> set_atpg -patterns [expr [sizeof_collection \
[get_patterns -all]] + 100]
TEST-T> run_atpg -auto
```

See Also

[Performing Scan Chain Diagnosis](#)

[Preparing for ATPG](#)

Performing Scan Chain Diagnostics

Functional logic diagnostics assumes that scan data is properly loaded and unloaded. If patterns show failures during the chain test, a chain defect is interfering with the loading and unloading processes. TetraMAX scan chain diagnostics isolates the defects that affect scan chain shifting.

You can use both standard scan patterns and DFTMAX patterns for scan chain diagnostics. If you are testing an X-tolerant design, TetraMAX ATPG can generate additional chain test patterns that use the X-tolerant modes to directly observe a group of chains at the scan outputs. For more information on this process, see the "[Creating Test Patterns for Diagnosing Scan Chain Failures](#)" section.

The following sections explain how to perform scan chain diagnostics:

- [Running Scan Chain Diagnostics](#)
- [Understanding the Scan Chain Diagnosis Report](#)
- [Diagnosing Defects Related to Power Issues](#)

Running Scan Chain Diagnostics

Chain diagnostics are enabled by default, and can be disabled using the `set_diagnosis -noauto` command. For optimal accuracy, you should use failure data from ten or more patterns. Always provide TetraMAX ATPG with as many failures as possible, including failures that occur when running the chain test pattern. The following example shows how to set up and run scan chain diagnosis:

```
set_patterns -external pat.stil  
set_diagnosis -auto  
run_diagnosis fail.log
```

Understanding the Scan Chain Diagnosis Report

Scan chain diagnosis identifies several types of defects that affect shifting, including slow clock signals that cause a hold time violation and reset lines stuck at an active value.

The output diagnosis report identifies the location of stuck-at, slow-to-rise, slow-to-fall, fast-to-rise or fast-to-fall faults. The latter two fault types address hold time problems that affect the scan chain shift operation.

To isolate the location of the defect, TetraMAX scan chain diagnosis analyzes the control and observability of scan cells. For example, assume that a stuck-at fault prevents scan cell A from shifting to scan cell B. In this case, scan cell A, and all cells located before it, drive valid values to functional logic. These cells appear as tied cells when they are unloaded, and are therefore unobservable. Scan cell B, and all cells that follow it, drive invalid values to functional logic, but they might capture observable valid values.

The diagnosis report includes a set of possible defect locations (chain, cell position, and instance name). It also includes a match percentage score that indicates the confidence of each location. This score is a percentage that measures the degree to which a failure on the tester matches a

simulated chain defect at that location. The predicted type of defect is also included in the diagnosis report. For example:

```
fail.log scan chain diagnosis results: #failing_patterns=79
-----
defect type=fast-to-rise
match=100% chain=c0 position=178 master=CORE/c_rg0 (46)
match=100% chain=c0 position=179 master=CORE/c_rg2 (57)
match= 98% chain=c0 position=180 master=CORE/c_rg6 (54)
```

The example report indicates that a fast-to-rise defect is likely the cause of the failures. It also identifies the three scan cell locations that have an output with the physical defect. Some chain test patterns do not fail on the tester, even though the failures appear to be related to a chain defect. Also, the tester might not collect all failures for the chain test patterns. In both cases, scan chain diagnostics cannot analyze or locate the defect location. To address these situations, use the `-assume_chain_defect` option of the `run_diagnosis` command to specify a defect location and force TetraMAX ATPG to obtain the scores.

Diagnosing Defects Related to Power Issues

TetraMAX ATPG can also improve the characterization and diagnosis of chain defects related to power issues. To screen failures based on switching activity, TetraMAX ATPG uses the quiet chain test patterns instead of regular chain test patterns. Specify the `-quiet_chain_test` option of the `set_atpg` command to enable the `run_atpg` command to generate quiet chain test patterns.

For more information, see the "[Applying Quiet Test Patterns](#)" section.

Creating End-of-Cycle Measures in ATPG Patterns

The TetraMAX combinational ATPG algorithm is based on a `preclock` measure of scan outputs and regular design outputs. This preclock measure requires a fundamental event order within a tester cycle of:

- Force inputs
- Measure outputs
- Pulse capture clocks (optional)

This preclock measure has been chosen because it enables superior ATPG pattern generation performance without compromising on pattern count or tester cycle count.

Many ASIC vendors and users prefer to have patterns with an event order using `postclock` or End-of-Cycle measures. A `postclock` measure seems to be a more comfortable form because it matches the event order of most functional patterns and is perhaps easier to debug.

Many ASIC vendors claim that they can only accept `postclock` measure format. It is rare to find an ASIC tester which does not support the `preclock` measure. More often than not it is a software translation limitation rather than a tester limitation. The fundamental event order for a `postclock` measure cycle is:

1. Force inputs
2. Pulse capture clocks (optional)
3. Measure outputs

The TetraMAX combinational ATPG algorithm will not produce this postclock form of patterns. However, the postclock style of patterns can be created using some post processing techniques applied during the `write_patterns` command.

Drawbacks of Using End-of-Cycle Measures

Here are some drawbacks of creating End-of-Cycle style ATPG patterns:

- The internal pattern format is in preclock format and attempting to compare internal patterns to an external form in STIL, Verilog, VHDL, etc. is more difficult.
- At least one additional tester cycle is needed for every ATPG pattern. This additional cycle is placed in the `load_unload` procedure and performs a scan chain pre-measure before the `Shift` procedure.
- Capture Clock procedures cannot be condensed into a single tester cycle and must be defined with a minimum of 2 tester cycles. The first cycle performs a force PI, measure PO, and the second cycle performs an optional clock pulse.

In general terms, the cost of implementing the End-of-Cycle measure is two additional tester cycles for every ATPG pattern generated. There is no increase or decrease to overall test coverage or the number of ATPG patterns produced by choosing End-of-Cycle measures over preclock measure. This can or cannot be significant, depending upon your budget for test cycles or tester time.

Requirements Needed to Produce End-of-Cycle Measures

To create End-of-Cycle style ATPG patterns with the `write_patterns` command the following setup steps are required:

1. The DRC procedure file must contain a timing definition block and the time at which outputs and scan outputs are measured must be defined to occur at the end of a test cycle, after any potential clock pulses.
2. All capture procedures must be defined using two or more test cycles and the event order must be:
`cycle 1: force PI's, measure PO's`
`cycle 2: mask PO's, pulse clocks`
3. The `load_unload` procedure must pre-measure the first scan chain output before the first scan shift is performed.

In this case, you are still measuring outputs before a clock. You do not change the fundamental event order which must continue to be: 1) force PI's, 2) measure PO's, 3) pulse clocks; make sure that relative to a single tester cycle timing, the measures occur after any clock pulses. For example, if you define tester timing for a 100nS period in which PI's are forced at offset zero, a

clock is pulsed from 50 to 70ns and outputs are measured at 99ns, then your "capture_XXX" procedures produce a 2-cycle timing of:

time	action	cycle
000	force PI's	1
050	assert clock (but inhibited)	1
070	remove clock	1
099	measure PO's	1
100	force PI's (no change needed)	2
150	assert clock	2
170	remove clock	2
199	measure PO's (masked)	2

The fundamental event order is still that of the preclock timing under which the ATPG patterns are generated but the per cycle timing is such that measures are performed at the end of a tester cycle.

See Also

[write_patterns](#)

[End-of-Cycle Measures and Load_Unload](#)

[End-of-Cycle Measures and Timing](#)

[End-of-Cycle Measures and Capture Procedures](#)

Deleting Top-Level Ports From Output Patterns

Some netlist formats include nonlogic top-level ports (for example, power and ground). ATPG patterns that include power and ground can create problems with simulation. You can eliminate these and other unwanted top-level ports from the generated patterns using the `add_net_connections` command.

The following example removes the top-level input ports `pwr1`, `pwr2`, and `pwr3` from the generated patterns:

```
BUILD-T> add_net_connections pwr1 pwr2 pwr3 -remove
```

Note: This command modifies only the in-memory image of the design. These changes do not appear in the output from the `write_netlist` command.

Detecting Faults Multiple Times Using N-Detect

The N-detect feature attempts to detect faults n times in ATPG. The default is one fault detection. During fault simulation, the fault is kept in the active list until it is detected n times. Studies have shown that detecting faults with multiple patterns helps catch defects that cannot be modeled with standard fault models. Examples include transistor stuck-open or cell-level faults.

Pattern size, memory consumption, and runtime is larger than with the default one fault detection.

With the exception of the IDDQ and path delay fault models, all other fault models are supported. Multicore ATPG, distributed processing, Full-Sequential ATPG, and fault simulation are not supported.

The N-detect capability is implemented with options of the `run_atpg`, `run_fault_sim`, and `report_faults`. See Online Help for each of these commands for descriptions of the N-detect options.

N-detect ATPG should be used in conjunction with the `set_atpg -decision random` command to increase the probability of detecting the faults in different ways. Note that TetraMAX ATPG does not guarantee that each fault is detected in different ways.

See Also

[Distributed ATPG Limitations](#)

WGL Pattern Generation Options

The following sections explain the various WGL pattern generation options:

- [Creating LSI-Compatible WGL Patterns](#)
- [Creating NEC-Compatible WGL Patterns](#)
- [Scan Chain Padding](#)
- [Scan Chain Definition Choices](#)
- [Macro Usage](#)
- [Grouping Bidirectional Port Data](#)
- [Controlling Port Data Order](#)
- [Specifying Windowed Measures in WGL](#)
- [Delayed Input Force Timing and Force Prior](#)
- [Balancing Vector and Scan Statements via Last Scan](#)
- [Mapping Bidirectional Ports Within Vector Statements](#)
- [Mapping Bidirectional Ports Within Scan Statements](#)
- [Adjusting Pattern Data for Serial vs. Parallel Interpretation](#)
- [Selecting Scan Chain Inversion Reference](#)
- [Effect of CELLDEFINE](#)
- [Ambiguity of the Master Cell](#)

See Also

`set_wgl`
`set_buses`

How Do I Create LSI-Compatible WGL Patterns?

Answer:

To produce LSI-compatible WGL output you need to use the `set_drc`, `set_buses`, `set_simulation`, and `set_wgl` commands, as shown in the following example:

```
set_drc -nomulti_captures_per_load
set_buses -external_z x
set_simulation -xclock_gives_xout
set_rules c13 error
set_rules z4 error
set_wgl -nolast_scan
set_wgl -scan_map keep
set_wgl -pre_measured
set_wgl -inversion_reference master
set_wgl -chain_list shift
set_wgl -nomacro -nopad -nogroup_bidis
set_wgl -bidi_map { 0x 0- 1x 1- xx x- z0 -0 z1 -1 zx -x zz -z }
```

Note the following:

- Scan shifts must use a single tester cycle. For more information, see "[Defining the Shift Procedure](#)."
- Scan Chain names defined in the STIL procedure file must not contain spaces or other white space. For example, use "chain_1" instead of "chain 1".
- You must define the end-of-cycle timing, as follows:
 - a. The timing block must define the end-of-cycle measure. For more information, see "[Creating End-of-Cycle Measures in ATPG Patterns](#)."
 - b. The load_unload procedure must use pre-measure scan outputs. For more information, see "[Defining the load_unload Procedure](#)."
- You can use the ReflectIO protocol. However, unless all bidirectional pins are fully controlled, you should avoid this protocol since it can create patterns which fail in simulation and might contain contention when all BIDI pins are not controlled.

For a design with bidirectional ports, the ReflectIO protocol causes each capture_XXX procedure to use the reflectIO style of syntax. For example, you can define all clocks and then issue the `set_drc -bidi_control_pin` command followed by a `write_drc` command to create a template STIL procedure file. Then modify the capture_XXX procedures to appear similar to the following 3-cycle protocol:

```
capture_CLK {
    W _default_WFT_;
    V { _pi=\r15 # ; _po=\j \r44 % ; } # force PI, TN=1
    V { TN=0; _io=\r32 z ; _po=\j \r44 X ; } # disable bidis
```

```
V { _io=\m \r32 % ; CLK=P; } # reflect bidis, pulse CLK
}
```

- All capture_XXX procedures for clocks must have the same number of tester cycles, V{...} constructs. If you use a three cycle capture for 'CLK', then you must also use a three-cycle capture for 'RST', 'CLK2', etc. This includes the non-clocking capture procedure named `capture`.
- Use a [test_setup procedure](#) to initialize all input pins to a known value in the first test cycle. Initialize bidirectional pins to Z.
- If inputs are applied with a delay on the tester, then the Timing block of the STIL DRC procedure file should include a "ForcePrior" or "P" character at time offset zero of each cycle before applying the required value within that cycle. This generates a V6 warning during DRC which will have to be ignored. There is an example of ForcePrior at the end of topic: Controlling Pin Timing in STIL
- You can use only one timing block.
- Use the `-order_pins` option of the `write_patterns` command when writing WGL patterns.
- Do not use the `-measure_forced_bidis` option of the `write_patterns` command when writing WGL patterns
- Contact LSI for the latest advice and application notes concerning the use of TetraMAX ATPG.

See Also

[set_wgl](#)
[set_drc](#)
[set_simulation](#)
[set_contention](#)
[write_drc_file](#)
[write_patterns](#)
[End-of-Cycle Measures and Load_Unload](#)
[End-of-Cycle Measures and Timing](#)
[End-of-Cycle Measures and Capture Procedures](#)

How Do I Create NEC-Compatible WGL Patterns?

Answer:

To produce NEC-compatible WGL output, you need to use both the `set_simulation` and `set_wgl` commands, as shown in the following example:

```
set_simulation -strong_bidi_fill
set_wgl -nomacro
set_wgl -nopad
set_wgl -notester_ready
set_wgl -inversion_reference master
```

```
set_wgl -scan_map dash
set_wgl -bidi_map { 0x 0- 1x 1- xx x- z0 -0 z1 -1 zx -x zz -z -x -
- z- -- }
```

Note the following:

- Scan shifts must use a single tester cycle. For more information, see "[Defining the Shift Procedure](#)."
- You must define the end-of-cycle timing, as follows:
 - a. The timing block must define the end-of-cycle measure. For more information, see "[Creating End-of-Cycle Measures in ATPG Patterns](#)".
 - b. The load_unload procedure must use pre-measure scan outputs. For more information, see "[Defining the load_unload Procedure](#)".
 - c. The clock capture procedures must use the two-cycle end-of-cycle measure format. For more information, see "[Defining Capture Procedures in STIL](#)".
- You must explicitly initialize bidirectional ports to non-Z values in the load_unload procedure.
- Use the test_setup procedure to eliminate uninitialized ports at T=0. For more information, see "[Defining the test_setup Procedure](#)".
- Use the test_setup procedure to eliminate floating ports at T=0.
- Do not use the -measure_forced_bidis option of the write_patterns command when writing WGL patterns.
- Use the WGL to ALB to Verilog translation path. Other paths, such as WGL to ALB to CPT, have not been validated to work.

See Also

[set_wgl](#)
[set_simulation](#)
[set_contention](#)
[write_patterns](#)
[End-of-Cycle Measures and Load_Unload](#)
[End-of-Cycle Measures and Timing](#)
[End-of-Cycle Measures and Capture Procedures](#)

WGL Scan Chain Padding

When a design has more than one scan chain and the scan chains are not all the same length then you have the option of causing the WGL patterns to be written so that all scan load and unload data is the same length (`set_wgl -pad`) or is only the length of the scan chain (`set_wgl -nopad`). The default is not to pad, and this is preferred by most vendors.

When padding is enabled, the pad value may be any one of 0, 1, or X and you select which by the `-pad_character` option of the `write_patterns` command when the WGL patterns are written. The default when padding is enabled, is to pad with a zero. Note, however, that

when padding is enabled and a particular pad character is chosen that this will have no effect on the padding used for the chain test patterns. The padding for chain test patterns is always the continuation of the repeating string 0011.

The first example shows a portion of the WGL `SCANSTATE` block for a design with three scan chains of length 2, 3, and 8 bits where padding is disabled.

```
# scan chain padding disabled
scanstate
c1L0 := c1G(11);
c2L1 := c2G(011);
c3L2 := c3G(00110011);
c1E3 := c1G(00);
c2E4 := c2G(100);
c3E5 := c3G(11001100);
```

The second example shows the same data with scan chain padding enabled and a pad character of X used so that it is easier to see where the padding occurs. For scan load strings the padding occurs on the left (first shifted in) for all shorter chains. For scan unload strings the padding occurs on the right (last shifted out).

```
# scan chain padding enabled with pad = X
scanstate
c1L0 := c1G(XXXXXX11);
c2L1 := c2G(XXXXX011);
c3L2 := c3G(00110011);
c1E3 := c1G(00XXXXXX);
c2E4 := c2G(100XXXXX);
c3E5 := c3G(11001100);
```

See Also

`set_wgl`
`set_buses`

WGL Scan Chain Definitions

By convention, the `scanchain` block in WGL defines the instances in the physical sequence of each scan chain, starting at the scan input, and traversing to the scan output. The number of instances in the scan chain matches the number of bits called for in the `scanstate` block for loading or observing from the scan chain.

On some designs, generally those with JTAG used during ATPG, the final scan chain shift is done outside of the scan loop. This translates into the "scan()" vector being shortened by one bit and an additional vector() or more being added to the procedure to handle the final shift outside of the scan statement. Now most WGL translators require that the number of bits defined in the `scanchain` block match the physical length of the scan chain. However, a few require that the number of bits match the length of data to be loaded by the "scan()" statements. The `-chain_list` option controls how the scan chain is listed in the `scanchain` block. The default is `all`.

which causes all instances in the scan chain to be included in the defining list. Optionally specifying `shift` causes the list to match only those bits loaded by the "scan()" statements.

The first examples shows the default `scanchain` block for a design with two scan chains of 5 and 4 bits.

```
# set_wgl -chain_list all
scanchain
  chain1 ["sil", "A4", !, "A3", "A2", "A1", "A0", "so1" ];
  chain2 ["si2", "B3", "B2", "B1", "B0", !, "so2" ];
end
```

The second example shows the same scanchain block when the final shift of the scan chain is done outside of the Shift procedure and a selection of -chain_list shift is used. The final instance in each scan chain "A1", and "B1" have been omitted from the scan chain definitions.

```
# set_wgl -chain_list shift
scanchain
  chain1 ["sil", "A4", !, "A3", "A2", "A1", "so1" ];
  chain2 ["si2", "B3", "B2", "B1", !, "so2" ];
end
```

See Also

`set_wgl`
`set_buses`

Macro Usage in WGL

WGL supports the definition of macros. Macros can be used to represent commonly repeated sequences and the use of macros can lead to more compact WGL pattern files. TetraMAX ATPG will write WGL using macros if the `set_wgl -macro` option has been used. Most vendors do not support macros as this requires a more complex WGL reader and so the TetraMAX default is not to use macros.

When macros are enabled, TetraMAX ATPG adds various macro definitions to the WGL pattern file. The following example is a macro for a capture procedure for the port CLK. There will generally be a macro for each procedure in the DRC file.

```
# an example macro definition
macro capture_CLK (SDI3_I, SDO1_I, D0_I, D2_I, CLK, RSTB, SDI,
    INC, SCAN_9, SDI3_O, SDO1_O, D0_O, D2_O, P, SDO, CO)
vector(tp1) := [ @SDI3_I @SDO1_I @D0_I @D2_I @CLK @RSTB @SDI
    @INC @SCAN_9 X X X X XX XX X ];
vector(tp1) := [ @SDI3_I @SDO1_I @D0_I @D2_I @CLK @RSTB @SDI
    @INC @SCAN_9 @SDI3_O @SDO1_O @D0_O @D2_O @P
    @SDO @CO ];
vector(tp1) := [ @SDI3_I @SDO1_I @D0_I @D2_I 1 @RSTB @SDI
    @INC @SCAN_9 X X X X XX XX X ];
endmacro
```

The first following example shows a segment from a WGL *PATTERN* block which does not use macros and the second example is the same information using macros.

```
# example patterns without macros
pattern group_ALL ("SDI3":I, "SDO1":I, "D0":I, "D2":I, "CLK",
"RSTB", "SDI[1]", "SDI[2]", "INC", "SCAN", "SDI3":O, "SDO1":O,
"D0":O, "D2":O, "P[0]", "P[1]", "SDO[2]", "SDO[3]", "CO")
{ test_setup }
vector(tp1) := [ Z Z Z Z 0 1 0 0 0 0 X X X X X X X X X ];
vector(tp1) := [ Z Z Z Z 0 0 0 0 0 0 X X X X X X X X X ];
vector(tp1) := [ Z Z Z Z 0 1 0 0 0 0 X X X X X X X X X ];

{ scan_test }
{ pattern 0 }
{ load_unload }
vector(tp1) := [ X X X X 0 1 X X 0 0 X X X X X X X X X ];
vector(tp1) := [ X Z X X 0 1 X X 0 1 X X X X X X X X X ];
scan(tp1) := [ -- X X 1 1 -- 0 1 -- X X X X -- X ],
output [c1:c1U0], output [c2:c2U1], output [c3:c3U2],
input [c1:c1L0], input [c2:c2L1], input [c3:c3L2];
{ capture_RSTB }
vector(tp1) := [ Z Z Z Z 0 1 0 0 0 1 X X X X X X X X X ];
vector(tp1) := [ - Z -- 0 0 0 0 1 Z 0 Z Z Z 0 1 0 0 ];

{ pattern 1 }
{ load_unload }
vector(tp1) := [ X X X X 0 1 X X 0 0 X X X X X X X X X ];
vector(tp1) := [ X Z X X 0 1 X X 0 1 X X X X X X X X X ];
scan(tp1) := [ -- X X 1 1 -- 0 1 -- X X X X -- X ],
output [c1:c1U3], output [c2:c2U4], output [c3:c3U5],
input [c1:c1L3], input
[c2:c2L4], input [c3:c3L5];
{ capture_CLK }
vector(tp1) := [ Z Z 0 Z 0 1 1 1 0 0 X X X X X X X X X ];
vector(tp1) := [ -- 0 - 0 1 1 1 0 0 Z Z X Z Z 0 1 0 1 ];
vector(tp1) := [ Z Z 0 Z 1 1 1 1 0 0 X X X X X X X X X ];

{ pattern 2 }
{ load_unload }
vector(tp1) := [ X X X X 0 1 X X 0 0 X X X X X X X X X ];
vector(tp1) := [ X Z X X 0 1 X X 0 1 X X X X X X X X X ];
scan(tp1) := [ -- X X 1 1 -- 0 1 -- X X X X -- X ],
output [c1:c1U6], output [c2:c2U7], output [c3:c3U8],
input [c1:c1L6], input
[c2:c2L7], input [c3:c3L8];
capture_RSTB }
vector(tp1) := [ Z Z Z Z 0 1 1 1 1 X X X X X X X X X ];
vector(tp1) := [ - Z Z Z 0 0 1 1 1 Z 0 1 0 Z 0 0 0 0 ];
```

```
# example patterns using macros
pattern group_ALL ("SDI3":I, "SDO1":I, "D0":I, "D2":I, "CLK",
    "RSTB", "SDI[1]", "SDI[2]", "INC", "SCAN", "SDI3":O, "SDO1":O,
    "D0":O, "D2":O, "P[0]", "P[1]", "SDO[2]", "SDO[3]", "CO")
{ test_setup }
test_setup

{ scan_test }
{ pattern 0 }
load_unload(c1U0, c2U1, c3U2, c1L0, c2L1, c3L2)
capture_RSTB(-, Z, -, -, 0, 1, 00, 0, 1, Z, 0, Z, Z, Z0, 10, 0)

{ pattern 1 }
load_unload(c1U3, c2U4, c3U5, c1L3, c2L4, c3L5)
capture_CLK(-, -, 0, -, 0, 1, 11, 0, Z, Z, X, Z, Z0, 10, 1)

{ pattern 2 }
load_unload(c1U6, c2U7, c3U8, c1L6, c2L7, c3L8)
capture_RSTB(-, Z, Z, Z, 0, 1, 11, 1, 1, Z, 0, 1, 0, Z0, 00, 0)
```

See Also

[set_wgl](#)
[set_buses](#)

Grouping Bidirectional Port Data in WGL

In WGL patterns a bidirectional port appears as two characters, one for the force input value and another for the measure output value. These two characters can appear side by side (grouped), or in independent locations within the data (split columns). The `set_wgl -group_bidis` command causes the two characters to appear as a single column of two characters, with the first representing the input action and the second representing the output action. The default is to present the bidirectional port data as two separate columns.

The first following example uses grouped bidis and in this example there are four bidirectional ports which appear as the first four columns of each `vector()` statement. The characters "ZX" indicate a force of Z (no force) and a measure of X (mask measure).

```
# example patterns using grouped bidis
pattern group_ALL ("SDI3", "SDO1", "D0", "D2", "CLK",
    "RSTB", "SDI[1]", "SDI[2]", "INC", "SCAN", "P[0]",
    "P[1]", "SDO[2]", "SDO[3]", "CO")
{ test_setup }
vector(tp1) := [ ZX ZX ZX ZX 0 1 0 0 0 0 X X X X X ];
vector(tp1) := [ ZX ZX ZX ZX 0 0 0 0 0 0 X X X X X ];
vector(tp1) := [ ZX ZX ZX ZX 0 1 0 0 0 0 X X X X X ];
```

In the second following example split bidis are used. Notice that the pattern data no longer has any two character columns. The port order list now lists each bidirectional port twice and follows each by either :I or :O to indicate direction. The two parts of the bidirectional port data do not appear as adjacent data in the vector, they can appear at any position.

```
#example patterns using split bidis
pattern group_ALL ("SDI3":I, "SDO1":I, "D0":I, "D2":I,
    "CLK", "RSTB", "SDI[1]", "SDI[2]", "INC", "SCAN", "SDI3":O,
    "SDO1":O, "D0":O, "D2":O, "P[0]", "P[1]", "SDO[2]", "SDO[3]",
    "CO")
{ test_setup }
vector(tp1) := [ Z Z Z Z 0 1 0 0 0 0 X X X X X X X X X ];
vector(tp1) := [ Z Z Z Z 0 0 0 0 0 0 X X X X X X X X X ];
vector(tp1) := [ Z Z Z Z 0 1 0 0 0 0 X X X X X X X X X ];
```

See Also

[set_wgl](#)
[set_buses](#)

Controlling Port Data Order in WGL

The default pin data order of the WGL pattern data follows the order in which the ports are defined in the design's top module. By changing the order of the ports in the top module you can affect the order of the WGL data.

There is also the `-order_pins` option of the `write_patterns` command. Use of this option causes the ports to occur in the order: inputs, bidis, and outputs. Within each grouping the port data order matches the order the ports are defined in the design's top module.

For a top-level design with port order:

```
module TOP (I1,B1,O1,O2,O4,O3,B3,B2,I3,I2);
```

the following two examples illustrate the difference in data order.

```
# default port order using grouped bidis
pattern group_ALL ("I1", "B1", "O1", "O2", "O4", "O3", "B3",
    "B2", "I3", "I2")
{ test_setup }
vector(tp1) := [ 0 ZX X X X X ZX ZX 0 0 ];
vector(tp1) := [ 0 ZX 1 1 1 ZX ZX 0 0 ];
vector(tp1) := [ 1 0X 1 1 1 0X 0X 1 1 ];

# port order using ORDER_PINS option
pattern group_ALL ("I1", "I3", "I2", "B1", "B3", "B2", "O1",
    "O2", "O4", "O3")
{ test_setup }
vector(tp1) := [ 0 0 0 ZX ZX ZX X X X X ];
vector(tp1) := [ 0 0 0 ZX ZX ZX 1 1 1 1 ];
vector(tp1) := [ 1 1 1 0X 0X 0X 1 1 1 1 ];
```

See Also

[set_wgl](#)
[set_buses](#)

Specifying Windowed Measures in WGL

The default WGL patterns written will define timing which performs a strobed measure (single time measure) when outputs are to be measured. If your tester supports window measure (measure over a continuous range of time) and you would like to have a windowed measure, this type of measure can be created. This time you do not use any `set_wgl` options, but instead make edits to the `Timing` block of the DRC procedure file. Note that these edits must be made before performing the `run_drc` command and before generating ATPG patterns.

The following example illustrates a window measure for the symbolic group `out_ports` defined elsewhere in the DRC file. The STIL language specifies that the uppercase {H,L,T,X} characters indicate a strobed measure, and the lowercase characters {h,l,t,x} call for a window measure. In this specific example the ports associated with the symbolic group `out_ports` is continuously measured for high/low/tristate values between an offset of 450 nS and 490 nS from the beginning of the tester cycle. The '`490ns' x;` text specifies the window measure is turned off at this time and is text which is not needed for a strobed measure.

```
Timing {
    WaveformTable "WINDOW_COMPARE" {
        Period '1000ns';
        Waveforms {
            clocks { P { '0ns' D; '500ns' U; '600ns' D; } }
            input_ports { 01Z { '0ns' D/U/Z; } }
            out_ports { X { '0ns' X; } }
            out_ports { HLT { '0ns' X; '450ns' h/l/t; '490ns' X; } }
            bidi_ports { X { '0ns' X; } }
            bidi_ports { 01Z { '0ns' D/U/Z; } }
            bidi_ports { HLT { '0ns' X; '450ns' H/L/T; } }
        }
    }
}
```

See Also

[set_wgl](#)
[set_buses](#)

Delayed Input Force Timing and Force Prior in WGL

It is a common requirement when running the pattern timing to require that one or more pins have their inputs applied at some delayed offset from the beginning of the tester cycle. This is another adjustment that is made in the `Timing` block of the DRC file rather than with a `set_`

wgl command. In the following example the symbolic pin group `input_grp2` has its pattern data applied at an offset of 5ns into the tester cycle.

What is the value on the pins of group `input_grp2` from the start of the cycle to offset 5ns? The answer is that the value is undefined unless you specify some value in the timing block such as 0, 1, X, or perhaps Z. What if you just want the port to continue the value from the previous tester cycle? In WGL as well as STIL there is a "Force Prior" cone timept which indicates the value is to be whatever was previously assigned.

To cause the WGL output to call for a Force Prior, edit the `Timingblock` of the DRC file before performing a `run_drc` command and before generating any ATPG patterns and add the "P" character to the beginning of the timing definition for those inputs which are applied after a delay. Note that this use of the "P" waveform character will produce a V6 warning which you can ignore. In the following example, the symbolic pin group `input_grp2` calls for the Force Prior value.

```
WaveformTable "FORCE_PRIOR_EXAMPLE" {
    Period '1000ns';
    Waveforms {
        CLOCK { P { '0ns' D; '500ns' U; '600ns' D; } }
        CLOCK { 01ZN { '0ns' D/U/Z/X; } }
        RESETB { P { '0ns' U; '400ns' D; '800ns' U; } }
        RESETB { 01ZN { '0ns' D/U/Z/X; } }
        input_grp1 { 01ZN { '0ns' D/U/Z/X; } }
        input_grp2 { 0 { '0ns' P; '5ns' D; } }
        input_grp2 { 1 { '0ns' P; '5ns' U; } }
        input_grp2 { Z { '0ns' P; '5ns' Z; } }
        out_ports { HLT{ '0ns' X; '490ns' H/L/T/X; } }
        bidi_ports { 01ZN { '0ns' Z; '20ns' D/U/Z/X; } }
        bidi_ports { X { '0ns' X; } }
        bidi_ports { HLT { '0ns' X; '490ns' H/L/T; } }
    }
} # end FORCE_PRIOR_EXAMPLE
```

See Also

[set_wgl](#)
[set_buses](#)

Balancing Vector and Scan Statements in WGL

By default, the last event in the WGL pattern file is a scan chain unload to observe the measure values of the final capture clock. This corresponds to a `scan()` statement in the WGL file. Some vendors require that the final event in the WGL pattern file be a `vector()` statement to ensure that clocks are off and to provide a symmetric order where the scan statements are always followed by an identical number of vector statements. You can cause the final events in the WGL file to be vector statements by using the `set_wgl -nolast_scan` option to change the default behavior.

The first following example shows the default final pattern where the last event is a scan() statement. The second example shows the effect of using -nolast_scan.

```
#example made with -last_scan
{ pattern 26 }
{ load_unload }
vector(tp1) := [ X- X- X- X- 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ X- -- X- X- 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ -- -- X- X- 1 1 - - 0 1 X X - - X ],
output [c1:c1U78], output [c2:c2U79], output [c3:c3U80],
input [c1:c1L78], input [c2:c2L79], input [c3:c3L80];
{ capture
vector(tp1) := [ -z -0 -0 -1 0 1 1 1 1 Z 1 0 0 0 ];
{ load_unload }
vector(tp1) := [ X- X- X- X- 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ X- -- X- X- 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ -- -- X- X- 1 1 - - 0 1 X X - - X ],
output [c1:c1U81], output [c2:c2U82], output [c3:c3U83],
input [c1:c1L81], input [c2:c2L82], input [c3:c3L83];
end
```

```
#example made with -nolast_scan
{ pattern 26 }
{ load_unload }
vector(tp1) := [ X- X- X- X- 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ X- -- X- X- 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ -- -- X- X- 1 1 - - 0 1 X X - - X ],
output [c1:c1U78], output [c2:c2U79], output [c3:c3U80],
input [c1:c1L78], input [c2:c2L79], input [c3:c3L80];
{ capture_CLK }
vector(tp1) := [ -z -0 -0 -1 0 1 1 1 1 Z 1 0 0 0 ];
{ load_unload }
vector(tp1) := [ X- X- X- X- 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ X- -- X- X- 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ -- -- X- X- 1 1 - - 0 1 X X - - X ],
output [c1:c1U81], output [c2:c2U82], output [c3:c3U83],
input [c1:c1L81], input [c2:c2L82], input [c3:c3L83];
{ nocapture }
vector(tp1) := [ -X -X -X -X 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ -X -X -X -X 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ -X -X -X -X 0 1 X X 0 0 X X X X X ];
end
```

See Also

[set_wgl](#)
[set_buses](#)

Mapping Bidirectional Ports Within Vector Statements in WGL

You've seen an example earlier of how TetraMAX ATPG supports creating WGL patterns with bidirectional port data represented as either a single column of two characters (grouped) or as two columns of single characters (non-grouped or split). In addition to this choice in grouping there is also the ability to change or map the characters used. Not every vendor agrees on what the WGL character representation should be for bidirectional port data so TetraMAX ATPG has been designed to provide flexibility by use of the `set_wgl -bidi_map` option.

The syntax for this option is: `set_wgl -bidi_map <from> <to>`

There are 9 mappings that can be adjusted: 3 for which the bidirectional port is an input, 4 for which the bidirectional port is an output, and 2 for when the bidirectional port is a scan input or scan output. This argument may be repeated on the same command line or across multiple commands to specify more than one mapping. If the same from designator is repeated then the later one will replace the earlier ones.

The from designator is a two-character string that represents the TetraMAX internal data. The to designator is a two-character string that specifies the characters which will appear in the WGL pattern output in place of this internal representation.

```
Definition of TetraMAX Internal Representation = "from"
```

```
from
=====
0x : force 0, no measure
1x : force 1, no measure
xx : force unknown, no measure

z0 : no force, measure 0
z1 : no force, measure 1
zx : no force, no measure
zz : no force, measure Z

-x : bidi is in scan input mode
-z : bidi is in scan output mode
```

The preceding table defines all the legal combinations available for the `from` portion of the mapping option. Any other combination is illegal. The `to` designator is also made up of characters 0/1/x/z/- but the mapping is checked to ensure that you are not destroying the intent of the data or masking measures that would affect the test coverage reported. As an example of a mapping the following table represents a commonly requested map in which one of the bidirectional characters is always a dash:

```
A common mapping
```

```
from : to
===== : ==
0x : 0- # force 0, no measure
1x : 1- # force 1, no measure
xx : x- # force unknown, no measure
```

```

z0 : -0 # force Z, measure 0
z1 : -1 # force Z, measure 1
zx : -x # force Z, no measure
zz : -z # force Z, measure Z
-x : -- # bidi is a scan input
z- : -- # bidi is a scan output

```

With the exception of the {zz,-z} mapping above, this table represents the default mapping.

The `set_wgl` command which would implement the previous table is:

```
BUILD> set_wgl -bidi_map { 0x 0- 1x 1-  xx x- z0 -0 \
                           z1 -1 zx -x zz -z x -- z- -- }
```

Note in the previous example that you can specify the `-bidi_map` option only one time, and the parameters must be in a list structure. Alternatively, you can repeat the entire command line for each entry, as shown in the following example:

```

set_wgl -bidi_map {0x 0-}
set_wgl -bidi_map {1x 1-}
set_wgl -bidi_map {xx x- }
set_wgl -bidi_map {z0 -0 }
set_wgl -bidi_map {z1 -1}
set_wgl -bidi_map {zx -x}
set_wgl -bidi_map {zz -z}
set_wgl -bidi_map {x --}
set_wgl -bidi_map {z- --}

```

Note: Not all mappings are allowed. For example, you cannot map the dash for scan input or scan output to any other character. Also, you can map "zz" to "-z", but you cannot map "zz" to "z-". because of the loss of measure and to unambiguously read back in the WGL which is written out. The "zz"->-z" mapping still indicates a measure must be performed but a "zz"->z-" mapping could be confused with a "zx"->z-" mapping which generally is interpreted to mean there is no force and no measure.

Note: The ability to use some bidi mappings is affected by whether the tester can measure Z values or not. If the tester can measure Z values then the default setting of `set_buses -external_z Z` should be used and the WGL patterns can contain both ZZ and ZX data (no force, measure Z and no force, no measure). If the tester cannot measure Z values or you want to generate patterns for which no Z-measure is needed you would set the `set_buses -external_z X` option before generating patterns. This would result in WGL patterns with "ZX" data for bidirectional pins but no "ZZ". If "ZZ" does not appear in the WGL you can define a bidi map of "ZX"->Z-" or "ZX"->-Z" which you could not do if the Z measure were enabled and "ZZ" were possibly present.

Note: Most vendors do not support a simultaneous force and measure on the same port in the same cycle. With that in mind you should not use the `-measure_forced_bidis` option of the `write_patterns` command as this allows for a simultaneous force and measure whenever possible.

To report the current bidirectional map settings use the `report_settings wgl` command. The output is similar to the following example and the mapping will appear as a series of (from,to) settings.

```
wgl = macro_usage=off, nopad=on, scan_map=dash
  group_bidis=off, inversion_reference=master, tester_ready=on
  bidi_map=(Z0,-0) (Z1,-1) (0X,0-) (1X,1-) (XX,X-) (ZX,-X) (ZZ,-Z) (Z-,--)
```

As an example of how the `vector()` statement data changes for bidirectional ports the first following example shows some pattern data with four bidirectional pins (grouped as single column of two characters each) where the mapping is identical to the TetraMAX internal representation. The second example uses a common mapping in which the bidirectional character pair always has one character represented as a dash.

An example where the mapping matches TetraMAX internal representation.

```
{ pattern 1 }
{ load_unload }
vector(tp1) := [ 0X 1X XX ZX 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ 0X 1X XX ZX 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ 0X 1X XX ZX 1 1 - X 0 1 X X X - X ],
output [c1:c1U0], input [c1:c1L1];
{ capture_CLK }
vector(tp1) := [ ZX ZX ZX ZX 0 1 0 1 0 1 X X X X X ];
vector(tp1) := [ Z0 Z1 ZX ZZ 0 1 0 1 0 1 Z 0 0 1 0 ];
vector(tp1) := [ ZX ZX ZX ZX 1 1 0 1 0 1 X X X X X ];

{ pattern 2 }
{ load_unload }
vector(tp1) := [ ZX ZX ZX 0X 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ ZX ZX ZX 0X 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ ZX ZX ZX 0X 1 1 - X 0 1 X X X - X ],
output [c1:c1U1], input [c1:c1L2];
{ capture_CLK }
vector(tp1) := [ 0X 0X 0X 0X 0 1 1 1 1 0 X X X X X ];
vector(tp1) := [ 0X 1X Z0 ZZ 0 1 1 1 1 0 Z 0 1 0 0 ];
vector(tp1) := [ 0X 1X ZX ZX 1 1 1 1 1 0 X X X X X ];
```

The same patterns after defining a mapping of:
`(0x,0-)` `(1x,1-)`, `(xx,x-)`, `(z0,-0)`, `(z1,-1)`, `(zx,-x)`, `(zz,-z)`

```
{ pattern 1 }
{ load_unload }
vector(tp1) := [ 0- 1- X- Z- 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ 0- 1- X- Z- 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ 0- 1- X- Z- 1 1 - X 0 1 X X X - X ],
output [c1:c1U0], input [c1:c1L1];
{ capture_CLK }
vector(tp1) := [ -X -X -X -X 0 1 0 1 0 1 X X X X X ];
vector(tp1) := [ -0 -1 -X -Z 0 1 0 1 0 1 Z 0 0 1 0 ];
vector(tp1) := [ -X -X -X -X 1 1 0 1 0 1 X X X X X ];
```

```

{ pattern 2 }
{ load_unload }
vector(tp1) := [ -X -X -X 0- 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ -X -X -X 0- 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ -X -X -X 0- 1 1 - X 0 1 X X X - X ],
output [c1:c1U1], input [c1:c1L2]];
{ capture_CLK }
vector(tp1) := [ 0- 0- 0- 0- 0 1 1 1 1 0 X X X X X ];
vector(tp1) := [ 0- 1- -0 -Z 0 1 1 1 1 0 Z 0 1 0 0 ];
vector(tp1) := [ 0- 1- -X -X 1 1 1 1 1 0 X X X X X ];

```

See Also

[set_wgl](#)
[set_buses](#)

Mapping Bidirectional Ports Within Scan Statements in WGL

The vector() statements in WGL correspond to the application of tester cycles. The scan() statements correspond to the serial loading and unloading of scan chains. The various vendor rules for character mapping of the vector() statements cannot be the same as for the scan() statement and so TetraMAX ATPG supports the `set_wgl -scan_map` option to allow somewhat independent control of characters in the scan() statement. The available choices for scan mapping are: dash, bidi, keep, and none. The default is dash.

The following examples show some of the variations of `-scan_map`. The patterns are for a design with three scan chains and the first bidirectional port is a scan input and the second bidirectional port is a scan output.

For a setting of `dash`, every scan input and output position in the scan() statement contains a dash, and all bidirectional ports acting as a scan input or output contain a double dash.

```

# set_wgl -scan_map dash

vector(tp1) := [ 0- -X X- X- 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ -- -- X- X- 1 1 - - 0 1 X X - - X ],

```

For a setting of `bidi`, every scan input and output position in the scan() statement contains a dash, and any bidirectional port acting as a scan input or output follows the mapping defined by the `-bidi_map` options. For the following example, assume a BIDI mapping of `(-x,--)` for scan inputs, and `(z-,z-)` for scan outputs.

```

# set_wgl -scan_map bidi -bidi_map {-x --} -bidi_map {z- z- }

vector(tp1) := [ 0- -X X- X- 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ -- z- X- X- 1 1 - - 0 1 X X - - X ],

```

For a setting of `keep`, every scan input and output position in the scan() statement keeps the same characters as from the previous vector() statement in the `load_unload` procedure,

including any scan inputs or outputs on bidirectional ports. **Note:** It is important that the load/unload procedure have at least one vector() statement before the Shift procedure in order for a selection of keep to work properly.

```
# set_wgl -scan_map keep

vector(tp1) := [ 0- -X X- X- 0 1 X X 0 1 X X X X X ];
scan(tp1)  := [ 0- -X X- X- 1 1 X X 0 1 X X X X X ],
```

For a setting of `none`, every scan input and output position in the `scan()` statement contains a dash, and any bidirectional port acting as a scan input or output uses the TetraMAX internal representation of "-X" for input and "Z-" for output.

```
# set_wgl -scan_map none

vector(tp1) := [ 0- -X X- X- 0 1 X X 0 1 X X X X X ];
scan(tp1)  := [ -X Z- X- X- 1 1 - - 0 1 X X - - X ],
```

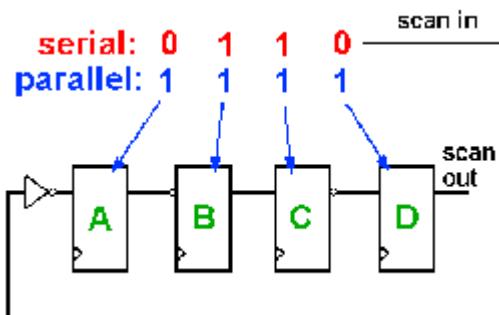
See Also

[set_wgl](#)
[set_buses](#)

Adjusting Pattern Data for Serial Versus Parallel Interpretation in WGL

The scan load data in the WGL patterns can be represented in two different ways, depending upon the reference point required by your WGL pattern translation tool. The `set_wgl -tester_ready` setting selects a data format that is ready to serially shift into the device without further processing for scan cell inversions. The `-set_wgl -notester_ready` option selects a data format that is ready to parallel load directly into the scan cells without further processing for inversions.

In the following figure, if you desire to have all devices A,B,C, and D loaded with 1's after a scan load, and your WGL translation application expects the data in parallel (`-notester_ready`) format, then the WGL scan data must be written as all 1's. However, if your WGL translation application expects the data in serial format (`-tester_ready`), then the WGL scan data must be adjusted for internal inversions that it passes through before being shifted into place. As you can see, the data is not the same: "1111" vs. "0110". So it is very important to know which data format your WGL translation application is expecting. The parallel format is the more popular, so if you do not know you should try the `-notester_ready` option first.



Note that both the serial and parallel load formats may be sensitive to the referencing scheme for determining inversion if the final WGL translator is doing a parallel-form to serial form translation or a serial-form to parallel-form translation.

One additional variant of WGL output is needed if the WGL is to be interpreted for a parallel simulation and the end-of-cycle protocol is used. This end of cycle protocol results in a scan output pre-measure before beginning the "scan()" statement for the balance of the scan load/unload. The expected scan output vector needs to be shifted by the single bit of the pre-measure. To accomplish this, use the `-pre_measured` option instead of the `-notester_ready` option.

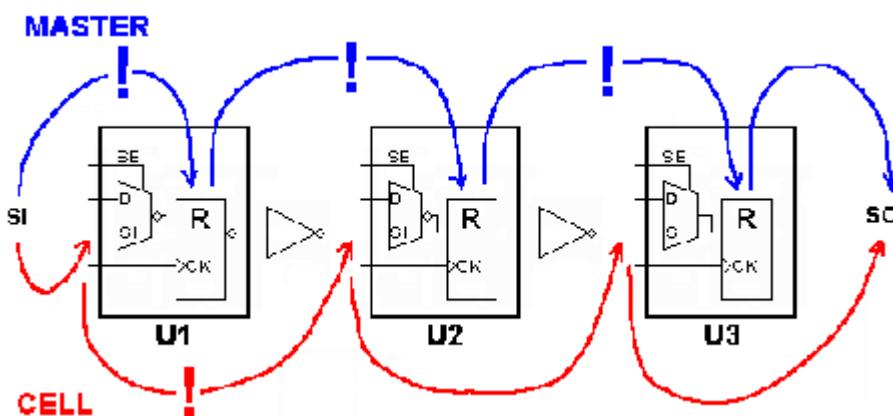
See Also

`set_wgl`
`set_buses`

Selecting Scan Chain Inversion Reference in WGL

The `scanchain` block of the WGL pattern file defines each scan chain in physical order from input port to output port. When an inversion exists between positions in the scan order, and exclamation mark "!" is inserted to indicate an inversion of the data has occurred between the two positions. This inversion information is crucial for the correct translation of the scan chain load and unload data by the WGL-to-simulator or WGL-to-tester tools supported by your vendor.

More than one interpretation of the reference scheme for calculating inversions exists and so TetraMAX ATPG offers options of master, cell, and omit for the `set_wgl -inversion_reference` command.



The previous diagram can provide some insight into the two different referencing schemes for inversion markers. When TetraMAX ATPG calculates the inversion markers for a setting of `master` the reference point begins at the scan input pin, and then looks at whether the data is inverted from that point to the actual sequential simulation primitive functioning as the "master cell" where the value is stored. This is often a Verilog UDP level underneath the vendor's library cell. For a library cell with only one sequential element there is only one answer but for a library cell with two or more sequential elements, the answer might be ambiguous. As shown in the diagram, for an inversion reference of `master` there are inversions between the scan input and U1, between U1 and U2, and between U2 and U3. The corresponding WGL scanchain definition is shown in the following example.

```
# set_wgl -inversion_reference master

scanchain
  c1 ["si", !, "U1/R", !, "U2/R", !, "U3/R", "so" ];
end
```

When TetraMAX ATPG calculates the inversion markers for a setting of `cell` it begins at the scan in pin and then determines whether an inversion of the data occurs relative to the scan input pin of each library cell. This reference is used by some WGL translators in forming the FORCE/RELEASE statements needed for a parallel Verilog simulation. The location of the inversion markers is unambiguous and not affected by which cell is classified as the "master" cell by TetraMAX ATPG during DRC. Using an inversion reference of `cell` and the preceding diagram, there is an inversion only between the scan input of cell U1 and the scan input of U2. The corresponding WGL scanchain definition is shown in the following example:

```
# set_wgl -inversion_reference cell

scanchain
  c1 ["si", "U1/R", !, "U2/R", "U3/R", "so" ];
end
```

Sometimes, no matter which inversion reference you select the external WGL translator seems to come up with patterns that mismatch in simulation. If the simulation environment serially processes scan load information, then there is one more inversion reference that might be of use and that is the `omit` option. This option leaves out all inversion markers. By combining both the `-inversion_reference omit` and `-tester_ready` options, TetraMAX ATPG produces scan load/unload data that is preprocessed for inversions and is ready to shift into the device unchanged, and omits the inversion markers so the external WGL translator is mydesigned into thinking that no data adjustments for inversion are needed. The corresponding WGL scanchain data when `omit` is used is shown in the following example:

```
# set_wgl -inversion_reference omit

scanchain
  c1 ["si", "U1/R", "U2/R", "U3/R", "so" ];
end
```

See Also

[set_wgl](#)
[set_buses](#)

Effect of CELLDEFINE in WGL

The previous examples showed the effect of different choices of inversion reference on the placement of the inversion markers "!" in the scanchain definition block. Another item which affects the scanchain block is the presence or absence of the `celldefine compiler directive in the definition of the library model. Consider the following two examples:

```
# Verilog library module without celldefine
module SDFF (Q, CLK, SE, D, SI);
    input CLK, SE, D, SI;
    output Q;
    uMUX M (di, SE, D, SI);
    uDFFQ R (Q , CLK, di);
endmodule

# WGL scanchain shows instance "R"
scanchain
    c1 ["si", "U1/R", !, "U2/R", "U3/R", "so" ];
end
```

```
# Verilog library module with celldefine
`celldefine
module SDFF (Q, CLK, SE, D, SI);
    input CLK, SE, D, SI;
    output Q;
    uMUX M (di, SE, D, SI);
    uDFFQ R (Q, CLK, di);
endmodule
`endcelldefine

# scanchain instances have no "R"
scanchain
    c1 ["si", "U1", !, "U2", "U3", "so" ];
end
```

In the first example the Verilog module definition was not defined inside a `celldefine/endcelldefine pair. The resulting WGL scanchain definition shows instance pathnames that include the R of the uDFFQ device.

In the second example the Verilog module definition was within a `celldefine/endcelldefine pair. The resulting WGL scanchain definition does not include the instance references beneath the SDFF module.

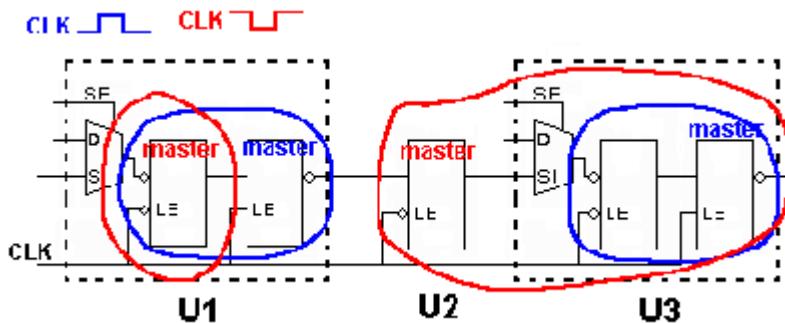
Note: Reading a netlist with the `-library` option has the same effect as enclosing the module with `'celldefine'/endcelldefine` pair and is yet another way to affect the WGL output.

See Also

`set_wgl`
`set_buses`

Ambiguity of the Master Cell in WGL

The diagram below provides one simple example of the potential for ambiguity when using an inversion reference of master. In this example some DFF functions are created with a library cell using two latches. TetraMAX ATPG defines the "master" based on which sequential device in a scan chain shifts first due to the leading edge of the defined shift clocks. So with the CLK port defined as active high, the "master" becomes the second LATCH in U1 and U3, with U2 acting as a lockup latch. If the polarity of CLK is reversed, then the first latch in U1 is classified as the master and the lockup latch is classified as the master for cell U3! Both polarities of CLK generates ATPG patterns but most likely only one resulting WGL inversion set is correct.



See Also

`set_wgl`
`set_buses`

Running Multicore ATPG

Multicore ATPG is used to parallelize and improve ATPG runtime by leveraging the resources provided by multicore machines. Multicore ATPG launches multiple slaves to parallelize ATPG work on a single host. You can specify the number of processes to launch, based on the number of CPUs and the available memory on the machine.

The following sections describe multicore ATPG:

- [Comparing Multicore ATPG and Distributed ATPG](#)
- [Invoking Multicore ATPG](#)
- [Multicore Interrupt Handling](#)

- [Understanding the Processes Summary Report](#)
- [Multicore Limitations](#)

See Also

[Running Distributed ATPG](#)

Comparing Multicore ATPG and Distributed ATPG

Multicore ATPG is different than distributed ATPG, which is described in [“Running Distributed ATPG.”](#) Distributed ATPG launches multiple slave processes on a computer farm or on several standalone hosts. The slave processes read an image file and execute a fixed command script. ATPG distributed technology does not differentiate between multiple CPUs and separate workstations.

When compared to ATPG distributed technology, multicore ATPG has several advantages:

- It is easier to use. You simply need to specify the number of slaves to use. There is no need to set up the environment for slaves, and no need to debug network or computer farm issues. Also, it is not necessary for the master to write a binary image file or for the slaves to read it.
- It is more efficient in using compute resources. Multicore ATPG shares netlist information among slaves and the master. This means the overall memory usage is much lower than the total memory usage of distributed ATPG.
- It reduces communication overhead by running all involved processes on one machine. It also improves the efficiency of parallelism by sharing more information among processes. This often results in better QoR compared to distributed ATPG.

Although multicore ATPG offers better memory utilization (<50 percent increase per core) compared to distributed ATPG (~100 percent increase per slave), the entire memory must reside on a single machine.

Multicore ATPG and distributed ATPG provide similar efficiency in reducing ATPG runtime. The runtime improvement from multicore processing is limited by the number of cores and CPUs on a single host. With distributed ATPG, however, runtime continues to improve as more hosts are added across the network.

Invoking Multicore ATPG

Multicore ATPG is activated using the following `set_atpg` command:

```
set_atpg -num_processes < number | max >
```

The `number` specification refers to the number of slave processes that are used in ATPG. If `max` is specified, then TetraMAX ATPG computes the maximum number of processes available in the host, based on number of CPUs. If TetraMAX ATPG detects that the host has only one CPU, then single-process ATPG is performed instead of multicore ATPG with only one slave.

To turn off multicore ATPG, specify `set_atpg -num_processes 0`.

Do not specify more processes than the number of CPUs available on the host. You should also consider whether there are other CPU-intensive processes running simultaneously on the host when running the number of processes. If too many processes are specified, performance will degrade and might be worse than single-process ATPG. On some platforms, TetraMAX ATPG cannot compute the number of CPUs available and will issue an error if max is specified.

Multicore Interrupt Handling

To interrupt the multicore ATPG process, use “Control-c” in the same manner as halting a single process. If a slave crashes or is killed, the master and remaining slaves will continue to run. This behavior is consistent with the default behavior of distributed ATPG.

If the master crashes or is killed, the slaves will also halt. In this case, there are no ongoing processes, dangling files, or memory leakage.

You can also interrupt the multicore ATPG process from the TetraMAX GUI by clicking the Stop button. At this point, the master process sends an abort signal to the slave processes and waits for the slaves to finish any ongoing interval tasks. If this takes an extended period of time, you can click the Stop button twice. This action causes the master process to send a kill signal to the slaves, and the prompt immediately returns. Note that clicking the Stop button twice terminates all slave processes without saving any data gathered since the last communication with the master. For more information on the Stop button, see "[Command Entry](#)."

Understanding the Processes Summary Report

Memory consumption needs to be measured to tune the global data structure to improve the scalability of multicore architecture. Legacy memory reports are not sufficient because they do not deal with issues related to “copy-on-modification.” To facilitate collecting performance data, a summary report of multicore ATPG is printed at the end of ATPG when the -level expert option is specified with the `set_messages` command. The summary report appears as shown in [Example 1](#).

Example 1: Processes Summary Report

Processes Summary Report

Process Patterns	Time (s)	Memory (MB)						
ID	pid	Internal	CPU	Wall	Shared	Private	Total	Pattern
<hr/>								
-	0	7611	1231	0.53	35.00	67.78	30.54	98.32 5.27
-	1	7612	626	35.68	35.00	64.87	22.31	87.18 0.00
-	2	7613	605	35.50	35.00	64.71	22.47	87.18 0.00
-	Total	1231	71.71	35.00	67.78	75.32	143.10	5.27
<hr/>								
--								

The report in [Example 1](#) contains one row for each process. The first process with an ID of “0” is the master process. The child processes have IDs of 1, 2, 3, and so forth. The last row is the sum for each measurement across all processes.

The “pid” is the process ID of that process. The “Patterns” are the total number of patterns stored by the master or the number of patterns generated by the slave in this particular ATPG session. The “Time(s)” includes CPU time and wall time.

The “Memory” measurements are obtained by parsing the system-generated file /proc/pid/smaps. The file contains memory mapping information created by the OS while the process still exists. The /proc/pid/ directory cannot be found after the process terminates. The tool parses this file at the proper time to gather memory information for the reporting at the end of parallel ATPG.

The “Memory” measurement includes “Shared”, “Private”, “Total” and “Pattern”. “Shared” means all processes share the same copy of the memory. “Private” means the process stores local changes in the memory. The “Total” is the sum of “Shared” and “Private”. The “Pattern” refers to memories allocated for storing patterns. The total memory consumption of the entire system is the “Total” item in the row “Total,” which is the sum of total shared memory (the maximum of shared memories for each process) and the total private memory (the sum of all private memory for all processes). Although the memory for patterns is listed separately, it is part of the master private memory.

The memory section of the summary report is only available on Linux and AMD64 platforms. No other platform gives “shared” or “private” memory information in a copy-on-write context. On other formats, the memory reports gives all “0s” for items other than the pattern memory.

Multicore Limitations

The following ATPG features are not supported by multicore ATPG:

- Streaming Pattern Validation
- Distributed ATPG
- The `-per_cycle` option of the `report_power` command is not recognized.

Running Logic Simulation

Using TetraMAX ATPG, you can run logic simulation and use the graphical schematic viewer (GSV) to view the logic simulation results.

For combinational and sequential patterns, you can perform the following tasks:

- Perform logic simulation using either the internal or external pattern set.
- Check simulated against expected values from the patterns.
- Perform simulation in the presence of a single failure point to determine the patterns that would show differences.
- View the effect of any single point of failure for any single pattern.

For combinational patterns, you can also view the logic simulation value from any single pattern in the most recent 32 patterns in the simulation buffer.

For sequential patterns, you can also save the logic simulation value from any range of patterns and view this data.

The following sections describe how to run logic simulation:

- [Comparing Simulated and Expected Values](#)
- [Patterns in the Simulation Buffer](#)
- [Sequential Simulation Data](#)
- [Single-Point Failure Simulation](#)
- [GSV Display of a Single-Point Failure](#)

Note: In addition to standard logic simulation, you can also improve simulation runtime by launching multiple slaves to parallelize fault and logic simulation to work on a single host. This process is described in [“Running Multicore Simulation”](#).

Comparing Simulated and Expected Values

You can compare the simulation results against the expected values contained in the patterns during logic simulation. To do this, use the Compare option of the Run Simulation dialog box, or the `-nocompare` option of the `run_simulation` command. For more information, see [“Performing Good Machine Simulation”](#).

[Example 1](#) shows a transcript of a simulation run that had no comparison errors; 139 patterns were simulated with zero failures.

Example 1 Simulation With No Comparison Errors

```
TEST-T> run_simulation
Begin good simulation of 139 internal patterns.
Simulation completed:
#patterns=139, #fail_pats=0(0),
#failing_meas=0(0), CPU time=4.61
```

[Example 2](#) shows a transcript of a simulation run with comparison errors. In this report, the first column is the pattern number, and the second column is the output port or scan chain output. The third column is present if the port is a scan chain output and contains the number of scan chain shifts that occurred to the point where the error was detected. The last column, shown in parentheses, is the simulated/expected data.

Example 2 Simulation With Comparison Errors

```
TEST-T> run_simulation
Begin simulation of 139 internal patterns.
  1  /o_sdo2  23  (exp=0, got=1)
  4  /o_sdo2  23  (exp=1, got=0)
  6  /o_sdo2  23  (exp=1, got=0)
  7  /o_sdo2  23  (exp=1, got=0)
  8  /o_sdo2  23  (exp=0, got=1)
  :
  123 /o_sdo2  23  (exp=0, got=1)
  124 /o_sdo2  23  (exp=1, got=0)
  129 /o_sdo2  23  (exp=1, got=0)
```

```
132 /o_sdo2 23 (exp=1, got=0)
Simulation completed: #patterns=139, #fail_pats=41(0),
#failing_meas=41(0), CPU time=4.97
```

Patterns in the Simulation Buffer

During ATPG, TetraMAX ATPG processes potential patterns in groups of 32 using an internal buffer called the Simulation Buffer. Immediately after completion of ATPG, you can select any of the last 32 patterns processed and display the resulting logic values on the pins of objects in the GSV window. You can use the Setup dialog box to select pattern data and provide an integer between 0 and 31 for the pattern number.

Alternatively, you can execute the following commands:

```
TEST-T> set_pindata pattern NN
TEST-T> refresh schematic
```

For an example, see [“Displaying Pattern Data”](#).

Sequential Simulation Data

Sequential simulation data is typically from functional patterns. This type of data is stored in the external pattern buffer. When the simulation type in the Run Simulation dialog box is set to Full Sequential, you can select a range of patterns to be stored. After the simulation is completed, you can display selected data from this range of patterns using the pin data type “sequential sim data.”

For example, with gates drawn in the schematic window, execution of the following commands generates the display shown in [Figure 1](#).

```
TEST-T> set_simulation -data 85 89
TEST-T> run_simulation -sequential
```

The pin data in the display shows the sequential simulation data values from the five patterns; each pattern has a dash (–) as a separator. Some patterns result in a single simulation event value and other patterns result in three values.

Figure 1 Sequential Simulation Data for Five Patterns

Single-Point Failure Simulation

You can simulate any single point of failure for any single pattern by checking the Insert Fault box in the Run Simulation dialog box and running the error site and stuck-at value, or by using a command such as the following:

```
TEST-T> run_simulation -max_fails 0 amd2910/ incr/U42/A 1
```

[Example 3](#) shows the result of executing this command. TetraMAX ATPG reports the signature of the failing data to the transcript as a sequence of pattern numbers and output ports with differences between the expected data and the simulated failure.

Example 3 Signature of a Simulated Failure

```
TEST-T> run_simulation -max_fails 0 /amd2910/ incr/U42/A 1
Begin simulation of 139 internal patterns with pin /amd2910/ incr/
U42/A stuck at 1.
85  /o_sdo2 23  (0/1)
94  /o_sdo2 23  (0/1)
Simulation completed: #patterns=139, #fail_pats=2(0),
#failing_meas=2(0), CPU time=2.02
```

GSV Display of a Single-Point Failure

You can display simulation results for a single-point failure in the GSV. To do so, click the SETUP button on the GSV toolbar to display the Setup dialog box.

To view the difference between the good machine and faulty machine simulation for a specific pattern,

1. In the Setup dialog box, under Pin Data, choose Fault Sim Results.
2. Click the Set Parameters button. The Fault Sim Results Parameters dialog box appears.
3. Enter the pin path name or gate ID of the fault site, the stuck-at-0 or stuck-at-1 fault type, and the pattern number that is to be simulated in the presence of the fault.
4. Click OK to close the Fault Sim Results Parameters dialog box.
5. Click OK again to close the Setup dialog box.

The GSV displays the fault simulation results, as shown in [Figure 2](#).

Figure 2 Fault Simulation Results Displayed Graphically

In [Figure 2](#), pin A of gate 1216 is the site of the simulated stuck-at-1 fault. The output pin X shows 0/1, where the 0 is the good machine response and the 1 is the faulty machine response. You can trace the effect of the faulty machine throughout the design by locating logic values separated by a forward slash (/), representing the good/bad machine response at that pin.

Data Volume and Test Application Time Reduction Calculations

The equations for calculating data volume and test application time reduction for running TetraMAX ATPG on compressed scan designs are as follows:

```
Test Data Volume Reduction =
(Scan Test Data Volume) /
(Scan Compression Test Data Volume)

Test Application Time Reduction =
(scan mode test application time) /
(ScanCompression_mode test application time)
```

These calculations are explained in the following sections:

- [Test Data Volume Calculations](#)
 - [Test Application Time Reduction Calculations](#)
-

Test Data Volume Calculations

The following information is stored on the tester in each test cycle:

- Forced value on input (signal or clock waveform)
- Value expected on output (strobed output)
- Whether output value should be strobed or not (output mask)

On every output, there are two bits of information per cycle. In the following two equations, this accounts for the factor of three in the scan-test-data-volume equation and the factor of two in the scan-compression-test-data-volume equation. The compression calculation is written differently because it accounts for the number of inputs and outputs to the compression logic.

You can use the following formulas to expand the test data volume reduction equation:

```
Scan Test Data Volume =
3*(length of the longest Scan mode scan chain) *
(number of scan chains in Scan mode) *
(number of Scan mode patterns)
Scan Compression Test Data Volume =
(length of longest ScanCompression_mode scan chain) *
(number of scan_in + 2*(number of scan_out)) *
(number of patterns)
```

The test data volume might not match the memory used by the tester because each ATE uses the test data volume differently. However, the tester can optimize the memory content. It can allocate memory differently, depending on the brand or version of the tester and the channels and cycles used. In such cases, the factors 2 and 3 in the scan-test-data-volume and scan-compression-test-data volume formulas, respectively, might not match the data in the tester memory.

The following ratio indicates the test data volume reduction that can be achieved:

```
Test Data Volume Reduction =
(Scan Test Data Volume) /
(Scan Compression Test Data Volume)
```

The test data volume reduction value calculated with this formula is just an estimate of the improvements you can get by using compression.

Test Application Time Calculations

The test application time reduction is an estimate for the improvements you can achieve by using compression. You can determine this reduction by taking the ratio of scan versus scan-compression test-application time.

The test-application-time-reduction equation can be expanded by using the following formulas:

```
Scan Test Application Time =  
(longest chain in Scan mode) *  
(number of patterns in Scan mode)  
Scan Compression Test Application Time =  
(longest scan chain in ScanCompression_mode) *  
(number of patterns in ScanCompression_mode)
```

The test application time reduction that can be achieved as follows:

```
Test Application Time Reduction =  
(scan mode test application time) /  
(scanCompression_mode test application time)
```

If you expand this equation, using the previous test application time equations for scan and scan compression, you get the following:

```
Test Application Time Reduction =  
((longest chain in Scan mode) *  
(number of patterns in Scan mode)) /  
((longest scan chain in ScanCompression_mode) *  
(number of patterns in ScanCompression_mode))
```

See Also

[Distributed ATPG Limitations](#)

13

Fault Lists and Faults

TetraMAX ATPG puts faults into various fault classes, each of which are organized into categories.

The following topics describe the fault classes and explain how TetraMAX ATPG calculates test coverage statistics:

- [Working with Fault Lists](#)
- [Fault Categories and Classes](#)
- [Fault Summary Reports](#)
- [Using Clock Domain-Based Faults](#)

Working with Fault Lists

TetraMAX ATPG maintains a list of potential faults for a design, along with the categorization of each fault. A fault list is contained in an ASCII file that can be read and written using the `read_faults` and `write_faults` commands.

The following topics describe how to work with fault lists:

- [Using Faults Lists](#)
- [Collapsed and Uncollapsed Fault Lists](#)
- [Random Fault Sampling](#)
- [Fault Dictionary](#)

As shown in [Example 1](#), a fault list contains one fault entry per line. Each entry consists of three items separated by one or more spaces. The first item indicates the stuck-at value (`sa0` or `sa1`), the second item is the two-character fault class code, and the third item is the pin path name to the fault site. Any additional text on the line is treated as a comment.

If the fault list contains equivalent faults, then the equivalent faults must immediately follow the primary fault on subsequent lines. Instead of a class code, an equivalent fault is indicated by a fault class code of “`--`”.

Example 1: Typical Fault List Showing Equivalent Faults

```
// entire lines can be commented
sa0 DI /CLK ; comments here
sa1 DI /CLK
sa1 DI /RSTB
sa0 DS /RSTB
sa1 AN /i22/mux2/A
sa1 UT /i22/reg2/lat1/SB
sa0 UR /i22/mux0/MUX2_UDP_1/A
sa0 -- /i22/mux0/A # equivalent to UR fault above it
sa0 DS /i22/reg1/MX1/D
sa0 -- /i22/mux1/X
sa0 -- /i22/mux1/MUX2_UDP_1/Q
sa1 DI /i22/reg2/r/CK
sa0 DI /i22/reg2/r/CK
sa1 DI /i22/reg2/r/RB
sa0 AP /i22/out0/EN
sa1 AP /i22/out0/EN
```

Note the following:

- TetraMAX ATPG ignores blank lines and lines that start with a double slash and a space (`//`).
- You can control whether the fault list contains equivalent faults or primary faults by using the `-report` option of the `set_faults` command or the `-collapsed` or `-uncollapsed` option of the `write_faults` command.

See Also

[Persistent Fault Model Operations](#)

Using Fault List Files

You can use fault list files to manipulate your fault list in the following ways:

- Add faults from a file, while ignoring any fault classes specified
- Add faults from a file, while retaining any fault classes specified
- Delete faults specified by a fault list file
- Add nofaults (sites where no faults are to be placed) specified by a fault list file

To access fault list files, you use the `read_faults` and `read_nofaults` commands, which have the following syntax:

```
read_faults file_name [-retain_code] [-add | -delete]
```

```
read_nofaults file_name
```

The `-retain_code` option retains the fault class code but behaves differently depending on whether the faults in the file are new or replacements for existing faults:

- **New Faults**

For any new fault locations encountered in the input file, if the fault code is DS or DI, the new fault is added to the fault list as DS or DI, respectively. For all other fault codes, TetraMAX ATPG determines whether the fault location can be classified as UU, UT, UB, DI, or AN. If the fault location is determined to be one of these fault classes, the new fault is added to the fault list and the fault code is changed to the determined fault class. If the fault location was not found to be one of these special classes, the new fault is added with the fault code as specified in the input file.

- **Existing Faults**

For any fault locations provided in the input file that are already in the internal fault list, the fault code from the input file replaces the fault code in the internal fault list. TetraMAX ATPG does not perform any additional analysis.

Collapsed and Uncollapsed Fault Lists

To improve performance, most ATPG tools collapse all equivalent faults and process only the collapsed set. For example, the stuck-at faults on the input pin of a BUF device are considered equivalent to the stuck-at faults on the output pin of the same device. The collapsed fault list contains only the faults at one of these pins, called the primary fault site. The other pin is then considered the equivalent fault site. For a given list of equivalent fault sites, the one chosen to be the primary fault site is purely random and not predictable.

You can generate a fault summary report using either the collapsed or uncollapsed list using the `-report` option of the `set_faults` command.

Example 1: Collapsed and Uncollapsed Fault Summary Reports

```
TEST-T> set_faults -report collapsed
TEST-T> report_faults -summary

Collapsed Fault Summary Report
-----
fault class          code #faults
-----
Detected             DT   120665
Possibly detected    PT   3749
Undetectable         UD   1374
ATPG untestable     AU   6957
Not detected         ND   6452
-----
total faults          139197
test coverage        88.91%
-----

TEST-T> set_faults -report uncollapsed
TEST-T> report_faults -summary

Uncollapsed Fault Summary Report
-----
fault class          code #faults
-----
Detected             DT   144415
Possibly detected    PT   4003
Undetectable         UD   1516
ATPG untestable     AU   8961
Not detected         ND   7607
-----
total faults          166502
test coverage        88.74%
-----
```

Random Fault Sampling

Using a sample of faults rather than all possible faults can reduce the total runtime for large designs. You can create a random sample of faults using the `-retain_sample percentage` option of the `remove_faults` command.

The `percentage` argument of the `-retain_sample` option indicates a probability of retaining each individual fault and does not indicate an exact percentage of all faults to be retained. For example, if `percentage = 40`, for a fault population of 10,000, TetraMAX ATPG does not retain exactly 4,000 faults. Instead, it processes each fault in the fault list and retains or discards each fault according to the specified probability. For large fault populations, the exact percentage of faults kept is close to 40 percent, but for smaller fault populations, the actual

percentage might be a little bit more or less than what is requested, because of the granularity of the sample.

For example, the following sequence requests retaining a 25 percent sample of faults in `block_A` and `block_B` and a 50 percent sample of faults in `block_C`.

```
TEST-T> add_faults /spec ASIC/block_A
TEST-T> add_faults /spec ASIC/block_B
TEST-T> remove_faults -retain_sample 50
TEST-T> add_faults /spec ASIC/block_C
TEST-T> remove_faults -retain_sample 50
```

You can combine the `-retain_sample` option with the capabilities of defining faults and nofaults from a fault list file for flexibility in selecting fault placement.

As an alternative to the `remove_faults` command, you can choose Faults > Remove Faults to access the Remove Faults dialog box.

Fault Dictionary

In some products, a “fault dictionary” is used to translate a fault location into a pattern that tests that location, and to translate a pattern number into a list of faults detected by that pattern.

TetraMAX ATPG does not produce a traditional fault dictionary. Instead, it supports a diagnostics mode that translates tester failure data into the design-specific fault location identified by the failure data. For more information, see “[Diagnosing Manufacturing Test Failures](#)”.

Fault Categories and Classes

Faults are assigned to classes corresponding to their current fault detection or detectability status. A two-character code is used to specify a fault class. Fault classes are hierarchically defined: low-level fault classes can be grouped together to form a higher level fault classes. Faults are only assigned the low fault classes but the high level fault classes may be used for reporting. The fault class hierarchy for all fault classes is as follows:

Fault Class Hierarchy

DT - Detected

DR - Detected Robustly

DS - Detected by Simulation

DI - Detected by Implication

D2 - Detected clock fault with loadable nonscan cell faulty value of 0 and 1

TP - Transition partially detected

PT - Possibly Detected

AP - ATPG Untestable Possibly Detected

NP - Not analyzed, Possibly Detected

P0 - Detected clock fault and loadable nonscan cell faulty value is 0

P1 - Detected clock fault and loadable nonscan cell faulty value is 1

UD - Undetectable

UU - Undetectable Unused

UO - Undetectable Unobservable

UT - Undetectable Tied

UB - Undetectable Blocked

UR - Undetectable Redundant

AU - ATPG Untestable

AN - ATPG Untestable Not-Detected

AX - ATPG Untestable Timing Exceptions

ND - Not Detected

NC - Not Controlled

NO - Not Observed

DT (Detected) = DR + DS + DI + D2 + TP

The "detected" fault class is comprised of faults which have been identified as "hard" detected. A hard detection guarantees a detectable difference between the expected value and the fault effect value. The detection identification can be performed by simulation or implication analysis.

- **DR (Detected Robustly)**

DR faults are hard detected by the fault simulator using weak non-robust (WNR), robust (ROB), or hazard-free robust (HFR) testing criteria to mark path delay faults. During ATPG, at least one pattern that caused the fault to be placed in this class is retained. This classification applies only to Path Delay ATPG.

- **DS (Detected by Simulation)**

DS faults are hard detected by explicit simulation of patterns. During ATPG, at least one pattern that caused the fault to be placed in this class is retained.

- **DI (Detected by Implication)**

DI faults are detected by an implication analysis. Faults which reside on pins which are in the scan chain path are declared detected due to the application of a scan chain functional test. Faults on ungated circuitry that connect to the shift clock line of scan cells are also considered detected by implication. Faults on ungated circuitry that connect to the set/reset lines of scan cells and cause the set/reset to be active are also considered detected by implication. No credit is given when the scan chain path is multiply sensitized. Faults are immediately placed into this fault class when they are added to the fault list.

- D2

A fault is classified as D2 if a clock fault is detected and the loadable nonscan cell faulty value is set to both 0 and 1. Note that the loadable nonscan cells feature must be active. For more information, see "Using Loadable Nonscan Cells in TetraMAX."

- TP (Transition Partially-Detected)

TP faults are detected with a slack that exceeds the minimum slack by more than value specified by the `-max_delta_per_fault` option of the `set_delay` command. A TP fault can continue to be simulated with the intention of getting a better test for the fault.

PT (Possibly Detected) = AP + NP + P0 + P1

- AP (ATPG Untestable, Possibly Detected)

AP faults are possibly detected faults. A faulty machine response will simulate an "X" rather than a 1 or 0. Analysis has determined that the fault cannot be detected with the current ATPG constraints and restrictions so the fault is removed from the active fault list and no further patterns for detecting this fault is attempted.

- NP (Not Analyzed, Possibly Detected)

NP faults are identical to AP faults except that either analysis was not completed or could not prove that the fault would always simulate as an X. It is still possible that a different pattern could detect the fault and it's classification could become DS, until then it's classification remains NP and it remains in the active fault list

- P0

A fault is classified as P0 fault if a clock fault is detected and the loadable nonscan cell faulty value is set to 0. This classification applies only if the loadable nonscan cells feature is active. For more information, see "Using Loadable Nonscan Cells in TetraMAX."

- P1

A clock fault is classified as P1 if a clock fault is detected and the loadable nonscan cell faulty value is set to 1. Note that the loadable nonscan cells feature must be active. For more information, see "Using Loadable Nonscan Cells in TetraMAX."

UD (Undetectable) = UU + UO + UT + UB + UR

The "undetectable" fault classes include faults which cannot be detected (either hard or possible) under any conditions. When calculating [test coverage](#), these faults are not considered because they have no logical effect on the circuit behavior and cannot cause failures.

- UU (Undetectable Unused)

UU faults are located on circuitry with no connectivity to an externally observable point. During the creation of the simulation model, the default is to remove this unused circuitry which results in these faults not existing. To expose these faults, you need to select the `-nodelete_unused_gate` option of the `set_build` command. Faults are immediately placed into this fault class when they are added to the fault list.

- **UO (Undetectable Unobservable)**

UO faults are similar to UU faults, except they are located on unused gates *with* fanout (i.e., gates connected to other unused gates). Faults on unused gates *without* fanout are identified as UU faults.

- **UT (Undetectable Tied)**

A UT fault is located on a pin that is tied to a value that is the same as the fault value. Faults are immediately placed into this fault class when they are added to the fault list.

- **UB (Undetectable Blocked)**

A UB fault is located on circuitry that is blocked from propagating to an observable point due to tied logic. Faults are immediately placed into this fault class when they are added to the fault list.

- **UR (Undetectable Redundant)**

URs fault are undetectable (using both hard detection and possible detection). Test generation fault analysis is performed when adding faults, during pattern-by-pattern test generation (as a result of the `run_atpg` command), and as a dedicated analysis of local or global redundancies (also as a result of `run_atpg`). When adding faults (using the `add_faults` command), an analysis is performed to identify and remove from the active list those faults which can easily be shown to be AU or UR. A simple form of ATPG is used during this analysis. Fault grading can never place a fault in this class.

AU (ATPG Untestable) = AN

"ATPG Untestable" faults include faults which can neither be hard detected under the current ATPG conditions nor proved redundant. When calculating test coverage, these faults are considered the same as untested faults because they have the potential to cause failures.

- **AN (ATPG Untestable, Not Detected)**

AN faults have not been possibly detected and an analysis was performed to prove it cannot be detected under current ATPG conditions. The analysis also failed the redundancy check. Faults can immediately be placed in this class if they are inconsistent with the pre-calculated constrained value information. Others can require test generation analysis. After they are placed in this class, they are removed from the active fault list and not given any further opportunity to become possible detected. Primary reasons for faults in this classification include:

- Fault untestable due to a constraint which is in effect.
- Fault requires sequential patterns for detection.
- Fault can only be possible detected.
- Fault requires using an unresolvable Z state for detection.

- **AX (ATPG Untestable, Timing Exceptions)**

For each fault affected by SDC (Synopsys Design Constraints) timing exceptions, if all the gates in both the backward and forward logic cones are part of the same timing exception simulation path, then the fault is marked AU and is assigned an AX subclass. This analysis

finds the effects of setup exceptions, so it does not affect exceptions that are applied only to hold time.

To enable this type of analysis, use the `set_atpg -timing_exceptions_au_analysis` command. To configure separate reporting of these faults, use the `set_faults -summary verbose` command.

Note that AX analysis is applied only for transition delay faults. The commands used for AX analysis are accepted for other fault models, but the results will not show any AX faults.

ND (Not Detected) = NC + NO

An ND fault indicates that test generation has not yet been able to create a pattern that controls or observes the fault. For these faults, it is possible that increasing the ATPG effort with the `set_atpg -abort_limit` command will result in these faults becoming some other classification.

- NC (Not Controlled)

The NC fault class indicates that no pattern was yet found that would control the fault site to the state necessary for fault detection. This is the initial default class for all faults.

- NO (Not Observed)

The NO fault class indicates that, although the fault site is controllable, that no pattern has yet been found to observe the fault so that credit can be given for detection.

Fault Summary Reports

The following sections describe the various types of summary reports:

- [Fault Summary Report Examples](#)
- [Test Coverage](#)
- [Fault Coverage](#)
- [ATPG Effectiveness](#)

Fault Summary Report Examples

By default, TetraMAX ATPG displays fault summary reports using the five categories of fault classes, as shown in [Example 1](#).

Example 1: Fault Summary Report: Test Coverage

```
-----
Uncollapsed Fault Summary Report
-----
fault class          code #faults
-----
Detected            DT   144361
```

Possibly detected	PT	4102
Undetectable	UD	1516
ATPG untestable	AU	8828
Not detected	ND	7695
<hr/>		
total faults		166502
test coverage		88.74%
<hr/>		

For a detailed breakdown of fault classes, use the `-summary verbose` option of the `set_faults` command:

```
TEST-T> set_faults -summary verbose
```

[Example 2](#) shows a verbose fault summary report, which includes the fault classes in addition to the fault categories.

Example 2: Verbose Fault Summary Report

Uncollapsed Fault Summary Report		
fault class	code	#faults
Detected	DT	144415
detected_by_simulation	DS	(117083)
detected_by_implicitation	DI	(27332)
Possibly detected	PT	4003
atpg_untestable-pos_detected	AP	(403)
not_analyzed-pos_detected	NP	(3600)
Undetectable	UD	1516
undetectable-unused	UU	(4)
undetectable-tied	UT	(565)
undetectable-blocked	UB	(469)
undetectable-redundant	UR	(478)
ATPG untestable	AU	8961
atpg_untestable-not_detected	AN	(8961)
Not detected	ND	7607
not-controlled	NC	(503)
not-observed	NO	(7104)
<hr/>		
total faults		166502
test coverage		88.74%

The test coverage figure at the bottom of the report provides a quantitative measure of the test pattern quality. You can optionally choose to see a report of the fault coverage or ATPG effectiveness instead.

The three possible quality measures are defined as follows:

- Test coverage = detected faults / detectable faults
- Fault coverage = detected faults / all faults

- ATPG effectiveness = ATPG-resolvable faults / all faults

Test Coverage

Test coverage gives the most meaningful measure of test pattern quality and is the default coverage reported in the fault summary report. Test coverage is defined as the percentage of detected faults out of detectable faults, as follows:

$$\text{Test Coverage} = \frac{\text{DT} + (\text{PT} \times \text{PT_credit})}{\text{All_Faults} - \text{UD} - (\text{AN} \times \text{AU_credit})} \times 100$$

PT_credit is initially 50 percent and AU_credit is initially 0. You can change the settings for PT_credit or AU_credit using the `set_faults` command.

By default, the fault summary report shows the test coverage, as in [Example 1](#) and [Example 2](#).

Fault Coverage

Fault coverage is defined as the percentage of detected faults out of all faults, as follows:

$$\text{Fault Coverage} = \frac{\text{DT} + (\text{PT} \times \text{PT_credit})}{\text{All_Faults}} \times 100$$

Fault coverage gives no credit for undetectable faults; PT_credit is initially 50 percent.

To display fault coverage in addition to test coverage with the fault summary report, use the `-fault_coverage` option of the `set_faults` command.

[Example 3](#) shows a fault summary report that includes the fault coverage.

Example 3: Fault Summary Report: Fault Coverage

```
TEST-T> set_faults -fault_coverage
TEST-T> report_faults -summary
-----
Uncollapsed Fault Summary Report
-----
fault class          code #faults
-----
Detected             DT   144361
Possibly detected    PT   4102
Undetectable         UD   1516
ATPG untestable      AU   8828
Not detected         ND   7695
-----
total faults          166502
test coverage        88.74%
fault coverage       87.93%
```

ATPG Effectiveness

ATPG effectiveness is defined as the percentage of ATPG-resolvable faults out of the total faults, as follows:

$$\text{ATPG_eff} = \frac{\text{DT} + \text{UD} + \text{AN} + (\text{NP} \times \text{PT_credit})}{\text{All_Faults}} \times 100$$

In addition to faults that are detected, full credit is given for faults that are proven to be untestable by ATPG. PT_credit is initially 50 percent.

To display ATPG effectiveness with the fault summary report, use the `-atpg_effectiveness` option of the `set_faults` command. [Example 4](#) shows a fault summary report that includes the ATPG effectiveness.

Example 4: Fault Summary Report: ATPG Effectiveness

```
TEST-T> set_faults -atpg_effectiveness
TEST-T> report_faults -summary
-----
Uncollapsed Fault Summary Report
-----
fault class          code #faults
-----
Detected             DT   144361
Possibly detected    PT   4102
Undetectable         UD   1516
ATPG untestable      AU   8828
Not detected         ND   7695
-----
total faults          166502
test coverage        88.74%
fault coverage       87.93%
ATPG effectiveness   94.30%
```

See Also

[Direct Fault Crediting](#)

Using Clock Domain-Based Faults

TetraMAX ATPG includes a set of command options that enable you to report fault coverage for transition or stuck-at faults on a per-clock domain basis. You can also add or remove faults for particular clock domains so that ATPG or fault simulation targets only those clock domains that are of interest.

Note the following when using this feature:

- TetraMAX ATPG distinguishes faults captured by a clock and launched by a clock:
 - Faults are considered to be captured by a clock when they feed a logic cone that enters the data input of a flip-flop clocked by that clock.
 - Faults are considered to be launched by a clock when they are fed by a logic cone starting from the output of a flip-flop clocked by that clock.
 - The clock, set, and reset inputs of flip-flops are not considered when determining capture; faults leading to them are captured by the NO_CLOCK domain.
- Faults within the logic core of more than one clock are not considered to belong to either domain. Instead, they are put into a separate category called MULTIPLE. Thus, the clock domain faulting is called exclusive because each clock domain excludes the effects of other clocks.
- Faults given the status Detected by Implication (DI) are detected by the scan chain load/unload sequence. This sequence uses shift constraints which can differ dramatically from the capture constraints that are used to calculate launch and capture clocks for reporting faults by clock domain. This often results in DI faults being reported as captured by the NO_CLOCK domain if the shift path is blocked by the capture constraints. If shift-only clocks are used, this can result in DI faults being both launched and captured by the NO_CLOCK domain.
- Faults that can be launched by one clock and PI/PIO, or that can be captured by one clock and PO/PIO, are not considered MULTIPLE faults. These faults are added, removed, or reported when only the one clock is specified as the launch or capture clock, and they are considered exclusive faults.
- When the special domains PI, PO or NO_CLOCK are specified, the only faults added are those launched or captured exclusively by the specified domain, unless shared faults are also specified. These domains are treated in a more restricted way because they generally cannot be used to test transition delay faults, so the ability to add them is included mainly for the sake of completeness.

When adding faults launched and captured by specific clocks and also other clocks, as many as four commands may be required. For example:

- The `add_faults -launch A -capture B` command adds faults launched exclusively by A and captured exclusively by B.
- The `add_faults -launch A -capture B -shared` command adds faults launched by A and another clock and captured by B and another clock.
- The `add_faults -launch A -capture B -shared_launch` command adds faults launched by A and another clock and captured exclusively by B.
- The `add_faults -launch A -capture B -shared_capture` command adds faults launched exclusively by A and captured by B and another clock.

[Table 1](#) list all the commands and command options associated with reporting clock domain-based faults.

Table 1: Commands and Options Used for Reporting Clock Domain-Based Faults

Command	Description
<code>add_faults -launch <i>launch_clock</i></code>	Specifies the launch clock of the faults to be added. You can use this switch independently, or in conjunction with the <code>-capture</code> switch.
<code>add_faults -capture <i>clock_name</i></code>	Specifies the capture clock of the faults to be added. You can use this switch independently, or in conjunction with the <code>-launch</code> switch.
<code>add_faults -exclusive</code>	Specifies that only the faults that are driven and captured exclusively (using a single launch and a single capture) are to be added. Faults exclusively driven by PI or observed by PO are also added.
<code>add_faults -shared</code>	Specifies that only the faults that are launched or captured by multiple clocks should be added. This excludes all PI and PO faults described in the <code>add_faults</code> options described previously.
<code>add_faults -shared_launch</code>	Specifies that faults launched by the specified clock and other clocks are added. An error is reported if you specify this switch without also using the <code>-launch</code> option.
<code>add_faults -shared_capture</code>	Specifies that faults captured by the specified clock and other clocks are added. An error is reported if you use this switch without also using the <code>-capture</code> option.
<code>add_faults -inter_clock_domain</code>	Adds only exclusive faults that are driven and captured by different clock domains.
<code>add_faults -intra_clock_domain</code>	Adds only exclusive faults that are driven and captured by the same clock domains.
<code>remove_faults -launch <i>clock_name</i></code>	Specifies the launch clock of the faults to be removed. You can use this switch independently, or in conjunction with the <code>-capture</code> switch (described later).

<code>remove_faults -capture clock_name</code>	Specifies the capture clock of the faults to be removed. You can use this switch independently, or in conjunction with the –launch switch (described previously).
<code>remove_faults -exclusive</code>	Specifies that only the faults that are driven and captured exclusively (using a single launch and a single capture) are to be removed. Faults exclusively driven by PI or observed by PO are also removed.
<code>remove_faults - shared</code>	Specifies that only the faults that are launched or captured by multiple clocks should be removed. This excludes all PI and PO faults (described in the <code>remove_faults</code> options previously).
<code>remove_faults -inter_clock_domain</code>	Removes only exclusive faults that are driven and captured by different clock domains.
<code>remove_faults -intra_clock_domain</code>	Removes only exclusive faults that are driven and captured by the same clock domains.
<code>report_faults -per_clock_domain</code>	All specified faults are reported with extra information for their launch and capture clocks. Note that all clocks are reported, even for "shared" or "multiple" categories.
<code>report_summaries faults -per_clock_domain</code>	Specifies that the clock report should be divided on a per clock domain basis as shown in the following example. All shared faults are reported as one category.
<code>report_summaries faults -launch clock_name</code>	Specifies the launch clock of the faults to be reported on. This switch can be used independently, or in conjunction with the –capture switch.
<code>report_summaries faults -capture clock_name</code>	Specifies the capture clock of the faults to be reported on. This switch can be used independently or in conjunction with the –launch switch (described previously).
<code>report_summaries faults -exclusive</code>	Excludes the multiple launch and capture section from the report.
<code>report_summaries faults -shared</code>	Reports only the section relating to multiple launch and capture clocks.

report_summaries faults -inter_clock_ domain	Reports on only the exclusive faults that are driven and captured by different clock domains.
report_summaries faults -intra_clock_ domain	Reports on only the exclusive faults that are driven and captured by the same clock domains.

Using Signals That Conflict With Reserved Keywords

The MULTIPLE, NO_CLOCK, PI, and PO names are reserved keywords when you use the `-launch` and `-capture` options. If a clock signal uses one of these names, the clock signal always takes priority when these options are used.

For example, if a clock is named MULTIPLE, then the command `add_faults -launch MULTIPLE` adds faults launched exclusively by the clock named MULTIPLE. In this case, if you want to add faults launched by multiple clocks, you can use the command `add_faults -launch multiple`. This command works as expected because the reserved names can be all uppercase or all lowercase; however, the actual clock names are case-sensitive.

Finding Particular Untested Faults Per Clock Domain

If you specify the `report_summaries faults -per_clock` command, TetraMAX ATPG provides only aggregate results. To find individual faults, specify the `report_faults -per_clock_domain` command, then use UNIX editing commands to manipulate the faults of interest.

14

Fault Simulation

Fault simulation determines the test coverage obtained by an externally generated test pattern. To perform fault simulation, you use functional test patterns that were developed to test the design and have been previously simulated in a logic simulator to verify correctness. The functional test patterns should contain the expected values, unless you are using the Extended Value Change Dump (VCD) format. The expected values tell TetraMAX ATPG when and what to measure.

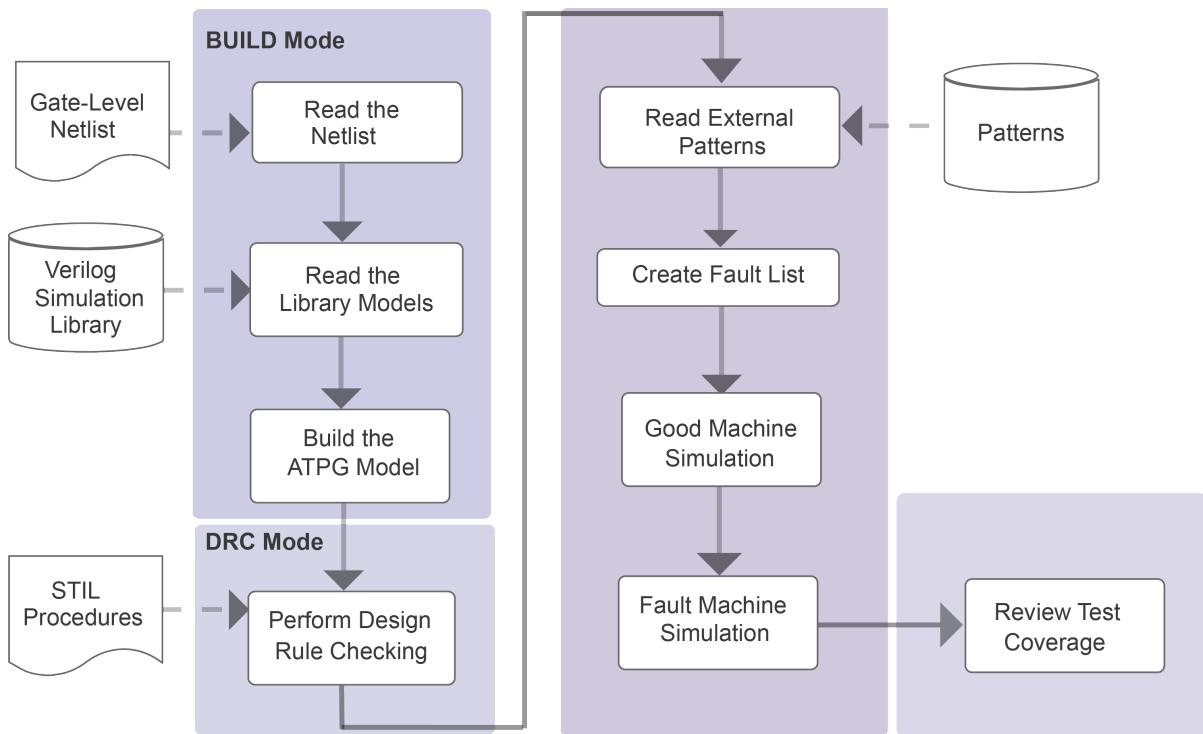
The following topics describe fault simulation:

- [Fault Simulation Design Flow](#)
- [Preparing Functional Test Patterns for Fault Simulation](#)
- [Reading the Functional Test Patterns](#)
- [Initializing the Fault List](#)
- [Performing Good Machine Simulation](#)
- [Performing Fault Simulation](#)
- [Combining ATPG and Functional Test Patterns](#)
- [Running Multicore Simulation](#)
- [Per-Cycle Pattern Masking](#)

Fault Simulation Design Flow

The fault simulation design flow prepares functional test patterns for fault simulation, reads the test patterns, initializes the fault list, performs good machine simulation, performs fault simulation, and reviews the test coverage.

Figure 1 Fault Simulation Design Flow



Preparing Functional Test Patterns for Fault Simulation

The ATPG and fault simulation algorithms emphasize speed and efficiency over the ability to use or simulate gate timing delays. There are corresponding limitations on functional test patterns. The requirements for test patterns are described in the following sections:

- [Pattern Compliance with Automated Test Equipment \(ATE\)](#)
- [Checking Patterns for Timing Insensitivity](#)
- [Checking Patterns for Recognizable Formats](#)

Pattern Compliance with ATE

Because the functional test patterns are used by ATE, you must verify that the patterns comply with requirements of ATE. Each brand and model of ATE has its own list of restrictions. The following general list of characteristics is usually acceptable:

- The input stimuli, clocks, and expected response outputs can be divided into a sequence of identical tester cycles.
- Each tester cycle is associated with a timing set. There are a fixed number of timing sets.
- The test cycle defines the state values to be applied as inputs and measured as outputs, and the associated timing set defines the cycle period and the timing offsets within the cycle when inputs are applied, clocks are pulsed, and outputs are sampled.
- The functional patterns are regular, with the timing of input changes and clock pulse locations constant from one cycle to the next.
- The functional pattern set maps into four or fewer timing sets.

Every ATE has its own set of rules for timing restrictions, including the following examples:

- Minimum and maximum test cycle period
- Minimum and maximum pulse width
- Proximity of a pulsed signal to the beginning or end of a cycle
- Proximity of two signal changes to one another
- Accuracy and placement of measure strobes
- Placement accuracy of input transitions

Checking Patterns for Timing Insensitivity

Functional test patterns must be timing insensitive within each test cycle. The design can have no race conditions that depend on gate delays that must be resolved on nets that reach a sequential device, a RAM or ROM, or a primary output port.

To check for timing insensitivity:

1. Simulate the design in a logic simulator without timing and use a unit-delay or zero-delay timing mode.
2. If the simulation passes, simulate it again with all timing events expanded in time by five or ten times.

If the functional test patterns pass under these conditions, they can be considered timing insensitive.

Timing Sensitivity

The following examples cause timing sensitivity:

- **A pulse generator:** An edge transition of an input port results in a pulse on an output port or at the data capture input of an internal register. This pulsed value occurs at a specific delay from the input event and, unless the output is measured at the correct time or the internal register is clocked at the correct time, the pulsed value is lost. This type of design fails simulation in the absence of actual timing.

You can correct this situation in one of two ways:

- Hold the triggering port fixed to a constant value in the functional patterns.
- Add some shunting circuitry (enabled in some sort of test mode) that blocks the internal propagation of the pulsed value.

- **Timing-critical measurements:** An input port event at offset 0 ns turns on an output driver in 100 nanoseconds (ns), but the patterns are set to measure a Z value at 90 ns before the driver is turned on. Although this measurement is correct in the real device, TetraMAX ATPG uses only unit delays and reports a simulation mismatch.

You can correct this situation by measuring at 110 ns and changing the expected data to the appropriate non-Z value.

- **Multiple active clocks or asynchronous set/reset ports in the same cycle:** With careful attention to timing, correct use of clock trees, and good analysis tools, you can design blocks of logic with intermixed clock zones that operate correctly with functional patterns when more than one clock is active. However, because TetraMAX ATPG uses zero delay and not gate timing, simulating designs that contain more than one active clock can result in the erroneous identification of internal race conditions and subsequent elimination of functional test patterns.

- Use master-slave clocking in your design.
- Use resynchronization latches between clock domains.
- Arrange your test patterns so that in any one cycle you have only one active clock.

Preparing Your Design for Fault Simulation

The process for preparing your design for fault simulation is generally the same as preparing for the ATPG design flow:

1. [Preprocessing the Netlist](#)
2. [Reading the Design and Libraries](#)
3. [Building the ATPG Design Model](#)
4. [Declaring Clocks \(optional\)](#)
5. [Running DRC](#)

Preprocessing the Netlist

If necessary, preprocess the netlist for compatibility with TetraMAX ATPG. For more information, see [Netlist Requirements](#).

Reading the Design and Libraries

As with ATPG, for TetraMAX ATPG fault simulation you first invoke TetraMAX ATPG, read in the design netlist, and read in the library models. For details, see [Reading the Netlist](#) and [Reading Library Models](#).

Note the following example command sequence:

```
% tmax  
  
BUILD-T> read_netlist spec ASIC.v  
BUILD-T> read_netlist spec_lib/*.v -noabort
```

Building the ATPG Design Model

To build the ATPG design model for fault simulation, you use the same `run_build_model` command as for ATPG. For fault simulation, enter the following command:

```
BUILD-T> run_build_model top_module_name
```

Example 1 run_build_model Transcript

```
BUILD-T> run_build_model spec ASIC  
-----  
Begin build model for topcut = spec ASIC ...  
-----  
End build model: #primitives=101004, CPU_time=13.90 sec,  
Memory=34702381  
-----  
Begin learning analyses...  
End learning analyses, total learning CPU time=33.02
```

Declaring Clocks

Although the nonscan functional stimuli provide all inputs, you might want to declare clocks so that TetraMAX ATPG can perform its clock-related DRC checks. Declaring clocks is optional. Some clock violations found during `run_drc` can affect the simulator and it might be necessary to remove `add_clocks` commands.

If certain ports in the functional stimuli are operated in pulsed fashion within a cycle, you might want to provide this information to TetraMAX ATPG by declaring these ports to be clocks.

A typical command sequence for declaring a clock is shown in the following example:

```
DRC-T> add_clocks 0 CLK  
DRC-T> add_clocks 1 RESETB
```

Running DRC

Running DRC with nonscan functional test patterns tends to be simpler than running DRC for ATPG, because the additional check for scan chains and other ATPG-only checks do not need to be performed.

DRC for Nonscan Operation

For nonscan operation, if you have defined a clock, you do not need to specify an STL procedure file unless it is necessary for defining port timing. To run DRC without a file, enter the following commands:

```
DRC-T> set_drc -nofile
DRC-T> run_drc
```

To run DRC with a file, enter the following command:

```
DRC-T> run_drc filename
```

Note the following:

- If you encounter DRC violations that apply to ATPG but are not relevant to the fault grading of nonscan functional patterns, adjust the DRC rule severity by using the `set_rules rule_id warning` command, and then execute the `run_drc` command again.
- In some cases, external functional VCDe patterns are not always compliant with TetraMAX behaviors -- particularly when the clocks are active at the same time that PIs change state. The basic rule is to define clocks in DRC if there are no C-rule violations in the design. If there are C violations, consider passing all signals as inputs and not defining any signals as clocks.

Example 2 shows a transcript of `run_drc` for a nonscan operation.

Example 2 Running DRC for Nonscan Operation

```
DRC-T> set_drc -nofile
DRC-T> run_drc
-----
Begin scan design rule checking...
-----
Begin Bus/Wire contention ability checking...
Bus summary: #bus_gates=4, #bidi=4, #weak=0, #pull=0, #keepers=0
    Contention status: #pass=0, #bidi=4, #fail=0, #abort=0,
#not_analyzed=0
    Z-state status   : #pass=0, #bidi=4, #fail=0, #abort=0,
#not_analyzed=0
Bus/Wire contention ability checking completed, CPU time=0.02 sec.

-----
Begin simulating test protocol procedures...
Test protocol simulation completed, CPU time=0.00 sec.
```

```
Begin scan chain operation checking...
Scan chain operation checking completed, CPU time=0.00 sec.
-----
Begin clock rules checking...
Warning: Rule C3 (no latch transparency when clocks off) failed 5
times.
Clock rules checking completed, CPU time=0.02 sec.
-----
Begin nonscan rules checking...
Warning: Rule S23 (unobservable potential TLA) failed 5 times.
Nonscan cell summary: #DFF=0 #DLAT=10 tla_usage_type=none
Nonscan behavior: #CX=5 #LS=5
Nonscan rules checking completed, CPU time=0.03 sec.
-----
Begin contention prevention rules checking...
Contention prevention checking completed, CPU time=0.00 sec.

Begin DRC dependent learning...
DRC dependent learning completed, CPU time=0.00 sec.
-----
DRC Summary Report
-----
Warning: Rule S23 (unobservable potential TLA) failed 5 times.
Warning: Rule C3 (no latch transparency when clocks off) failed 5
times.
There were 10 violations that occurred during DRC process.
Design rules checking was successful, total CPU time=0.21 sec.
```

DRC for Scan Operation

For scan operation, the STL procedure file you specify should contain, at a minimum, the scan chain definitions, the waveform timing definitions, and the `load_unload` and `Shift` procedure definitions. You can define clocks, primary inputs constraints, and primary input equivalences on the command line or within the STL procedure file, or you can use a combination of both.

To run DRC with a STIL procedure file, enter the following command:

```
DRC-T> run_drc filename
```

Reading Functional Test Patterns

You can read functional test patterns using the Set Patterns dialog box, or by running the `set_patterns` command at the command line.

If you are reading external patterns in VCDE format, you need to specify the trigger conditions for measurement. In the Set Patterns dialog box, use the Strobe Position option and related options; or in the `set_patterns` command, use the `-strobe` option.

The following sections describe how to read functional test patterns:

- [Using the Set Patterns Dialog Box](#)
 - [Using the set_patterns Command](#)
 - [Specifying Strobes for VCDE Pattern Input](#)
-

Using the Set Patterns Dialog Box

To read in the functional test patterns using the Set Patterns dialog box:

1. From the menu bar, choose Patterns > Set Pattern Options. The Set Patterns dialog box appears.
 2. Click External.
 3. In the Pattern File Name text field, enter the name of the pattern file, or locate it using the Browse button.
 4. Click OK.
-

Using the set_patterns Command

The following example shows how to read functional test patterns using the set_patterns command:

```
TEST-T> set_patterns -external data.vcde -strobe rising CLK \
    -strobe offset 50 ns
```

TetraMAX ATPG automatically determines the type of patterns being read and whether they are in standard or GZIP format, and handles all variations automatically.

The following example transcript show output from the set_patterns external command:

```
TEST-T> set_patterns ext patterns.v
End parsing Verilog file patterns.v with 0 errors;
End reading 41 patterns, CPU_time = 0.02 sec, Memory = 2952
```

For examples of functional patterns, see Pattern Input.

Specifying Strobes for VCDE Pattern Input

Functional patterns in VCDE format do not contain measure information. Therefore, when you read in VCDE patterns with the Set Patterns dialog box or the set_patterns command, you need to specify the trigger conditions for measuring expected values. You can specify strobes that occur at a fixed periodic interval, or you can specify strobe trigger conditions based upon events occurring at a specified primary input port, output port, or bidirectional port.

In the Set Patterns dialog box, when you select External as the pattern source, the Strobe Position option and related options are displayed. These options apply to reading VCDE patterns only. The set of options changes according to the Strobe Position setting.

The Strobe Position can be set to any one of the following states:

- **None:** This option is not supported for VCDE input.
- **Period:** Strobes occur at a fixed periodic interval, starting in each cycle at the offset value

specified in the Offset field.

- **Event:** A strobe is triggered by any event occurring on the port specified in the Port Name field. Any event at that port causes a strobe, including a transition with no level change such as 1 to 1 or 0 to 0.
- **Rising:** A strobe is triggered by each transition to 1 on the port specified in the Port Name field. Any transition to 1 causes a strobe, including 0 to 1, 1 to 1, X to 1, or Z to 1.
- **Falling:** A strobe is triggered by each transition to 0 on the port specified in the Port Name field. Any transition to 0 causes a strobe, including 1 to 0, 0 to 0, X to 0, or Z to 0.

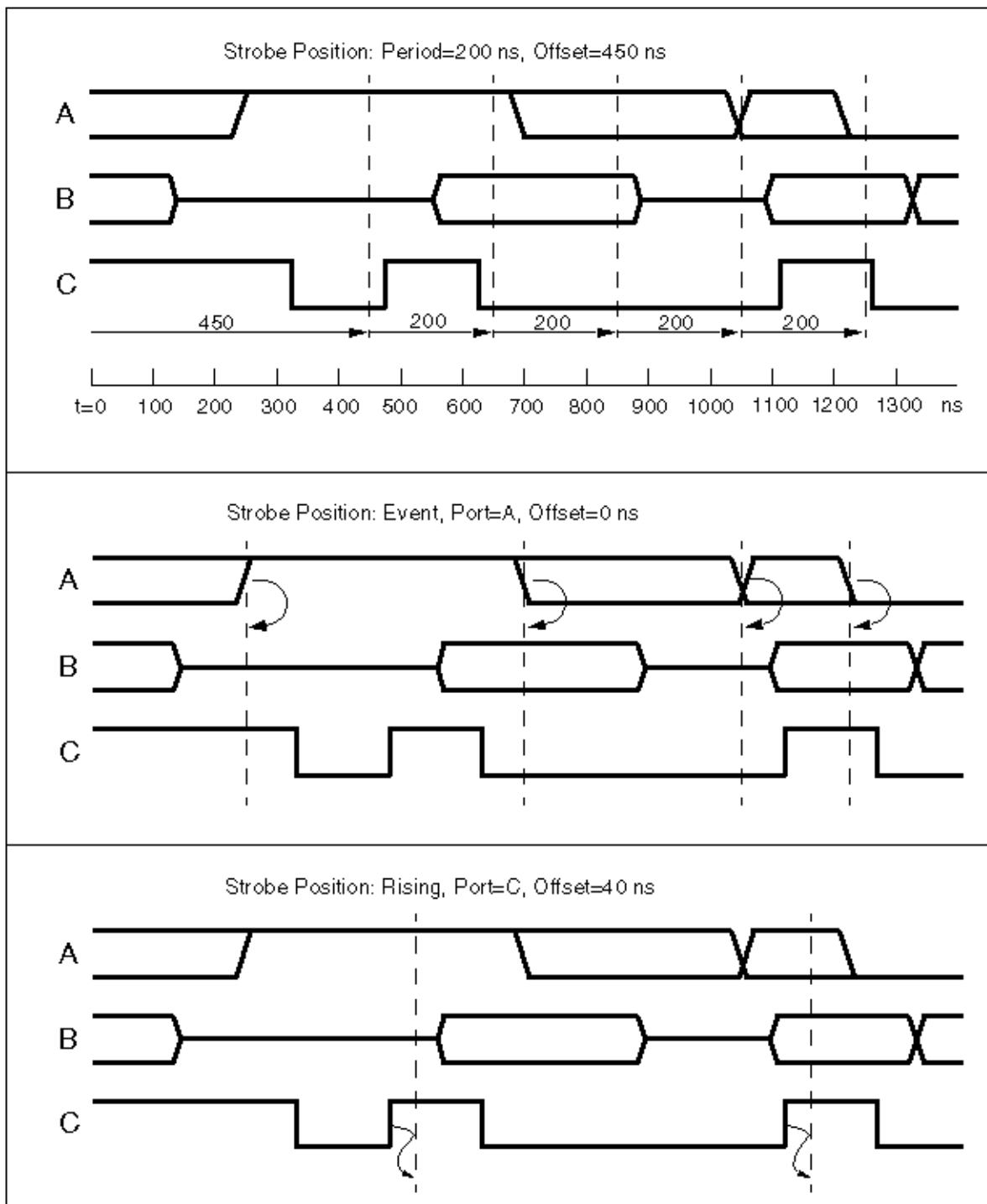
For the Event, Rising, and Falling strobe modes, you can specify an offset value in the Offset field. By default, the offset is 0, which causes the strobe to occur just before the trigger event. In other words, the measure occurs just before processing of the VCDE data change that is the trigger event.

To make the strobe occur at a specific time after the trigger event, specify a positive offset value. Negative offsets for strobes are not supported.

Each period and offset setting must be a positive integer or zero. You specify the time units in the Unit fields: seconds, milliseconds, microseconds, nanoseconds, picoseconds, or femtoseconds.

To specify the strobes using the command-line interface, use the `-strobe` option of the `set_patterns` command. For details on the command syntax, see the online help for the `set_patterns` command.

Figure 1 shows some timing diagrams with the strobe points resulting from various strobe specification settings.

Figure 1 VCDE Strobe Specification Examples

Each timing diagram shows the primary I/O signals A, B, and C. The vertical dashed lines represent the strobe times. In the first example, the strobes are periodic and independent of the data stream. In the second and third example, the strobes are based on port A and port C, respectively.

Initializing the Fault List

The following sections show you how to initialize a fault list:

- [Using the Add Faults Dialog Box](#)
 - [Using the add_faults Command](#)
-

Using the Add Faults Dialog Box

To initialize the fault list using the Add Faults dialog box,

1. From the Faults menu, choose Add Faults. The Add Faults dialog box appears.
For descriptions of these controls, see Online Help for the `add_faults` command.
 2. To add all potential faults (the most common usage), click All.
 3. Click OK.
-

Using the add_faults Command

You can also initialize the fault list for all faults using the `add_faults -all` command:

```
TEST-T> add_faults -all
```

You can also define fault lists by reading faults or nofaults from a file or by defining specific hierarchical blocks for adding or removing faults:

```
TEST-T> read_faults saved_faults_file  
TEST-T> read_faults saved_faults_file -retain
```

The double-read sequence shown in this example is necessary to restore the exact fault codes saved to the file.

In addition, you can read in a fault list generated by the TetraMAX ATPG patterns and thereby determine the cumulative fault grade for a combination of ATPG and functional test patterns. For details, see Setting Up the Fault List and Combining ATPG and Functional Test Patterns.

Performing Good Machine Simulation

You should perform a good machine simulation using the functional patterns before running a fault simulation, to compare the TetraMAX simulation responses to the expected responses found in the patterns. If the good machine simulation reports errors, there is little value in proceeding to run fault simulation.

As part of setting up the good machine simulation, refer to contention checking as described in ["Choosing Settings for Contention Checking."](#)

The following sections show you how to set up other good machine simulation parameters:

- [Using the Run Simulation Dialog Box](#)
- [Using the set_simulation and run_simulation Commands](#)

Using the Run Simulation Dialog Box

To set up the good machine simulation parameters using the Run Simulation dialog box,

1. Click the Simulation button in the command toolbar at the top of the TetraMAX ATPG main window. The Run Simulation dialog box appears.
For descriptions of these controls, see the online help for the `run_simulation` command.
2. Select required options.
3. Click Set to set the simulation options, or click Run to set the options and begin the good machine simulation.

Using the `set_simulation` and `run_simulation` Commands

To set up the fault simulator from the command line, use a combination of the `set_simulation` command and appropriate options of the `run_simulation` command:

```
DRC-T> set_simulation -measure pat -oscillation 20 2 -verbose  
TEST-T> run_simulation -sequential
```

For the complete syntax and option descriptions, see Online Help for each command.

[Example 1](#) shows a transcript of a simulation run that has no mismatches between the simulated and expected data. For an example with simulation mismatches, see Comparing Simulated and Expected Values.

Example 1 Good Machine Simulation Transcript

```
TEST-T> run_simulation -sequential  
Begin sequential simulation of 36 external patterns.  
Simulation completed: #patterns=36/102, #fail_pats=0(0),  
#failing_meas=0(0)
```

Performing Fault Simulation

After performing a good machine simulation to verify that the functional patterns and expected data agree, you can perform a fault grading or fault simulation of those patterns. Performing fault simulation includes setting up the fault simulator, running the fault simulator, and reviewing the results.

The following sections describe how to run fault simulation:

- [Using the Run Fault Simulation Dialog Box](#)
- [Using the `run_fault_sim` Command](#)
- [Writing the Fault List](#)

Note: The `set_simulation` command described in the ["Performing Good Machine Simulation"](#) section sets the environment for fault simulation as well as for good machine simulation. Many of the options in the Run Simulation dialog box are also included in the Run Fault Simulation dialog box.

Using the Run Fault Simulation Dialog Box

To set up fault simulation parameters using the Run Fault Simulation dialog box,

1. Click the Fault Sim button in the command toolbar at the top of the TetraMAX ATPG main window. The Run Fault Simulation dialog box appears.
For descriptions of these controls, see Online Help for the `run_fault_sim` command.
2. Select required options.
3. Click Set to close the dialog box and set the simulation options, or click Run to set the options and begin the faulty machine simulation.

Using the `run_fault_sim` Command

You can also set up fault simulation parameters using the `run_fault_sim` command:

```
TEST-T> run_fault_sim -sequential
```

The following example shows a typical transcript of a fault simulation run is shown in Example 1.

```
TEST-T> run_fault_sim -sequential
-----
Begin sequential fault simulation of 4540 faults on 36 external
patterns.
-----
#faults      pass #faults      cum. #faults      test      process
simulated    detect/total    detect/active   coverage    CPU time
-----
1675          550     1675      550     3990    13.57%    3.72
3326          669     1651      1219    3321    29.36%    7.41
4540          390     1214      1609    2931    40.22%   11.13
Fault simulation completed: #faults_simulated=4540,test_
coverage=40.22%
```

You review test coverage in the same way as for ATPG. For details, see Reviewing Test Coverage.

The following command generates a summary of fault simulation.

```
TEST-T> report_summaries
```

Writing the Fault List

You write fault lists for fault simulation in the same way as you do for the ATPG flow. The following `write_faults` command writes (saves) a fault list.

```
TEST-T> write_faults file.dat -all -uncollapsed -rep
```

Combining ATPG and Functional Test Patterns

If your design supports scan-based ATPG, you can create ATPG test patterns in addition to functional test patterns. Combining ATPG patterns with functional test patterns can often

produce a more thorough and more complete set of test patterns than using either method alone.

If your design allows both ATPG and functional testing, you can combine the resulting test patterns. The following sections describe the various methods for combining test patterns:

- [Creating Independent Functional and ATPG Patterns](#)
 - [Creating ATPG Patterns After Functional Patterns](#)
 - [Creating Functional Patterns After Creating ATPG Patterns](#)
-

Creating Independent Functional and ATPG Patterns

If you do not want to combine the effects of functional test patterns and ATPG patterns, you can create them independently. The functional test patterns are fault-graded in an appropriate tool, and you obtain a test coverage value for the ATPG patterns that you create using TetraMAX ATPG.

To determine the test coverage overlap, you must perform a detailed comparison of the fault lists from both methods. You should expect overlap. In fact, you might prefer redundancy.

Creating ATPG Patterns After Functional Patterns

If complete functional patterns are to be part of the test flow, use a combined approach with ATPG patterns following functional patterns. The goal of ATPG is to create patterns to test faults not tested by the functional patterns.

The following steps show a typical flow:

1. Use TetraMAX ATPG to fault-grade the functional patterns.
2. Review the resulting test coverage.
3. Write the uncollapsed fault list resulting from the fault simulation.
4. Use TetraMAX ATPG to create ATPG patterns for the fault list you created in step 3.

Example 1 shows a command file that implements this flow.

Example 1 Creating ATPG Patterns After Functional Patterns

```
#  
# --- ATPG follows Fault Grade flow  
#  
read_netlist spec_design.v -del      # read netlist  
read_netlist spec_lib.v              # read library modules  
run_build_model                     # form in-memory design image  
add_clocks 0 CLK                   # define clock  
add_clocks 1 RESETB                # define async reset  
run_drc                            # DRC without a procedure file  
set_patterns -external b010.vin    # read in external patterns  
set_simulation -measure pat       # set up for fault sim  
run_simulation -sequential        # perform good machine simulation  
  
add_faults -all                    # add all faults
```

```

run_fault_sim -sequential          # perform fault grade
report_summaries                  # report results
write_faults pass1.flt -all -uncol -rep      # save fault list
#
# --- switch to SCAN-based ATPG for more patterns
#
drc -force                         # return to DRC mode
set_patterns -delete                # clear out external patterns
set_patterns -internal              # switch to int pattern
generation
run_drc spec_design.spf           # define scan chains and
procedures
read_faults pass1.flt -retain      # start with fault list from
pass1
set_atpg -abort 20 -merge high    # setup for ATPG
run_atpg                           # create ATPG patterns
report_summaries                   # report coverage results
write_patterns pat.v -form verilog -replace  # save patterns
write_faults pass2.flt -all -uncollapsed -rep  # save cumulative
fault
list

```

Creating Functional Patterns After ATPG Patterns

Use a combined approach with functional patterns following ATPG patterns if you want to minimize the effort of creating functional test patterns. On a full-scan design, the ATPG patterns achieve a very high coverage and the functional patterns can be created to test for faults that are untestable with ATPG methods.

The following steps show a typical flow:

1. Use TetraMAX ATPG to create ATPG patterns.
2. Review the resulting test coverage.
3. Save the uncollapsed fault list resulting from ATPG.
4. Save the collapsed fault list of the nondetected faults, which are the faults in the ND, AU, and PT categories. For an explanation of these categories, see Fault Categories and Classes.
5. Use the nondetected fault list to guide your construction of functional patterns to test for the remaining faults.
6. When the functional patterns are ready, fault-grade them using the uncollapsed fault list from the ATPG (generated in step 3 above) as the initial fault list.

Example 2 shows a command file sequence that illustrates this flow.

Example 2 Creating Functional Patterns After ATPG Patterns

```

#
# --- ATPG before Fault Grade
#
read_netlist spec_design.v -del      # read netlist

```

```

read_netlist spec_lib.v          # read library modules
run_build_model                 # form in-memory design image
add_clocks 0 CLK                # define clock
add_clocks 1 RESETB             # define async reset
add_pi_constraints 1 TEST       # define constraints
run_drc spec_design.spf        # define scan chains and
procedures
add_faults -all                 # seed faults everywhere
run_atpg -auto_compression     # create ATPG patterns
write_patterns pat.v -form verilog -replace   # save patterns
write_faults pass1.flt -all -uncollapsed -rep  # save cumulative
fault list
#
# --- switch to Fault Grade mode
#
drc -force                      # clocks will still be defined
remove_pi_constraints -all       # don't constrain when using ext
patterns
set_drc -nofile
run_drc                          # switch to test mode
set_patterns -external b010.vin  # read in external patterns
set_simulation -measure pat     # set up for fault sim
run_simulation -sequential      # perform good machine simulation

read_faults pass1.flt           # seed the fault list
read_faults pass1.flt -retain   # start with fault list from ATPG

run_fault_sim -sequential       # perform fault grade
report_summaries                 # report results
write_faults pass2.flt -all -uncollapsed -replace  # save fault
list

```

Running Multicore Simulation

Multicore simulation is a methodology that enables you to improve simulation runtime by launching multiple slaves to parallelize fault and logic simulation to work on a single host. You can specify the number of processes to launch based on the number of CPUs and available memory on the machine.

Note: Multicore simulation provides similar runtime reductions and works the same way as the multicore ATPG architecture described in Running Multicore ATPG.

The following topics describe how to use multicore simulation and analyze its performance:

- [Invoking Multicore Simulation](#)
- [Interrupt Handling](#)
- [Understanding the Processes Summary Report](#)

- [Resimulating ATPG Patterns](#)
 - [Limitations](#)
-

Invoking Multicore Simulation

Multicore simulation is activated by the following `set_simulation` command:

```
set_simulation -num_processes <number | max>
```

The `number` specification refers to the number of slave processes used during simulation. If `max` is specified, then TetraMAX ATPG computes the maximum number of processes available in the host, based on number of CPUs. If TetraMAX ATPG detects that the host has only one CPU, then single-process simulation is performed instead of multicore simulation with only one slave. Note that you should not specify more processes than the number of CPUs available on the host. You should also consider whether there are other CPU-intensive processes running simultaneously on the host when running the number of processes. If too many processes are specified, performance will degrade and might be worse than single-process simulation. On some platforms, TetraMAX ATPG cannot compute the number of CPUs available and will issue an error if `max` is specified.

Interrupt Handling

To interrupt the multicore simulation process, use Control-c ; in the same manner as a single process. If a slave ends or is killed, the master and remaining slaves will continue to run.

If the master ends or is killed, the slaves will also halt. In this case, there are no ongoing zombie processes, dangling files, or memory leakage.

Understanding the Processes Summary Report

Memory consumption needs to be measured to tune the global data structure to improve the scalability of multicore architecture. Legacy memory reports are not sufficient because they do not deal with issues related to copy-on-modification. ; To facilitate collecting performance data, a summary report of multicore simulation is printed automatically at the end of the simulation process when the `-level expert` option is used with the `set_messages` command. The summary report appears as shown in Example 1.

Example 1 Example Processes Summary Report

Processes Summary Report							
Process	Patterns	Time (s)	Memory (MB)				
ID	pid	Internal	CPU	Elapsed	Shared	Private	Total

0	7611	1231	0.53	35.00	67.78	30.54	98.32	5.27
1	7612	626	35.68	35.00	64.87	22.31	87.18	0.00
2	7613	605	35.50	35.00	64.71	22.47	87.18	0.00
Total		1231	71.71	35.00	67.78	75.32	143.10	5.27

The report in Example 1 contains one row for each process. The first process with an ID of 0 ; is the master process. The child processes have IDs of 1, 2, 3, and so forth. The last row is the sum for each measurement across all processes.

The `pid` ; column lists the process IDs. The `Patterns` ; are the total number of patterns stored by the master or the number of patterns generated by the slave in this particular simulation session. The columns listed under `Time (s)` ; include CPU time and wall time.

The `Memory` ; measurements are obtained by parsing the system-generated file `/proc/pid/smaps`. The file contains memory mapping information created by the OS while the process still exists. The `/proc/pid/` directory cannot be found after the process terminates. The tool parses this file at the proper time to gather memory information for the reporting at the end of parallel simulation.

The `Memory` ; measurement includes `Shared`,; `Private`,; `Total`,; and `Pattern`.; The `Shared` ; column refers to all processes that share the same copy of the memory. The `Private` ; column refers to the process stores local changes in the memory. The `Total` ; column is the sum of `Shared` ; and `Private`.; The `Pattern` ; column refers to memories allocated for storing patterns. The total memory consumption of the entire system is the `Total` ; item in the row `Total`,; which is the sum of total shared memory (maximum of shared memories for each process) and the total private memory (sum of all private memory for all processes). Although the memory for patterns is listed separately, it is part of the master private memory.

Due to a lack of OS support, the Memory section of the summary report is only available on Linux and AMD64 platforms. No other platform gives shared or private memory information in a copy-on-write context. On other formats, the memory reports all 0 ;s for items other than the pattern memory.

Note: The report in Example 1 is printed only when the `set_messages` command is set to - expert. Otherwise, a default summary report, similar to the following example, is printed out:

```
End parallel ATPG: Elapsed time=35.00 sec, Memory=143.10MB.
Processes Summary Report
```

Resimulating ATPG Patterns

You can resimulate ATPG patterns to mask out the observe values for any mismatched patterns verified with `run_simulation` command. This feature is enabled when both multicore ATPG and ATPG pattern re-simulation are enabled, as shown in the following example:

```
set_atpg -resim_atpg fault_sim
set_atpg -num_processes 2
run_atpg -auto
```

The command output is similar to single-process ATPG pattern simulation with mismatch masking messages. The process summary report is automatically printed out at the end of

ATPG, logic simulation, and fault simulation; this report is similar to the process summary report for the corresponding standalone commands.

Limitations

There are several `run_fault_sim` and `run_simulation` command options that are not supported by multicore simulation.

The unsupported `run_fault_sim` options are as follows:

- `-detected_pattern_storage` — This option stores the first detection pattern for each fault. In multicore fault simulation, the patterns are not simulated in the order of the pattern number occurrence.
- `-distributed` — This option is used to launch distributed fault simulation only. It cannot be used in conjunction with multicore fault simulation.
- `-nodrop_faults`

The unsupported `run_simulation` options are as follows:

- `-sequential`
- `-sequential_update`
- `-update`

See Also

[Running Multicore ATPG](#)

Per-Cycle Pattern Masking

A common practice for test engineers is to replace 0s and 1s with Xs in scan patterns on the tester. The goal, in this case, is to mask specific measures that mismatch on the tester.

The per-cycle pattern masking feature enables you to use a masks file to identify the measures to mask out. Then, masked patterns can be written out, and, optionally, test coverage can be recalculated, or the patterns can be simulated.

The following sections describe per-cycle pattern masking:

- [Flow Options](#)
- [Masks File](#)
- [Running the Flow](#)
- [Limitations](#)

Flow Options

There are two flows available for running per-cycle pattern masking: the tester flow and the simulation flow.

The following steps are for the tester flow:

1. The original patterns are written out from TetraMAX ATPG.
2. A few mismatches occur on the tester.
3. The patterns and mismatches are read into TetraMAX ATPG.
4. Mismatches are masked in the pattern.
5. Masked patterns are optionally fault simulated again.
6. Masked patterns are written out from TetraMAX ATPG.
7. All patterns pass on the tester.

The following steps are for the simulation flow:

1. The original patterns are written out from TetraMAX ATPG.
2. Mismatches occur during fault simulation.
3. The patterns and mismatches are read into TetraMAX ATPG.
4. Mismatches are masked in the pattern.
5. Masked patterns are optionally fault simulated again.
6. Masked patterns are written out from TetraMAX ATPG.
7. All patterns pass during simulation.

Masks File

A masks file contains the measures used to mask in the patterns. It uses the same format as the failure log file used for diagnostics and can be pattern-based or cycle-based. The pattern-based format with chain name from parallel STILDPV simulation is also supported. See “Providing Tester Failure Log Files” for details of the file format.

You can create a masks file as a result of running patterns on a tester. Note that only STIL or WGL patterns files can be used with a cycle-based format masks file. A binary pattern file cannot be masked with the cycle-based format masks file.

You can also create the masks file by collecting mismatches that occur during simulation, in serial or parallel mode, of STIL patterns. See “Predefined Verilog Options” in the *Test Pattern Validation User Guide* for information on the `+tmax_diag` option that controls this process.

Running the Flow

The flow consists of first reading the patterns in the external buffer along with the masks file. This read step will perform the masking of the patterns. You can then write the updated patterns so you can use them. Finally, you can optionally calculate the new test coverage with the masked cycles. It is possible to update binary, WGL or serial-STIL patterns with failures from the parallel simulation of STIL patterns; and then, to write the parallel STIL masked patterns for simulation.

To read the patterns in the external buffer and read in the masks file, use the following `set_patterns` command:

```
set_patterns -external patterns_file -resolve_differences masks_file
```

For example, the following command reads in the `pat.stil` patterns file and the `mask.txt` masks file, and creates a report that indicates the total number of X measures added in the external patterns:

```
set_patterns -external pat.stil -resolve_differences mask.txt
End parsing STIL file pat.stil with 0 errors.
End reading 200 patterns, CPU_time = 33.40 sec, Memory = 5MB
6 X measures were added in the external patterns.
```

Next, use the `write_patterns -external` command to write out the new vectors stored in the external patterns buffer. Then, if you want to calculate the new test coverage, it is recommended that you fault simulate the new patterns with `run_fault_sim`.

The flow is shown in the following example:

```
TEST-T> set_patterns -external pat.stil.gz -resolve_differences
mask.txt
TEST-T> write_patterns pat.masked.stil.gz -format STIL \
    -compress gzip -external
TEST-T> run_fault_sim
```

An alternate method for fault simulating the patterns and saving them so they can run on the tester is to use first `run_atpg -resolve_differences` and then `write_patterns`. In this case, the difference with previous method is that the `run_atpg -resolve_differences` command fault grades the external patterns with the added masks, and, patterns that don't contribute to the test coverage are removed.

The advantage of using the alternate method is that if a large number of failures are used during per-cycle pattern masking, it is likely that many patterns run on the tester are useless and thus removing them will reduce the test time. The drawback is that new failures could appear because of the patterns suppression. This is why it is recommended that you perform a check with the `run_simulation` command after `run_atpg -resolve`. If new failures occur, you must mask the patterns another time using `set_patterns -resolve_differences`.

An alternate flow is shown in the following example:

```
TEST-T> set_patterns -external pat.stil.gz -resolv_differences
mask.txt
TEST-T> add_faults -all
TEST-T> run_atpg -resolve_differences
TEST-T> run_simulation
TEST-T> write_patterns pat.masked.stil.gz -format STIL -compress
gzip \
    -external
```

You can also use this feature when the patterns are split after ATPG. In this case, make sure you generate the failures used for masking by using the log of the cycle-based failures from the tester. Also, the cycle count must be reset from the execution of a particular split pattern set to the next split pattern set. The flow is shown in following example:

```
set_patterns -external <pattern_filename_0_to_mask> -resolve
<failures_reset_0>
write_patterns ...
set_patterns -delete
```

```
set_patterns -external <pattern_filename_1_to_mask> -resolve  
<failures_reset_1>  
write_patterns ...  
set_patterns -delete  
etc. ...
```

Note: You should specify the `set_diagnosis -cycle_offset` command when using a cycle-based failures log file for masking and an offset is applied to the cycle.

Limitations

The following limitations apply to this flow:

- It is not possible to write masked full-sequential patterns in parallel format.
- The binary pattern file cannot be masked with a cycle-based format masks file.
- For patterns with multiple load-unloads with measures on the scanout in each unload, only the failures for the first unload can be masked.

15

On-Chip Clocking Support

On-Chip Clocking (OCC) support is common to all scan ATPG and Adaptive Scan environments. This implementation is intended for designs that require ATPG in the presence of PLL and clock controller circuitry.

OCC support includes phase-locked loops, clock shapers, clock dividers and multipliers, etc. In the scan-ATPG environment, scan chain load and unload are controlled through an ATE clock. However, internal clock signals that reach state elements during capture are PLL-related.

The following sections describe on-chip clocking support:

- [OCC Background](#)
- [OCC Definitions, Supported Flows, Supported Patterns](#)
- [OCC Limitations](#)
- [DFT Compiler to TetraMAX Flow](#)
- [OCC Support in TetraMAX](#)
- [OCC-Specific DRC Rules](#)

OCC Background

At-speed testing for deep submicron defects requires not only more complex fault models for ATPG and fault simulation, like transition faults and path delay faults, but also requires the accurate application of two high-speed clock pulses to apply the tests for these fault models. The time delay between these two clock pulses, referred to as the launch clock and the capture clock, is the effective cycle time at which the circuit is tested.

A key benefit of scan-based at-speed testing is that only the launch clock and capture clock need to operate at the full frequency of the device under test. Scan shift clocks and shift data might operate at much slower speed, thus reducing the performance requirements of the test equipment. However, complex designs often have many different high frequency clock domains, and the requirement to deliver a precise launch and capture clock for each of these from the tester can add significant or prohibitive cost on the test equipment. Furthermore, special tuning is often required to properly control the clock skew to the device under test.

One common alternative for at-speed testing is to leverage existing on-chip clock generation circuitry. This approach uses the active controller, rather than off-chip clocks from the tester, to generate the high speed launch and capture clock pulses. This type of approach generally reduces tester requirements and cost, and can also provide high speed clock pulses from the same source as the device in its normal operating mode without additional skews from the test equipment or test fixtures.

To use this approach, additional on-chip controller circuitry is included to control the on-chip clocks in test mode. The on-chip clock control is then verified, and at-speed test patterns are generated which apply clocks through proper control sequences to the on-chip clock circuitry and test mode controls. DFT Compiler and TetraMAX ATPG support a comprehensive set of features to ensure that:

- The test mode control logic for the OCC operates correctly and has been connected properly.
- Test mode clocks from the OCC circuitry can be efficiently used by TetraMAX ATPG for at-speed test generation.
- OCC circuitry can operate asynchronously to shift and other clocks from the tester.

OCC Definitions, Supported Flows, Supported Patterns

Note the following definitions as they apply to OCC:

- **Reference Clocks** — The frequency reference to the PLL. It must be maintained as a constantly pulsing and free-running oscillator or the circuitry will lose synchronization.
- **PLL Clocks** — The output of the PLL. A free-running source that also runs at a constant frequency which might not be the same as the reference clock.

- **ATE Clocks** — Shifts the scan chain typically slower than a reference clock. You must manually add this signal (a port) when inserting the OCC. Note that the ATE clock cannot be a reference clock, and it does not capture.
- **Internal Clocks** — The OCC is responsible for gating and selecting the PLL clocks and ATE clocks, and for creating the internal clocks, which satisfy ATPG requirements.
- **External Clocks** — The primary inputs of a design which clock flip-flops directly through combinational logic not generated from PLLs.

OCC is supported in the following flows:

- DFT Compiler-to-TetraMAX flow (for details, see Chapter 7, “Using On-Chip Clocking,” in the *DFT Compiler User Guide Vol. 1: Scan*)
- Non-DFT Compiler to TetraMAX Flows:
 - Basic Scan with On-Chip Clocking
 - Adaptive Scan with On-Chip Clocking

Note the following pattern support available in OCC:

Format	Synchronous Single Pulse	Synchronous Multi-Pulse	Asynchronous
STIL	Yes	Yes	Yes
STIL99	Yes	Yes	No
WGL	Yes	Yes	No
Others	Yes	No	No

OCC Limitations

Note the following limitations for OCC support:

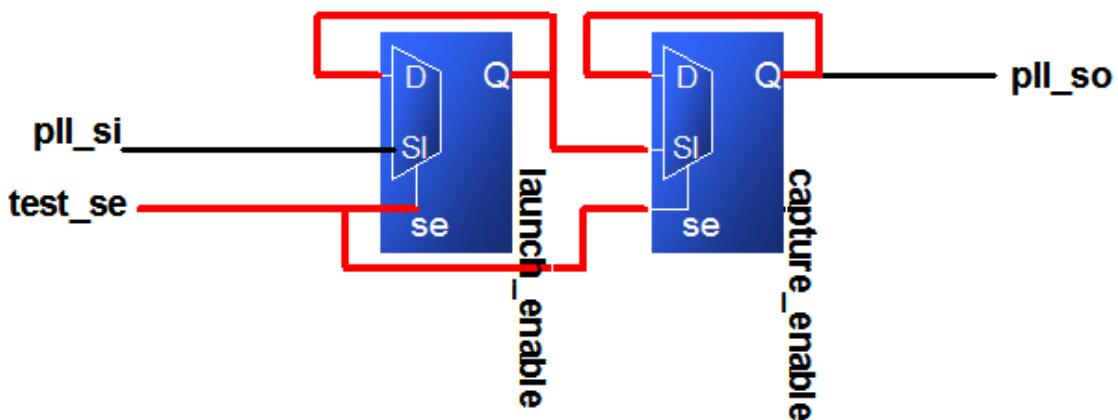
- You must use generic capture procedures for internal/external clocking. For more information, see “[Creating Generic Capture Procedures](#).”
- You cannot use the OCC from DFT Compiler with the `set_delay -launch_cycle last_shift` command. However, you can use it with the `set_delay -launch_cycle extra_shift` command if it is used in combination with pipelined scan enable. In this case, the `scan_en` pin must be connected to the non-pipelined scan enable input.
- Multi-cycle paths can only be tested when they are defined in a `MultiCyclePath` block for synchronized multi frequency clocking. You must also specify the `set_drc -multiframe_paths` command.
- The clock frequency of the PLL generating internal clocks cannot change dynamically — must be constant (i.e., programmable bits must be nonscan and constant during ATPG).
- Do not use the reference clock as your ATE clock or shift clock.
- End-of-cycle measure is not compatible with PLL reference clocks. With PLL reference clocks defined, ATPG can generate patterns with the following sequence of events:

- forcePI
- measurePO
- pulse reference clocks

When writing such patterns out in STIL (or any other external format), the vector that contains the measurePO must also pulse reference clocks (by definition reference clocks must be pulsed in every vector). But the end-of-cycle measure timing means the order of events is reversed in this vector: pulse reference clocks measurePO. This is incorrect and the pattern will likely fail on silicon. A new message has been added that will flag you to correct the timing:

Warning: Reference Clock <ON_time> < measure_time> in waveformtable. All PO measures were masked. (M664)

- Clock bits must hold state during capture.



- Avoid using reference clock for flip-flops inside the design.
- Programmable PLLs (test_setup is critical and must not become corrupt during the entire ATPG process).

```
MacroDefs {
  "test_setup" {
    W "_default_WFT_";
    C {
      "all_inputs" = \r26 N;
      "all_outputs" = \r8 X;
    }
    V {
      "ateclk" = P;
      "clk" = P;
      "pll_reset" = 1;
    }
    V {
      "test_mode" = 1;
      "pll_bypass" = 0;
      "pll_reset" = 0;
      "test_se" = 0;
    }
  }
}
```

```
    }
}
}
```

The `pll_reset` must be constrained to stay in a consistent state while shifting data from the clock chain. The OCC Controller goes through the initialization sequence one time and returns to a state to be controlled from the clock chain only. Therefore, the `pll_reset` must be constrained to stay in a consistent state.

- If the reference clock period is an integer divisor of the `test_default_period`, then patterns can be written in the STIL, STIL99 and WGL formats.
- If the reference clock is not an integer divisor to the `test_default_period`, the only format that can be written in a completely correct way is STIL. Other formats (including STIL99) cannot include the reference clock pulses and a warning is printed indicating that these pulses must be added back to the patterns manually.
- Make sure you constrain the scan enable to the off-state in the TetraMAX command file since it is not specified in the OCC protocol file.
- The `tmax2pt.tcl` script supports OCC. However, since there is no timing information for internal clocks in the TetraMAX database, the timing that is written out is nominal and might not match the design's actual clock timing.

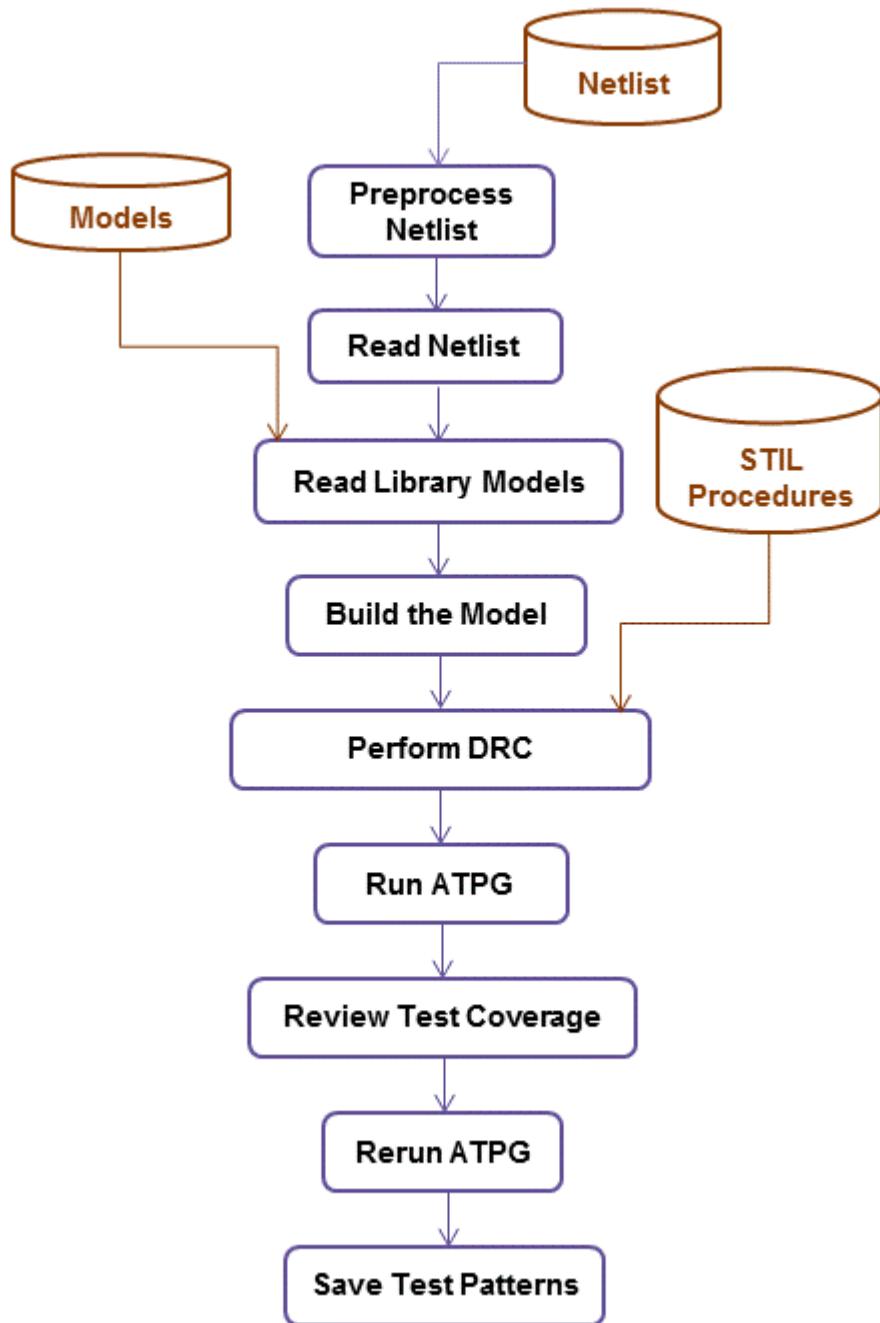
DFT Compiler to TetraMAX Flow

This flow automatically writes out the STIL procedure file for TetraMAX ATPG and the Verilog netlist.

For details on this flow, refer to “Using On-Chip Clocking,” in the *DFT Compiler User Guide* Vol. 1: Scan).

[Figure 1](#) illustrates the basic DFT Compiler to TetraMAX ATPG design flow.

Figure 1: DFT Compiler to TetraMAX ATPG Flow



The basic DFT Compiler-to-TetraMAX ATPG design flow consists of the following steps:

1. Edit your netlist to meet the requirements of TetraMAX ATPG (see "[Netlist Requirements](#)").
2. Read the netlist (see "[Reading in the Netlist](#)").
3. Read the library models (see "[Reading Library Modules](#)")

4. Build the ATPG design model (see "[Building the ATPG Model](#)")
5. Read in the STIL test protocol file, automatically generated by DFT Compiler (see "[Selecting the Pattern Source](#)").
6. Perform DRC and make any necessary corrections (see "[Performing Test Design Rule Checking](#)").

To run with PLL active, specify the following command:

```
run_drc <STIL_file> -patternexec <test_mode>
```

To run with PLL bypassed, specify the following command:

```
run_drc <STIL_file> -patternexec <test_mode>_occ_bypass
```

When using default test modes, use one of the following:

```
run_drc <scan_STIL_file> -patternexec Internal_scan
run_drc <scan_STIL_file> -patternexec Internal_scan_occ_bypass
run_drc <compression_STIL_file> -patternexec ScanCompression_mode
run_drc <compression_STIL_file> -patternexec ScanCompression_mode_occ_bypass
```

7. Prepare the design for ATPG by setting up the fault list, and setting the ATPG options (see "[Preparing for ATPG](#)").

Depending on the ratio between the `_default_WFT_` and the OCC clocks, the `set_atpg -min_ateclock_cycles` command might be needed.

The capture sequence for OCC clocks uses the `multiclock_capture` procedure (if generic capture procedures are used). There are as many of these as the number of launch and capture clocks required. The Synopsys OCC controller requires an ATE clock falling edge to occur after the scan enable has become inactive to start its count, then emits its first clock to correspond with the sixth following clock coming from the PLL. If the scan enable becomes active again before all of the pulses required from the OCC controller are emitted, then the capture pulses are truncated and the patterns will fail simulation.

When the ratio of the slowest PLL clock period to the ATE clock period is not high enough to ensure that all OCC clock pulses are emitted, the `set_atpg -min_ateclock_cycles` command should be used to add to the number of ATE clock cycles.

8. Run ATPG (see "[Running ATPG](#)").
9. Review the test coverage and rerun ATPG if necessary (see "[Reviewing Test Coverage](#)").
10. Save the test patterns and fault list (see "[Writing ATPG Patterns](#)").

Note: You should not use any OCC IP that is not created by DFT Compiler with TetraMAX ATPG. If you have this type of IP, you should refer to the “User-Defined Instantiated Clock Controller and Chain Insertion Flow” section in the *DFT Compiler User Guide*.

OCC Support in TetraMAX

OCC support in TetraMAX ATPG provides for automated handling of internal clocks in a generic manner. This automation is enforced by using clock design rules that validate user-specified clock controller settings.

The following sections describe OCC support in TetraMAX ATPG:

- [Design Set Up](#)
 - [OCC Scan ATPG Flow](#)
 - [Waveform and Capture Cycle Example](#)
 - [Using Synchronized Multi Frequency Internal Clocks](#)
 - [Using Internal Clocking Procedures](#)
-

Design Set Up

When a design contains both internal clocks (commonly driven by PLL sources), and external (primary input) clocks, the TetraMAX ATPG default operation is to use both clock sources for test generation. In some clock-tracing situations, internal clocks will take precedence over external sources, however this might not eliminate all ambiguity, especially when both clock sources are presented to the same internal element.

TetraMAX ATPG allows for control of capture clocks that are issued during ATPG on a per-pattern basis. This gives ATPG the flexibility of deciding what internal clocks that should be pulsed in a given capture cycle, instead of incurring the overhead of pulsing all internal clocks every capture cycle. Note that generic capture procedures should be used exclusively. Also, because the pulse placements of different OCC clocks cannot be predicted, you should always use the following command:

```
set_delay -common_launch_capture_clock
```

Note: If you are using synchronous multi frequency internal clocks, you should not use this example. Instead, TetraMAX ATPG offers a specific flow for designs that use synchronous multi frequency internal clocks. For details on this process, see "[Using Synchronized Multi Frequency Internal Clocks](#)." However, if your design contains asynchronous internal clocks, then you should use the example cited above.

Black boxes are often the sources of the PLL clocks. When PLL clocks are driven by logic, DRC might fail because of how these clocks are simulated. Simulation events are driven on the defined PLL clock source, and these events are used to trace the OCC controller functionality. Other simulation events that propagate through the logic confuse this DRC analysis. To prevent this problem, you should replace the instances driving the PLL clocks with TIEX primitives:

```
set_build -instance_modify {pl1864/U93 TIEX}  
set_build -instance_modify {pl1923/U45 TIEX}
```

OCC Scan ATPG Flow

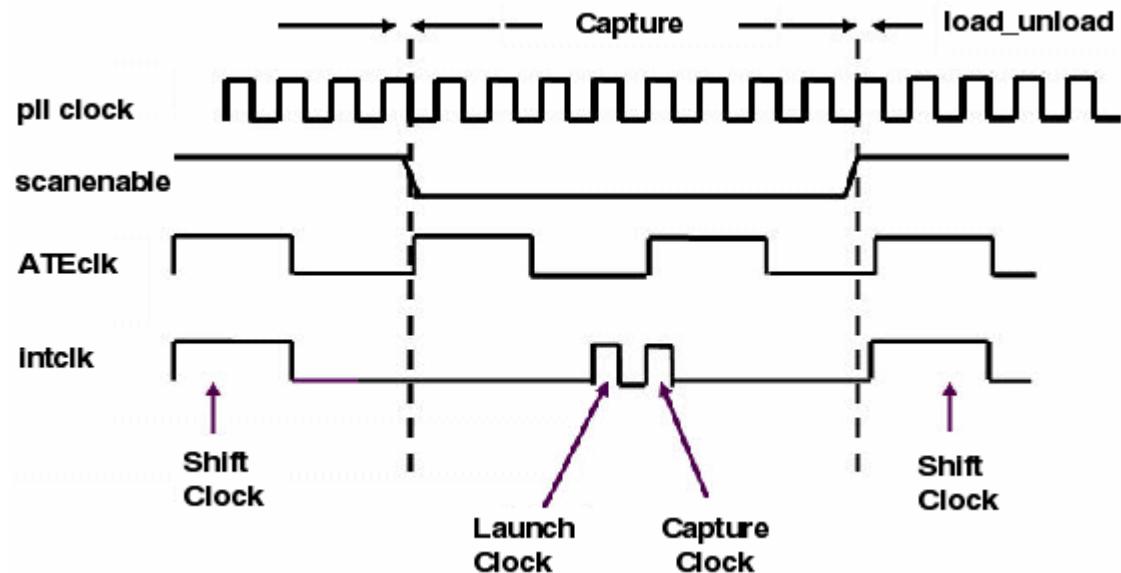
The OCC Scan ATPG flow consists of the following steps:

1. Read the design files (see "[Reading the Library Modules](#)").
 2. Build the design (see "[Building the ATPG Model](#)").
 3. Run DRC with the TetraMAX STIL procedure file created by DFT Compiler after scan insertion in presence of PLL circuitry (see "[Performing Design Rule Checking](#)").
 4. Run ATPG (see "[Running ATPG](#)").
-

Waveform and Capture Cycle Example

[Figure 1](#) shows an example of the relationship between various clocks when the design contains an OCC controller.

Figure 1: Waveform and Capture Cycle Example



Note in [Figure 1](#) that the `refclk` must pulse in every vector. This figure also contains information about `pllclk`, `ateclk`, and `intclk`.

Using Synchronized Multi Frequency Internal Clocks

By default, internal clocks derived from an OCC Controller are considered by TetraMAX ATPG to be asynchronous to each other. However, you can specify the timing relationships of internal clocks, thus improving the test quality. This section describes the process for implementing synchronized internal clocks at one or multiple frequencies in an OCC Controller.

It is important to note that this capability requires the PLL clocks to be synchronized in the design and requires the OCC Controllers to actually synchronize their output pulses. TetraMAX ATPG

uses the information provided to it and does not do any checking to ensure that this reflects the actual circuit design.

The following sections describe how to specify synchronized multi frequency internal clocks:

- [Enabling Internal Clock Synchronization](#)
- [Clock Chain Reordering](#)
- [Clock Chain Resequencing](#)
- [Finding Clock Chain Bit Requirements](#)
- [Reporting Clocks](#)
- [Reporting Patterns](#)

Enabling Internal Clock Synchronization

To enable internal clock synchronization, specify the `ClockTiming` block in the STIL Procedure File . There are several command switches, described later in this section, that can be used one time this feature is enabled.

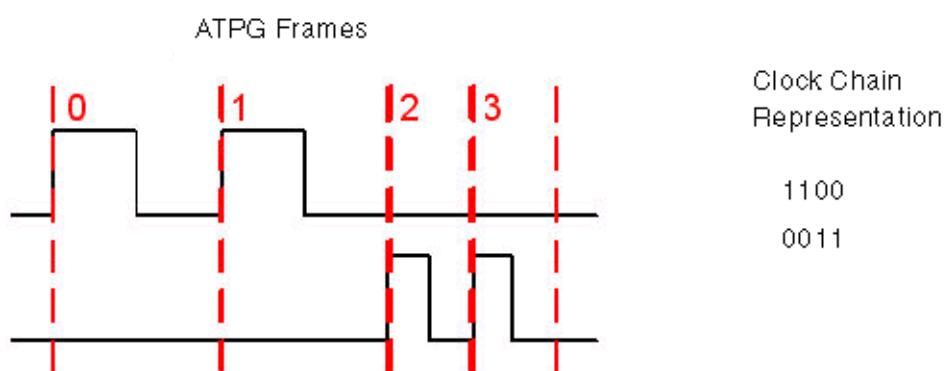
The `ClockTiming` block is placed in the top level of the `ClockStructures` block, which already describes other aspects of the internal clocks.

For details on how to enable internal clock synchronization in the STL procedure file, see the "[Specifying Synchronized Multi Frequency Internal Clocks for an OCC Controller](#)" section.

Clock Chain Reordering

The clock chain has one register bit per clock cycle. The value loaded into this register controls whether the OCC controller allows a clock pulse from the PLL to propagate during its cycle. ATPG calculates the pattern by ordering the clock pulses, and this initial order must be re-sequenced to reflect period and latency differences between the clocks. See [Figure 1](#).

Figure 1: Clock Chains Before Reordering



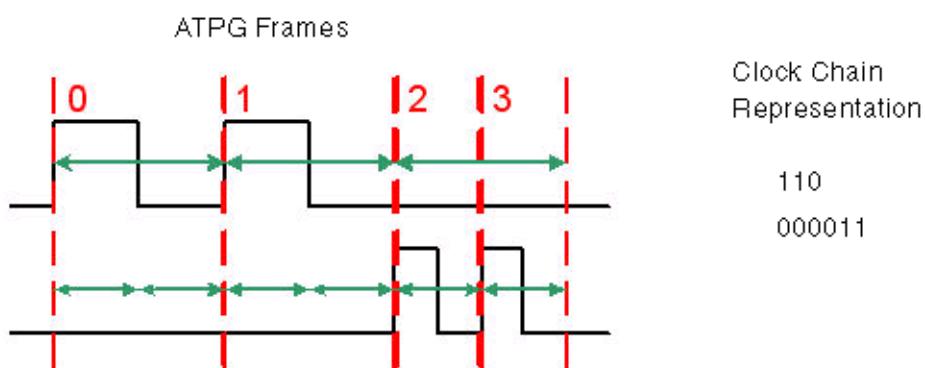
In [Figure 1](#), note that the order of the clock chain bits is the same as defined in the `Cycle` statements of the `PLLStructures` block, with ATPG frame 0 representing Cycle 0, and so forth.

Clock Chain Resequencing

By default, clock chain resequencing is done to convert the ATPG frame sequence to an equivalent duration in terms of clock periods. Since different clocks might have different periods, this might result in very different sequence lengths to cover the same capture time duration.

Latency is ignored when all clocks pulsed in a capture sequence are defined in the same `PLLStructures` block, or when the latency period (that is the latency number times the minimum clock period within the `PLLStructures` block) is the same even though the clocks are in different `PLLStructures` blocks. In this case, resequencing is based on period times and whether `MultiCyclePath` blocks are defined. See [Figure 2](#).

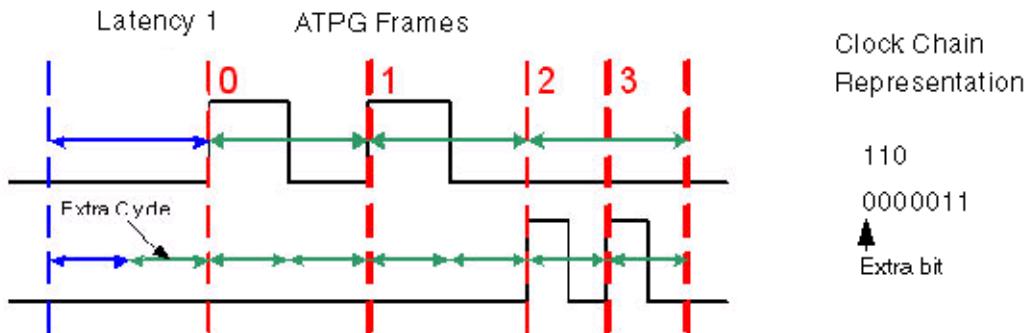
Figure 2: Clock Chain Resequencing with the Same Latency



Note that in [Figure 2](#), twice as many bits are needed to represent the 2X clock. For this reason, clock chains are allowed to have different lengths when a `ClockTiming` block is used.

Latency must be considered when a capture sequence contains clock pulses of clocks having different latency periods. In this case, extra padding cycles are added for the clock with the shorter latency period so that the clock periods coincide at the first ATPG frame. See [Figure 3](#).

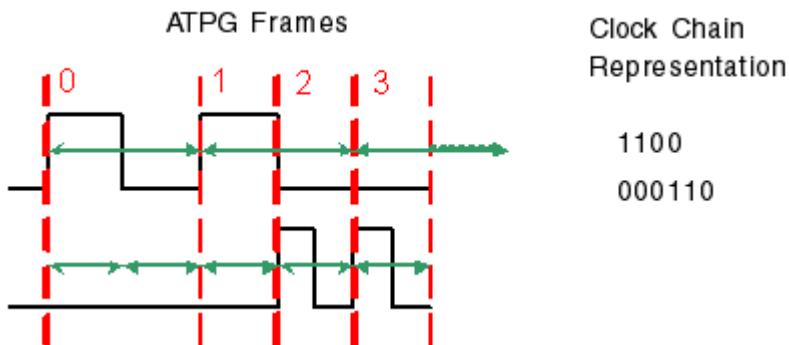
Figure 3: Clock Chain Resequencing with Different Latencies



Note that in [Figure 3](#), two clocks from different PLL structures blocks with the same latency number have different latency periods because of their different frequencies. This requires an extra padding bit to be added to its clock chain.

When clock overlapping is enabled, either by the `MultiCyclePath` statement in the STL procedure file or by the `set_drc -fast_multifrequency_capture` on command, clock chain re-sequencing is required to get the final result. For example, in the Latency 0 case, see [Figure 4](#).

Figure 4: Clock Chain Resequencing When Clock Overlapping is Enabled



Finding Clock Chain Bit Requirements

The required clock chain lengths can be calculated and combined with the number of internal clock pulses that are used, based on the `set_atpg -capture_cycles` specification.

You can also determine the clock chain bit requirements using the following command:

```
set_messages -level expert
```

After pattern generation, before the summary is printed, the following message will appear:

Warning: 238 clock pulses rejected. Clock 895 has a 6 bit clock chain, but needs 13 bits. (M720)

The clock number refers to the clock source, whose instance name can be found using the `report_primitives` command. The clock chain length reported is the maximum needed. If a M720 message is not printed, then the clock chains meet or exceed the required length.

Reporting Clocks

To report the structure of the synchronized clock groups as they are used by ATPG, use the command `report_clocks intclocks`. If any synchronization groups are active, two extra columns are printed with the headings sync and period. The example STL procedure file shown in “ClockTiming Block Example” uses `ClockTiming CTiming_2`, and looks like the following:

```
int_clock_inst_name gate_id off source sync period cycle
conditions
-----
--  

TOTO/U2 895 0 20 1 1 0 1468=1 (0,4)
... one line for each extra pulse condition ...
TOTO/U5 825 0 19 1 2 0 1487=1 (0,4)
... one line for each extra pulse condition ...
TOTO/U8 755 0 18 1 4 0 1506=1 (0,4)
... one line for each extra pulse condition ...
```

The `sync` heading indicates the synchronization group number. The number is arbitrary, but all internal clocks that are synchronized to each other are in the same synchronization group.

The `period` heading indicates the period of the clock in units of the fastest clock in the same synchronization group. They are normalized to 1 since the actual period is not used by ATPG, only the relationships between the different periods. Note that each synchronization group will have a clock with a period of one. This does not mean that their periods are the same, since the different groups are asynchronous to each other.

To get clock pulse overlapping information, use the `report_clocks -capture_matrix` command. The output from this command takes one of two forms. The default form is as follows:

```
report_clocks -capture_matrix
Warning: Requested report contained no entries. (M13)
```

This means that overlapping is not allowed between any clock pairs. This would be expected in the example STL procedure file (see “[ClockTiming Block Example](#)”) if `set_drc -internal_clock_timing CTiming_2` was used because of the lack of `Waveform` and `MultiCyclePath` statements. The non-default form is as follows:

```
report_clocks -capture_matrix
id# clock_gate period 0 1
-----
0 895 10.0 10.0 10.0
1 825 30.0 10.0 30.0
```

This means that clock pulse overlapping is allowed. All numbers in the matrix are the time between the launch and capture pulses when this pair of clocks is used. In this example, captures between any pairs of clocks can be made at the minimum of the two clocks’ periods, or in other words, at single-cycle timing.

The timing of the periods and edges of the internal clocks is reported by using the command `report_clocks -intclocks -verbose`. For example:

```
report_clocks -intclocks -verbose
#int_clk_inst_nm gt_id off source sync period LE TE lat cycle
conditions
#----- -----
#pll_control_M1/U2 6698 0 138 1 1 0 10 5      0 13337=1 (0,4)
# 1 13336=1 (0,5)
# 13 13324=1 (0,17)
#pll_control_M2/U2 7347 0 139 1 1 0 10 5 0 13362=1 (0,4)
# 1       13361=1 (0,5)
#
# 13349=1 (0,17)
#pll_control_M3/U2 8567 0 187 1 2 5 25 5      0 13387=1 (0,4)
#
# 13386=1 (0,5)
# 13       13374=1 (0,17)
```

Note that the leading/trailing edge information comes from the STL procedure file. Here is the block that produced the example report:

```
SynchronizedClocks M_clocks {
    Clock ICLK1 {Location "pll_controller_M1/U2/Y"; Period
'20ns';
    Waveform '0ns' '10ns';
}
    Clock ICLK2 {Location "pll_controller_M2/U2/Y"; Period
'20ns';
    Waveform '0ns' '10ns';
}
    Clock ICLK3 {Location "pll_controller_M3/U2/Y"; Period
'40ns';
    Waveform '5ns' '25ns';
}
```

Reporting Patterns

The `report_patterns` command is useful for finding out the intention of ATPG, but the report can be too verbose when only the clocking information is required. To get a report that is tightly focused on the clocking, use the command `report_patterns -clocking`. For example:

```
TEST-T> report_patterns 7 -clocking
Clocking only:
Pattern 7 (fast_sequential-parallel_clocking)
Cycle-based clocking sequence:
0: TOTO/U2/Z:0100000000
1: TOTO/U5/Z:1-0-0-0-0-
Clock Instruction Registers:
```

```

0: 0010000000
1: 1000000000
# PLL internal clock pulse: capture_cycle=0, node=TOTO/U5 (191)
# PLL internal clock pulse: capture_cycle=1, node=TOTO/U2 (242)

```

The cycle-based clocking sequence field is the test in terms of ATPG frames and the `Clock` `Instruction Registers` field is the clock chain contents after re-sequencing. A dash is inserted to indicate that the clock operation is determined by a previous value and its period has not finished yet. It allows columns representing the same time to line up even though they refer to clocks of different periods.

Using Internal Clocking Procedures

Internal clocking procedures enable you to specify which combinations of internal clock pulses you want to use and how to generate them.

The following sections describe how to use internal clocking procedures in TetraMAX ATPG:

- [Enabling Internal Clocking Procedures](#)
- [Performing DRC with Internal Clocking Procedures](#)
- [Reporting Clocks](#)
- [Performing ATPG with Internal Clocking Procedures](#)
- [Grouping Patterns By ClockingProcedure Blocks](#)
- [Writing Patterns Grouped by Clocking Procedure](#)
- [Reporting Patterns](#)
- [Limitations](#)

Enabling Internal Clocking Procedures

To enable internal clocking procedures, you can use either the `ClockTiming` block or the `ClockConstraints` block within the top level of the `ClockStructures` block.

The `ClockTiming` block is used for synchronized multi frequency clocks. In many cases, you can use either the `ClockTiming` block or `ClockConstraints` block to describe synchronized OCC controllers. However, you should first consider using the `ClockTiming` block because it provides greater freedom to ATPG and results in fewer patterns for the same coverage. You should use the `ClockConstraints` block when the synchronized OCC controllers are limited to providing a small fixed set of clock waveforms.

Note that you cannot combine the `ClockConstraints` and `ClockTiming` blocks.

For complete details on enabling internal clocking procedures in the STL procedure file, see the "[Specifying Internal Clocking Procedures](#)" section.

Performing DRC with Internal Clocking Procedures

The presence of the `ClockConstraints` block in the STL procedure file disables some of the on-chip clocking checks normally performed during DRC. In particular, no checking is done to ensure that the specified `InstructionRegister` values cause the required clock pulses to be generated. In this case, the intention is to support clock controllers whose behavior cannot be understood through zero-delay gate-level simulation. Clock effects from the defined clock pin

name are simulated to ensure that capture behavior is valid. Clock-grouping checks are not performed.

You can define more than one named `ClockConstraints` block, but you can use only one for any single DRC or ATPG run. You must select the required `ClockConstraints` block using the `set_drc -clock_constraints` command, as shown in the following example:

```
set_drc -clock_constraints constraints1
```

If you do not specify the `set_drc -clock_constraints` command, none of the `ClockConstraints` blocks is used.

Timing information is not provided, which means clocks are assumed to be in the order specified. All clocks that pulse in the same frame are assumed to pulse simultaneously without disturbing each other. The trailing edges of all clock pulses in the first frame are assumed to occur before the leading edges of the clocks in the second frame. If these assumptions are violated in the actual design, timing exceptions must be used to prevent simulation mismatches.

You can use the `set_drc -num_pll_cycles` command to specify the sequential depth of the constraints. Procedures with a small number of frames are padded with clock-off values. Procedures with a large number of frames are degenerated if all of the extra frames are at clock-off values; otherwise, they are unusable. This enables the definition of multiple constraints of different depth in a single `Constraints` block while ensuring that only the procedures of the appropriate depth are used. The `set_drc -num_pll_cycles` and `set_atpg -capture` commands must match, but they can differ from the `PLLcycles` declaration in the `ClockStructures` block. The commands specify the sequential depth to be used in this particular run, while the `PLLcycles` declaration indicates the maximum sequential depth supported by the clock controller.

Reporting Clocks

You can use the `-constraints` option of the `report_clocks` command to report information on clocking procedures as they are used by ATPG. To report details for a given procedure, use the `report_clocks -constraints -procedure name` command. To report more detail for all procedures, use the `report_clocks -constraints -all` command.

For example,

```
TEST> report_clocks -constraints -all
-----
Clock Constraints constraints1:
  Maximum sequential depth: 2
  Defined Clocking Procedures: 3
  Usable Clocking Procedures: 3
  PLL clocks off Procedure: ClockOff

U0to1:
  CLKIR=10010
  dutm/ctrl1/U17/Z=P0
  dutm/ctrl2/U19/Z=0P
-----
U1to0:
  CLKIR=01010
```

```

dutm/ctrl11/U17/z=0P
dutm/ctrl12/U19/z=P0
-----
ClockOff:
  CLKIR=00000
  dutm/ctrl11/U17/z=00
  dutm/ctrl12/U19/z=00
-----

```

Note: When procedures with different frame counts are reported, the shorter procedures are shown with zeros padded to the left so that all procedures are reported with the same depth. This does not mean that the procedures should be written this way. ATPG is more efficient when all procedures are written with as few frames as possible.

Performing ATPG with Internal Clocking Procedures

The internal clocking procedures feature fully supports two-clock optimized ATPG, basic scan ATPG, and fast-sequential ATPG. Full-sequential ATPG is not supported and no patterns are generated when internal clocking procedures are defined.

When two-clock optimized ATPG is used, all usable clocking procedures must have two frames for each clock. When basic scan ATPG is used, all usable clocking procedures must have one frame for each clock.

As a result of using internal clocking procedures, ATPG can use only a subset of the available clock pulse sequences. The sequences cannot be used to force ATPG to generate a pattern that it could not otherwise generate.

When a procedure has multiple clocks and multiple frames, ATPG can only capture transition or fault effects using clocks that pulse in the last frame. Clocks whose last pulse is in a preceding frame can only be used to launch transitions or set up conditioning to detect faults captured by other clocks. Make sure you provide other procedures where these clocks pulse in the last frame; otherwise, fault coverage is reduced.

Grouping Patterns By ClockingProcedure Blocks

In some situations, you might want to group patterns into sets, each of which uses only one of the defined `ClockingProcedure` blocks. To group patterns, specify the following command before the `run_atpg` command:

```
set_atpg -group_clk_constraints { first_pass middle_pass final_pass }
```

The three arguments are specified in terms of percentages of the fault list. These numbers are specified just one time, but they are applied for each individual clocking procedure. ATPG categorizes each fault by the clocking procedures that can test it; it considers only the appropriate subset as it generates tests for each clocking procedure.

The `first_pass` specification is the percentage of the fault list that is targeted in the first pass through each clocking procedure. The first pass results in long blocks of patterns with just one clocking procedure.

The `middle_pass` specification is the percentage of the fault list that is targeted by subsequent passes through each clocking procedure. These passes are repeated until the `final_`

pass number is reached. The middle passes result in shorter blocks of patterns with just one clocking procedure.

The *final_pass* specification is the percentage of the fault list targeted by the final pass in which any clocking procedure is used. In this pass, there is no guarantee that any two consecutive patterns share the same clocking procedure.

Forcing a Single Group Per Clocking Procedure

The following example forces a single group for each clocking procedure with no exceptions:

```
set_atpg -group_clk_constraints { 100 0 0 }
```

The drawback of this particular specification is that ATPG efficiency, both in runtime and in fault detections per pattern, decreases significantly after most of the fault list has been targeted. All faults that are detectable by the first clocking procedure must be targeted before moving on to the next clocking procedure, which results in a larger pattern count than if other arguments are chosen.

Enabling ATPG to Achieve Better Efficiency

You can define a set of numbers that allow ATPG to achieve better efficiency and results in a lower overall pattern count, as shown in the following example:

```
set_atpg -group_clk_constraints { 85 5 2 }
```

This command creates a set of pattern groups by clocking procedure:

```
ClockingProcedure_1 (0-85%)
ClockingProcedure_2 (0-85%)
.....
ClockingProcedure_N (0-85%)
ClockingProcedure_1 (85-90%)
ClockingProcedure_2 (85-90%)
.....
ClockingProcedure_N (85-90%)
ClockingProcedure_1 (90-95%)
ClockingProcedure_2 (90-95%)
.....
ClockingProcedure_N (90-95%)
ClockingProcedure_1 (95-98%)
ClockingProcedure_2 (95-98%)
.....
ClockingProcedure_N (95-98%)
Mixed ClockingProcedure's (98-100%)
```

Note: The drawback to this approach is that the grouping is less strict.

Writing Patterns Grouped by Clocking Procedure

By default, the `write_patterns` command saves all patterns into a single pattern file. You can use the `write_patterns -occ_load_split` command to split patterns into a separate file for each clocking procedure. This command is compatible with all pattern formats.

When patterns are grouped using the command `set_atpg -group_clk_constraints { 100 0 0 }`, only one pattern file is saved for each clocking procedure. If clocking procedures are grouped less strictly, or are not grouped at all, more pattern files are saved. A new pattern file is saved each time the clocking procedure changes from one pattern to the next, which can result in a large number of pattern files. Because of this, you should use the `write_patterns -occ_load_split` command only in combination with the `set_atpg -group_clk_constraints` command.

Reporting Patterns

You can use the `report_patterns -clocking` command to find out which clocking procedure is used in each capture cycle. For example,

```
TEST> report_patterns 7 -clocking
Clocking only:
Pattern 7 (fast_sequential)
Clocking Procedures: U0to1
// PLL internal clock pulse: capture_cycle=0, node=dutm/ctrl11/U17
(64)
// PLL internal clock pulse: capture_cycle=1, node=dutm/ctrl12/U19
(94)
```

To get a summary of the number of clocking procedures of each type that was used in the pattern set, specify the `report_patterns -clk_summary` command:

```
TEST> report_patterns -all -clk_summary
Pattern Clocking Constraints Summary Report
-----
#Used Clocking Procedures
#U0to1 6
#U1to0 5
-----
```

Limitations

The following limitations apply when using internal clocking procedures in TetraMAX ATPG:

- TetraMAX DRC does not perform checking to ensure that the specified `InstructionRegister` values cause the generation of the required clock pulses.
- TetraMAX DRC does not perform clock-grouping checks, and accepts all clock pulses specified in the same frame as simultaneous pulses without disturbing each other.
- TetraMAX ATPG assumes that the trailing edges of all clock pulses in one frame occur before the leading edges of the clocks in the next frame. It is not possible to specify overlapping clock pulses.
- Full-sequential ATPG is not supported because it can generate bad patterns.
- When a procedure has multiple clocks and multiple frames, TetraMAX ATPG can only capture transition or fault effects using clocks that pulse in the last frame. Clocks with a last pulse in a preceding frame can only be used to launch transitions or set up conditioning to detect faults captured by other clocks.
- Only single-load patterns are supported. You do not need to explicitly disable the

generation of multi load patterns because TetraMAX ATPG will not attempt to generate them.

- The grouping performed by the `-group_clk_constraints` option of the `set_atpg` command does not apply to fast sequential patterns. It applies to two-clock-optimized transition delay patterns and basic scan patterns for stuck-at.

See Also

[Specifying Internal Clocking Procedures](#)

OCC-Specific DRC Rules

Test DRC involves analysis of many aspects of the design. Among other things, DRC checks the following:

- C28 - Invalid PLL source for internal clock
- C29 - Undefined PLL source for internal clock
- C30 - Scan PLL conditioning affected by nonscancells
- C31 - Scan PLL conditioning not stable during capture
- C34 - Unsensitized path between PLL source and internal clock
- C35 - Multiple sensitizations between PLL source and internal clock
- C36 - Mistimed sensitizations between PLL source and internal clock
- C37 - Cannot satisfy all internal clocks off for all cycles
- C38 - Bad off-conditioning between PLL source and internal clock
- C39 - Nonlogical clock C connects to scancell
- C40 - Internal clock is restricted

Reference clocks are used only during design rule checking and are non-logical for pattern generation. PLL clocks are used during scan design rule checking (Category S – Scan Chain Rules) and clock design rule checking (Category C – Clock Rules). Pattern generation does not consider PLL clocks. Internal clocks are used for all capture operations, and normal clock rule checking is applied to these so that TetraMAX ATPG can perform these and other DRC checks, you must provide information about clock ports, scan chains, and other controls by means of a STIL test protocol file. The STIL file can be generated from DFT Compiler, or you can create one manually as described in “[STIL Procedure Files](#).”

16

Diagnosing Manufacturing Test Failures

The presence of defects in silicon has a direct impact on yield ramp during the manufacturing process. When a device fails testing, TetraMAX diagnostics can quickly isolate the cause and location of the failure, and simplify the analysis process.

The following sections describe the process for applying TetraMAX scan diagnostics to manufacturing test failures:

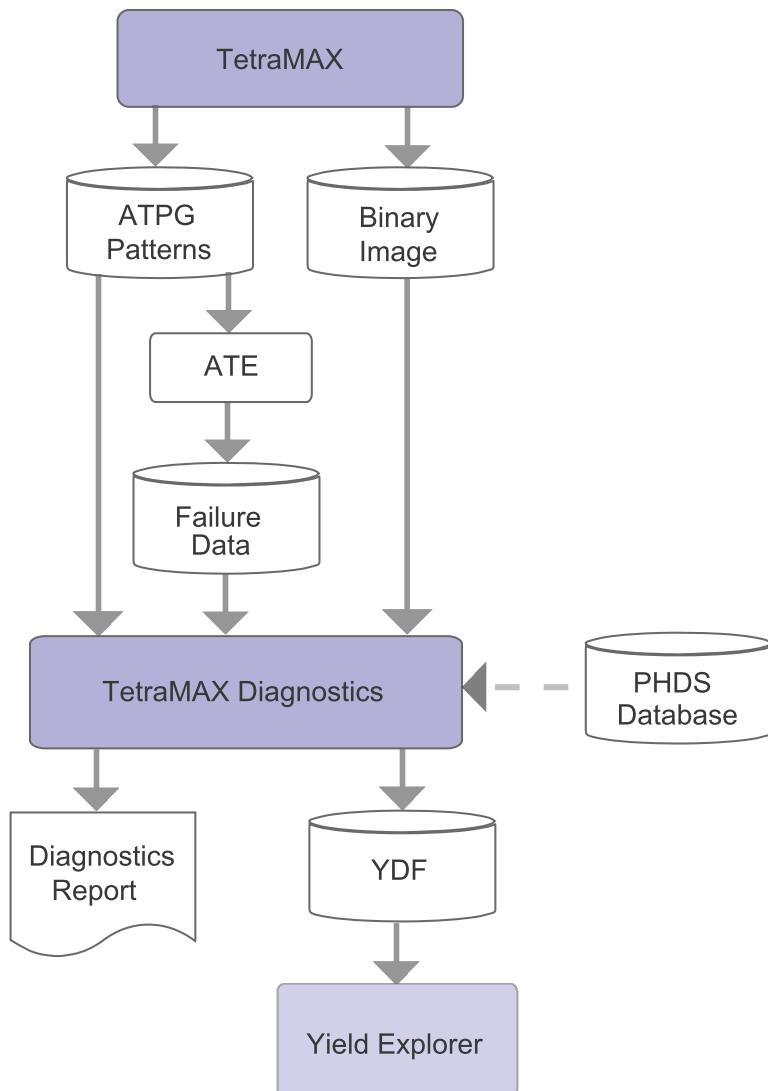
- [Diagnostics Flow Overview](#)
- [Running the Diagnostics Flow](#)
- [Writing and Reading Binary Images](#)
- [Reading Pattern Files](#)
- [Failure Data Files](#)
- [Class-Based Diagnostics Reporting](#)
- [Fault-Based Diagnostics Reporting](#)
- [Failure Mapping Report for DFTMAX Patterns](#)
- [Composite Fault Model Data Report](#)
- [Parallel Diagnostics](#)

Diagnostics Flow Overview

Diagnostics determine the root cause of failures observed during the testing of a chip. The data obtained from diagnostics identifies cells and scan chains with defects, and any logic with defects.

The following input files are required by TetraMAX diagnostics: a pattern set created by TetraMAX ATPG in basic-scan or fast-sequential mode, a failure data file from the ATE, and a binary design image file. Alternatively, you can use a netlist, library model, and STIL procedure file (SPF) as input in place of the image file. However, this method is more time-consuming because you have to rebuild the library model and rerun DRC.

Figure 1 TetraMAX Diagnostics Flow



Running the Diagnostics Flow

To run TetraMAX diagnostics:

1. Establish the original ATPG environment used for creating the patterns, including reading the design netlist, reading the library model, running DRC, and running ATPG. For details on preparing for and running ATPG, see the "[ATPG Design Flow](#)" section.
2. Write the image to a binary file using the `write_image` command, as shown in the following example:

```
write_image image.gz -compress gzip
```

This is an optional step. You can run diagnosis using the original netlist, library model, and rerun DRC. However, a binary image file contains all this information in single file and eliminates the need to rerun the ATPG process before each diagnosis run. For more information on binary image files, see the "[Writing and Reading Binary Image Files](#)" section.

3. Connect to the PHDS database (optional)

For more information on creating and validating a PHDS database, see the "[Using TetraMAX to create a PHDS Database](#)" section.

4. Read in the original patterns used to detect the failure. You can use patterns generated by the basic-scan and fast-sequential modes, but not the full-Sequential mode. For more information on using pattern files, see "[Reading Pattern Files](#)."
5. Obtain the failure data file produced from the ATE. For more information, see "[Failure Data Files](#)."
6. Use the `set_diagnosis` command to specify parameters for the diagnostics run. For example, you might want to [perform scan diagnostics](#) or [perform parallel diagnostics](#).
7. Read the failure data file and start diagnostics using the Run Diagnosis dialog box in the TetraMAX GUI or specify the `run_diagnosis` command at the command line.
8. Analyze the results in the diagnostics summary report.

TetraMAX ATPG determines the cause of the failing patterns and generates the diagnostics report. By default, TetraMAX diagnostics searches for defects in functional logic. If pattern 0 is the chain test pattern and it fails, then chain diagnostics is performed. For more information, see either the "[Class-Based Diagnostics Reporting](#)" section or the "[Fault-Based Diagnostics Reporting](#)" section."

Note the following:

- DFTMAX compression is supported for both logic and scan chain diagnosis. Serialized DFTMAX compression is also supported. Each individual failure is mapped to a scan cell. To produce a detailed failure mapping report, use the `-mapping_report` option of the `set_diagnosis` command. Always review the mapping report for accuracy.
- If a large set of failures cannot be mapped using DFTMAX patterns, you can create patterns that bypass the output compressor. For more information on this process, see

the ["Translating DFTMAX Patterns Into Normal Scan Patterns"](#) section.

- By default, a defect is not linked to the type of fault tested. This means, for example, that any failures collected while running transition patterns could include stuck-at faults. However, you can use the `-delay_type` option of the `set_diagnosis` command to cause the diagnostics report to include delay defects.

Writing and Reading Binary Image Files

A binary image offers many advantages when running diagnostics. It simplifies file management because you use only a single file that encapsulates the design netlist, library, and SPF. It is also faster to load, and can be password-protected. You can also garble the instance names in an image file, if necessary.

Prior to creating an image file, you need to read the design netlist and library model, run DRC, and run ATPG (see the ["ATPG Design Flow"](#) section for details). You then use the `write_image` command to write a binary image that contains the netlist, library, SPF, and DRC data. During the diagnostics process, you can use the `read_image` command to read the image file as many times as necessary without rerunning the ATPG flow.

The following example shows how to write and read a binary image file for diagnostics:

```
// First time through (ATPG flow)
BUILD> read_netlist top.v
BUILD> read_netlist spec_lib.v -library
BUILD> run_build_model spec_chip
DRC> run_drc spec_chip.spf
TEST> write_image spec_image.gz -compress gzip

// For subsequent runs during diagnosis
BUILD> read_image spec_image.gz
```

For more information on writing and reading secure binary images, see ["Binary Image Files."](#)

Reading Pattern Files

The diagnostics process requires either a single pattern file corresponding to the patterns that were run on the tester when the device failed or an entire set of split patterns files, including the associated failure files.

The binary pattern format works best in TetraMAX ATPG. STIL or WGL patterns also work, however if you use these patterns, the fast-sequential patterns might be interpreted as a full-sequential patterns, and errors are reported.

The following sections describe how to read and use pattern files for diagnostics:

- [Reading Patterns](#)
- [Reading Multiple Pattern Files](#)

- [Translating DFTMAX Scan Patterns Into Normal Scan Patterns](#)

See Also

[Writing ATPG Patterns](#)

Reading Patterns

TetraMAX diagnostics accepts either basic-scan or fast-sequential ATPG patterns. When reading patterns into TetraMAX ATPG, use binary formats whenever possible.

You can perform a sanity check to verify that the simulation passes with the patterns you read in by running the `run_simulation` command before performing diagnostics.

Use the `set_patterns` command to read a set of patterns, as shown in the following example:

```
set_patterns -external patterns.bin
```

For details on reading patterns, see "[Selecting the Pattern Source](#)."

See Also

[Using Split Datalogs to Perform Parallel Diagnostics for Split Patterns](#)

Reading Multiple Pattern Files

Some designs require the ATE to run multiple TetraMAX pattern files. Each pattern file is typically run individually in separate test programs on the tester. When a device fails, the tester generates one failure log file per TetraMAX pattern file.

TetraMAX diagnostics can read multiple pattern files and multiple failure data files so you can get a single result from a single diagnosis run. This is supported for DFTMAX compression.

There can be as many failure log files as there are pattern files. A failure log file is expected to contain the failures for only the patterns in the corresponding pattern file. Otherwise, an error is generated.

If there are no failures for any patterns in a particular pattern file, the corresponding failure log file might not exist. The correspondence between pattern files and failure log files is specified by a required directive in the failure log file, as explained in the "[Failure Data Files](#)" section. An error is generated otherwise.

To use multiple patterns files, specify the following `set_patterns` command:

```
set_patterns -external file -split_patterns
```

When split pattern files are read, you can specify multiple failure log files using the `run_diagnosis` command.

By default, TetraMAX diagnostics considers that the cycle count recorded in the failures file in cycle-based format is reset to 1 (or the recorded pattern count is reset to 0 for the pattern-based format) from the execution of one pattern set to the next set. You can use the `.first_pattern` directive to change this behavior if the failures are in the pattern-based format.

See Also

[Using Split Datalogs to Perform Parallel Diagnostics for Split Patterns](#)

Translating DFTMAX Compressed Patterns Into Normal Scan Patterns

If your design uses DFTMAX compression, you can perform diagnostics on the patterns that include compression, or create patterns that bypass the output compression. Diagnosing patterns in compressor mode could reduce diagnostic resolution due to compressor effects. After a device in compressor mode fails on the tester, if the diagnostic resolution is not high enough, you can retest it in scan mode. The translated patterns detect the same defects, but diagnostic resolution is higher because the compressor no longer affects the unloaded values.

The translation process involves writing out a special netlist-independent version of the pattern in binary format along with the DFTMAX pattern set. The netlist-independent pattern file contains a mapping of the scan cells and primary inputs to their ATPG generated values. This pattern set can be read back into TetraMAX ATPG after a design is put into the reconfigured scan mode by reading the scan mode STL procedure file. When the patterns are read back, an internal simulation is performed to compute the expected values to complete the translation process. The internal patterns can then be written out for the tester to use in scan mode for diagnostics.

Example Flow

To translate DFTMAX compressed patterns to normal scan patterns:

1. Read the design with DFTMAX in compressor mode and write out the netlist-independent pattern format.

```
run_build_model ...
# read STL procedure file for adaptive scan mode
run_drc scan_compression.spf
run_atpg -auto
# write out adaptive scan mode patterns
write_patterns compressed_pat.bin -format binary
# write_netlist independent patterns that can be translated
set_patterns -netlist_independent
write_patterns compressed_pat.net_ind.bin
```

2. Read the design with DFTMAX compression in scan mode. Translate the patterns into scan mode.

```
run_build_model ...
# read STL procedure file for normal scan mode
run_drc scan.spf
# read netlist independent patterns
set_patterns -external compressed_pat.net_ind.bin
# optional sanity check to verify that simulation passes
run_simulation
# write out translated patterns to be re-run on the tester
```

```
write_patterns scan_pat.pats -external -format <any format>
# write out translated patterns in binary format for
diagnostics
write_patterns scan_pat.bin -external -format binary
```

Translation Limitations

The following limitations apply when translating DFTMAX compression patterns to normal scan mode patterns:

- Translation is one-way. You cannot translate scan patterns to compression mode.
- Only basic-scan and fast-sequential patterns are supported .
- Configuration differences between compressor mode and scan mode might result in slightly different coverage numbers.

See Also

DFTMAX User Guide

Failure Data Files

A failure data (or log) file is an ASCII text file that provides the failure information from a device necessary to perform diagnostics. This file captures the test results of a failing device, including failing patterns and failing outputs and scan cells. Most ATE vendors automatically generate failure data files in a format recognizable by TetraMAX.

When testing a chip, if the value measured by the ATE is different than the expected value indicated in the patterns file, a failure is recorded in the failure data file. Each recorded failure includes the vector number, the output port where the mismatch occurs, the cycle number within the mismatched vector, and optional expected data.

TetraMAX supports either a pattern-based or cycle-based failure data file.

The following sections describe failure data files:

- [Pattern-Based Failure Data File](#)
 - [Cycle-Based Failure Data File](#)
 - [Failure Data File Extensions](#)
 - [Adding Header Information to a Failure Data File](#)
 - [Limitations](#)
-

Pattern-Based Failure Data File

Each line in a failure data file describes a pattern in which the output values detected by the test equipment did not match the expected values. An example is as follows:

```
// Pattern Output Cell
```

```

50 vout    55
50 abus    57
58 vout    57
82 xstrb
82 vout    57
82 vout    5
83 abus    90

```

The format of each line is as follows:

pattern_num *output_port* [*cell_position*] [*expected_data*]

Where:

pattern_num

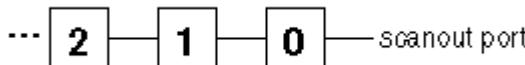
The TetraMAX pattern number in which the failure occurred, starting with 0 for the first pattern.

output_port

The name of the output port at which the failure was detected, or the scan chain name when the pin name is shared among scan groups.

cell_position

The cell position must be provided if the failure occurred during a scan shift cycle. The position is the number of tester shift cycles that have occurred since the start of the scanout process. From this value, TetraMAX ATPG determines the position of the scan chain cell that captured the erroneous data. The cell position of the scan chain cell closest to the output port is 0, the next one is 1, and so on; for example:



expected_data

This is an optional parameter that describes the expected value of the measured failure. The expected value is either 0 or 1. By default, if this parameter is present, it is checked against the expected data recorded in the patterns. If this step succeeds, it is a good indicator that the files used for diagnostics and on the tester are consistent. To change the default, use the `-nocheck_expected_data` option of the `set_diagnosis` command.

To specify the expected data on a primary output during a capture cycle, which is also used as a scan chain output, you must use the `exp=` syntax to avoid any ambiguity with this being a cell position. For example:

```

103 scan_out3 1 //invalid
103 scan_out3 (exp=1, got=0) //valid
103 scan_out3 exp=1 //valid

```

Any line in the failure data file that begins with two slash characters is considered a comment line.

The following example shows another tester data file. In this example, five failing patterns are reported: pattern numbers 3, 4, 10, 11, and 12.

```
// pattern 3, port REQRDYO
3 REQRDYO

// pattern 4, port MA[9], scan chain 'c9', 30 shifts
4 MA[9] 30

//pattern 10-12, port NRD, scan chain 'c29', 3 shifts
10 NRD 3
11 NRD 3
12 NRD 3
```

The `-failure_memory_limit` option of the `set_diagnosis` command helps ease the failure log file truncation task. This option enables you to specify the maximum number of failures that can be captured by the tester. It also enables TetraMAX ATPG to automatically truncate the patterns considered during diagnosis.

Pattern-Based Failure Data File for DFTMAX Serialized Adaptive Scan

The format for DFTMAX serialized adaptive scan technology is similar to the regular format except that it includes additional piece of information that identifies the bit of a serialized bitstream containing the failure (*bit_position*). The pattern-based format of the failure data file is as follows:

pat_num *output_ports* *cell_pos* *bit_pos* [*expected_data*]

Where:

pat_num

The TetraMAX pattern number on which the failure occurred. The first pattern is 0.

output_ports

The name of the output port on which the failure was detected.

cell_pos

This is the position of the scan chain cell that captured the data that was in error. The cell position of the scan chain cell closest to the output port is 0, the next one in is 1, and so on.

bit_pos

For each scan chain shift cycles, the serializer is capturing the parallel output of the output compressor. Then, this information is serialized and shift out on the scanout pin. The *bit_position* is the bit of the serialized bitstream where there is a failure. The first *bit_position* is 0 and it corresponds to serializer bit close to the scan out.

expected_data

This optional parameter is a 0 or 1 depending on the expected value specified by the pattern. By default, this parameter is checked against the expected data recorded in the patterns. If this step succeeds, it is a good indicator that the files used for diagnostics and on the tester are consistent. To change the default, use the `-nocheck_expected_data` option of the `set_diagnosis` command.

Cycle-Based Failure Data File

Most ATE vendors generate failure data directly in the pattern-based failure data file format. However, some testers do not support this format. For the unsupported testers, you can use a cycle-based (or vector-based) failure data file format (TetraMAX patterns contain multiple cycles or vectors). Cycle-based failure log files in TetraMAX format are easier to generate than pattern-based failure log files.

TetraMAX ATPG supports both basic-scan and fast-sequential patterns in STIL or WGL format. The binary pattern format and the WGL flat format are not supported for cycle-based failure log file diagnostics.

If the external pattern buffer contains an unsupported pattern format, TetraMAX ATPG displays an error message when you execute the `run_diagnosis` command with a cycle-based failure log file.

Cycles (V statements in STIL format or vector statements in WGL format) are counted when you read patterns using the `set_patterns external` command. This count identifies the vectors at pattern boundaries, and the time when shift cycles start within each pattern. If you or the tester make adjustments that cause the failing cycle/vector to deviate from the corresponding vectors in the STIL/WGL patterns used for diagnosis (such as combining multiple STIL/WGL vectors into a single tester cycle), you must make a corresponding change in the cycle-based failure log to map back to the vectors in the pattern file.

The following `set_diagnosis` options are associated with the cycle-based failure log file:

- `-cycle_offset integer` — You can use this option to adjust the cycle count when the cycle numbering does not start at 1.
- `-show cycles` — This option causes the translated pattern-based failure log file to be reported by the `run_diagnosis` command.

The following example shows sample output. Comments indicate the failure cycle used to generate the pattern-based failure. It also shows whether the cycle was a capture or a shift cycle.

```
4 po0 # Cycle conversion from cycle 34; fail in capture  
4 so 2 # Cycle conversion from cycle 38; fail in shift
```

The following set of commands show an example flow:

```
run_drc ...  
set_patterns -external pat.stil  
set_diagnosis -cycle_offset 1  
run_diagnosis fail.log
```

Cycle-Based Failure Data File Format

A failure data file contains only failed cycles. The format of each line is as follows:

`C output_name cycle [expected_value]`

Where:

C

The first character on the line, indicating that the line specifies tester cycles and not TetraMAX pattern numbers. It helps you identify the type of failure log file (pattern- or cycle-based).

output_name

A string that can be a PO or a scan chain name (no output compressor).

cycle

An integer that identifies the cycle in the external pattern set that failed on the tester. If the first cycle is numbered 0, use the `set_diagnosis -cycle_offset 1` command to make an adjustment. TetraMAX ATPG expects failures only in cycles in which measurements occur (for example, during shift or capture cycles). Invalid failure cycles can provide inaccurate diagnostics results.

expected_value

This optional parameter is a 0 or 1 depending on the expected value specified by the pattern. By default, this parameter is checked against the expected data recorded in the patterns. If this step succeeds, it is a good indicator that the files used for diagnostics and on the tester are consistent. To change the default, use the `-nocheck_expected_data` option of the `set_diagnosis` command.

TetraMAX ATPG ignores all other characters in the line, and treats them as comments.

Failure Data File Extensions

The failure data file can contain the directives to specify settings specific to that failure log file. These directives must be at the top of the failure data file before any failure data. The directives cannot be abbreviated. Any other line in the failure log file is interpreted as failure data.

`.pattern_file_name string`

This directive is required when you are using the split patterns feature. It specifies the name of the corresponding pattern file to associate the failure log files to the pattern file. If there are no failures in the patterns corresponding to a pattern file, this directive is used to make correspondences between pattern and failure log files. This name is assumed to be just the file name, without the directory hierarchy.

If the failure log file does not contain this directive, or if the name does not match, diagnosis is aborted.

`.attr_file_name string`

This directive can be used to set user-defined attributes for a particular failure log file; you can then access the specified *string* value using the Tcl API. For example, the attribute could describe the ATE clock frequency or the pattern type used for testing the chip. You could then retrieve the *string* using the attribute <*attr_file_name*> returned by the `get_diag_files` Tcl API

command. If the name of the directive does not match, the diagnosis process is aborted.

.cycle_offset <d | continue>

This directive adjusts the cycle count when the cycle numbering does not start at 1 for cycle-based failure log files. The *d* parameter is an integer in tester cycles. The purpose of *continue* is for ease of use. The string argument *continue* indicates that the cycle count is not reset from the previous pattern set. The default is to reset the cycle count to 1. This directive only applies to split pattern diagnosis.

When used, this directive overrides the *-cycle_offset* option of the *set_diagnosis* command for this pair of pattern/failure log files. Normally, the cycle count is reset to 1 for every pattern set.

.split_pattern_offset *d*

Specifies the number of offset cycles for each failure log file. This directive is used for diagnosing cycle-based failures in split patterns. For more information on using the *.split_pattern_offset* directive, see the "Split Pattern Diagnostics for Cycle-Based Failures" topic in TetraMAX Online Help.

.truncate *d*

All patterns numbered greater than *d* in this failure log file are ignored. The pattern numbers begin at 0 for each failure log file. The argument *d* specifies the last pattern for which complete failures were captured. It should not exceed the number of patterns in the corresponding pattern file.

When used, this directive overrides the *-truncate* option of the *run_diagnosis* command.

.incomplete_failures

Ignores patterns in the range beginning with the last failing pattern recorded in this failure log file, to the last pattern in the corresponding pattern file, unless there is only one failing pattern in this file.

When used, this directive overrides the *-incomplete_failures* option of the *set_diagnosis* command.

.failure_memory_limit *d*

Ignores patterns in the range beginning with the last failing pattern recorded in this failure log file, to the last pattern in the corresponding pattern file, if the number of failures in this file is at least *d*. The argument *d* is a decimal number that specifies the number of failures that the tester can capture.

When used, this directive overrides the *-failure_memory_limit* option of the *set_diagnosis* command.

Adding Header Information to a Failure Data File

You can insert a header section into a failure data file to include additional data from the ATE, such as key-value pairs information, the device name, job name, or truncation status. This

information is passed to the output of the `write_ydf` command during physical diagnostics (for more information on physical diagnostics, see "[Using Physical Data for Diagnostics](#)").

Not all information in the header section is passed to the Yield Explorer Data Format (YDF) file used for physical diagnostics. Only data described in a configuration file, called the *header schema file*, is retrieved when running diagnostics.

The following sections describe how to insert a header section into a failure data file:

- [Creating a Header Section](#)
- [Creating a Header Schema File](#)
- [Examples](#)

Creating a Header Section

You can place the header section in a failure log file either immediately before or after the `.pattern_file_name` directive. All key-value pairs in the header section are associated with the corresponding pattern file specified by this directive.

To start the header section, specify the `.header` keyword. To finish the header section, use the `.end_header` keyword. Each line in the header section is a key-value pair. A key is a single word separated by tab or space, and the value may be one or more words excluding the special symbols tab, “#” and “\”.

The following example shows a typical header section:

```
.pattern_file_name patterns.bin
.header
DEVICE TOPDUT1
LOT K382
WAFER 03
DIEX 112
DIEY 124
VDD_CORE 1.32
VDD_PAD 3.3
TEMP 0300
START_T Nov 25 2015 18:40:10
TRUNCATE Y
.end_header
6977 PAD_34 1
6981 PAD_34 1
6985 PAD_34 1
6989 PAD_34 1
...
...
```

Note the following:

- After the header information is read by the `write_ydf` command, it is included in the DFTCandidates table in the YDF file. Each keyword constitutes a column with entries specified as strings.
- Only one header section is used for a set of failure log files associated with a single `run_diagnosis` command. When split patterns are used, the header section is defined only one time.
- The header section can be included in any failure log file.
- If duplicate value names are included in the header, the last defined value is used.
- If a custom field in the header matches a standard DFTCandidates Table field, the custom field is ignored. For example:

```
TEST-T> set_ydf schema.txt -schema
-----
YDF Schema Set Summary
-----
LOT used as a standard column in DFTCandidate segment ...
Skipping ...
WAFER used as a standard column in DFTCandidate segment ...
Skipping ...
DIEX used as a standard column in DftCandidate segment ...
Skipping ...
DIEY used as a standard column in DftCandidate segment ...
Skipping ...
YDF Schema has been set for 6 keywords.
CPU_time: 0.00 sec
Memory Usage: 0MB
-----
```

- You can use the `set_diagnosis -show key_value_pairs` command to print the values from the header section to the diagnostics report.

Creating a Header Schema File

The header schema file defines the custom columns that are included in the YDF file. The schema file specifies the keywords and their respective string argument field sizes. If the size is not specified for a keyword, a default string size of 256 is used. The following example is a typical schema file:

```
DEVICE 128
LOT 256
WAFER 128
DIEX 64
DIEY 64
... ...
... ...
... ...
```

After you create the header schema file, you define it using the `-schema` option of the `set_ydf` command, as shown in the following example:

```
set_ydf header_schema_file -schema
```

You need to set the header schema file only once. All successive diagnostics results appended to the same YDF file adhere to the original specified header schema file. When performing an append operation on an existing YDF file, TetraMAX ATPG retrieves the header schema from the YDF file and fills in the appropriate values for the keywords from the failure log file.

However, if you update the diagnosis results for a YDF file to a new file, you must define a new header schema file using the `set_ydf` command. If a new schema is not specified, TetraMAX ATPG uses the header schema file specified earlier in the session. If a schema file is not specified, TetraMAX ATPG does not write the custom columns.

The script in [Example C](#) implements the custom columns in the YDF file during diagnosis.

If a value name is duplicated in the schema file, an error is issued when the `write_ydf` command is executed.

Examples

The examples in this section include the following cases:

- [Example A: Header Schema File for Split Pattern Set With Two Pattern Files](#)
- [Example B: Header Schema File for Split Pattern Set With Three Pattern Files](#)
- [Example C: Flow for Handling Custom columns in the YDF File](#)

Example A: Header Schema File for Split Pattern Set With Two Pattern Files

In this example, the 15 key-value pairs are defined in the header section and passed to the `write_ydf` command using one list of 15 string pairs.

```
.header
DEVICE 1604
LOT PL924
WAFER 03
DIEX 122
DIEY 122
VDD_CORE 1.32
VDD_PAD 3.3
V2_PAD N/A
TEMP 0300
JOB_NAM 1604_SW
JOB_REV 02
FLOOR_ID AF6E
FLOW_ID EWS1
START_T Nov 25 2015 18:40:10
TRUNCATE Y
.end_header
.pattern_file_name pattern_file1.bin
```

```

6977 PAD_34 1
6981 PAD_34 1
6985 PAD_34 1
6989 PAD_34 1
...
.pattern_file_name pattern_file2.bin
7977 PAD_34 1
7981 PAD_34 1
7985 PAD_34 1
7989 PAD_34 1

```

Example B: Header Schema File for Split Pattern Set With Three Pattern Files

In this example, TetraMAX ATPG associates the header with all pattern files: pattern_file1.bin, pattern_file2.bin, and pattern_file3.bin. The header section associated with the second file includes eight key values.

```

.pattern_file_name pattern_file1.bin
6977 PAD_34 1
6981 PAD_34 1
6985 PAD_34 1
6989 PAD_34 1
...
.pattern_file_name pattern_file2.bin
.header
DEVICE 1604
LOT PL924
WAFER 03
DIEX 122
DIEY 122
VDD_CORE 1.32
START_T Nov 25 2015 18:40:10
TRUNCATE Y
.end_header
7977 PAD_34 1
7981 PAD_34 1
7985 PAD_34 1
7989 PAD_34 1
....
.pattern_file_name pattern_file3.bin
9977 PAD_34 1
9981 PAD_34 1
9985 PAD_34 1
9989 PAD_34 1
.....

```

Example C: Flow for Handling Custom Columns in the YDF File

```

TEST-T> set_messages -log example.log -replace
TEST-T> set_command noabort
TEST-T> read_image mydesign.phy.img.gz
TEST-T> set_physical_db -hostname host01 -port_number 9998

```

```
TEST-T> set_physical_db -top_design top_specdevice
TEST-T> set_physical_db -device [list "specDevice" "1"]
TEST-T> match_names -verify all
TEST-T> run_drc mydesign_scan.spf
TEST-T> set_patterns -external mydesign_pat.bin
TEST-T> run_diagnosis sample1.ff
TEST-T> set_ydf schema1.sch -schema
TEST-T> set_diagnosis -show key_value_pairs

// A new YDF file is created with the results of last
// diagnostics run (note the -replace option).

TEST-T> write_ydf mydesign-diag1.ydf -replace \
    -device TESTDEVICE -version 1 \
    -candidates -cell -instance_cell \
    -cell_instance_pin_net -net_path \
    -net_contact_position -net_layer
TEST-T> run_diagnosis sample2.ff

// The same YDF file is updated with the results of last
// diagnostics run (note the -append option). The same header
// schema file is used.

TEST-T> write_ydf mydesign-diag1.ydf -append -candidates \
    -cell -instance_cell -cell_instance_pin_net \
    -net_path -net_contact_position -net_layer
TEST-T> run_diagnosis sample3.ff
TEST_T> set_ydf schema2.sch -schema

// A new YDF file is created with the results of last
// diagnostics run (note the -replace option). A new header schema
// file is used.

TEST-T> write_ydf mydesign-diag2.ydf -replace \
    -device TESTDEVICE -version 2 \
    -candidates -cell -instance_cell \
    -cell_instance_pin_net -net_path \
    -net_contact_position -net_layer
TEST-T> run_diagnosis sample4.ff

// The same YDF file is updated with the results of last
// diagnostics run (note the -append option). The same header
// schema file is used.

TEST-T> write_ydf mydesign-diag1.ydf -append -candidates \
    -cell -instance_cell -cell_instance_pin_net \
    -net_path -net_contact_position -net_layer
TEST-T> exit
```

Failure Data File Limitations

The following limitations are associated with failure data files:

- Only STIL and WGL patterns with cycle-based failure data files are supported. Binary patterns are also supported with pattern-based failure files.
- The `-truncate` option of the `run_diagnosis` command is not supported with cycle-based diagnosis.
- WGL flat patterns are not supported for diagnostics.

Class-Based Diagnostics Reporting

The class-based diagnostics report includes cell, net, subnet, and bridgeable area physical data, and other information required for physical failure analysis (PFA). You can enable this report using the following command:

```
set_diagnosis -organization class
```

The following sections describe how to configure, create, and read a class-based report:

- [Filtering Candidates](#)
 - [Filtering Bridge Candidates](#)
 - [Resetting User-Specified Filters](#)
 - [Reporting Detailed Candidate Information](#)
 - [Example Flow](#)
 - [Understanding the Class-Based Report](#)
 - [Class-Based Cell-Aware Diagnostics](#)
-

Filtering Candidates

You can use the `-filter_candidates` option of the `set_diagnosis` command to apply several different types of filters when generating the class-based report.

If you use the `score_distance` parameter of the `-filter_candidates` option, you can exclude all candidates that are further than the specified percentage distance from the maximum candidate percentage match score:

```
set_diagnosis -organization class -filter_candidates \
  {score_distance 10}
```

The default for the `score_distance` parameter is 20 percent.

You can also use the `min_score` parameter to filter candidates with a match score less than a specified minimum value (the default is 50). In this case, at least one candidate is reported even if it is below the minimum value. You can also use the `max_number` parameter to limit the number of candidates reported for each defect. Since candidates are reported in order of decreasing scores, this effectively specifies to report the top *N* candidates.

In the following example, all candidates with a match score less than 10 and a score further than 20 are removed from the diagnostics report:

```
set_diagnosis -organization class -filter_candidates \
  {min_score 75 max_number 20}
```

The number of filtered candidates are reported by an M804 message:

Warning: Filtered 2 candidates with match score outside the score_distance of 20. (M804)

Filtering Bridge Candidates

An ambiguous bridge is any bridge candidate in which every explained pattern has the same value on the aggressor node. These bridge candidates are not distinguishable from a stuck candidate on the victim node. A same-cell bridge is any bridge candidate between two nodes that are connected to the same library cell. Such bridge candidates are not distinguishable from bridges inside the cell.

You can use the `-filter_bridges` option of the `set_diagnosis` command to remove bridge candidates exercised by either an ambiguous bridge or any bridge candidates between two nets connected to the same cell.

To remove bridge candidates with the same value on the aggressor node and that behave the same as a stuck candidate, use the `ambiguous` parameter:

```
set_diagnosis -filter_bridges {ambiguous on}
```

To remove bridge candidates between two nets connected to the same cell, use the `same_cell` parameter:

```
set_diagnosis -filter_bridges {same_cell on}
```

Resetting User-Specified Filters

To reset any filters you specified using the `-filter_candidates` or `-filter_bridges` options, specify the `-reset_filters` option of the `set_diagnosis` command:

```
set_diagnosis -reset_filters
```

Note that all class-based filters must be specified before the `run_diagnosis` command to affect a candidate list reported by the `report_defects` or `write_ydf` command.

Reporting Detailed Candidate Information

You can obtain more detailed information about net connectivity and physical locations by specifying the `set_diagnosis -verbose` command, or report the same data separately using the `-candidates` option of the `report_nets` command or the `report_physical` command. For example, the `-candidates` option of the `report_nets` command returns the following net connectivity information:

```
TEST-T> report_nets -candidates all
Candidate 1:
  Net connections:
  -----
  I_RISC_CORE/I_ALU/n57 (695)
  O I_RISC_CORE/I_ALU/U108/Z
  I I_RISC_CORE/I_ALU/U90/A2
```

```
I I_RISC_CORE/I_ALU/U82/A2
-----
Net connections:
-----
I_RISC_CORE/I_ALU/n111 (889)
O I_RISC_CORE/I_ALU/U149/ZN
I I_RISC_CORE/I_ALU/U147/I0
-----
```

The `-candidates` option of the `report_physical` command reports the following physical data:

```
TEST-T> report_physical -candidates all
Candidate 1:
Physical details:
-----
~ METAL3 (619530 622190) (623420 622400) TB
-----
```

To view candidates for all defects or a particular defect after specifying the `run_diagnosis` command, use the `-defect` option of the `report_defects` command:

```
TEST-T> report_defects -defect all
Defect 1:
#failing_patterns=5
#observe_points=1
#simulated_failures=5
#candidates=1
bbox=(619530 622190) (623420 622400), area=816900
Candidate 1:
class=Bridge
net1_id=695, net2_id=889
driver1=I_RISC_CORE/I_ALU/U108, pin=Z
driver2=I_RISC_CORE/I_ALU/U149, pin=ZN
layers=METAL3
bbox=(619530 622190) (623420 622400), area=816900
behavior=bAND, match_score=100.00% (TFSF=5/TFSP=0/TPSF=0)
```

You can generate a fault list for high-resolution ATPG using the `-faults` option of the `report_defects` command:

```
TEST-T> report_defects -faults
ba0 NC I_RISC_CORE/I_ALU/U108/Z I_RISC_CORE/I_ALU/U149/ZN
ba0 NC I_RISC_CORE/I_ALU/U149/ZN I_RISC_CORE/I_ALU/U108/Z
```

Example Flow

A typical class-based diagnostics reporting flow is as follows:

```
read_image CHIP.img  
  
set_patterns -external pat.bin  
  
set_physical_db -database ./PHDS  
set_physical_db -port_number 9998  
  
open_physical_db  
  
set_physical_db -hostname localhost  
set_physical_db -device {"TOP" "1"}  
  
set_diagnosis -organization class  
set_diagnosis -filter_candidates {min_score 75}  
  
run_diagnosis die123.tmx  
  
report_physical -candidates all  
set_ydf -version 1.2  
  
write_ydf die123.ydf -replace  
  
close_physical_db
```

Understanding the Class-Based Diagnostics Report

The class-based diagnostics report includes cell, net, subnet, and bridgeable area physical data, and information required for physical failure analysis (PFA). This data includes the defect area and the overall bounding box coordinates.

The header section of the report contains the number of tester failures, tester patterns, patterns simulated by diagnosis, and identified defects:

```
-----  
Diagnostics Candidate Report  
-----  
#tester_failures=256 (#ignored=41/#used=215)  
#patterns=51, #simulated_patterns=17 (#fail=7/#pass=10)  
#defects=1
```

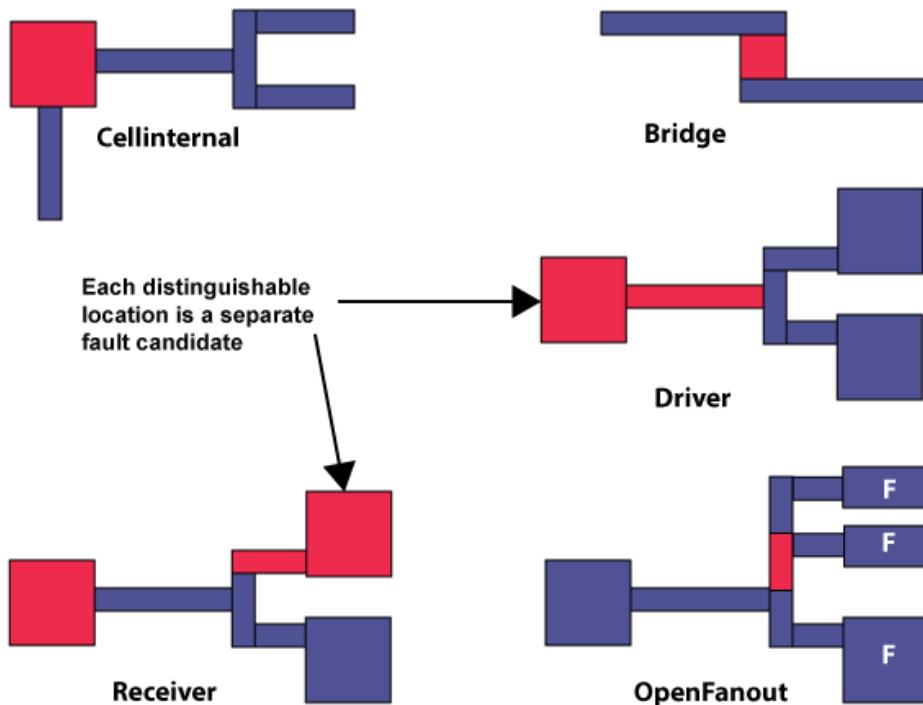
Each defect is reported, and includes the number of failing patterns, observe points, simulated failures, and list of one or more candidates:

```
Defect 1:  
#failing_patterns=7
```

```
#observe_points=65
#simulated_failures=70 (#unique=68/#potential=2)
#candidates=2
```

Each defect candidate is assigned a predefined class which defines the logical location and physical details of the candidate (shown in red in Figure 1). Classes include Cellinternal, Bridge, Receiver, Driver, and OpenFanout. These classifications match Yield Explorer classifications.

Figure 1: Candidates are Assigned Predefined Classes, such as Cellinternal or Bridge.



A candidate includes a specific set of physical details. For example, the description of a Receiver class candidate consists of a cell instance and the net or subnet connected to an input pin of that cell. A Bridge class candidate includes the bridgeable area between two metal shapes on the same layer.

Candidate 1:

```
class=Receiver
instance=c/iproc/U14417, pin=E, module=A010LL
net_id=27094, fanout_id=2
behavior=sa0, match_score=91.43% (TFSF=64/TFSP=6/TPSF=0)
```

Candidate 2:

```
class=Bridge
net1_id=12443, net2_id=27094
driver1=c/iprociarc_portxdati_regx12x, pin=QN
driver2=c/iprociarc_portxdati_regx12x, pin=Z
behavior=bAND, match_score=100.00% (TFSF=70/TFSP=0/TPSF=0)
```

Note the following:

- A single candidate can have multiple fault associations, and each fault has a particular type of behavior. For example, a Bridge class candidate may share both bAND and bDOM behavior between the same net pairs. In this case, each behavior is scored separately.
- Candidates with different behaviors can be reported in the same defect — as long as they are associated with a common set of tester failures and they are in separate logic cones. This means candidates with different behaviors are scored based upon the same set of tester failures.
- Nets are identified by the primitive ID of the driving gate for candidate classes with a net, subnet, or net pair. For OpenFanout class candidates, the subnet ID is always the physical subnet ID, even if the `set_diagnosis -show physical_subnet_id` command is specified. For Bridge class candidates, the instance and pin name for both net drivers are reported.
- If physical data is available from the PHDS database, a physical summary is also included with each candidate. This summary includes the layers associated with the candidate and the candidate bounding box and area:

```
layers=METAL2  
bbox=(619320 659710) (619530 660730), area=214200
```

Class-Based Cell-Aware Diagnostics

The class-based diagnostics report supports cell-aware diagnostics. Cell-aware behaviors may be included in Driver, Receiver, CellInternal, and CellAware class candidates.

When cell-aware diagnostics is enabled by the `set_diagnosis -fault_type all` command, several behaviors may be reported differently than the fault-based report: ca0, ca1, ca01, car, caf, carf. These six behaviors correspond to the sa0, sa1, sa01, str, stf, strf faults, except that their match score is calculated assuming that the defect may be inside the cell rather than on the pin.

If cell test models have been read using the `read_cell_model` command, CTM behaviors will also be scored during diagnosis and any matching candidates will be reported as a CellAware class candidate with the CTM ID and any equivalent CTM IDs.

Fault-Based Diagnostics Reporting

In the fault-based diagnostics report, a collapsed fault is a unique candidate. A fault and all its equivalent faults are considered a single candidate and separate faults are created for different behaviors.

The fault-based report is created by default when you specify the `run_diagnosis` command. If you previously ran class-based reporting (described in the "[Class-Based Diagnostics Reporting](#)" section), you can revert back to fault-based reporting using the following command:

```
set_diagnosis -organization fault
```

The following example shows a typical fault-based diagnosis report produced by the `run_diagnosis` command.

```
TEST-T> run_diagnosis /project/mars/lander/chipA_failure.dat \
Diagnosis summary for failure file /project/mars/lander/chipA_
failure.dat
#failing_pat=4, #failures=5, #defects=2, #faults=3, CPU_time=0.05
Simulated : #failing_pat=4, #passing_pat=35, #failures=5
-----
Fault candidates for defect 1: stuck fault model, #faults=1,
#failing_pat=3,
#passing_pat=36, #failures=3
-----
match=100.00%, #explained patterns: <failing=3, passing=36>
sa1 DS de_d/data3_reg_0/Q (S003)
sa1 -- de_d/U211/A (SELX2)
-----
Fault candidates for defect 2: stuck fault model, #faults=2,
#failing_pat=2,
#passing_pat=37, #failures=2
-----
match=100.00%, #explained patterns: <failing=2, passing=37>
sa1 DS de_encrypt/C264/U36/O (L434ND)
sa0 -- de_encrypt/C264/U36/I1 (L434ND)
sa0 -- de_encrypt/C264/U36/I2 (L434ND)
sa0 -- de_encrypt/C264/U28/O (L434ND)
sa1 -- de_encrypt/C264/U26/I2 (L434ND)
-----
match=50.00%, #explained patterns: <failing=1, passing=37>
sa1 DS de_encrypt/C264/U28/I1 (L434ND)
```

This example shows that the four failing patterns in the failure log file were resolved to two defects. The first defect came from three failing patterns and was resolved to one fault location and its fault-equivalent location. The second defect came from two failing patterns and was resolved to two fault locations. The first fault location has a 100 percent match score and has four faults-equivalents. The second fault location of the second defect has a 50 percent match score.

The fields in this report are described as follows:

`#failing_patterns`

Identifies the total number of failing patterns in the failure file. A pattern is assumed to include both a measure of all POs and an unload of the scan chain.

`#failures`

Located in the main header, this field identifies the number of failures in the failure log file. In each defect's header, it shows the number of failures the candidates in that defect caused.

`#defects`

Indicates the number of different defects that appear to be causing the failures.

#faults

Indicates the number of collapsed faults. In the main header, it indicates the total number of faults. In each defect's header, it shows the number of faults in that defect group.

Simulated : #failing_pat=, #passing_pat= #failures

Displays the number of failing and passing patterns that were simulated, and the number of failures in the simulation.

Fault candidates for defect : <> fault model

The header for each defect displays the fault model used for that defect group. Then, there is the list all the fault candidates for a given defect. The fault list is given in the following format:

- First column: fault type. It could be sa0 for stuck-at-0 or sa1 for stuck-at-1.
- Second column: detection technique. It could be: “DS” (detected by simulation). This is the representative fault. “– –”. This is an equivalent fault.
- Third column: fault location (pin pathname)
- Fourth column: module name of the defective cell.

match=%

Indicates the match score of the set of fault candidates based on how well they match the defective device response on the tester.

#explained pattern: <failing: , passing:>

Indicates the number of failing and passing patterns that are explained with the fault candidate.

If logic diagnostics fails to find any candidates, the report appears as shown in the following example:

```
#failing_patterns=7, #defects=0, #unexplained_fails=7, CPU=19.56
```

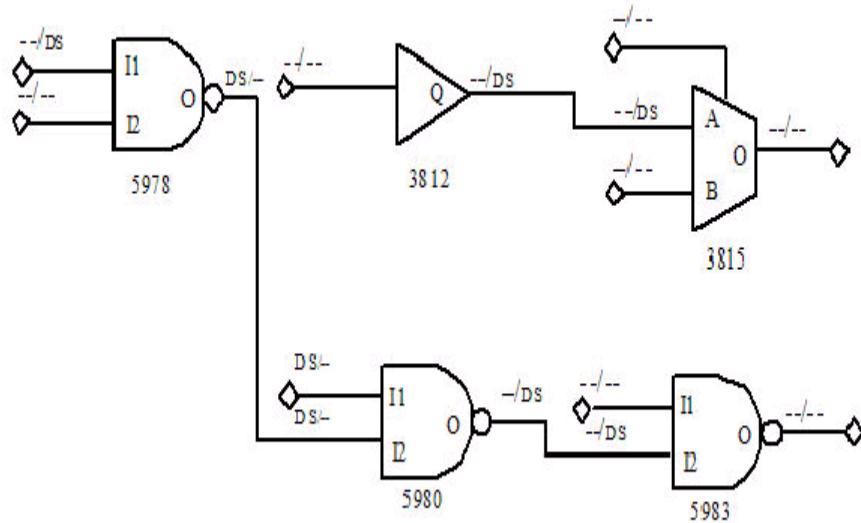
Unexplained pattern list:

3 6 8 12 13 25 67

No candidate because all failing patterns are unexplained.

By using the -display option of the run_diagnosis command or by checking the Display Results in Viewer check box in the Run Diagnosis dialog box of the TetraMAX GUI, you can display the instances and fault locations graphically, as shown in [Figure 1](#). For this schematic, the pin display data format has been set to Fault Data, where the format is stuck-at-0/stuck-at-1.

Figure 1 Diagnosis Data Displayed Graphically



You can identify each defect location by the DS (detected by simulation) code on the pin corresponding to either a fault site or a fault equivalent. The DS notation marks all potential fault sources that could cause the same failing data pattern. The notation **DS/-** indicates that a stuck-at-0 fault at that point in the design would cause the failure, and the notation **--/DS** indicates that a stuck-at-1 at that point in the design would cause the failure. TetraMAX ATPG shows all potential failure sites that would cause the same failure data patterns.

In this example, the diagnosis by TetraMAX ATPG finds two independent areas of failure in the design. The graphical schematic viewer (GSV) display of [Figure 1](#) shows the two corresponding independent groups of logic. According to the diagnosis, the faulty circuit location for each failure is displayed along the path.

Verbose Format

When the `-verbose` option is used for either the `set_diagnosis` or `run_diagnosis` commands, additional information is added to the diagnostics report. An example is as follows:

```
#failing_pat=15, #failures=17, #defects=2, #faults=8, CPU_
time=0.01
Simulated : #failing_pat=15, #passing_pat=36, #failures=17
-----
Fault candidates for defect 1: stuck fault model, #faults=1,
#failing_pat=14, #passing_pat=37, #failures=14
Observable points:
782
-----
Explained pattern list:
2 16 20 21 23 25 34 37 38 39 40 45 47 49
-----
```

```
match=100.00%, (TFSF=14/TFSP=0/TPSF=0), #perfect/partial match:
<failing=14/14, passing=37>
sa0 DS mic0/pc0/add_247/U41/Z (ND2I)
sa0 -- mic0/pc0/add_247/U40/B (ENI)
```

Note the following definitions:

- Observable points — The list of gate IDs in which the failures generated by all fault candidates from this defect group occurred.
- Explained pattern list — The list of the all patterns which could be explained by all fault candidates from this defect group. Some fault candidates in the same defect could explain some patterns and other candidate other patterns. But the list is the union of all explained patterns.
- TFSF=N1/TFSP=N2/TPSF=N3 — These are the match score components. See `run_diagnosis -rank_fault` for more details.
- #perfect/partial match: <failing=N1/N2, passing=N3> — This data indicates the number of failing patterns that are perfectly or partially explained by the fault candidate. A perfect match means that the failures observed on the tester are perfectly matching (without any other failure either in simulation or on the tester). It also indicates the number of passing patterns that are explained with the fault candidate.

Physical Diagnosis Format

```
TEST-T> run_diagnosis fail_56.log
Setting top-level physical design name to 'RISC_CHIP'
Check expected data completed: 241 out of 241 failures were
checked
Diagnosis summary for failure file fail_56.log
#failing_pat=30, #failures=241, #defects=1, #faults=1, CPU_
time=1.08
Simulated : #failing_pat=30, #passing_pat=96, #failures=194
-----
Defect 1: stuck-at fault model, #faults=1, #failing_pat=30,
#passing_pat=96, #failures=241
Observable points:
1693 1687 1696 1699 1672 1530 1529
-----
Explained pattern list:
40 41 47 48 50 51 54 55 58 59 67 69 70 71 72 73 77 78 79 80 85
88
92 93 94 95 97 98 99 101
-----
match=100.00%, #explained patterns: <failing=30, passing=96>
sa01 DS I_RISC_CORE/I_ALU/U14/ZN (inv0d1)
Pin_data: X=747110 Y=652790, Layer: METAL (38)
Cell_boundary: L=746115 R=747345 B=650175 T=653865
Subnet_id=4
-----
Total Wall Time = 22.97 sec PHDS Query Time = 22.13 sec
```

```
PHDS queries: subnets (added/total)=10/40 bridges  
(added/total)=28/58
```

The physical diagnosis report includes the physical location of the failing pin and cell.

Where:

Pin_data:

- `X` is the horizontal coordinate of one of the vertices of the pin associated with the location of the fault candidate.
- `Y` is the vertical coordinate of one of the vertices of the pin associated with the location of the fault candidate.
- `Layer` is the physical layer where the pin object is defined.

Cell_boundary:

- `L` is the horizontal coordinate of the leftmost boundary of the cell identified with the fault candidate.
- `R` is the horizontal coordinate of the rightmost boundary of the cell identified with the fault candidate.
- `B` is the vertical coordinate of the bottommost boundary of the cell identified with the fault candidate.
- `T` is the vertical coordinate of the topmost boundary of the cell identified with the fault candidate.

Also note that the performance data in the physical diagnosis report is slightly different than the standard diagnosis report:

- The `CPU_time` is strictly the time that TetraMAX is active. This time does not include the PHDS query time. In the previous example, the `CPU_time` is approximately 1 second, even though the diagnosis run took almost 23 seconds.
- The PHDS diagnosis report includes the wall time and the time it takes to query the PHDS database during diagnosis.
- The second line in the report displays the number of extracted subnets and the number of bridge pairs queried in the PHDS database compared to the total number of subnets and bridging pairs identified during the previous diagnosis run. This data helps you identify any matching issues between the logical and physical names during diagnosis.
- The subnets and bridges extracted during a diagnosis run are displayed as “added”. The “total” includes subnets and bridges extracted in previous diagnosis runs. For example, if the next diagnosis run extracts 10 subnets and 22 bridges, the second line appears as follows:

```
PHDS queries: subnets (added/total)=10/50 bridges  
(added/total)=22/80
```

Scan Chain Diagnosis Format

```

fail.log scan chain diagnosis results: #failing_patterns=79
-----
  defect type=stuck-at-1
  match=100% (TFSF=500/TFSP=0/TPSF=0) chain=c0 position=178
  master=CORE/c_rg0 (46)
  match=100% (TFSF=500/TFSP=0/TPSF=0) chain=c0 position=179
  master=CORE/c_rg2 (57)
  match= 98% (TFSF=500/TFSP=10/TPSF=0) chain=c0 position=180
  master=CORE/c_rg6 (54)
  CPU=0.26 #sim_patterns=57 #sim_cells=64
-----

#failing_patterns
  Indicates the total number of failing patterns in the failure file.

defect type
  The predicted type of defect. It could be stuck-at, slow-to-rise, slow-to-fall,
  fast-to-rise, or fast-to-fall. Note: The polarity of the reported defect affects
  the scan output of the top candidate for each scan chain. It is not necessarily
  the same for all scan cells because an inverter might be present in the scan
  path. The exact polarity can be retrieved using the Tcl API.

match
  A percentage score that measures how well failures seen on the tester match a
  simulated chain defect at that location. The components of the match score
  (TFSF, TFSP, TPSF) calculation are displayed in the verbose report.

chain
  Indicates the chain name where the defect is diagnosed.

position
  Indicates the position in the chain where the defect is diagnosed.

master
  Indicates the scan cell instance name of the diagnosed defect.

  Next, follow these performance indicators:

CPU
  Indicates the CPU time of the chain diagnosis run.

#sim_patterns
  Indicates the number of patterns used during the chain diagnosis simulation.

#sim_cells
  Indicates the number of cells used during the chain diagnosis simulation.

  The chain diagnostics can also appear as follows:

./failures/fail8g16.log scan chain diagnosis results: #failing_
patterns=1
-----
```

Warning: Insufficient data to locate stuck-at-0 fault in chain
48.

This example reports that the number of failures contained in the failures log file is sufficient to determine the behavior and the chain name of the fault candidate. But it fails to accurately locate the failing scan cells. More failures are needed.

The report can also appear as follows:

```
./failures/fX7.log scan chain diagnosis results: #failing_
patterns=1
```

```
Scan chain diagnosis failed to identify any fault candidate.
```

This example indicates that the scan chain diagnostics failed to find a fault candidate that match the failures seen on the tester.

```
/i_p0/E (mx2a3)
# subnet_id=2
```

Failure Mapping Report for DFTMAX Patterns

When you run diagnostics for DFTMAX patterns, the report includes a section in its header that displays the mapping of the failures. This report is an intermediate report and is not intended to be a complete report. It includes only those cases that have a unique choice during the first phase of DFTMAX failure mapping procedure.

To print a complete mapping report, you have to use the

`run_diagnosis -only_report_failures` command or the `set_diagnosis -mapping_report` command. The format of the later report is explained in "Understanding the Failure Mapping Report." The format of `-only_report_failures` is documented in the description of the `run_diagnosis` command. You should use this report because it is short and easy to understand.

The following example shows how this additional section appears in the diagnosis report:

```
-----  
pattern chain pos# output pin_names  
-----  
9 1 4 test_so1 test_so2 test_so3  
14 1 3 test_so1 test_so2 test_so3  
15 1 4 test_so1 test_so2 test_so3  
48 1 4 test_so1 test_so2 test_so3  
52 1 4 test_so1 test_so2 test_so3  
-----  
Failure mapping completed: #failing_pats=5, #skipped_pats=0,  
#masked_cycles=0, CPU_time=0.00  
-----
```

This section includes a header which describes the column printed after it. The failures mapping results are printed next, followed by a failure mapping one line summary.

The columns are described as follows:

- `pattern` -- the failing pattern number
- `chain` -- the failing scan chain
- `pos#` -- the failing scan shift present in the failures log file
- `output pin_names` -- the names of the output pins where the failures are observed

The default number of failures reported are 10. You can change this by running the command `set_diagnosis -max_report_failures <N>`.

The failure mapping one-line summary shows the number of failing patterns that were processed (`#failing_pats`). When failures for a particular cycle could not be mapped back, the `#masked_cycles` is incremented. In the previous example, no cycle has been masked. When too many cycles per pattern are masked, the entire pattern is skipped. This is indicated by the `#skipped_pats`. In the previous example, no pattern has been skipped by the mapping process. When `#masked_cycles` and `#skipped_pats` are equal to 0, this is an indicator that the mapping step of the diagnosis is doing a good job.

When tail pipeline registers are present, the failures log file indicates the scan cell index which is failing + `N`, where `N` is the depth of the tail pipeline register. Then, the column `pos#` in the previous report is not the failing scan cells index. To precisely determine the failing scan cell index, execute the command `run_diagnosis <failure_log> -only_report_failures`.

Composite Fault Model Data Report

Composite faults are based on TetraMAX ATPG component faults. They are only used in a diagnosis report to better describe the observed behavior of a defect on the tester. If this defect behavior is observed, the composite fault model behaviors may be reported by default. For the ranking flow, TetraMAX diagnostics ranks only component fault types, unless the `set_diagnosis -composite` command is specified. The composite fault types are as follows:

- `sa01` — The fault location can behave as a stuck-at 0 on some patterns and a stuck-at 1 on others. This could be a coupled open defect or a bridge type defect. On nets with fanout branches, it is possible for this fault type to appear as stuck-at 0 on some patterns and stuck-at 1 on others. For ranking, this fault model can produce optimistic scores.
- `strf` — The fault location can cause a delay on both rising and falling transitions (slow-to-rise-fall). The traditional fault models of `str` and `stf` are unidirectional.
- `bAND` or `bOR` — The defect location behaves as a wired-AND or wired-OR type bridge. Both nodes of the bridging fault are simulated and reported.
- `bDOM` — The defect location behaves as the victim node of a dominant bridge. Ranking scores are based on the fault simulation at the fault site for failing and passing patterns. This might result in optimistic ranking scores since this model always matches the tester for passing patterns. The scores are optimistic only when the aggressor is unknown. Only the victim node is reported.

The following example report show how the diagnosis report appears with composite fault model data:

```

#failing_pat=6, #failures=20, #defects=4, #faults=5, CPU_
time=0.44
Simulated : #failing_pat=6, #passing_pat=75 #failures=23
-----
Fault candidates for defect 1: stuck fault model, #faults=1,
#failures=2
-----
match=100.00%, #explained patterns: <failing=2, passing=72>
sa01 DS ENC/I_RC/U1569/B (and2c3)
-----
Fault candidates for defect 2: transition fault model, #faults=1,
#failures=1
-----
match=100.00%, #explained patterns: <failing=1, passing=73>
str DS ENC/I_RC/U1685/Y (inv1a3)
stf -- ENC/I_RC/U1685/A (inv1a3)
-----
Fault candidates for defect 3: stuck fault model, #faults=2,
#failures=2
-----
match=100.00%, #explained patterns: <failing=2, passing=72>
sa0 DS ENC/I_RC/U1697/Y (inv1a3)
sa1 -- ENC/I_RC/U1697/A (inv1a3)
sa0 DS ENC/I_RC/U2074/B (ao4f3)
-----
Fault candidates for defect 4: bridging fault model, #faults=1,
#failures=1
-----
match=33.33%, #explained patterns: <failing=1, passing=71>
bAND DS ENC/I_RC/U1685/Y ENC/I_RC/U1697/Y (inv1a3)

```

Fault types:

sa01:

Fault location behaves like a sa0 on some patterns, and a sa1 on others. This could be a coupled open or a bridge type defect.

strf:

Fault location can cause a delay on both rising and falling transitions (slow-to-rise-fall).

bAND or bOR:

The defect location behaves as a wired AND or wired OR type bridge. Examples are provided below:

```

-----
match=100.00%, (TFSF=15/TFSP=0/TPSF=0), #perfect/partial
match: <failing=15/15, passing=31>
bAND DS mic0/pc0/add_247/U14/Z (ENI) mic0/alu0/U89/Z
(ND2I)
-----
match=100.00%, (TFSF=24/TFSP=0/TPSF=0), #perfect/partial

```

```
match: <failing=24/24, passing=51>
bOR DS mic0/pc0/add_238/U76/Z (ENI) mic0/alu0/U12/Z
(ND2I)
```

Note that both nodes of the bridge are reported. The library cells are also provided in parenthesis.

bDOM:

The defect location behaves as the victim node of a dominant bridge.

By default, only the victim node is reported. An example is as follows:

```
-----
match=100.00%, (TFSF=24/TFSP=0/TPSF=0), #perfect/partial
match: <failing=24/24, passing=56>
bDOM DS mic0/pc0/add_233/U39/Z (ND2I)
bDOM -- mic0/pc0/add_233/U26/A (AN2I)
```

However, if the likely bridging pairs or ranking flow is used, then both nodes can be included in the report. An example is provided below:

```
-----
match=100.00%, (TFSF=75/TFSP=0/TPSF=0), #perfect/partial
match: <failing=75/75, passing=92>
bDOM DS mic0/pc0/add_233/U39/Z (ND2I) mic0/alu0/U16/ZN
(INV2I)
```

Note that the library cells are provided in parenthesis.

The following example shows how stuck-open faults in the diagnosis subnets report:

```
#-----
# Defect 1: stuck fault model, #faults=1, #failing_pat=6,
#passing_pat=14, #failures=35
#-----
# match=100.00%, #explained patterns: <failing=6, passing=14>
# sa01 DS top/i_p0/E (mx2a3)
# subnet_id=2
```

Parallel Diagnostics

You can diagnose multiple failure logs in parallel in a single TetraMAX session with a single `run_diagnosis` command. This approach, called *parallel diagnostics*, improves volume diagnostics throughput and is much more memory efficient than invoking multiple TetraMAX sessions. It is especially useful when processing a large number of failure files.

The following sections describe how to run parallel diagnostics:

- [Specifying Parallel Diagnostics](#)
- [Converting Serial Scripts to Parallel Scripts](#)
- [Using Split Datalogs to Perform Parallel Diagnostics for Split Patterns](#)
- [Diagnosis Log Files](#)
- [Parallel Diagnostics Limitations](#)

See Also

[Running Multicore ATPG](#)

Specifying Parallel Diagnostics

To specify parallel diagnostics, use the `-num_processes` option of the `set_diagnosis` command. This option sets the number of cores to use during parallel diagnostics. You can specify the number of processes to launch based on the number of CPUs and the available memory on the multicore machine.

The following example configures parallel diagnostics to use four cores:

```
set_diagnosis -num_processes 4
```

You can also define a post processing procedure to run concurrently with a parallel diagnostics run, as shown in the following example:

```
proc pp {} {
    write_ydf -append my_ydf.ydf
    set datalog [get_attribute [index_collection \
        [get_diag_files -all] 0] name]
    foreach_in_collection cand [get_candidates -all] {
        echo [get_attribute $cand pinpath] \
            $datalog >> candidates.list
    }
}
set_diagnosis -post_procedure pp
```

In the previous example, a procedure called `pp` specifies the `write_ydf` command and several Tcl API commands. This procedure is executed using the `-post_procedure` option of the `set_diagnosis` command.

Note that when you use multiple slave cores, each process writes the same report file.

To disable a post processing procedure, specify the following command:

```
set_diagnosis -post_procedure none
```

When parallel diagnostics is enabled, you can specify a list of data logs in the `run_diagnosis` command, as shown in the following example:

```
set_diagnosis -num_processes 4
run_diagnosis [list {datalogs/ff_[1-9].log} \
```

```
{datalogs/ff_[1-9][0-9].log} \
{datalogs/ff_100.log}]
```

Note that wildcards are also accepted when specifying data logs, as shown in the following example:

```
run_diagnosis datalogs/ff_*.log
```

Converting Serial Scripts to Parallel Scripts

You can convert an existing serial mode script to a parallel mode script, and then run parallel diagnostics for any number of cores.

The following example is a script snippet used for volume diagnosis in serial mode:

```
for {set i 1} {$i <= 100} {incr i} {
    set fail_log datalogs/ff_${i}.log
    run_diagnosis $fail_log
    write_ydf -append my_ydf_file.ydf
    set datalog [get_attribute [index_collection \
        [get_diag_files -all] 0] name]
    foreach_in_collection cand [get_candidates -all] {
        echo [get_attribute $cand pinpath] \
            $datalog >> candidates.list
    }
}
```

The following example shows how the script snippet in the previous example appears as a parallel script that uses four cores:

```
set_diagnosis -num_processes 4
proc pp {} {
    write_ydf -append my_ydf_file.ydf
    set datalog [get_attribute [index_collection \
        [get_diag_files -all] 0] name]
    foreach_in_collection cand [get_candidates -all] {
        echo [get_attribute $cand pinpath] \
            $datalog >> candidates.list
    }
}
set_diagnosis -post_procedure pp
run_diagnosis [list {datalogs/ff_[1-9].log} \
    {datalogs/ff_[1-9][0-9].log} \
    {datalogs/ff_100.log}]
```

Using Split Datalogs to Perform Parallel Diagnostics for Split Patterns

You can use split datalogs to perform parallel diagnostics for split patterns.

The following example shows a serial mode script snippet used for diagnostics with split datalogs:

```
set_patterns -external p1.bin -split
set_patterns -external p2.bin -split
set_patterns -external p3.bin -split
for {set i 1} {$i <= 100} {incr i} {
    run_diagnosis datalogs/ff_${i}.p1.log \
        -file "datalogs/ff_${i}.p2.log datalogs/ff_${i}.p3.log"
    write_ydf -append my_ydf.ydf
    set datalog [get_attribute [index_collection \
        [get_diag_files -all] 0] name]
    foreach_in_collection cand [get_candidates -all] {
        echo [get_attribute $cand pinpath] \
            $datalog >> candidates.list
    }
}
```

The following example shows how the script snippet in the previous example appears as a parallel mode script that uses four cores:

```
set_patterns -external p1.bin -split
set_patterns -external p2.bin -split
set_patterns -external p3.bin -split
set_diagnosis -num_processes 4
proc pp {} {
    write_ydf -append my_ydf.ydf
    set datalog [get_attribute [index_collection \
        [get_diag_files -all] 0] name]
    foreach_in_collection cand [get_candidates -all] {
        echo [get_attribute $cand pinpath] \
            $datalog >> candidates.list
    }
}
set_diagnosis -post_procedure pp
run_diagnosis datalogs/ff_*.p1.log \
    -file "datalogs/ff_*.p2.log datalogs/ff_*.p3.log"
```

See Also

[Reading a Split Patterns File](#)

Diagnosis Log Files

When you run parallel diagnosis, the diagnosis log is stored in multiple files; one file is created for each core. The name of the diagnosis log file is based on the name of the tool log file specified by the `set_messages` command and is appended with the core ID.

The following example specifies a log file called `diag.log`:

```
set_messages -log diag.log -replace -level expert
```

When multiple cores are used for parallel diagnostics, a diagnosis log file is created for each core, as shown in the following example:

```
diag.log.1
diag.log.2
diag.log.3
diag.log.4
```

Each datalog file is processed for a different slave core, as specified in the tool log file:

```
run_diagnosis datalogs/ff_*.log
Perform diagnosis with 100 failure files.
Starting parallel processing with 4 processes.
-----
Failure file >>                                output log
-----
datalogs/ff_100.log          diag.log.1
datalogs/ff_101.log          diag.log.2
datalogs/ff_102.log          diag.log.3
datalogs/ff_103.log          diag.log.4
datalogs/ff_104.log          diag.log.4
datalogs/ff_105.log          diag.log.2
datalogs/ff_106.log          diag.log.3
datalogs/ff_107.log          diag.log.1
...
-----
End parallel diagnosis: Elapsed time=14.33 sec, Memory=596.61MB.
Processes Summary Report
-----
```

In the previous example, the total memory consumed by parallel diagnostics is 596.61MB, and the total elapsed runtime is 14.33 seconds.

A slave core diagnosis log file is similar to a single core diagnosis log file. The following example is an excerpt from the `diag.log1` file:

```
=====
Performing diagnosis with failure file datalogs/ff_100.log
Diagnosis will use 2 chain test patterns.
Check expected data completed: 196463 out of 196463 failures were
checked
Failures for COMPRESSOR patterns
-----
pattern  chain      pos#  output pin_names
-----
0        41         0      OUT_0   OUT_1   OUT_5
0        41         3      OUT_0   OUT_1   OUT_5
0        41         4      OUT_0   OUT_1   OUT_5
0        41         7      OUT_0   OUT_1   OUT_5
0        41         8      OUT_0   OUT_1   OUT_5
```

```

0      41      11      OUT_0    OUT_1    OUT_5
0      41      12      OUT_0    OUT_1    OUT_5
0      41      15      OUT_0    OUT_1    OUT_5
0      41      16      OUT_0    OUT_1    OUT_5
0      41      19      OUT_0    OUT_1    OUT_5
-----
datalogs/ff_100.log scan chain diagnosis results: #failing_
patterns=400
-----
defect type=stuck-at-1
match=100.00% chain=41 position=214 master=CORE_U1/vys2/U_L0/U_
FONTL/FF_pp1_reg_1_ (FSDX_1)
CPU_time=1.25 #sim_patterns=10 #sim_failures=5001
-----
YDF Candidates Schema with 0 entries retrieved. -----
Following physical data tables generated for all elements:
- YDF
CPU_time: 0.00 sec
Memory Usage: 0MB
-----
Performing diagnosis with failure file datalogs/ff_107.log
Diagnosis will use 2 chain test patterns.
Check expected data completed: 157974 out of 157974 failures were
checked
Failures for COMPRESSOR patterns
...

```

You can specify the `-level expert` option of the `set_messages` command to produce a parallel processing summary report for each core after the `run_diagnosis` command process is completed:

Process		Patterns	Time (s)			Memory (MB)			
ID	pid	External	CPU	Elapsed	Shared	Private	Total	Pattern	
0	3449	401	0.09	14.33	296.45	0.14	296.59	1.81	
1	3461	0	14.20	14.32	251.91	75.18	327.10	0.00	
2	3462	0	8.71	8.80	251.93	70.98	322.91	0.00	
3	3463	0	10.66	10.77	251.94	72.02	323.9	0.00	
4	3464	0	10.81	10.98	251.96	81.84	333.80	0.00	
Total		401	44.47	14.33	296.45	00.17	596.61	1.81	

Note: In the previous example, the total memory usage for parallel diagnostics is less than 600MB. However, running multiple TetraMAX sessions for the same diagnostics session requires almost 1.2GB.

Parallel Diagnostics Limitations

The `run_diagnosis` command checks out one Test-Diagnosis license key and one Test-Faultsim license key for each process enabled by the `set_diagnosis -num_processes` command. The `run_diagnosis` command issues a warning message if the number of specified failure data files is less than the number of enabled processes.

Note the following restrictions and limitations:

- If you specify more cores than the number of datalogs to be analyzed, the enhanced performance provided by parallel diagnostics is compromised because parallelization is applied to each datalog.
- For small designs you might not see a significant performance improvement, especially if diagnosis for a single datalog takes only a few seconds.

17

Using Physical Data for Diagnostics

Physical diagnostics provides significantly higher defect isolation accuracy and precision than standard scan diagnostics and improves the effectiveness of volume diagnostics.

Using the PHDS database, TetraMAX ATPG can dynamically extract likely bridging pairs, subnet information, and layout data for the diagnostics fault candidates during the diagnostics process. TetraMAX ATPG can also produce a dedicated physical diagnostics report that includes full physical descriptions of all diagnostics candidates organized by data type.

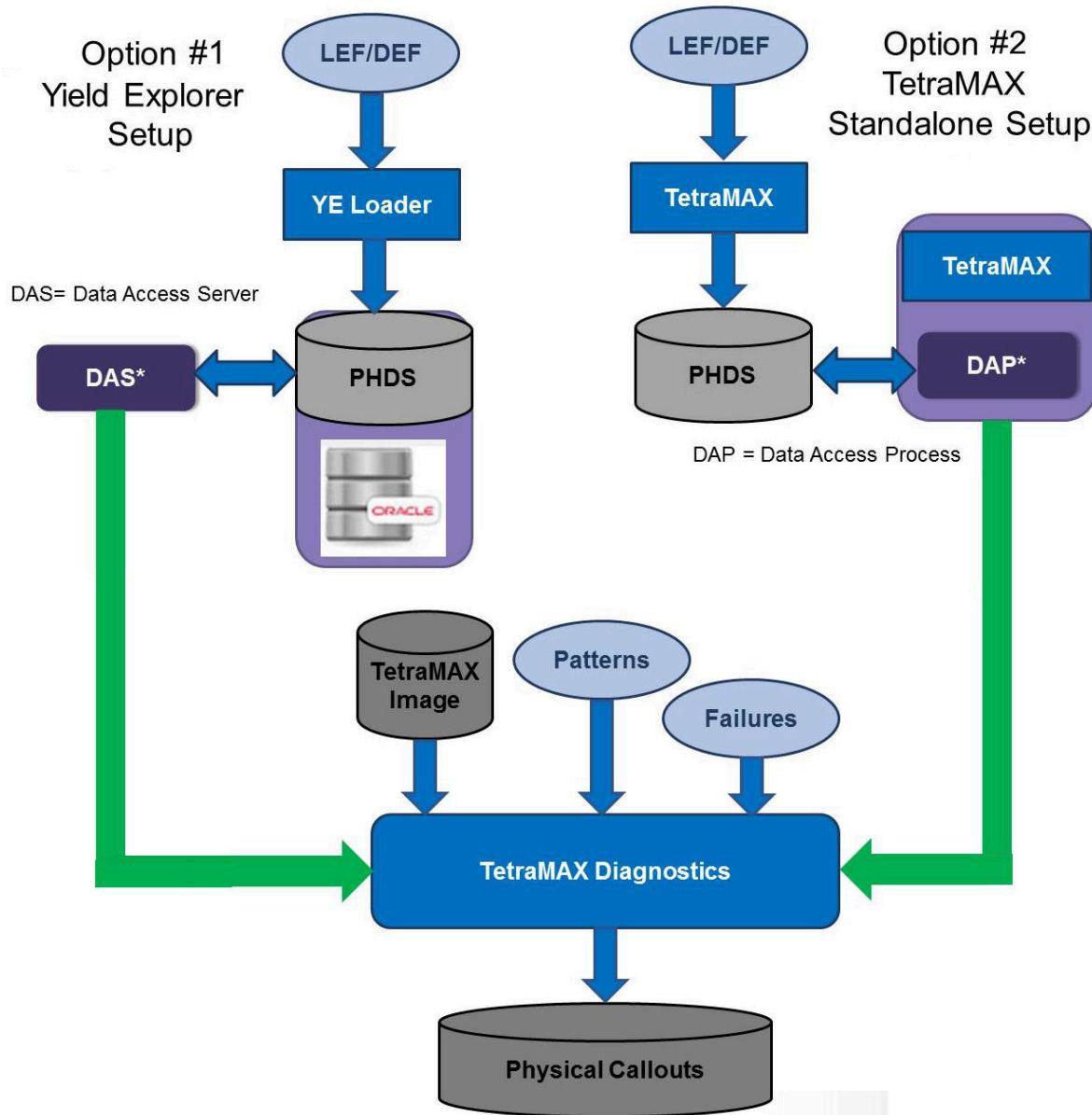
The following sections describe how to use physical data to perform diagnostics:

- [Physical Diagnostics Flow Overview](#)
- [Using TetraMAX to Create and Validate the PHDS Database](#)
- [Reading the PHDS Database](#)
- [Name Matching Using the PHDS Database](#)
- [Setting Up and Running Physical Diagnostics](#)
- [Static Subnet Extraction Using a PHDS Database](#)
- [Writing Physical Data for Yield Explorer](#)

Physical Diagnostics Flow Overview

You can use Yield Explorer or TetraMAX ATPG to create the PHDS database used for physical diagnostics. After loading the PHDS database into the data access process (DAP) server or the data access server (DAS), you can use TetraMAX ATPG to extract the physical information and perform diagnostics.

Figure 1 Flows for Using the PHDS Database for Physical Diagnostics



Note the following:

- For information on creating the PHDS database using Yield Explorer, see the *Yield Explorer User Guide*.
- For information on creating the PHDS database using TetraMAX, see the "[Using TetraMAX to Create and Validate the PHDS Database](#)" section.

See Also

[Reading the PHDS Database](#)

Using TetraMAX to Create and Validate the PHDS Database

The default flow for creating the PHDS database includes a validation run that reports a set of Y rules (described in TetraMAX Help). You can also create or validate the PHDS database in separate runs.

To create and validate the PHDS database, you need access to all the LEF/DEF files relevant to the partition of the design on which you are performing diagnostics. Although you can extract a physical database for an incomplete LEF/DEF set, you can only perform logical diagnostics for the blocks associated with any missing files. This is expected behavior in some cases, such as for memories or hardened IP.

A technology file (commonly referred to as a "techlef") is also usually required. This file contains a description of the physical properties of the design (metal layers, routing grid, etc.). The information in the technology LEF file is sometimes included directly in the design LEF files. If you include the string "tech" (case insensitive) in the name of the file (for example, "technology.lef" or "any.tech.lef.gz"), the file is automatically recognized as a technology file. You can also use the `set_physical_db -technology_lef_file` command to specify the technology file name.

Creating and Validating the PHDS Database

The following steps describe how to create and validate the PHDS Database:

1. Specify the locations of the LEF and DEF directories.

```
set_physical_db -lef_directory ./lef -def_directory ./def
```

The LEF and DEF directories can be merged, if necessary.

2. Specify the name of the top-level DEF file.

```
set_physical_db -top_def_file top_design.def
```

You can use any name for the `top_design.def` file.

3. Specify the location of the output PHDS directory.

```
set_physical_db -database ./phds
```

4. Specify the name of the design associated with the LEF/DEF database that you want to translate.

```
set_physical_db -device [list DES 4]
```

You can use any name regardless of the actual design name. The `-device` option also requires that you specify the device version (in the example, 4 is used).

5. Create and validate the PHDS database.

```
write_physical_db -replace -verbose
```

This command creates and validates the specified PHDS database (the `phds` directory is used in the example). You must use the `-replace` option to overwrite any previously loaded device with same name and version.

After the PHDS database is created, a confirmation message appears, as shown in the following example:

```
Writing Physical Database...
LEF input directory : ./lef
DEF input directory : ./def
Top DEF file name : top_design.def
PHDS output directory: ./phds
Device name : DES
Device version : 4
Running PHDS validation...
PHDS validation completed successfully.
-----
-----
Validation Summary Report
-----
-----
Warning: Rule Y18 (DEF Without Corresponding LEF) was violated 2 times.
There were 2 violations that occurred during Validation process.
Running PHDS creation...
PHDS creation completed successfully.
Total_time = 36.76
```

You can specify the `-verbose` option of the `write_physical_db` command to create the following detailed reports of the PHDS creation process:

- Log file from the LEF/DEF loader: `phds/server/log/di/trn`
- Log file from the PHDS loader: `phds/server/log/di/array`

The PHDS loader is a multithreaded process. By default, it uses four cores: one core is used for the main process and three cores are used to load a subset of the data.

Creating and Validating the PHDS Database in Separate Runs

You can specify the `-no_validation` option of the `set_physical_db` command to create the PHDS database in a separate run without validating it. You can also use the `-nocreate_phds` option of the `set_physical_db` command to only validate the PHDS database in a separate run.

The following example creates the PHDS database without validating it:

```
set_physical_db -lef_dir ./LEF -def_dir ./DEF
set_physical_db -top_def_file RISC_CHIP.def
set_physical_db -database PHDS_C
set_physical_db -device {"RISC" "1"}
set_physical_db -novid
write_physical_db
```

The following example validates the PHDS database in a separate run:

```
set_physical_db -lef_dir ./LEF -def_dir ./DEF
set_physical_db -top_def_file RISC_CHIP.def
set_physical_db -database PHDS_C
set_physical_db -device {"RISC" "1"}
set_physical_db -nocreate_phds
write_physical_db
```

See Also

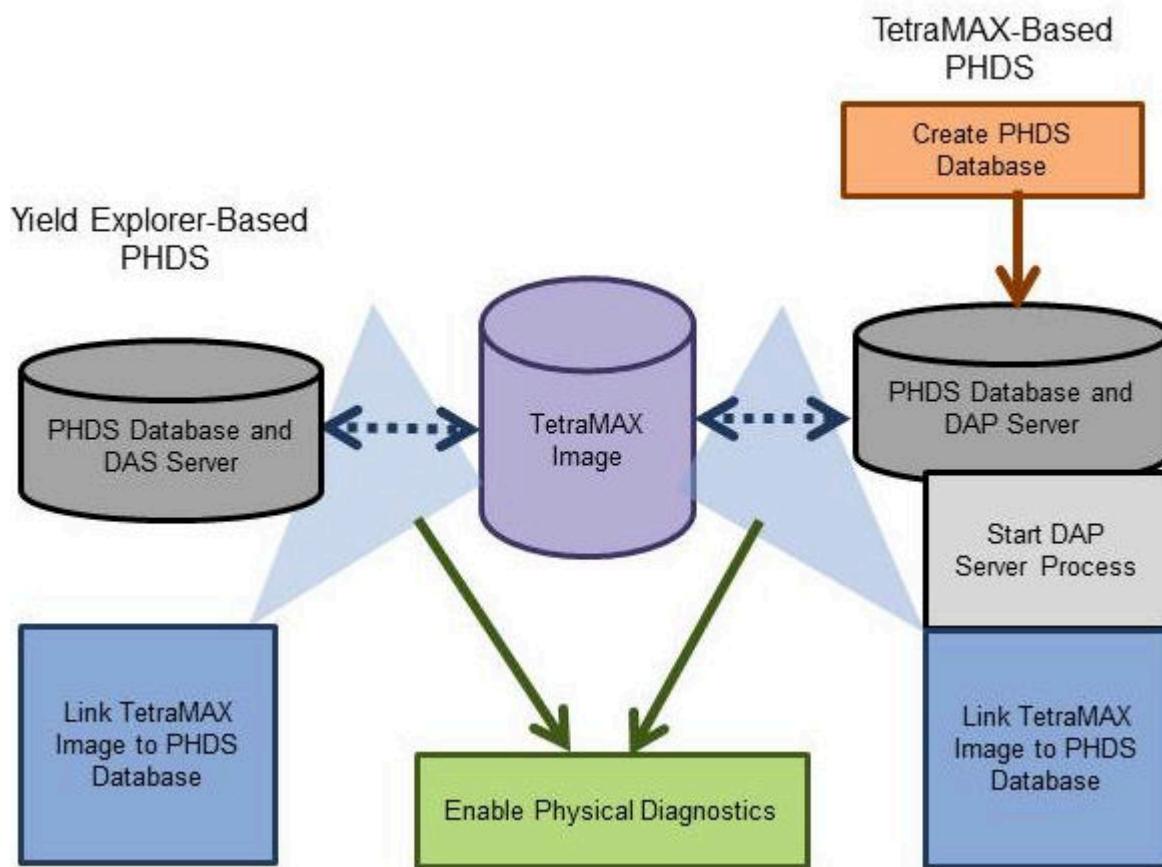
[Physical Diagnostics Flow Overview](#)
[Reading a PHDS Database](#)
[Setting Up and Running Physical Diagnostics](#)

Reading a PHDS Database

TetraMAX ATPG can use a PHDS database to perform dynamic extraction of likely bridging pairs, subnet information, and layout data for the diagnostic fault candidates.

This section describes how to set up a PHDS database for use in TetraMAX ATPG. The initial setup steps for the flow are different depending on whether you use Yield Explorer or TetraMAX ATPG to create a PHDS database. [Figure 2](#) shows how to read a PHDS database.

Figure 2: Reading a PHDS Database for Physical Diagnosis



The following sections describe how to use TetraMAX ATPG to read a PHDS database for physical diagnosis:

- [Starting and Stopping the DAP Server Process](#)
- [Setting Up a Connection to the PHDS Database](#)

See Also

[Using TetraMAX to Create a PHDS Database](#)
[Setting Up and Running Physical Diagnostics](#)

Starting and Stopping the DAP Server Process

The DAP (Data Access Process) server process is used to access a PHDS database created by TetraMAX ATPG. You must start this process before you can access and query a PHDS database created by TetraMAX ATPG.

A PHDS database created by Yield Explorer is stored on a DAS (Data Access Server) server. In this case, you do not need to start the DAP server process. Instead, your next step is establish a connection to the PHDS database (see the "[Setting Up a Connection to the PHDS Database](#)" section).

To start the DAP server process:

1. Identify the location of the PHDS database.

```
set_physical_db -database ./phds
```

2. Specify any available port on the host in which TetraMAX ATPG is currently running.

```
set_physical_db -port_number 9990
```

The default, 9998, is used if this command is not specified.

3. Start the data access process.

```
open_physical_db
```

When the DAP server process starts, the following message prints:

```
Starting Data Access Process...
Hostname : ighost101
Port Number : 9990
Physical Database Directory: ./phds
Successfully started Data Access Process.
```

If the process is already running, you will see the following message:

```
Starting Data Access Process...
Hostname : ighost101
Port Number : 9990
Physical Database Directory: ./phds
Data Access Process is already running.
```

To stop the DAP server process, specify the `close_physical_db` command:

```
BUILD-T> close_physical_db
Stopping Data Access Process...
Hostname : ighost101
Port Number : 9990
All kernel objects removed. Exiting the process...
Successfully stopped Data Access Process.
```

The PHDS database created from TetraMAX ATPG uses the DAPListener process. To keep the server process alive, make sure you exit the current TetraMAX session. The DAPListener process halts if TetraMAX ATPG runs in the background. You can perform this operation from any TetraMAX mode (BUILD, DRC or TEST).

Setting Up a Connection to the PHDS Database

Before performing physical diagnostics, you must establish a connection between the TetraMAX logical image and the PHDS database. To create this link:

1. Make sure the appropriate design image is loaded in DRC or TEST mode in your current TetraMAX session.

```
TEST-T> read_image i044_image.dat
```

2. Use the `set_physical_db` command to identify the hostname and port number of the PHDS server containing the PHDS database.

```
TEST-T> set_physical_db -hostname ighost101 -port_number 9990
      Setting host name ('ighost101') for physical connection.
      Setting port number ('9990') for physical connection.
      Connecting to physical database.
      Successfully connected to physical database.
      Available Devices:
-----
DES 1
DES 2
DES 3
DES 4
TST 1
```

If the connection is successful, a list of available devices is printed, as shown in the example.

Note: You should always specify the port number when connecting to an existing DAP. The default is not used in this case.

3. If you are using an Oracle-based PHDS database created by Yield Explorer, you must include a user name and password to establish a connection.

```
TEST-T> set_physical_db -hostname ighost101 \
      -port_number 9990 -user tester -password safe1234
      Setting user name ('tester') for physical connection.
      Setting password ('safe1234') for physical connection.
      Setting host name ('ighost101') for physical connection.
      Setting port number ('9990') for physical connection.
      Connecting to physical database.
      Successfully connected to physical database.
      Available Devices:
-----
DES 1
DES 2
DES 3
DES 4
TST 1
```

4. Use the `-device` option of the `set_physical_db` command to specify the current device and version.

```
TEST-T> set_physical_db -device "DES 4"
Connecting to physical database.
Successfully connected to physical database.
Setting device name ('DES') and device version ('4') for
physical connection.
```

5. Use the `-top_design` option of the `set_physical_db` command to specify the top-level DEF design name.

```
TEST-T> set_physical_db -top_design top_def_design_name
```

Name Matching Using a PHDS Database

TetraMAX diagnostics uses physical information mapped to logical instances to improve the accuracy and precision of diagnosis callouts. After diagnosis, TetraMAX writes the physical information for the diagnosis candidates for use in physical failure analysis. This process can be compromised due to logical pin names mismatching with the corresponding physical names in the LEF/DEF database.

To resolve instance name conflicts, matching should be performed on the logical names from the Verilog netlist to the physical names in the LEF/DEF database before running diagnosis. If the logical names and physical names match, name matching rules will be created for later use in diagnosis. The following sections describe how to perform name matching for all instance pins using a PHDS database:

- [Name Matching Overview](#)
- [Understanding the Name Matching Coverage Report](#)
- [Reporting the Name Matching Coverage](#)
- [Using Name Matching for Diagnostics](#)

Name Matching Overview

For standard physical diagnosis, TetraMAX diagnostics dynamically searches a PHDS database and matches logical candidate instance names with the corresponding physical names using existing name matching rules. The name matching feature performs this name matching process before running diagnosis. You can then create a set of rules to resolve name mismatches in subsequent diagnosis runs.

To perform name matching, you use both the `set_match_names` and `match_names` commands. The `set_match_names` command specifies the name matching rules you want to apply, if any, and the `match_names` command prints a report of the name matching coverage.

For example, you can use the `match_names` command to create an initial name matching report for a subset of instances. Next, you can use the `set_match_names` command to specify the replacement of a specific instance prefix with another instance prefix from the

flattened logical instance names. You can then perform name matching again to generate a final report that uses the preferred instance prefix.

The name matching report includes pin-level analysis data, hierarchical mismatch behavior, and a name match summary.

Understanding the Name Matching Coverage Report

The `match_names` command creates a name matching report that you can use to analyze the matching of logical candidate instance names with the corresponding physical names.

The following example shows sample output for the `match_names` command:

```
TEST-T> match_names
Setting top level physical design name to 'RISC_CHIP'
  Performing Pin Level Analysis
    Matched 564 of 884 instance pins
  Checking for logical wrapper
  Checking for physical wrapper
  Checking for differences in the lowest hierarchy levels
  Performing Hierarchy Level Analysis
    Module Inst Count Matched Unmatched Names
    -----
    STACK_TOP          1 0           1        320
    -----
Name Match Summary
-----
Number of instance names matched: 564
Number of mismatches found: 320
Percent Correct = 63.80%
CPU_time: 0.02 sec
Query_time: 2.61 sec
Total_time: 2.62 sec
Memory usage summary: 0MB
-----
Closing connection to physical database.
```

Note the following sections of the sample report:

- **Performing Pin Level Analysis** — TetraMAX diagnostics attempts to map every pin in the design to its logical equivalent, and displays the total results.
- **Checking for logical and physical wrappers** — For the logical and physical wrappers, TetraMAX diagnostics attempts to find the top-level hierarchies to explain mismatch behavior.
- **Checking for differences in the lowest hierarchy levels** — TetraMAX diagnostics attempts to explain mismatches at the lowest level hierarchies. The module level results are displayed in descending order relative to the highest number of unmatched names found. After reviewing this report, you should specify a series of `set_match_names`

commands to find the correct match for each name.

- **Name Match Summary** — This section summarizes the name matching results. It contains the following fields:
 - **Number of instance names matched** — indicates the number of logical names for which an instance can be found.
 - **Number of mismatches found** — indicates the number of logical names for which no match was found.
 - **Percent Correct** — indicates the final coverage of the name matching process

You can also use the `-auto` option of the `set_match_names` command to perform automatic name matching to resolve hierarchy conflicts.

Reporting the Name Matching Coverage

TetraMAX diagnostics creates a coverage report that displays the success of the static name matching process between the logical and physical names. The flow for this process is as follows:

1. Start TetraMAX.

For details, see "[Invoking TetraMAX](#)."

2. Read the design image.

```
read_image design.img.gz
```

3. Connect to an existing PHDS database.

```
set_physical_db -hostname host01 -port_number 9998  
set_physical_db -top_design top_design_name  
set_physical_db -device [list "Device_name" "1"]
```

4. Use the `match_names` command to perform name matching for a subset of the instances.

```
match_names -sample 1
```

This command reports name matching for 1% of the logical names.

5. Use the `set_match_names` command to specify the name matching rules, if needed.

```
set_match_names -sub_prefix [list "dut/" ""]
```

Note that the physical names include an extra level of hierarchy (`dut/`). The `set_match_names` command removes this string from the physical names and finds a match with the logical names.

6. Perform name matching again to get the final report.

```
match_names -sample 1
```

Using Name Matching Results for Diagnostics

You can use the name matching flow to identify logical to physical naming conflicts and create appropriate name matching rules that you can use later in diagnostics. The flow consists of the following steps:

1. Start TetraMAX.

For details, see "[Invoking TetraMAX](#)."

2. Read the design image.

```
read_image design.img.gz
```

3. Connect to an existing PHDS database.

```
set_physical_db -hostname host01 -port_number 9998  
set_physical_db -top_design top_design_name  
set_physical_db -device [list "Device_name" "1"]
```

4. Use the following command to automatically create the name matching rules (optional).

```
set_match_names -auto
```

5. Use the `match_names` command to perform name matching for all instances:

```
match_names
```

6. Use the following command to view the automatically created match name rules (optional):

```
report_settings match_names
```

7. Based on the remaining mismatches, use the `set_match_names` command to specify the name matching rules , then rerun the `match_names` command, as shown in Step 5..

```
set_match_names -sub_str [list "dut_0/" "DUT0/"]
```

8. Restart TetraMAX.

9. Read the design image.

```
read_image design.mapped.img.gz
```

10. Connect to an existing PHDS database.

```
set_physical_db -hostname host01 -port_number 9998  
set_physical_db -top_design top_design_name  
set_physical_db -device [list "Device_name" "1"]
```

11. Read the patterns into an external buffer.

```
set_patterns -external design.pat.bin.gz
```

12. Define the name matching rules. For example:

```
set_match_names -sub_prefix {top_i i_core}
```

13. Run the diagnosis

```
run_diagnosis design.datalogs
```

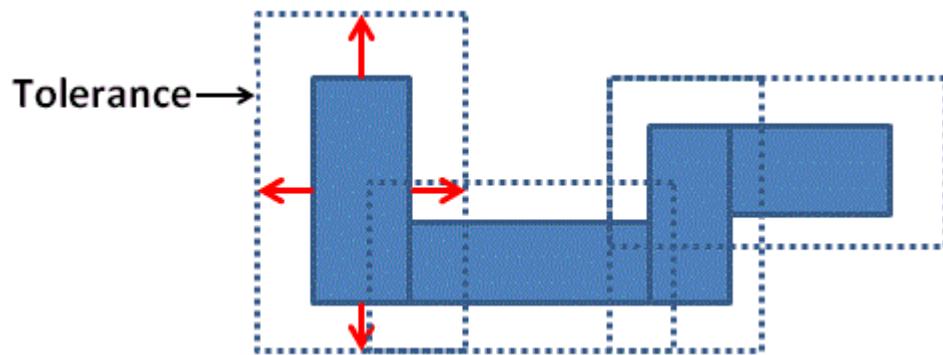
Setting Up and Running Physical Diagnostics

To perform physical diagnostics, you first need to extract the physical data structures from the PHDS database. You can then perform diagnostics using the `run_diagnosis` command.

You can use the `set_physical -tolerance` command and the `set_physical_db -device` command to specify a series of parameters for extracting specific types of data from the PHDS database.

When extracting bridges, TetraMAX ATPG searches and extracts neighbor nets based on a default distance per layer tolerance. This tolerance is measured from the boundary of the net, as shown in [Figure 1](#).

Figure 1: Net Tolerance for Bridge Extraction



The tolerance distance is equal to the pitch if this data exists in the technology information. If not, the default is 1000nm. You can determine the appropriate tolerance by analyzing technology data, such as pitch distance. To set a tolerance level, use the `-tolerance` option of the `set_physical` command.

Running Physical Diagnostics

The following steps describe how to set the extraction parameters from the PHDS database, extract physical data, run physical diagnostics, and write the physical data for Yield Explorer:

1. Use the `set_physical_db -device` command to query the PHDS database for technology information, including routing layers and tolerances for each layer.

```
set_physical_db -device [list "RISC" "1"]
Connecting to physical database.
Successfully connected to physical database.
Setting device name ('RISC') and device version ('1') for
physical connection.
Retrieving layers and tolerance values for device ('RISC')
and device version ('1')
Layer Tolerance
```

----- -----

```
METAL 410
METAL2 410
METAL3 410
METAL4 515
METAL5 810
METAL6 970
```

2. If required, use the `set_physical -tolerance` command to specify a tolerance for extracting neighbor nets for specific layers. Use the Tcl list syntax to specify each layer and its tolerance setting, as shown in following example:

```
set_physical -tolerance [list METAL 50 METAL2 100 METAL3 200 \
    METAL4 300 METAL5 400]
```

3. Perform physical diagnosis on the PHDS database using the `run_diagnosis` command. For example:

```
run_diagnosis /project/mars/lander/chipA_failure.dat
```

4. Use the `write_ydf` command to write the physical data, as shown in the following example:

```
write_ydf chipA.ydf -candidate -append
```

When running physical diagnostics, TetraMAX ATPG dynamically retrieves the physical data based on the instance names. If a match exists, TetraMAX ATPG accesses the physical data. You can also perform match naming using the physical IDs created before running diagnostics. For more information on this process, see "[Static Subnet Extraction Using a PHDS Database](#)".

See Also

[Using TetraMAX to Create a PHDS Database](#)

[Reading a PHDS Database](#)

Static Subnet Extraction Using a PHDS Database

You can improve the runtime for physical diagnostics by statically extracting subnet information from a PHDS database before running diagnostics. The default flow, dynamic subnet extraction, is performed during diagnostics. Static subnet extraction is only recommended when you run volume diagnostics on a large number of failing parts. Otherwise, the additional runtime required for static extraction is greater than the total reduction in diagnosis runtimes.

The static subnet extraction flow consists of the following steps:

1. Start TetraMAX.
For details, see "[Starting TetraMAX](#)".
2. Read the logical image.
`read_image original.img.gz`
3. Connect to an existing PHDS database.

```
set_physical_db -hostname host01 -port_number 9998
set_physical_db -top_design top_design_name
set_physical_db -device [list "Device_name" "1"]
```

4. Perform extraction of all subnet information. At the end of the extraction process, all subnet data is saved in the TetraMAX database.

```
extract_nets -all
```

Only driver pins with more than two fanouts are extracted since they are the only pins with subnets. Subnets from driver pins are extracted in groups of 500 to minimize server overloading.

5. Report statistics, as needed, for the extracted subnets (optional).

The following example reports statistics for a design in which 286 nets have a subnet:

```
TEST-T> report_layout -summary
Subnets : #nets=286, #subnets=1191, max_subnets=51, memory=0MB
Subnets_distribution: <10(88.46%) <20(98.25%) <30(98.95%) <50
(99.65%)
<60(100.00%)
Receivers_per_net: <10(84.62%) <20(97.90%) <30(98.95%) <50
(99.65%)
<60(100.00%)
```

6. Write the physical image containing the subnet information.

```
write_image new.img.gz -compress gzip -replace
```

7. Exit TetraMAX.

```
exit
```

8. Start TetraMAX.

For details, see "[Starting TetraMAX](#)."

9. Read the physical image.

```
read_image new.img.gz
```

10. Connect to an existing PHDS database.

```
set_physical_db -hostname host01 -port_number 9998
set_physical_db -top_design top_design_name
set_physical_db -device [list "Device_name" "1"]
```

11. Set diagnostics to query only candidate physical information and bridging information.

```
set_diagnosis -use_phds [list candidates bridges]
```

12. Perform Diagnostics.

```
run_diagnosis fail.log
```

13. Exit TetraMAX.

```
exit
```

Reporting Physical Subnet ID Data

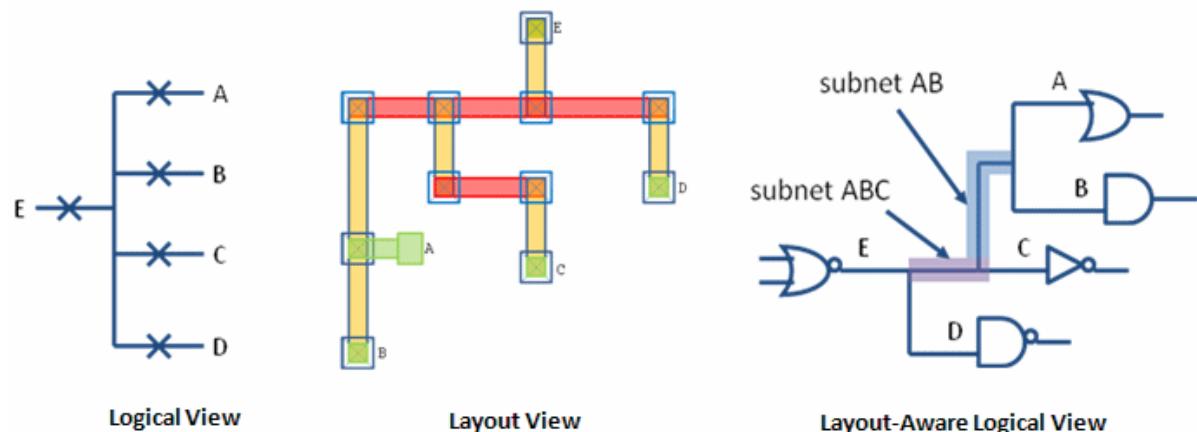
You can use TetraMAX diagnostics with a PHDS database to extract the physical structure of nets in a design. TetraMAX diagnostics uses this subnet data for any net that has more than two fanouts. When diagnostics is performed using the subnet data, the open fault candidates localized on a subnet are reported with the physical ID of the subnet when using the class-based candidate organization.

TetraMAX diagnostics normally uses the PHDS database of the design to extract the subnet data. TetraMAX reads in the extracted subnet data and maps the physical subnet to the logical image. If the subnet is an intermediate branch of the net, it is assigned a logical subnet ID. All branches of the net, including stem branches connected to the driver and receivers, are assigned a physical subnet ID.

The ability to display this type of information improves the precision of the diagnostics results because the physical failure analysis (PFA) area encompasses a much smaller portion of the net.

Understanding Physical Subnet ID Data

The following figure shows the logical, layout, and layout-aware logical views of the same net.



If a fault is localized in subnet ABC, it can be described by the actual subnet definition, and the subnet ID is reported in the diagnostics report. However, if a fault is localized in the metal segments of fanout C, the actual subnet definition is not sufficient. However, the physical subnet ID can address this situation.

The physical subnet ID begins to count from 0, starting with the driver. This means that all metal segments that belong to the driver side of the net are assigned the physical subnet ID of 0. The first fanout metal segments (in this case A), are assigned a physical subnet ID of 1. The physical subnet ID assignment process continues successively until all portions of the net, including the subnets, have received an ID.

The subnet definition of the net displayed in [Figure 1](#) is described in the following table. The left column contains the regular subnet definition. The middle column contains the logical subnet IDs. The right column displays the physical subnet IDs.

Subnet Definition	Logical Subnet IDs	Physical Subnet IDs
.net		
dut/impl/hier3/U12/E -----		0
dut/impl/hier3/U27/A -----		1
dut/impl/hier3/U21/B -----		2
dut/impl/hier3/U72/C -----		3
dut/impl/hier3/U9/D -----		4
.subnets		
1 2 3 1 5		
1 2 2 6		

Note the following:

- The physical subnet IDs are the same IDs reported in the Yield Explorer data file (YDF). If a net does not have a subnet definition, the physical subnet ID cannot be displayed.
- TetraMAX ATPG reports the physical subnet ID to match the contents of the YDF. When using the fault-based candidate organization, this can be enabled with the `set_diagnosis -show physical_subnet_id` command. This option is not necessary with the class-based organization.
- Physical subnet ID data is reported only for nets that have been extracted successfully and are in the subnet file. This means that nets with only one or two fanouts are not included. Physical subnet IDs for nets with three or more fanouts that cannot be extracted are also not reported.
- When the fault candidates are cell inputs, and a physical subnet ID must be reported, TetraMAX ATPG changes the instance name of the cell input to the instance of the cell which is driving the cell input.

Writing Physical Data for Yield Explorer

After completing the physical diagnostics process, you can write the diagnostics candidates, and all related physical information, to a Yield Explorer Data Format (YDF) file. Yield Explorer uses this file for volume diagnostics analysis.

To write physical data for Yield Explorer, specify the `write_ydf` command after each `run_diagnosis` command. The `write_ydf` command must include the name of the YDF file. You can specify this command using the `-candidates` option or without any options. For example:

```
run_diagnosis /project/mars/lander/chipA_failure.dat
write_ydf top_chip.ydf -candidates
```

The `write_ydf` command prints the physical data for the diagnostics candidates in all tables in the output YDF file. This data enables Yield Explorer to perform volume diagnostics analysis. You should use the `-replace` option if you want to create a new file to store each diagnostics candidate. You can use the `-append` option if you want to store all candidates in a single file. The following example reports the physical data elements for the diagnostics candidates to a single file:

```
write_ydf top_chip.ydf -candidates -append
```

By default, the `write_ydf` command does not write the chain definition table. To write this table, which displays chain definition data for the entire database, you must specify the `-chain_def` option. You only need to write this table one time, as shown in the following example:

```
write_ydf top_chip.chain_def.ydf -chain_def
```

You might prefer to write individual tables to a specific file. If you specify one or more physical data options, TetraMAX ATPG reports only the physical data tables related to the specified options.

If you specify the following example, TetraMAX ATPG reports only the physical data for the vias and the LEF macro cells in their respective tables:

```
write_ydf top_chip.ydf -replace -via -cell
```

See Also

[Using TetraMAX to Create a PHDS Database](#)
[Reading a PHDS Database](#)

18

Power Aware ATPG

A typical ATPG run targets as many faults as possible within a particular pattern. However, this approach can cause unintended ATE failures for designs containing a large number of flip-flops that toggle at any given time.

The TetraMAX power aware ATPG feature calculates the fanout of clock-gating structures and other clock sources during DRC. This approach enables you to specify capture and shift power budgets for generating power aware ATPG vectors. You can specify a budget as a percentage of scannable flip-flops and thereby limit the number of flip-flops that can toggle.

TetraMAX ATPG lowers the overall peak and average flip-flop switching by selectively turning on and off the respective clock-gating cells which control the flip-flops. This selective switching affects capture for stuck-at testing and launch and capture for transition fault testing.

Power aware ATPG is not intended to be used for power analysis. TetraMAX ATPG efficiently estimates the relative power of test patterns, which generally correlates well with actual power consumption. However, this approach is not a precise calculation of the actual power metrics. Performing a full power analysis during ATPG causes an unacceptable increase in runtime and is therefore not used for power aware ATPG.

The following sections describe how to prepare for and use power aware ATPG:

- [Input Data Requirements](#)
- [Setting a Power Budget](#)
- [Preparing Your Design](#)
- [Running Power Aware ATPG](#)
- [Applying Quiet Chain Test Patterns](#)
- [Testing with Asynchronous Primary Inputs](#)
- [Power Reporting By Clock Domain](#)
- [Setting a Capture Budget for Individual Clocks](#)
- [Retention Cell Testing](#)
- [Limitations](#)

Input Data Requirements

The following input data is required to use the power aware ATPG feature within TetraMAX ATPG:

- Netlists
- Library
- STIL procedure file
- Tcl command script containing the `build`, `run_drc`, `run_atpg` and other commands.

Setting a Power Budget

To run power aware ATPG, you need to set a power switching budget using the `-power_budget` option of the `set_atpg` command. You can specify the power switching budget using either of the following methods:

- Specify the maximum percentage of scannable flip-flops that are budgeted to change during capture. For example:

```
set_atpg -power_budget 48
```

- Specify the `min` keyword to use the minimum recommended switching budget based upon the clock-gating analysis. For example:

```
set_atpg -power_budget min
```

You can set the power switching budget any time before running the `run_atpg` command.

For complete information on how to determine the power switching budget, see the following section, "[Preparing Your Design](#)."

Preparing Your Design

Power aware ATPG is intended for designs that contain clock-gating cells in the context of ATPG. You will need to perform an initial analysis of your design to identify all clock-gating cells and calculate the recommended setting for the `set_atpg -power_budget` command.

The power analysis performed by TetraMAX ATPG uses information from the STL procedure file and data specified by the `add_pi_constraints` command. If your design has a constraint in which the clock-gating cells are always transparent, this power analysis will not show these clock-gating cells and they are not usable within the context of power aware ATPG. This means you need to constrain scan-enable ports to their respective off-state for basic-scan, two-clock, and fast-sequential modes for test pattern generation and gated-clock (latch) identification.

Also note the following:

- All global signals capable of enabling a large proportion of the clock gating cells must be disabled.
- All synchronous set and reset signals described as clocks with TetraMAX ATPG must be inactive or constrained to their respective off-state.

Reporting Clock-Gating Cells

After your design successfully passes the DRC process, use the `report_clocks -gating -verbose` command to report the clock-gating cells and calculate the recommended low-power ATPG budget percentage, as shown in the following example:

```
report_clocks -gating -verbose
Clock name: ife_clockdiv2_afe_wrap (0)
Number of cells directly controlled by the clock: 12077 (22.33%)

Number of cells controlled by clock through
a clock gating latch16605 (30.70%)

Number of cells directly controlled by clock + largest
clock gating domain: 12097 (22.36%)

Clock Gating Latch DecoderFrontEnd1/fedcod/dcqd_yc/ \
clk_gate_ramAddr_regx0x/U1 (693893)
drives 20 (0.04%) scan cells
...
...
Minimum Recommended Low-Power ATPG Budget: 22.36% (12097)
```

You should round-up the recommended low-power ATPG budget percentage to the next integer value. In the previous example, 22.36 should be rounded up to 23. This value is specified by the `-power_budget` option of the `set_atpg` command using either of the two methods:

- You can manually specify the budget as shown in the following example:
`set_atpg -power_budget 23`
- You can specify TetraMAX ATPG to automatically use the minimum recommended low-power ATPG budget, as shown in the following example:
`set_atpg -power_budget min`

Setting a Strict Power Budget

Use the `-power_effort` option of the `set_atpg` command to generate patterns that do not exceed the power budget specified for capture. The syntax for this option is as follows:

```
set_atpg -power_effort <high | low>
```

The default is `low`. If you set this option to `high`, TetraMAX ATPG generates patterns that do not exceed the budget specified by the `set_atpg -power_budget` command. Note that over-constraining the power budget might cause longer runtimes and generate fewer patterns when the `-power_effort` option is set to `high`. Because of this, you should not set power budgets below that recommended by the `report_clocks -gating` command.

Setting Toggle Weights

Using toggle weighting, you can place a 1 or higher integer number onto flip-flops to represent larger fan-out nets and cones of logic. You can specify separate weights for both shift and capture. This feature is specified using the `set_toggle_weights` command, which must be specified before issuing the `run_drc` command.

By default, each toggle or transition at a flip-flop is scored as a 1. This scoring only considers the flip-flop and does not take into account the fan-out of the flip-flop.

The following example shows a typical specification of the `set_toggle_weights` command:

```
set_toggle_weights path/to/spec/FF -weight 5 -shift -capture
```

You can also use the `report_toggle_weights` command to print a report of all non-default toggle weights you have applied, as shown in the following example:

```
report_toggle_weights
  Non-default Toggle Weights:
    Shift Weights:
      a_reg_2_: 5
    Capture Weights:
      a_reg_2_: 5
```

You can create a Tcl file comprised of numerous calls to the `set_toggle_weights` command and a `report_toggle_weights` command, as shown in the following example:

```
set_toggle_weights core1/ff1 -weight 5 -shift -capture
set_toggle_weights core1/ff2 -weight 2 -shift
set_toggle_weights core1/ff2 -weight 7 -capture
report_toggle_weights
```

Source this file, as shown in the following example:

```
source toggle_weight.tcl
```

Running Power Aware ATPG

After preparing your design, as described in the "[Preparing Your Design](#)" section, you are ready to perform a complete power aware ATPG run and use the `report_power` command to report the power data.

The following example script shows the use of the `set_atpg` and `report_power` commands in the complete power aware ATPG flow.

```
read_netlist -lib $spec_lib.v
read_netlist $spec_design.v
```

```

run_build_model $spec_design
add_pi_constraints 0 scan_enable
run_drc $spec_drc_file.spf
report_clocks -gating
set_atpg -power_budget 10
add_faults -all
run_atpg -auto
report_power -per_pattern -percentage

```

The `report_power` command produces the report shown in the following example:

```

-----
          Power Analysis Summary
-----
Number of Scan Cells      75053
Number of Patterns        0-2680
Average Shift Changes:   2400.38 3.20%
Average Capture Changes: 9058.04 12.07%
Maximum Shift Cell Changes: 37510 49.97% (pattern: 0 cycle: 3411)
Maximum Capture Cell Changes: 30742 40.96% (pattern: 1)

```

Applying Quiet Chain Test Patterns

Regular scan chain test patterns apply the 0011 sequence to all scan inputs, which can lead to power issues and unintended ATE failures. However, a quiet chain test pattern minimizes switching activity by loading a single scan input with specified pattern data and loads all other chains with a constant value.

The `-quiet_chain_test` option of the `set_atpg` command enables the automatic generation of quiet chain test patterns when the `run_atpg` command is executed.

The `set_atpg -load_mode` command enables the generation of the quiet chain test for a particular load mode or all load modes. The `set_atpg -load_value` command configures the constant value loaded into the quiet chains.

In legacy scan mode, the 0011 sequence, or any other specified pattern data, is independently applied to each scan chain, while all other scan chains are set to 0. This means that one pattern loads the 0011 sequence in a single scan chain at a time. To load N scan chains, where N is the total number of scan chains, TetraMAX ATPG generates N quiet chain test patterns.

In scan compression mode, the 0011 sequence or any other specified pattern data is independently applied to each scan channel, while all other scan channels are set to 0. The

compressor load mode is maintained to a constant value, which is 0. One scan channel fans to multiple chains due to the input load compressor. Thus, to load the P scan channels, where P is the total number of load compressor scan inputs, TetraMAX ATPG generates P quiet chain test patterns.

Testing with Asynchronous Primary Inputs

Use the `-power_aware_asyncs` option of the `set_atpg` command to test asynchronous sets and resets from primary inputs on legacy scan designs. Note that this feature is not implemented for DFTMAX designs.

To use this option, your design must be able to propagate the asynchronous signals to allow sufficient time for the signals to fit within the given ATE vector.

The following example shows the `-power_aware_asyncs` option of the power aware ATPG flow:

```
...
run_drc
...
set_atpg -power_aware_asyncs
run_atpg -auto
report_power -per_pattern -percentage
```

Power Reporting By Clock Domain

You can set the `-per_clock_domain` option of the `report_power` command to create individual capture power reports for each clock. By default, the `report_power` command creates a consolidated report for all clock domains.

The following example shows the type of report created using the `-per_clock_domain` option.

```
report_power -percentage -per_pattern -per_clock_domain
-----
          Power Analysis: Per Pattern
-----
Shift Results:
 Peak
  pattern    load cycle    shift cycle      switching      percentage
  0           0             87            344        48.93%
  1           0             35            361        51.35%
  2           0              5            341        48.51%
  3           0             80            351        49.93%
  4           0             11            337        47.94%
  5           0             23            343        48.79%
  6           0             64            340        48.36%
  7           0             75            361        51.35%
```

8	0	6	365	51.92%
9	0	48	348	49.50%
Average				
pattern	average	switching	percentage	
0		171.51	24.40%	
1		348.91	49.63%	
2		326.01	46.37%	
3		338.85	48.20%	
4		327.02	46.52%	
5		328.31	46.70%	
6		328.89	46.78%	
7		343.67	48.89%	
8		349.93	49.78%	
9		339.20	48.25%	

Capture Results:**Peak**

pattern	capture	cycle	switching	percentage
0		0	0	0.00%
1		2	68	9.67%
2		1	52	7.40%
3		1	54	7.68%
4		1	25	3.56%
5		1	16	2.28%
6		1	30	4.27%
7		1	26	3.70%
8		1	47	6.69%
9		1	60	8.53%

Average

pattern	average	switching	percentage
0		0.00	0.00%
1		41.67	5.93%
2		27.33	3.89%
3		30.33	4.31%
4		13.33	1.90%
5		7.00	1.00%
6		13.00	1.85%
7		14.00	1.99%
8		26.00	3.70%
9		32.67	4.65%

Capture Results For Clock CLK1:**Peak**

pattern	capture	cycle	switching	percentage
0		0	0	0.00%
1		0	0	0.00%
2		0	0	0.00%
3		0	0	0.00%
4		0	0	0.00%
5		0	0	0.00%
6		0	0	0.00%

7	0	0	0.00%
8	1	19	2.70%
9	0	0	0.00%
Average			
pattern	average	switching	percentage
0		0.00	0.00%
1		0.00	0.00%
2		0.00	0.00%
3		0.00	0.00%
4		0.00	0.00%
5		0.00	0.00%
6		0.00	0.00%
7		0.00	0.00%
8		12.33	1.75%
9		0.00	0.00%

Capture Results For Clock CLK2:

Peak			
pattern	capture	cycle	switching
0		0	0
1		0	0
2		1	32
3		0	0
4		0	0
5		0	0
6		0	0
7		1	26
8		0	0
9		0	0

Average			
pattern	average	switching	percentage
0		0.00	0.00%
1		0.00	0.00%
2		14.00	1.99%
3		0.00	0.00%
4		0.00	0.00%
5		0.00	0.00%
6		0.00	0.00%
7		14.00	1.99%
8		0.00	0.00%
9		0.00	0.00%

Capture Results For Clock CLK3:

Peak			
pattern	capture	cycle	switching
0		0	0
1		1	36
2		0	0
3		1	23
4		1	25
5		1	16

6	1	30	4.27%
7	0	0	0.00%
8	0	0	0.00%
9	1	23	3.27%
Average			
pattern	average switching	percentage	
0	0.00	0.00%	
1	22.67	3.22%	
2	0.00	0.00%	
3	13.00	1.85%	
4	13.33	1.90%	
5	7.00	1.00%	
6	13.00	1.85%	
7	0.00	0.00%	
8	0.00	0.00%	
9	11.33	1.61%	
...			
...			
...			
<hr/>			
Power Analysis Summary			
<hr/>			
Number of Scan Cells	703		
Number of Patterns	0-9		
Cycles Per Load	88		
Average Shift Switching	320.23	45.55%	
Average Capture Switching	22.00	3.13%	
Peak Shift Switching	365	51.92%	
(pattern: 8 cycle: 6)			
Peak Capture Switching	68	9.67%	
(pattern: 1)			
Peak Capture Switching (CLK1)	19	2.70%	
(pattern: 8)			
Peak Capture Switching (CLK2)	32	4.55%	
(pattern: 2)			
Peak Capture Switching (CLK3)	36	5.12%	
(pattern: 1)			
Peak Capture Switching (CLK4)	0	0.00%	
(pattern: 0)			
Peak Capture Switching (CLK5)	0	0.00%	
(pattern: 0)			
Peak Capture Switching (CLK6)	31	4.41%	
(pattern: 3)			
Peak Capture Switching (CLK7)	37	5.26%	
(pattern: 9)			
Peak Capture Switching (CLK8)	0	0.00%	
(pattern: 0)			

Peak Capture Switching (CLK9)	36	5.12%
(pattern: 1)	0	0.00%
Peak Capture Switching (CLK10)	0	0.00%
(pattern: 0)	28	3.98%
Peak Capture Switching (CLK11)	0	0.00%
(pattern: 8)	0	0.00%
Peak Capture Switching (SETN)	0	0.00%
(pattern: 0)	0	0.00%
Peak Capture Switching (RSTN)	0	0.00%
(pattern: 0)		

Setting a Capture Budget for Individual Clocks

You can use the `-power_budget` and `-domain` options of the `set_atpg` command to set a capture budget for individual clocks. You must specify the `-power_budget` option before the `-domain` option.

The following example sets a capture budget of 55 for clock1:

```
set_atpg -power_budget 55 -domain clock1
```

The next example sets a capture budget of 15 to the clock2 and clock3 clock domains:

```
set_atpg -power_budget 15 -domain {clock2 clock3}
```

The next example assigns the minimum recommended capture budget for clock4:

```
set_atpg -power_budget min -domain clock4
```

After setting a capture budget for the individual clock domains, you can produce a power report using the `-per_clock_domain` option of the `report_power` command, as shown in the following example:

```
report_power -per_clock_domain
```

Power Analysis Summary

```
-----
Number of Scan Cells 54093
Number of Patterns 0-64
Cycles Per Load 52
Average Shift Switching 12939.91 23.92%
Average Capture Switching 7867.95 14.55%
Peak Shift Switching 18373 33.97% (pattern: 5 cycle: 12)
Peak Capture Switching 10880 20.11% (pattern: 11)
Peak Capture Switching (clk1) 9860 18.23% (pattern: 16)
Peak Capture Switching (clk2) 154 0.28% (pattern: 62)
Peak Capture Switching (clk3) 6160 11.39% (pattern: 26)
Peak Capture Switching (clk4) 1389 2.57% (pattern: 23)
```

You can use the following command to set all clocks to use the specified capture budget:

```
set_atpg -power_budget {min} -domain [get_attribute \
[get_clocks -all] clock_name]
```

Retention Cell Testing

A retention cell, as recognized by TetraMAX ATPG, is a scan cell that contains a retention pin. When a retention pin is asserted, it disables all clocks, including asynchronous signals. When a retention pin is de-asserted, it functions as a normal sequential device. A retention cell is also designed to retain its current state during sleep mode.

The following sections describe how to use TetraMAX ATPG to test the unique properties of retention cells:

- [Creating the chain_capture Procedure](#)
 - [Identifying the Retention Cells](#)
 - [Performing Test DRC](#)
 - [Generating the Patterns](#)
 - [Running Fault Simulation](#)
 - [Limitations](#)
-

Creating the chain_capture Procedure

TetraMAX ATPG uses a special retention cell chain test to handle retention cell testing. You need to create a chain test procedure in the STIL procedure file , called the `chain_capture` procedure, to work with the retention chain test. TetraMAX ATPG initially runs the chain test using the data specified in the STL procedure file and then applies it again with the same data reverted.

For example, if you specify a repeating 0101 for the chain test, TetraMAX ATPG first applies the chain test with a repeating 0101 and then reapplys it with a repeating 1010.

The `chain_capture` procedure in the STL procedure file performs the following steps:

1. Turns the SLEEP signal on, and, if necessary, turns off the scan-enable signal.
2. Pulses the clocks.
3. Turns the SLEEP signal off, and turns the scan-enable signal back on if it was previously turned off.

The `chain_capture` procedure shown in the following example uses a four-bit retention register:

```
"chain_capture" {
    F { "test_mode"=1; _po=\r 78 X; }
    V { "scan_enable"=0; "ret_clk"=0; "reset"=0; "clk"=0; }

    // Load 0110 into retention register to turn retention on.
    V { "ret_in"=0; "scan_enable"=1; "ret_clk"=P; }
    V { "ret_in"=1; "scan_enable"=1; "ret_clk"=P; }
    V { "ret_in"=1; "scan_enable"=1; "ret_clk"=P; }
    V { "ret_in"=0; "scan_enable"=1; "ret_clk"=P; }

    // Pulse capture clock while retention is on.
```

```

V { "ret_clk"=0; "clk"=P; }

// Load 1001 into retention register to turn retention off.
V { "ret_in"=1; "scan_enable"=1; "ret_clk"=P; "clk"=0; }
V { "ret_in"=0; "scan_enable"=1; "ret_clk"=P; }
V { "ret_in"=0; "scan_enable"=1; "ret_clk"=P; }
V { "ret_in"=1; "scan_enable"=1; "ret_clk"=P; }
}

```

You can also implement the retention functionality in a circuit using a special retention register with a separate clock for that register. The `chain_capture` procedure enables the retention process by loading a special value into the retention register.

Identifying Retention Cells

To identify the retention cells you want to test, you must include two special notations in the cell library file: ``define retention` and ``undef retention`. The ``define retention` notation must be placed before the cell definition. This enables TetraMAX ATPG to detect the faults on that cell when the special retention cell test generation procedure is used. The ``undef retention` notation is placed after the retention cell definition and prevents the next cell in the cell library from being identified as a retention cell.

Performing Test DRC

After creating the STL procedure file and identifying the retention cells in the cell library file, you run DRC using the following command sequence:

```
set_drc -clock -chain_capture
run_drc STL procedure file_file_name
```

The `set_drc -clock -chain_capture` command sets the DRC process to use the retention cell chain test, and the `run_drc spf_filename` command runs DRC using the specified STL procedure file.

Generating the Patterns

To generate patterns for retention cell testing, specify the `run_atpg full_sequential_only` command.

Note that you must run the `run_atpg` command in full-sequential mode. You can also use the `set_atpg -chain_test` command to load additional data into the scan chains.

TetraMAX ATPG performs the following steps to create retention patterns:

1. Loads the scan chains with the SLEEP signal turn off.
2. Runs the `chain_capture` procedure.
3. Unloads the scan chains.

These steps are performed twice: one time with the data specified for the normal scan chain test and one time with the same data inverted.

Running Fault Simulation

After the patterns are generated, you need to set up and run the fault simulation using the `set_simulation` and `run_fault_sim` commands.

TetraMAX fault-simulates the generated patterns and the patterns are retained only if they detect faults. All detected faults are listed as DS (Detected by Simulation) faults. If the first pattern detects faults but the second pattern does not, only the first pattern is retained.

Limitations

The following limitations apply to retention cell testing:

- You cannot run retention cell testing using either of the following commands:
 - `run_atpg basic_scan_only`
 - `run_atpg fast_sequential_only`
- The `Procedure` section of the STL procedure file used for retention cell testing must contain only the `chain_capture` procedure and the `load_unload` procedure. Incorrect results are possible if any other capture procedures are included in the `Procedure` section when the `chain_capture` procedure is present.
- You cannot use the `run_atpg -auto` command if full-sequential test generation is turned off.

Power Aware ATPG Limitations

Note the following limitations related to power aware ATPG:

- Test-mode based clock gating is not supported. Only scan-enable based clock gating is supported.
- Latch-free clock gating is not supported. Only latch-based clock gating is supported, which includes cascaded latch-based clock gating structures.
- Only simple clock-gating latches are supported. Combinations of the output of two or more latches when logically combined with the clock are not supported.
- Full-sequential ATPG is not supported for either the `report_power` command or Power Aware ATPG.
- Scan-enable signals must be constrained to the off-state for basic-scan, two-clock, and fast-sequential for test pattern generation and gated-clock (latch) identification. In addition, all global signals that are capable of enabling a large proportion of the clock-gating cells must be disabled.
- Maximum switching overshoots might occur if ATPG requires more flip-flops to change in excess of the power budget to detect a fault.
- Memories are not supported.
- Asynchronous set and reset signals must be inactive (in their off state).

- The `-domain` option of the `set_atpg` command does not work when specified with the `-calculate_power` option of the `set_atpg` command.

19

Bridging Fault ATPG

A bridging defect (also known as a short), is a common defect in semiconductor devices. This defect causes two normally unconnected signal nets in a device to become electrically connected due to extra material or incorrect etching.

These type of defects can be detected if one of the nets (the aggressor) causes the other net (the victim) to take on a faulty value, which can then be propagated to an observable location. Although there is a strong correlation between stuck-at coverage and bridging coverage, there is no guarantee that a set of patterns generated to target stuck-at faults will achieve similar coverage for a set of bridge faults.

The following sections describe the bridging fault model and fault simulation ATPG flow:

- [Detecting Bridging Faults](#)
- [Bridging Fault Flows](#)
- [Using StarRC to Generate a Bridge Fault List](#)
- [Bridging Fault Model Limitations](#)
- [Running the Dynamic Bridging Fault ATPG Flow](#)

Note: You will need a Test-Fault-Max license to use this feature. This license is also checked out if you read an image that was saved with the fault model set to bridging.

See Also

[IDQ Bridging](#)

Detecting Bridging Faults

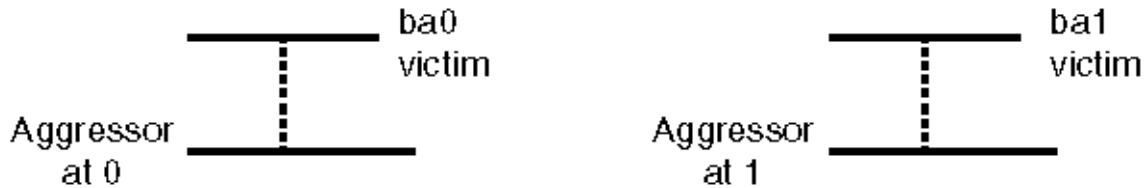
The following sections describe how to detect bridging faults:

- [How Bridging Faults Are Defined](#)
 - [Bridge Locations](#)
 - [Strength-Based Patterns](#)
-

How Bridging Faults are Defined

TetraMAX ATPG defines a bridging fault by type and a set of two nodes (which can be instance pins or net names). The type is either bridging fault at 0 (ba0) or bridging fault at 1 (ba1) (see [Figure 1](#)). The first node is called the victim node and the second node is called the aggressor node.

Figure 1 Bridging Fault Types ba0 and ba1



A ba0 bridging fault is considered detected if the stuck-at-0 fault at the victim node is detected at the same time the fault-free value of the aggressor node is at 0. Similarly, a ba1 bridging fault is considered detected if the stuck-at-1 fault at the victim node is detected at the same time the fault-free value of the aggressor node is at 1.

Bridge Locations

The victim and aggressor nodes are specified by *bridge location*. A bridge location is a hierarchical path to any of the following:

- Cell instance input pin
- Cell instance output pin
- Net name

Faults on bidirectional pins are ignored. Input pins can be used only if the `set_faults -bridge_input` command is specified.

Although a net can have many names as it traverses the hierarchy of a design, TetraMAX ATPG does not store them all. If you specify a net name as a bridge location that TetraMAX ATPG recognizes (those accepted by the `report_primitives` command), it is used to map the fault to the single output pin connected to that net.

Net names are internally translated to an instance pin. This pin path must be a valid stuck-at fault site. Instances dropped during the build process with a B22 warning message cannot be used. A warning is given if you specify an invalid bridge location.

Strength-Based Patterns

A bridge defect has complex analog effects due to parameters such as the strength of the driver, resistance of the bridge, and wire characteristics. Therefore, it is not always clear when a bridge is detected by the pattern generated considering only logical behavior. Some researchers have speculated that patterns can be adjusted to improve the odds of detecting bridging faults. The basic premise is that forcing the aggressor to drive stronger and the victim to drive weaker increases the chance of the bridge being detected.

Patterns that use this principle can be generated when the victim or aggressor is on the output pin of a primitive gate having a dominant value (AND, OR, NAND, or NOR). A more stringent detection criteria can then be imposed. The ATPG process can be given additional soft constraints to optimize the drive strengths after the normal bridging fault detection requirements are met. Soft constraints are those that the ATPG process attempts to meet on a best-effort basis. If the soft constraints are not met, the pattern is still retained for detection of bridging faults.

With the addition of strength-based patterns, bridge fault detection can be classified into the following detection types:

- Minimal detection – the minimum condition for the detection of ba0 & ba1 faults, as previously described.
- Fully optimized detection – a detection where the conditions specified in Table 1 are met. For maximizing inputs with a specific value, all inputs of the driving gate must be at the specified value. To minimize the inputs at a specific value, only one of the driving gate's inputs must be at the specified value.
- Partially optimized detection – a detected bridging fault that is neither minimal nor fully optimized.

Table 1 Strength-Optimized Detection of Bridging Faults

Driving Gate	ba0		ba1	
	Driver of victim	Driver of aggressor	Driver of victim	Driver of aggressor
AND		Maximize driver inputs with 0s	Minimize driver inputs with 0s	
NAND	Minimize driver inputs with 0s			Maximize driver inputs with 0s
OR	Minimize driver inputs with 1s			Maximize driver inputs with 1s
NOR		Maximize driver inputs with 1s	Minimize driver inputs with 1s	

Bridging Fault Flows

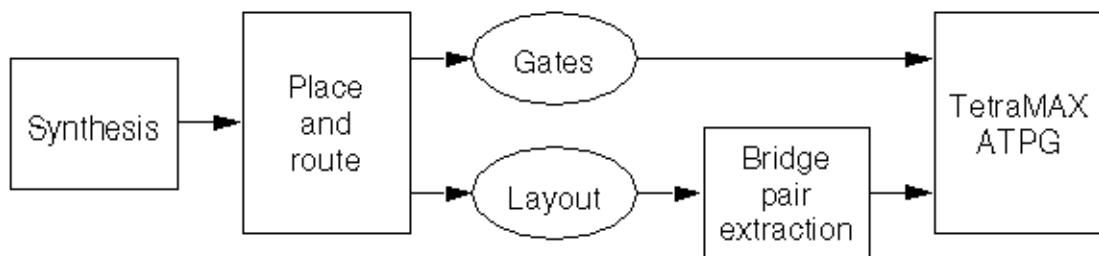
The following sections describe bridging faults and how they fit into an overall flow and in TetraMAX ATPG:

- [Bridging Faults and the Overall TetraMAX Flow](#)
 - [Bridging Fault Flow in TetraMAX](#)
-

Bridging Faults and the Overall TetraMAX Flow

The overall flow for bridging faults is shown in [Figure 1](#). Stuck-at fault ATPG is run immediately after synthesis, and before place and route in some flows. Bridging fault ATPG and fault simulation is usually run following the completion of place and route on full-chip designs.

Figure 1 Overall Flow for Bridging Faults



There are multiple ways you can generate bridge pairs. Two possibilities are

- Extract bridging pairs from the layout using an IFA-based scheme
- Use an extracted coupling capacitance report

A flow based on capacitance extraction is shown in [“Using StarRC to Generate a Bridge Fault List”](#). TetraMAX ATPG can read the coupling capacitance report directly from StarRC. Third-party capacitance extraction tools can be used to generate coupling capacitance reports if the node list is in the TetraMAX format.

It is not possible to accurately model fault effects for bridges that involve clock/set/reset lines and bridges that produce combinational loops. Therefore, you should filter out these types of bridges.

Bridging Fault Flow in TetraMAX

[Figure 2](#) shows the bridging fault flow in TetraMAX ATPG. The typical commands used in this flow are identified in the subsections that follow. For details regarding command usage and option descriptions, see Online Help.

Figure 2 Bridging Fault Model in TetraMAX



Setup

These commands are typically used at the beginning of a command file, because they have an effect on subsequent commands:

- `set_faults -model bridging` – This command is mandatory for bridging faults.
- `set_atpg -optimize_bridge_strengths` – This command can be used based on whether or not TetraMAX ATPG should optimize drive strength on the driving gates of the victim and aggressor nodes.
- `set_faults -bridge_input` – This command can be used to accept input pins of instances as bridge locations.

Input Faults

The list of bridging pairs can be supplied in any combination of the following three ways:

- Command file – A set of `add_faults` commands that can be sourced by a script: `add_faults [-bridge_location <bridge_location1> [bridge_location2>] [-bridge <0|1|01>] [-aggressor_node <first | second | both>]` This command can be used to add bridging faults.
- Fault list file – Generated by a `report_faults` or `write_faults` command and read by the `read_faults` command. Only ba0 and ba1 fault types are expected.
- Node file – A list of bridging node pairs: `add_faults <-node_file <name>> [-bridge <0|1|01>] [-aggressor_node <first | second | both>]` In its simplest form, the node file format is a pair of bridge locations per line, separated by a space. An unmodified coupling capacitance report from StarRC can also be used. For details, see topic “Node File Format for Bridging Pairs” in the online help. A suggested flow of using the Synopsys StarRC capacitance extraction tool for bridge fault list generation is described in [“Using StarRC to Generate a Bridge Fault List”](#).

A bridge fault list should not include clocks and asynchronous set or reset signals. Proper detection status cannot be guaranteed for these faults.

Manipulating the Fault List

The following commands and options are useful for manipulating the fault list:

- `add_nofaults` – This command can be used to set “no fault” status on victim nodes to prevent the associated bridging fault from being added to the fault list.
- `remove_faults <[-bridge_location <bridge_location1> <bridge_`

```
location2>] | -all | -retain_sample <d> | -class <fault_
class>> [-bridge <0|1|01>] [-clocks] [-aggressor_node <first |
second | both>] [-non_strength_sensitive]
```

This command can be used to remove bridging faults.

Examining the Fault List

The following options of the `report_faults` command can be used to examine a fault list, both before and after fault ATPG and simulation: [bridge_location1 [bridge_location2]] [-bridge <0|1|01>] [-aggressor_node <first | second | both>] [-bridge_feedback] [-bridge_strong]

The format of the report is in four columns:

- Fault type (ba0 or ba1)
- Fault detection status code
- Bridge location of the victim node
- Bridge location of the aggressor node

For example:

```
ba0    NC    nodeA    nodeB
ba1    NC    nodeA    nodeB
ba0    NC    nodeB    nodeA
ba1    NC    nodeB    nodeA
```

Fault Simulation

Fault simulation of bridging faults is usually done to determine which bridges are detected by other existing patterns, such as those generated for stuck-at faults. Typically, many bridges are detected by patterns targeting other fault models.

For bridging faults with either or both nodes driven by gates with dominant values (AND, OR, NAND, or NOR), use the `run_fault_sim -strong_bridge` command to require a fully optimized detection. When this option is used, the fault is marked as detected only if the criteria for fully optimized bridging fault detection is met.

Running ATPG

Bridging fault ATPG attempts to set the victim and aggressor bridge locations at opposite values, while attempting to detect the value of the victim net.

If you plan on issuing a `run_atpg -auto_compression` command, you first need to create an explicit fault list by either issuing an `add_faults` or `read_faults` command.

If you issue a `set_atpg -optimize_bridge_strengths` command, ATPG attempts to generate patterns with fully optimized detections on a best effort basis. This assumes that the TetraMAX libraries are modeled in a manner that would produce meaningful strength-based patterns. For example, gates with dominant values should be instantiated so that the correct transistors are activated or deactivated.

Analysis

After running ATPG or fault simulation, you can use the `report_faults` and `write_faults` commands to analyze fault detection status. You can invoke automated analysis and schematic display by using the `analyze_faults <bridge_location1 bridge_location2 -bridge <0|1>>` command.

Example Script

[Example 1](#) shows a script for bridging fault support. This script generates tests for bridging faults followed by stuck-at faults. You might want to experiment with the reverse order as well to see which method produces better results.

Example 1 Script for Bridging Faults

```
# read netlist and libraries, build, run_drc
read_netlist design.v -delete
run_build_model design
run_drc design.spf

# bridging faults
set_faults -model bridging

# allow instance input pins to be valid victim sites
set_faults -bridge_input

# to optimize strengths during atpg
set_atpg -optimize_bridge_strengths

# read in fault list
add_faults -node_file nodes.txt

# run atpg with merging
set_atpg -merge high
run_atpg -auto_compression

# write the bridging patterns out
write_patterns bridge_pat.bin -format binary -replace

# now fault simulate bridge patterns with stuck-at faults
# this part is intended to reduce the set of patterns by not
generating
# patterns for stuck-at faults detected by the bridging patterns
remove_faults -all
set_faults -model stuck
add_faults -all

# read in bridging pattern
set_patterns -external bridge_pat.bin
```

```
# fault simulate
run_fault_sim

# generate additional stuck-at patterns
set_atpg -merge high
set_patterns -internal
run_atpg -auto_compression
```

Using StarRC to Generate a Bridge Fault List

The bridging fault model requires a pair of locations to bridge. Because all pairs of nets would result in an intractably large fault set for ATPG or fault simulation, a set of bridge pairs that are likely candidates are used. This information is generated using layout information, because nets in close proximity are more likely to be bridged.

The flow currently supported by Synopsys uses finding nets with high coupling capacitance using StarRC. Some studies have shown there is a correspondence between capacitance and likelihood of bridging.

This section describes the process of generating a bridging fault list. It assumes you have some knowledge of StarRC and that your environment is properly set up and licensed. See the StarRC documentation for more details. If StarRC is not available, any capacitance extraction tool that generates a coupling capacitance report can be used. Synopsys does not support third-party tools or flows.

If high accuracy is a requirement, the coupling capacitance report from a normal run of StarRC can be used as a bridge pair list. If a small set of bridge faults is used, or if a higher accuracy bridge fault list is required, a separate run of TCAD characterization and extraction is required to remove the effect of varying dielectric constants across layers.

If a coupling capacitance report from a normal StarRC run is used, ensure that the report has enough net pairs for bridging fault ATPG and for fault simulation. The remainder of this section may be skipped if this is the flow you choose.

The process involves three primary steps, which are described in the following sections:

- [TCAD Characterization](#) – using grdgenxo, a part of the StarRC tool set. This is done only one time for a given fabrication process and bridge probability.
- [Capacitance Extraction](#) and coupling capacitance report generation using StarXtract. This is a separate run from the normal capacitance extraction run.
- [Running TetraMAX](#)

TCAD Characterization

TCAD characterization is run one time per process and bridging probability. It need not be run multiple times for each process corner. Only the layer locations are important. The process parameters themselves play almost no role in this procedure.

Edit a copy of the Interconnect Technology Format (ITF) file supplied by your foundry. It is used by StarRC to compute bridging likelihood. You edit the ITF file so the dielectric constants have less of an effect in generating the bridging pairs.

Generating a Resistance and Capacitance (GRD) Model.

The following steps assume that inter layer bridging is much less likely than intra layer bridging:

1. Change all `DIELECTRIC` statements between the top metal and poly layers as follows, depending on each dielectric's location.

To determine if a dielectric is between layers or between conductors on the same layer, create a diagram of the layers. See the section on `DIELECTRIC` statements in StarRC documentation for more information.

2. For dielectrics between different layers, change "ER" to 0.1.
3. For dielectrics between conductors on the same layer, change "ER" to some constant value; for example 1.

These numbers can be adjusted based on data from the foundry on bridge probabilities. For example, if some layers have higher defect densities, the intra layer ER values can be increased for that layer. This effectively sets the probability of intralayer bridges to be 10X the probability of interlayer bridges, based on the simplified parallel-plate capacitance equation ($C = \epsilon * A/d$, where ϵ is the dielectric constant, A is the area of the conductors facing each other, and d is the distance between the conductors).

Do TCAD characterization by running the field solver on the ITF file; for example, and create the `.nxtgrd` file.

4. `% grdgenvx spec.itf`

This produces a `.nxtgrd` (resistance and capacitance model) file that is used by StarRC.

This step can be done parallel to starting other jobs in the same directory on different machines. The speedup is almost linear relative to the number of processors. See the StarRC documentation for details.

Extracting Capacitance

Before beginning this process, you need to supply these files:

- Post-layout data in the form of a Milkyway database, LEF/DEF files, or Calibre files.
- The `.nxtgrd` file from the TCAD characterization step in the preceding section.
- A layer mapping file that maps the layer names in the ITF file to the layout layer names. This layer mapping file is created only one time per process and should already exist for normal extraction.

Make sure the `SYNOPSYS` environment variable (`$SYNOPSYS`) is set to the Synopsys root directory for StarRC.

Running StarRC in GUI or Batch Mode

To run StarRC in GUI Mode:

1. Start StarRC with the GUI option, for example:

```
% StarXtract -clean -gui &
```

The Milkyway database is assumed to exist. Other layout database formats have a similar flow.

2. Set up the extraction run.

3. Choose Setup > Timing. In the Timing Wizard, enter:

Drop down – Set to the layout data format (Milkyway is assumed).

Data format – If Milkyway or Hercules data:

BLOCK – Typically the design name. Most often there is a file with this name under the CEL view if you are using Milkyway. Check the CEL directory under the Milkyway directory.

MILKYWAY DATABASE – The directory name containing the Milkyway database. If LEF/DEF or Calibre, specify the appropriate input values.

TCAD GRD FILE – Path to the .nxtgrd file.

MAPPING FILE – Name of mapping file.

EXTRACTION – RC

COUPLE_TO_GROUND –

NO COUPLING_MULTIPLIER – 1

Then click OK.

4. Choose Setup > Noise. In the Noise Wizard, enter:

COUPLING REPORT FILE – Name of output file containing coupling report.

COUPLING ABS THRESHOLD – 0.1

COUPLING REL THRESHOLD – 1FF

COUPLING REPORT NUMBER – Number of the top net pair to report.

A rule of thumb for the number of pairs is that it should be on the order of magnitude of the stuck-at faults in the design.

Then click OK.

The default extraction should not consider power and ground signals as coupling partners. The preceding settings should prevent “smart decoupling,” because all coupling capacitances must be preserved.

5. Choose File > Run to begin the extraction.

Run StarRC in Batch Mode using a command file similar to the following example:

1. BLOCK: <design name>
MILKYWAY_DATABASE: <milkyway db>
TCAD_GRD_FILE: <nxtgrd file>
MAPPING_FILE: <mapping file>
EXTRACTION: C
COUPLE_TO_GROUND: NO

```

COUPLING_MULTIPLIER: 1
COUPLING_REPORT_FILE: <output coupling capacitance
report file>
COUPLING_ABS_THRESHOLD: 1e-15
COUPLING_REL_THRESHOLD: 0.1
COUPLING_REPORT_NUMBER: <number of nets to include in
report>

```

2. Specify the command file on the command line; for example:

```
StarXtract [options] star_cmd
```

Coupling Capacitance Report

An example coupling capacitance report follows. Note that some net pairs may be repeated in the opposite order with the victim and aggressor switched, as seen in the last 2 pairs. TetraMAX ATPG will add the faults twice.

```

* 632 worst couplings list in decreasing order:
* % coupling victim aggressor
100 3.33e-18 timer/se_7_cnt/K2370 timer/se_7_cnt/I2370
61.2 1.66e-15 io_c0[7] io_c1[9]
60.5 9.26e-17 timer/m_cnt/L2865 timer/m_cnt/T2128
60.4 2.77e-15 io_c3[5] io_c3[3]
59.7 2.29e-15 io_c4[7] io_c4[12]
57.5 2.33e-15 io_c4[14] io_c4[15]
54.9 7.97e-15 io_c2[14] io_c2[0]
54.1 7.97e-15 io_c2[0] io_c2[14]
...

```

Running TetraMAX

The StarRC coupling capacitance report can be directly read in by TetraMAX ATPG using the `add_faults -node_file` command. TetraMAX ATPG automatically recognizes a StarRC report and adds the necessary faults.

Do not use the netlist before place and route as is sometimes used. Use only the postroute netlist for TetraMAX ATPG. Otherwise, the signal names might not match.

Do not use an image file when adding faults that include net names, because net names are not preserved in the image file. An alternative is to perform the following steps:

1. Read the netlist
2. Build
3. Perform DRC
4. Add the net name-based bridge faults
5. Write out the fault list

The output fault list is pin-path based that can be read in subsequent TetraMAX runs that use the image file.

Bridging Fault Model Limitations

Using the bridging fault model has the following limitations:

- No oscillation effects are considered. The aggressor remains at the fault-free value. Fault effects from a victim in the fanin cone is dropped at the aggressor.
- Full-Sequential ATPG and Full-Sequential fault simulation are not supported.
- Bidirectional pins cannot be faulted.
- Basic-Scan ATPG and fault simulation assumes clocks and asynchronous sets/resets are at constant values per pattern.
- There is no fault collapsing for bridging faults.
- No detection by implication (DI) credit is given.
- No method for generating bridging node pairs is provided within TetraMAX ATPG.
- Net names cannot be used for bridging locations if the `read_image` command was used. Only net names given by the `report_primitives` command are supported.

Running the Dynamic Bridging Fault ATPG Flow

The following sections describe how to run the dynamic bridging fault ATPG flow:

- [Understanding the Dynamic Bridging Fault Model](#)
- [Preparing to Run Dynamic Bridging Fault ATPG](#)
- [Fault Simulation](#)
- [Running ATPG](#)
- [Analyzing Fault Detection](#)
- [Example Script](#)
- [Limitations](#)

Understanding the Dynamic Bridging Fault Model

The TetraMAX dynamic bridging fault model combines two fault models:

- The **static bridging fault model**, which observes whether the value on the aggressor node will override the value on the victim
- The **transition fault model**, which observes whether the transition at the fault site is too slow for the rated clock speed

Based on the combined usage of these two fault models, the dynamic bridging fault model can be used to analyze transition effects in the presence of a specified value on a bridge-aggressor node.

TetraMAX ATPG defines two types of dynamic bridging faults:

- **Bridge slow-to-rise (bsr)** — a slow-to-rise fault exists on the victim node while the aggressor node is at 0.
- **Bridge slow-to-fall (bsf)** — a slow-to-fall fault exists on the victim node while the aggressor node is at 1.

The fault location is the same as that used for (static) bridging faults, except that the cell instance input pin cannot be faulted. See [“Bridge Locations”](#) for more information.

Note that since a list of dynamic bridging nodes is required to run ATPG, the dynamic bridging fault model and fault simulation process is usually run after completing place and route on full-chip designs. Also note that you cannot add all faults using the dynamic bridging fault model. To add all faults, you will need to explicitly create a fault list before running ATPG using the `-auto` option.

Preparing to Run Dynamic Bridging Fault ATPG

To enable dynamic bridging fault ATPG, specify the following command:

```
set_faults -model dynamic_bridging
```

The following tasks are required to set up TetraMAX ATPG to run dynamic bridging fault ATPG:

- [Specifying a List of Input Faults](#)
- [Manipulating the Fault List](#)
- [Examining the Fault List](#)

Specifying a List of Input Faults

You can use any combination of the following three files to supply a list of dynamic bridging pairs (referred to as a fault list):

Command file — This file, which can be sourced from a script, contains a set of `add_faults` commands that specify dynamic bridging pairs. The syntax for using the `add_faults` command for this purpose is as follows:

- `add_faults [-bridge_location bridge_location1 bridge_location2] [-dynamic_bridge <r|f|rf>] [-dominant_node <first | second | both>]`
- **Fault list file** — This file is generated by either the `report_faults` command or the `write_faults` command and is read by the `read_faults` command. Note that only `bsr` and `bsf` fault types are valid in this list.

Node file — This file contains a list of dynamic bridging node pairs, specified in terms of nodes, using the following syntax for the `add_faults` command:

- `add_faults <-node_file name> [-dynamic_bridge <r|f|rf>] [-dominant_node <first | second | both>]`

The format for a node file is to specify a pair of bridge locations on each line, separated by a space. You can also use an unmodified coupling capacitance report generated from the

Synopsys StarRC capacitance extraction tool. Additional details are also covered in the “Node File Format for Bridging Pairs” topic in TetraMAX Online Help.

Note: The command file, fault list file, and node file should not include clocks and asynchronous set or reset signals, because proper detection status cannot be guaranteed for these faults.

Manipulating the Fault List

You can use the following commands and options to manipulate the fault list:

`add_nofaults` — This command can be used to set a “no fault” status on victim nodes to prevent the associated dynamic bridging fault from being added to the fault list. The syntax for this command is as follows:

- `add_nofaults < instance_name | pin.pathname | -Module name >`
- `remove_faults` — This command can be used to remove dynamic bridging faults. The syntax for this purpose is as follows:
 - `remove_faults < [-bridge_location bridge_location1 bridge_location2] | -all | -retain_sample <d> > [-dynamic_bridge <r|f|rf>] [-clocks] [-dominant_node <first | second | both>]`

Examining the Fault List

The following options from the `report_faults` command can be used to examine a fault list, both before and after fault ATPG and simulation:

```
[bridge_location1 bridge_location2]
[-dynamic_bridge <r|f|rf>]
[-dominant_node <first | second | both>]
```

The format of the generated report contains the following four columns:

- Column 1: Fault type (bsr or bsf)
- Column 2: Fault detection status code
- Column 3: Dynamic bridge location of the victim node
- Column 4: Dynamic bridge location of the aggressor node

Note the following example report:

```
bsr NC nodeA nodeB
bsf NC nodeA nodeB
bsr NC nodeB nodeA
bsf NC nodeB nodeA
```

Fault Simulation

You will need to perform fault simulation on the dynamic bridging faults to determine which dynamic bridging faults are detected by other existing patterns (such as those generated for stuck-at faults or transition faults). Typically, a large number of dynamic bridges are detected by patterns that target other fault models.

To run a fault simulation on the existing patterns, specify the `run_fault_sim` command. (You can see how to run this command in the “[Example Script](#)” section.)

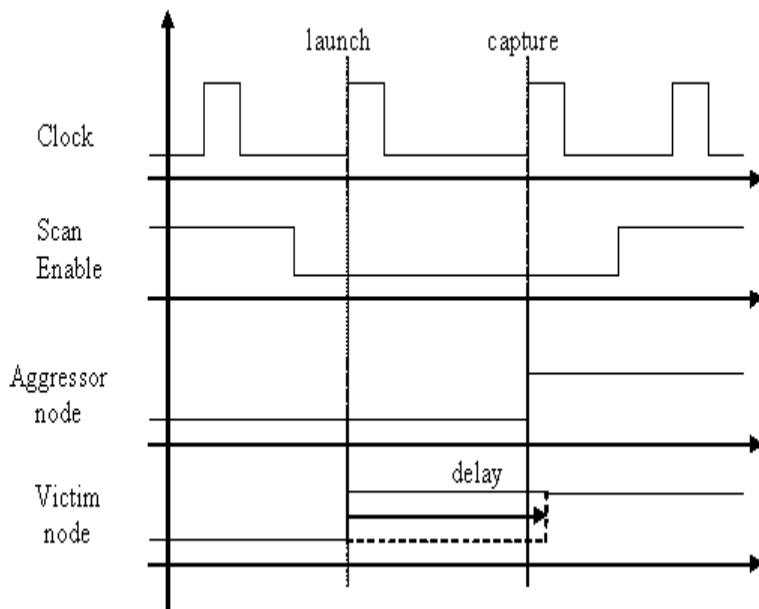
Note that fault simulation for dynamic bridging fault does not support Full-Sequential mode. An error is issued if you attempt to use this mode.

Running ATPG

The dynamic bridging fault ATPG process attempts to launch a transition along the victim while holding the aggressor at a static value. If you plan on issuing a `run_atpg -auto_compression` command, you will first need to create an explicit fault list by specifying either an `add_faults` or `read_faults` command.

Dynamic bridging fault ATPG can be run using the following ATPG modes: Basic-scan (launch on last shift), Two Clocks, and Fast-Sequential. An example of waveforms that are typically applied in the case of Fast-Sequential launch on system clock is shown in [Figure 1](#). In the presence of a bsr fault, the transition initiated because of the launch cycle at the victim node is delayed (dashed line) and the capture cycle detects the fault.

Figure 1 Dynamic Bridge Fault Detection Waveforms for Launch on System Clock



Note that dynamic bridging fault ATPG does not support Full-Sequential mode. An error is issued if this is attempted. Also, strength-based pattern generation similar to what exists for the TetraMAX bridging fault model is not supported.

Analyzing Fault Detection

After running ATPG or fault simulation, you can use the `report_faults` and `write_faults` commands to analyze the fault detection status. You can invoke automated analysis and schematic display by using the following `analyze_faults` command options:

```
analyze_faults <bridge_location1 bridge_location2 -dynamic_bridge
```

<r | f>

Example Script

The following example shows a script for dynamic bridging fault support. This script generates tests for dynamic bridging faults, followed by stuck-at faults. You might want to experiment by reversing the order to see which method produces better results.

```
# read netlist and libraries, build, run_drc
read_netlist design.v -delete
run_build_model design
run_drc design.spf

# set fault model to dynamic bridging
set_faults -model dynamic_bridging

# read in fault list
add_fault -node_file nodes.txt

# run_atpg
run_atpg -auto_compression

# write the bridging patterns out
write_patterns dyn_bridge_pat.bin -format binary -replace

# now fault simulate dynamic bridge patterns with stuck-at faults
# this part is intended to reduce the set of patterns
# by not generating patterns for stuck-at faults
# detected by the dynamic bridging patterns
remove_faults -all
set_faults -model stuck
add_faults -all

# read in dynamic bridging pattern
set_patterns -external dyn_bridge_pat.bin

# fault simulate
run_fault_sim

# generate additional stuck-at patterns
set_patterns -internal
run_atpg -auto_compression
```

Limitations

The dynamic bridging fault ATPG feature currently has the following limitations:

- Full-Sequential ATPG and Full-Sequential fault simulation are not supported.
- The dominant node effect is based on its fault-free value. There is no ability to consider feedback effects that result from a dynamic bridge.
- There is no fault collapsing for dynamic bridging faults.
- Strength-based pattern generation is not supported.
- Input and bidirectional pins cannot be faulted.
- Proper detection status cannot be guaranteed for dynamic bridging pairs, including clocks and asynchronous sets/resets.
- TetraMAX ATPG does not provide detection by implication (DI) credit.
- TetraMAX ATPG does not provide a method for internally generating dynamic bridging node pairs.
- Only net names given by the `report_primitives` command are supported.

20

Cell-Aware Test

Cell-aware test is a methodology for increasing defect coverage, lowering defective parts per million (DPPM) and improving diagnostics accuracy for emerging process nodes, [FinFETs](#) (multi-gate field-effect transistors), and automotive standards.

The following topics explain how to use cell-aware test:

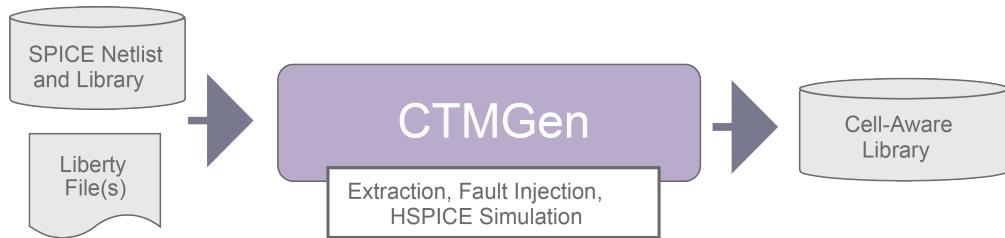
- [Cell-Aware Test Flow](#)
- [Cell-Aware Test Background](#)
- [Understanding Cell Test Models](#)
- [Generating Cell Test Models](#)
- [Running Cell-Aware ATPG](#)
- [Cell-Aware Diagnostics](#)

Cell-Aware Test Flow

Cell-Aware test is comprised of three primary phases:

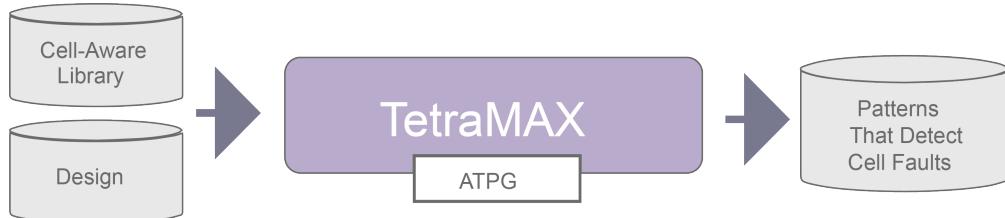
- **Cell Test Model Generation**

CTMGen, Synopsys' cell test model (CTM) generation utility, creates a CTM that lists all the detectable defects for each cell. The CTM guides the TetraMAX ATPG and diagnostics processes on how to target and isolate these defects. This process is described in more detail in "[Generating Cell Test Models](#)."



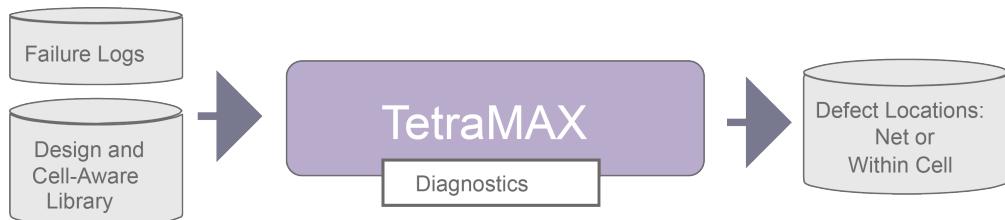
- **Pattern Generation**

Cell-aware ATPG uses the existing TetraMAX ATPG paradigm. All defects become faults in the fault lists, and both static and dynamic defects are supported. The ATPG process uses both primary and secondary faults, and merges and fault simulates the faults. This process is described in more detail in "[Running Cell-Aware ATPG](#)."



- **Diagnostics**

Cell-Aware test fits into the existing TetraMAX diagnostics process. Cell-aware faults are mapped to the cell excitation conditions. Detailed structural annotations allow for increased diagnostic resolution. This process is described in detail in "[Cell-Aware Diagnostics](#)."



The inputs to the characterization process include the following:

- SPICE netlist – The netlist for each cell typically includes extracted parasitics (not a strict requirement to generate a CTM).
- SPICE library – The library contains all the information needed to run HSPICE simulations on a cell.
- Liberty file – The .lib file contains timing information about the cell used to accurately model defect behavior.

Targeting Physical Cell Defects

Traditional ATPG and TetraMAX cell-aware test are significantly different in their approaches to target physical cell defects.

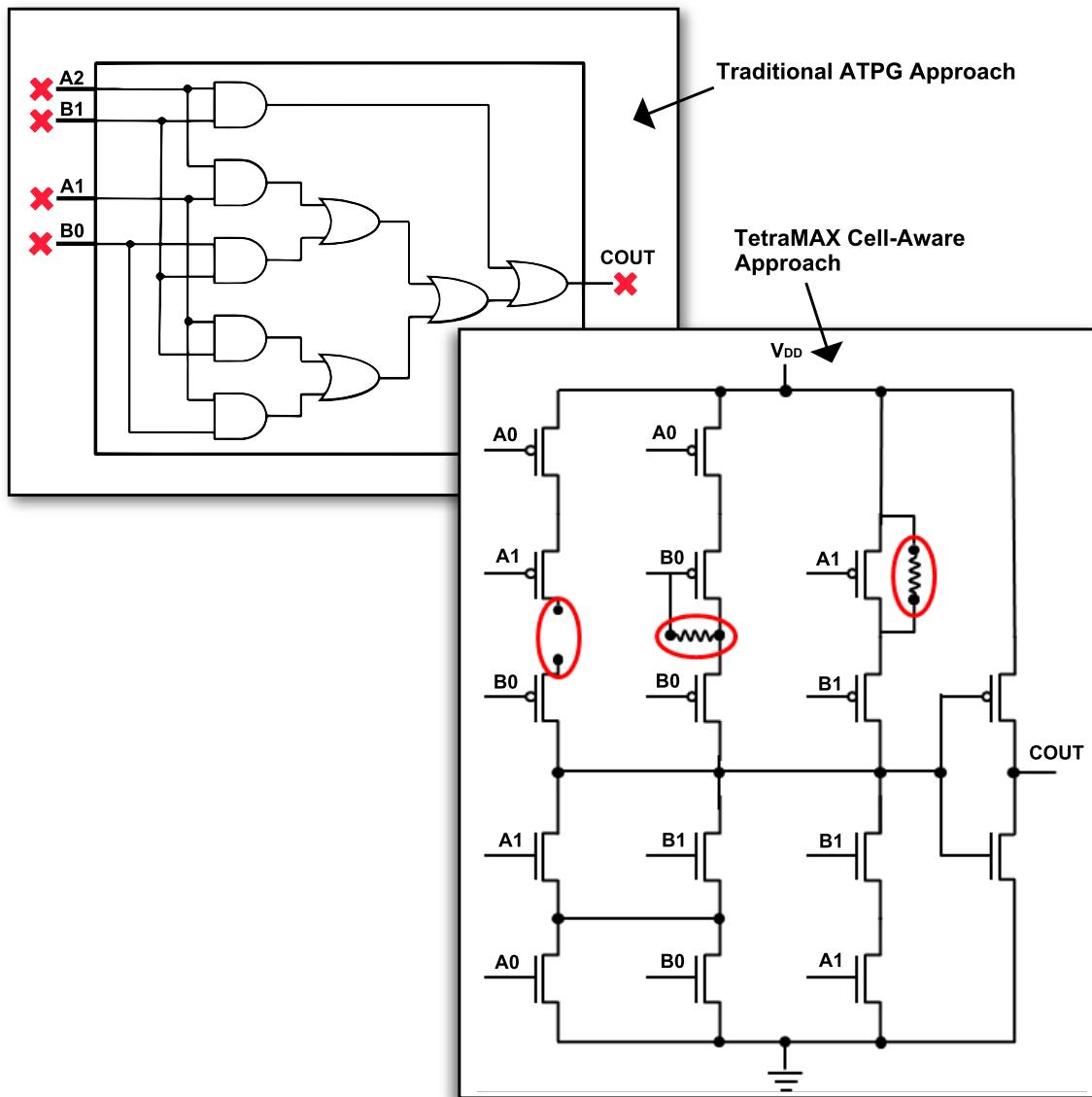
- Traditional ATPG targets faults between cells that are assigned to the input and output pins of cell instances. This approach can be effective in most situations, even though some of these faults originate as physical defects inside cells — referred to as “internal cell defects.”

However, traditional ATPG does not explicitly target internal cell defects. This is because as the complexity of a cell increases, the probability also increases that ATPG will not produce the input combinations required to cover all the defects likely to occur inside the cell. This observation particularly applies to high fan-in cells and cells that implement complex Boolean functions.

- TetraMAX cell-aware test explicitly targets internal cell defects during ATPG and diagnosis to increase defect coverage and improve diagnostics accuracy. It is particularly effective at testing faults in complex cells.

The example on the left in [Figure 1](#) is a complex cell using the traditional ATPG approach, which targets faults between cells that are assigned to the input and output pins of cell instances. The example on the right is a transistor-level view of same cell and shows examples of the defects targeted by TetraMAX cell-aware ATPG. In this case, the defects are targeted inside the cell.

Figure 1: Traditional ATPG Versus TetraMAX Cell-Aware Test



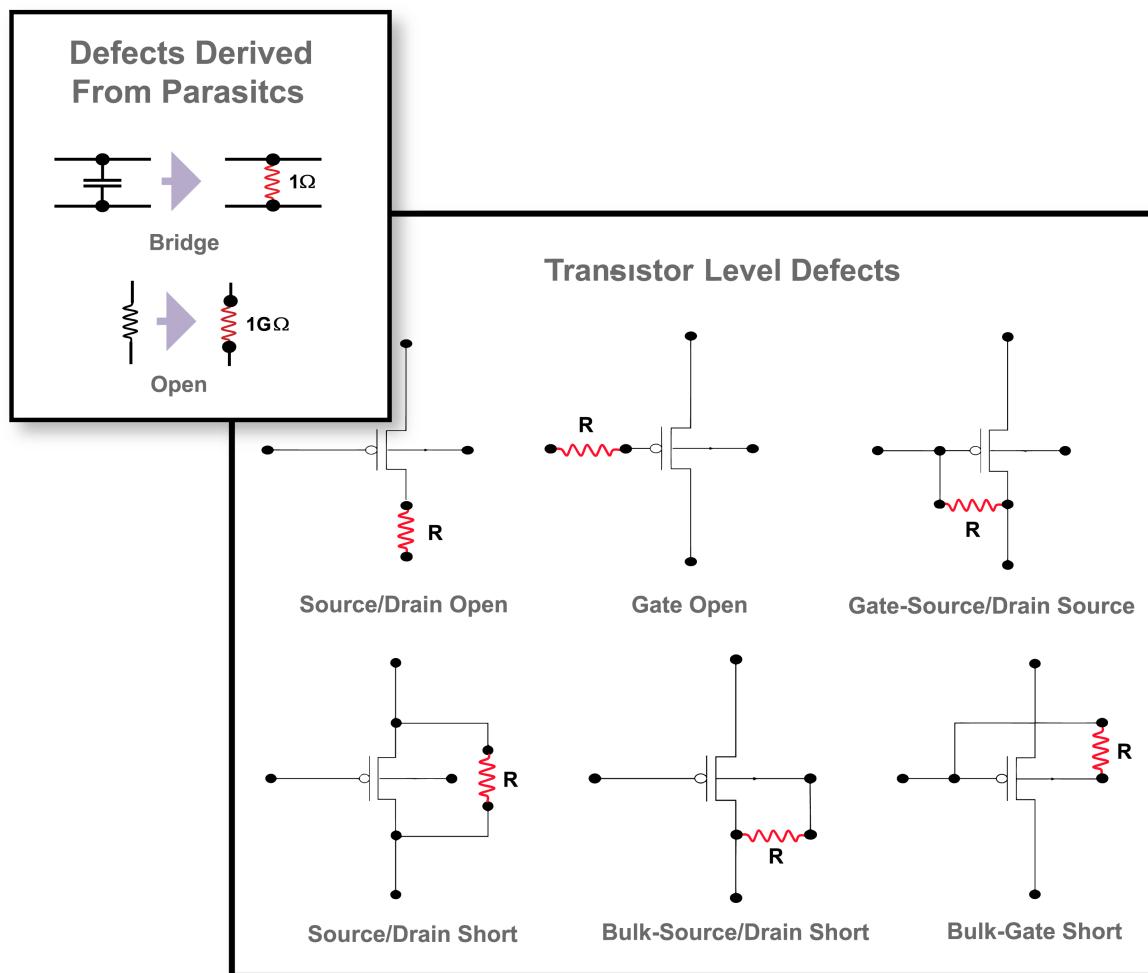
In TetraMAX cell-aware test, internal cell defects are characterized by simulating the HSPICE model of the cell under various short and open conditions. This characterization creates a single file, called a cell test model (CTM), that lists all the detectable defects for each cell. The CTM guides ATPG and diagnosis on how to target and isolate these defects.

Understanding Cell Test Models

Cell test models (CTMs) are the basis for performing cell-aware ATPG. A CTM is a text file that uses industry-standard YAML format and contains header information, such as the process corners derived from an HSPICE run.

As shown in [Figure 2](#), resistors can be connected to transistors and assigned values that approximate the behavior of various physical defect types, such as opens on drains, source-drain shorts, etc. If you insert a parameterized resistor into a circuit netlist and perform a transient analysis, you can compare good behavior versus faulty behavior at the outputs. The simulations need to be accurate enough to predict faulty behavior observable as stuck-at-1/0 or added delay to the output transitions.

Figure 2: Defect Injections



A CTM for a cell in a library includes a list of possible defects that could occur in a cell and the binary logic levels on the inputs and outputs. TetraMAX references to target and detect each

defect. A CTM can also include additional physical information about each defect, such as the mask layer and cell coordinates, that can be used for TetraMAX diagnostics. [Figure 3](#) shows a conceptual representation of a CTM of a cell with three defects: D0, D1, and D2.

Figure 3: Conceptual Representation of a CTM

Example Cell with Defects Added

Detect tables allow precise specification of detection conditions, good output values, and defects

Static Detections							
				Input condition	Good value	Defects covered	
A0	A1	B0	B1	COUT	D0	D1	D2
0	0	0	0	0	0	1	0
1	1	1	0	1	0	0	1

Dynamic Detections							
				COUT	D0	D1	D2
A0	A1	B0	B1	COUT	D0	D1	D2
0	0	F	1	0	1	0	0
0	0	F	0	0	0	1	0
1	R	1	0	1	0	0	1

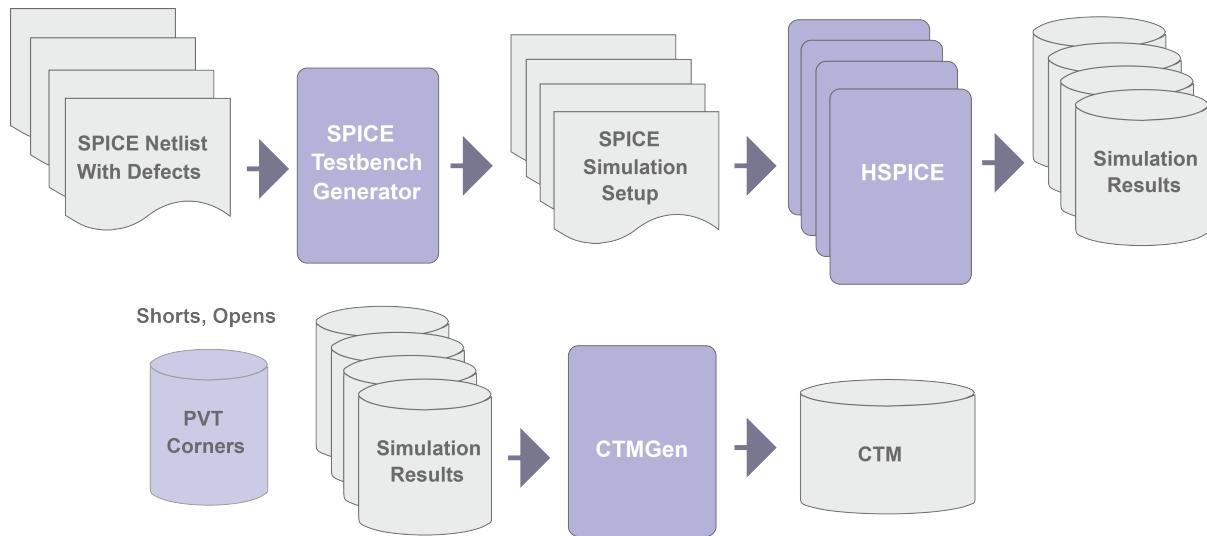
Defect list defines defect types and physical information

Defect List							
			Defect description		Physical bounding box		
Name	Type	Layer	X1	Y1	Width	Height	
D0	Open	P-Active	0.22	0.13	0.35	0.27	
D1	Short	Metal-1	0.63	0.19	0.38	0.34	
D2	Short	Metal-1	1.35	0.22	0.41	0.28	

Generating Cell Test Models

CTMs are generated using a utility called CTMGen, which is provided by TetraMAX ATPG. See the "Running CTMGen" document for details on using this utility.

To prepare for CTM generation, a SPICE testbench is generated and conditions are set up to inject defects to create faulty netlists. One testbench is used for each defect in the cell. HSPICE simulations are run on the fault-free and fault-injected netlists. The CTMGen utility processes the simulation results and generates the CTM. A library compiler compiles the source model to binary form.

Figure 4: CTM Generation Process

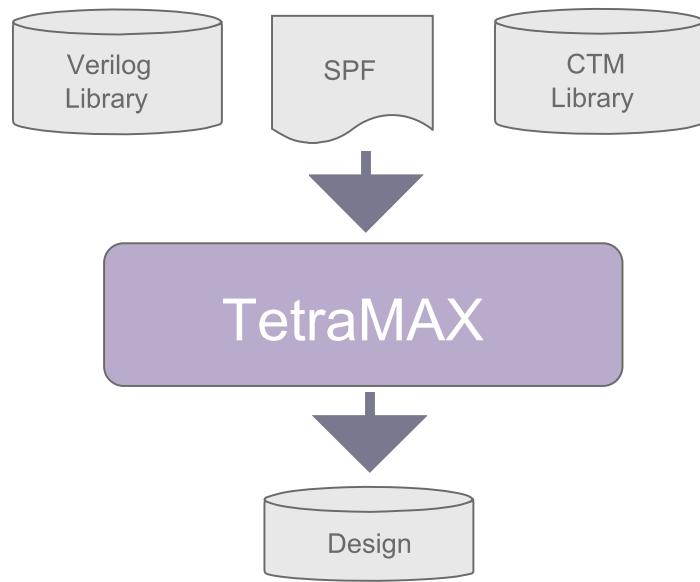
Running Cell-Aware ATPG

Cell-aware ATPG uses the existing ATPG paradigm. All defects become faults in the fault lists, and both static and dynamic defects are supported. The ATPG process uses both primary and secondary faults, and merges and fault simulates the faults.

The only additional requirement is that you use the `-cell_aware` option of the `add_faults` command.

[Figure 5](#) shows the inputs and outputs for running cell-aware ATPG, including the generated CTM library.

Figure 5: Inputs and Outputs for Running Cell-Aware ATPG



The following example shows a typical cell-aware ATPG top-off flow:

```

read_netlist ...
run_build
run_drc
...
set_faults -model <model>
read_cell_model /path/to/*.CTM
add_faults -all -cell_aware

#Fault grade existing patterns for internal cell defects
set_patterns -external my.ext.patternfile
report_summaries
run_fault_sim
report_summaries
...

#Top off patterns
set_patterns -internal
run_atpg -optimize
  
```

The next example shows a basic cell-aware ATPG flow:

```

read_netlist ...
run_build
run_drc
...
set_faults -model <model>
read_cell_model /path/to/*.CTM
  
```

```
add_faults -all -cell_aware
run_atpg -optimize
```

Cell-Aware Diagnostics

To run cell-aware diagnostics, do the following:

1. Identify a defect

Identify any defective cells and their observed behavior. First, use physical information to rule out defects on nets (such as bridges or open faults). Next, use the values observed on the cell inputs in the failing and passing patterns to characterize the defective behavior.

2. Identify defects within the cells

In this case, use the cell test model (CTM) to map defective behavior to defects modeled within the simulation

3. Review the cell input values

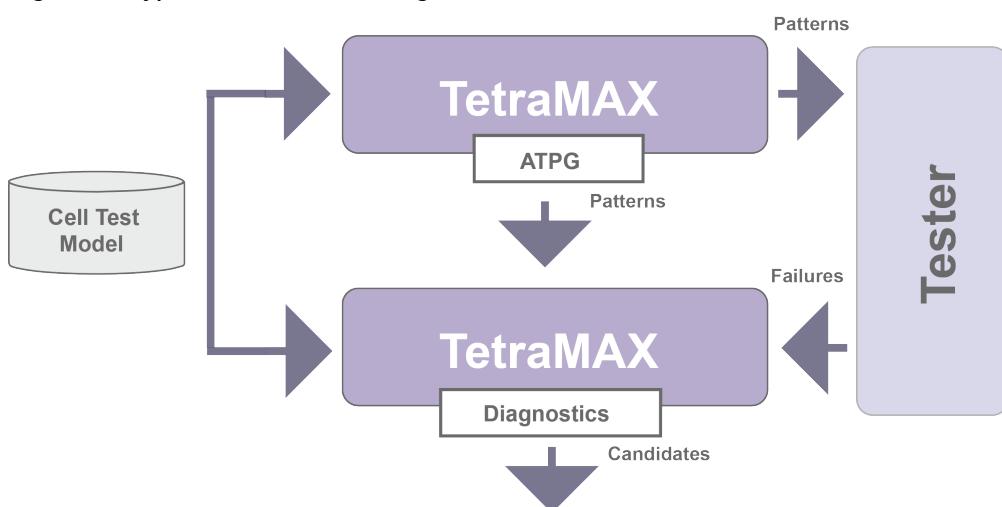
Examine the values applied to the cell inputs and consider the values in the response table. You can then create a model for the defective cell based on the observed behavior.

4. Perform Diagnostics

The only additional command you use in the diagnostics flow is the `read_cell_model` command.

Fault models commonly used for cell-aware diagnostics include stuck-at, transition, and IDQ models. After fabrication, you can use TetraMAX diagnostics to perform physical diagnostics and quickly and accurately isolate faults.

Figure 6: Typical Cell-Aware Diagnostics Flow

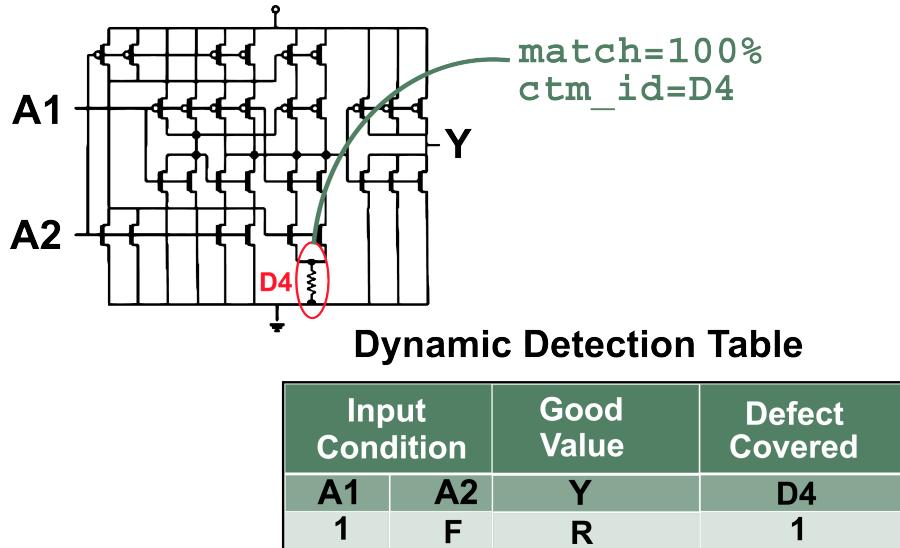


Identifying a Defect

During diagnostics, cell-aware faults are mapped to the cell excitation conditions. The CTM contains all of the necessary information to guide diagnosis, including detailed structural and behavioral annotations that enable increased diagnostic resolution.

A CTM can include both a static-detect (single-cycle) table and a dynamic-detect (two-cycle) table used for stuck-at and transition delay testing, respectively. Each table contains the input conditions required to target all the specified defects.

Figure 7 : How Cell-Aware Diagnostics Identifies an Actual Defect



Identifying Defects within a Cell

The following example shows how cell-aware diagnostics identifies defects within a cell. In this case, the detections match the observed behavior to the model.

```
CellTestModel:
- Cell: OR2
  InSignals: [A1, A2]
  OutSignals: [X]
  Defects:
    - Id: D1
      Type: short
    - Id: D2
      Type: short
    - Id: D3
      Type: short
    - Id: D4
      Type: short
    - Id: D5
      Type: short
```

```

- Id: D6
  Type: short
- Id: D7
  Type: short
- Id: D8
  Type: open
Detections:
- [Table, Static]
- [A1,A2, X, D1,D3,D4,D5,D7]
- [0, 0, 0, 1, 1, 1, 0, 1]
- [0, 1, 1, 0, 0, 1, 1, 1]
- [1, 0, 1, 1, 0, 1, 1, 0]
- [1, 1, 1, 1, 0, 1, 1, 1]
- [Table, Dynamic]
- [A1,A2, X, D2,D6,D8]
- [0, R, R, 0, 1, 0]
- [R, 0, R, 1, 0, 0]
- [0, F, F, 0, 0, 1]
- [F, 0, F, 0, 0, 1]

```

Running Diagnostics

To run TetraMAX cell-aware diagnostics, the only additional command you use in the diagnostics flow is the `read_cell_model` command. For example, the following command loads all CTMs in the `ctm_dir` directory:

```
read_cell_model ctm_dir/*.ctm
```

The following example shows how the `read_cell_model` command is used in a typical diagnostics flow:

```

read_image image101.dat
...
read_cell_model ctm_dir/*.CTM
run_diagnosis fail.txt

```

The following example shows how to use the `run_simulation` command to model a cell-aware defect and create a failure file for diagnostics:

```
run_simulation -failure_file [list outfile1 outfile2] \
  -cell_aware_fault instance_name1 defect_ID1 -replace
```

For complete information on running diagnostics, see "[Diagnosing Manufacturing Test Failures](#)."

The following example shows typical output after running cell-aware diagnostics with a CTM:

```

Diagnosis summary for failure file ./output/failure_logs/failure_
log-1.diag
#failing_pat=6, #failures=24, #defects=1, #faults=4, CPU_
time=53.05, Memory=216M
Simulated : #failing_pat=6, #passing_pat=96, #failures=24
-----
```

```

Defect 1: stuck fault model, #faults=4, #failing_pat=6, #passing_
pat=96, #failures=6
-----
match=100.00%, #explained patterns: <failing=6, passing=96>
sa0 DS frexlsdc/fr/FRCTL/p0010A1126972/Z (ddd222xss1uhd)
Internal_cell_type (cell_aware) cell_name=ddd222xss1uhd defect_
id=D68
-----
match=100.00%, #explained patterns: <failing=6, passing=96>
sa0 DS frexlsdc/fr/FRCTL/p0010A1126972/Z (hdoai222xss1uhd)
Internal_cell_type (cell_aware) cell_name=ddd222xss1uhd defect_
id=D60
-----
match=100.00%, #explained patterns: <failing=6, passing=96>
sa0 DS frexlsdc/fr/FRCTL/p0001A1126970/Z (hdao221xsslur)
Internal_cell_type (cell_aware) cell_name=ddd221xsslur defect_
id=D503
-----
match=100.00%, #explained patterns: <failing=6, passing=96>
sa0 DS frexlsdc/fr/FRCTL/p0001A1126970/Z (ddd221xsslur)
Internal_cell_type (cell_aware) cell_name=ddd221xsslur defect_
id=D504
-----
```

The following example shows the differences in output when using standard diagnostics and cell-aware diagnostics with and without a CTM.

Example without using cell-aware diagnostics:

```

-----  

Defect 2: transition fault model, #faults=1, #failing_pat=4, #passing_pat=96, #failures=4  

-----  

match=28.57%, #explained patterns: <failing=4, passing=86>  

stf  DS lym_i0/lym_i/lym_glm_i/i_lym_glm_cluster_36/U1750/Z  (t9_aoi22x1_anr)  

stf  -- lym_i0/lym_i/lym_glm_i/i_lym_glm_cluster_36/U414/B  (t9_nd3x2_anr)
```

Example using cell-aware diagnostics without a CTM:

```

Defect 2: transition fault model, #faults=1, #failing_pat=4, #passing_pat=96, #failures=4  

-----  

match=100.00%, #explained patterns: <failing=4, passing=96>  

stf  DS lym_i0/lym_i/lym_glm_i/i_lym_glm_cluster_36/U1750/Z  (t9_aoi22x1_anr)  

Internal_cell_type (driver)
```

Example using cell-aware diagnostics with a CTM:

```
Defect 2: transition fault model, #faults=2, #failing_pat=4, #passing_pat=96, #failures=4
-----
match=100.00%, #explained patterns: <failing=4, passing=96>
stf DS lym_i0/lym_i/lym_glm i/i lym_glm_cluster_36/U1750/7 (t9_aoi22x1_anr)
Internal_cell_type (cell_aware) cell_name=t9_aoi22x1_anr defect_id=D41
-----
match=100.00%, #explained patterns: <failing=4, passing=96>
stf DS lym_i0/lym_i/lym_glm i/i lym_glm_cluster_36/U1750/7 (t9_aoi22x1_anr)
Internal_cell_type (cell_aware) cell_name=t9_aoi22x1_anr defect_id=D42
```

21

Path Delay Fault and Hold Time Testing

The TetraMAX DSMTTest option enables you to use path delay fault testing to perform test generation to detect critical path delay faults. This option generates the most effective tests possible while providing the highest coverage of critical paths. TetraMAX ATPG also includes features to read, manage, and analyze paths from static timing analysis tools such as PrimeTime.

Most of the fault models supported by TetraMAX ATPG are intended to test maximum delays (or setup times), whether they are delay-based fault models (transition and dynamic bridging) or path-based fault models (path delay). Even the static fault models (stuck-at and bridging) are simulated so that the fault effect appears as a setup violation. The hold time fault model is different in that it tests minimum delays. In other respects, the hold time flow is very similar to the path delay ATPG flow.

The following sections describe path delay fault and hold time testing:

- [Path Delay Fault Theory](#)
- [Path Delay Testing Flow](#)
- [Obtaining Delay or Hold Time Paths](#)
- [Hold Time ATPG Test Flow](#)
- [Generating Path Delay Tests](#)
- [Handling Untested Paths](#)

Note: You will need a Test-Fault-Max license to use the path delay fault or hold time fault testing features. This license is also checked out if you read an image that was saved with the fault model set to path delay.

Path Delay Fault Theory

The single stuck-at fault model (stuck-at-0 or stuck-at-1) plays an important part in manufacturing test. However, you can achieve higher quality testing when you target other fault

models, such as the path delay fault model, in addition to the single stuck-at model.

The path delay fault model is useful for testing and characterizing critical timing paths in your design. Path delay fault tests exercise the critical paths at speed (the full operating speed of the chip) to detect whether the path is too slow because of manufacturing defects or variations.

Path delay fault testing targets physical defects that might affect distributed regions of a chip. For example, incorrect field oxide thicknesses could lead to slower signal propagation times, which could cause transitions along a critical path to arrive too late. By comparison, stuck-at, IDDQ, and transition delay faults are generally targeted at single-point defects.

Path delay faults are tested using the following sequence:

- The first vector initializes the path before applying the launch event, typically a clock pulse.
- The launch event generates the second vector, which propagates a logic transition along the entire path.
- A second clock pulse, occurring one at-speed cycle after the launch clock, captures the resulting transition at the end of the path.

The following sections describe the path delay fault testing theory:

- [Path Delay Fault Term Definitions](#)
- [Models for Manufacturing Tests](#)
- [Models for Characterization Tests](#)
- [Testing I/O Paths](#)

Path Delay Fault Term Definitions

[Table 1](#) lists the definitions for key terms used in path delay fault testing.

Table 1 Definitions of Terms

Terms	Definitions
at-speed clock	A pair of clock edges applied at the same effective cycle time as the full operating frequency of the device.
capture clock capture clock edge	The clock used to capture the final value resulting from the second vector at the tail of the path.
capture vector	The circuit state for the second of the two delay test vectors.
critical path	A path with little or no timing margin.

Table 1 Definitions of Terms (Continued)

Terms	Definitions
delay path	A circuit path from a launch node to a capture node through which logic transition is propagated. A delay path typically starts at either a primary input or a flip-flop output, and ends at either a primary output or a flip-flop input.
detection, robust (of a path delay fault)	A path delay fault detected by a pattern providing a robust test for the fault.
detection, non-robust (of a path delay fault)	A path delay fault detected by a pattern providing a non-robust test for the fault.
false path	A delay path that does not affect the functionality of the circuit, either because it is impossible to propagate a transition down the path (combinatorially false path) or because the design of the circuit does not make use of transitions down the path (functionally false path).
launch clock launch clock edge	The launch clock is the first clock pulse; the launch clock edge creates the state transition from the first vector to the second vector.
launch vector	The launch vector sets up the initial circuit state of the delay test.
off-path input	An input to a combinational gate that must be sensitized to allow a transition to flow along the circuit delay path.
on-path input	An input to a combinational gate along the circuit delay path through which a logic transition will flow. On-path inputs would typically be listed as nodes in the Path Delay definition file.
path	A series of combinational gates, where the output of one gate feeds the input of the next stage.

Table 1 Definitions of Terms (Continued)

Terms	Definitions
path delay fault	A circuit path that fails to transition in the required time period between the launch and capture clocks.
scan clock	The clock applied to shift scan chains. Typically, this clock is applied at a frequency slower than the functional speed.
test, non-robust	A pair of at-speed vectors that test a path delay fault; fault detection is not guaranteed, because it depends on other delays in the circuit.
test, robust	A pair of at-speed vectors that test a path delay fault independent of other delays or delay faults in the circuit.

Models for Manufacturing Tests

Path delay fault ATPG targets individual path delay faults and then simulates each test generated against the remaining undetected faults in the fault list using both robust and non-robust path delay fault models suitable for pass/fail manufacturing tests. By default, TetraMAX ATPG uses an auto relaxation scheme that provides both efficient ATPG and the flexibility of multiple path delay fault models.

The manufacturing test off-path inputs of various gates, for both the on-path input rising and falling, are shown in the following example:

```
set_delay -nodiagnostic_propagation (default manufacturing tests)
```

Path Delay Fault Class	AND Gate Off-path Inputs		
	Rising On-path Input	Falling On-path Input	
Robust (DR)	X1	S1	
Non-robust (DS)	X1	X1	

Note:

X1 : initial state don't care; final state is 1

S1: steady 1 state (hazard-free)

Path Delay Fault Class	OR Gate Off-path Inputs		
	Rising On-path Input	Falling On-path Input	
Robust (DR)	S0	X0	
Non-robust (DS)	X0	X0	

Note:

X0 : initial state don't care; final state is 0

S0: steady 0 state (hazard-free)

Path Delay Fault Class	XOR Gate Off-path Inputs			
	Rising On-path Input (rising output)	Rising On-path Input (falling output)	Falling On-path Input (falling output)	Falling On-path Input (rising output)
Robust (DR)	S0	S1	S0	S1
Non-robust (DS)	X0	X1	X0	X1

Path Delay Fault Class	MUX Gate Select Off-path Inputs		
	Rising/Falling Data0 On-path Input	Rising/Falling Data1 On-path Input	
Robust (DR)	S0	S1	
Non-robust (DS)	X0	X1	

Path Delay Fault Class	MUX Gate Data Off-path Inputs			
	Rising Select On-path Input (rising output)	Rising Select On-path Input (falling output)	Falling Select On-path Input (falling output)	Falling Select On-path Input (rising output)
Robust (DR)	S0, X1	S1, X0	X0, S1	X1, S0
Non-robust (DS)	X0, X1	X1, X0	X0, X1	X1, X0

Models for Characterization Tests

TetraMAX ATPG can also generate single-path sensitization tests that have unambiguous diagnostic results. Such tests are useful to measure individual path delays on a physical device for design characterization purposes. With these tests, any failure can be directly related to a specific path delay fault. You can determine the maximum operating frequency of each testable critical path by varying the at-speed test cycle time and associating failures to the paths being tested.

The characterization test off-path inputs of various gates, for both the on-path input rising and falling, are shown in the following example:

```
set_delay -diagnostic_propagation (characterization tests)
```

Path Delay Fault Class	AND Gate Off-path Inputs	
	Rising On-path Input	Falling On-path Input
Robust (DR)	S1	S1
Non-robust (DS)	11	11

Note:

11 : initial and final states are a 1

Path Delay Fault Class	OR Gate Off-path Inputs	
	Rising On-path Input	Falling On-path Input
Robust (DR)	S0	S0
Non-robust (DS)	00	00

Note:

00 : initial and final states are a 0

Path Delay Fault Class	XOR Gate Off-path Inputs			
	Rising On-path Input (rising output)	Rising On-path Input (falling output)	Falling On-path Input (falling output)	Falling On-path Input (rising output)
Robust (DR)	S0	S1	S0	S1
Non-robust (DS)	00	11	00	11

Path Delay Fault Class	MUX Gate Select Off-path Inputs	
	Rising/Falling Data0 On-path Input	Rising/Falling Data1 On-path Input
Robust (DR)	S0	S1
Non-robust (DS)	00	11

Path Delay Fault Class	MUX Gate Data Off-path Inputs			
	Rising Select On-path Input (rising output)	Rising Select On-path Input (falling output)	Falling Select On-path Input (falling output)	Falling Select On-path Input (rising output)
Robust (DR)	S0, S1	S1, S0	S0, S1	S1, S0
Non-robust (DS)	00, 11	11, 00	00, 11	11, 00

Testing I/O Paths

You can also use TetraMAX ATPG to generate test patterns that exercise paths from an input pin to a flip-flop or from a flip-flop to an output pin. Unlike internal paths, physical at-speed testing of I/O paths generally requires,

- High-speed, high-bandwidth ATE equipment
- A low-skew test fixture
- Very accurate placement of input signal edges
- Very accurate placement of output strobe delays

It is also important to be aware that the electrical environment of the test fixture might differ significantly from the system in which the device was designed to operate. Consequently, issues such as poorly terminated transmission lines and output driver simultaneous-switching current might cause excessive ringing on the input pins and additional delays on the output pins.

For these reasons, at-speed testing is not recommended for I/O paths unless ATE expertise exists for general high-speed testing issues and the electrical requirements for test fixtures are well understood in advance of their design.

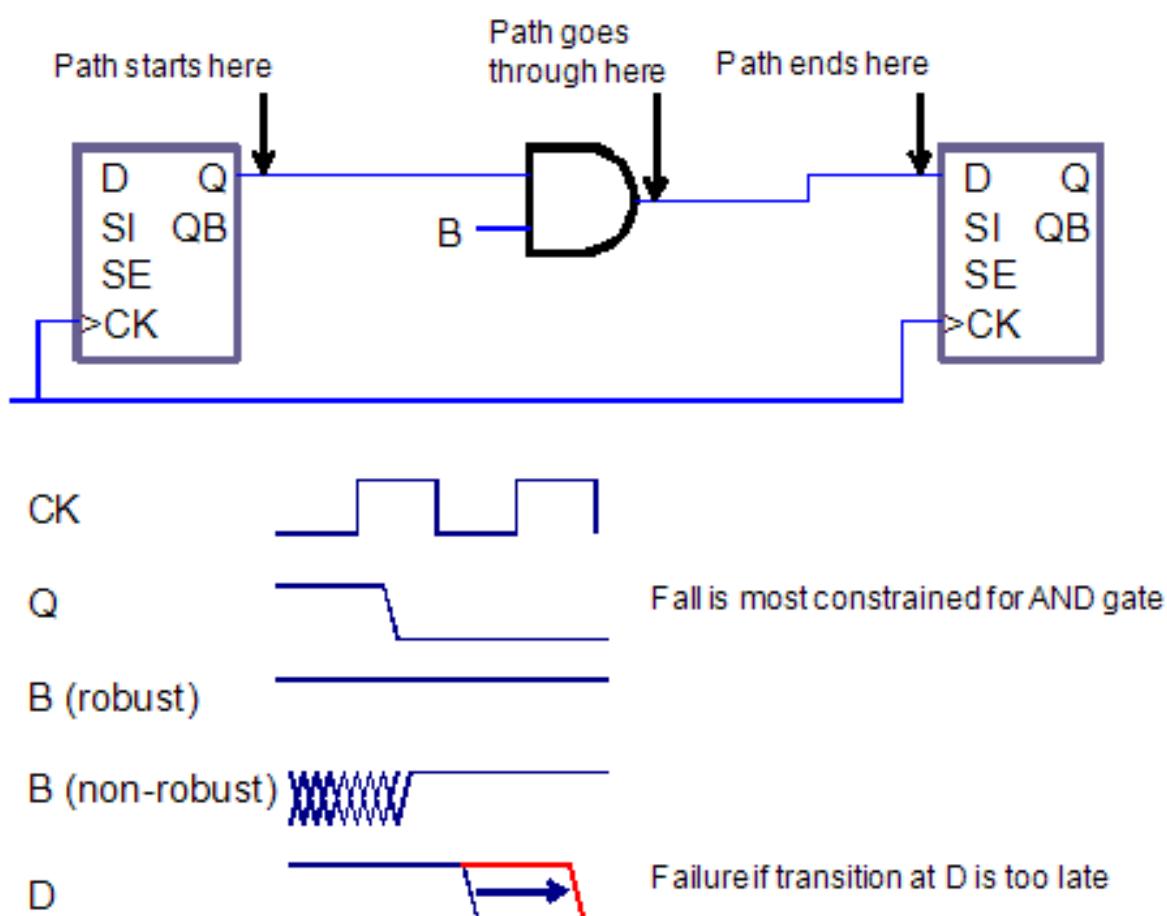
Path Delay Test Patterns

All delay paths must start with a state element (DFF, DLAT, or memory) and must end with an edge-triggered state element (DFF or edge-triggered memory); only combinational gates can be situated between the starting and ending elements. The source and destination points must capture on the same edge of the same clock. If the source and destination points are clocked by different clocks, the clocks must be either synchronized internal clocks (see [“Specifying Synchronized Multi Frequency Internal Clocks”](#)) or equivalent external clocks (see the description of the `add_pi_equivalences` command in TetraMAX Online Help). If these conditions are not satisfied, the path is declared ATPG Untestable (fault status AN).

The edge information provided in the path file is only used for the source point of the path. If the path goes through XOR gates or multiple paths, then the polarity at the destination point and the path actually taken by the transition might differ from what was specified.

In the fault modeled by the TetraMAX fault simulator, the launching node makes its transition too early. The captured node is assumed to be on time, and all off-path inputs are also assumed to be on time. If these assumptions result in a 0/1 difference in the output, then the fault is detected. See the representations of a path delay test pattern in [Figure 1](#).

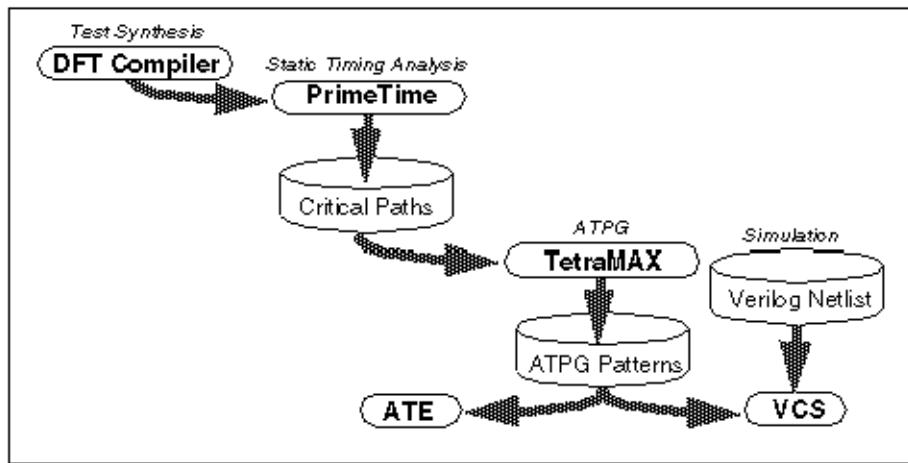
Figure 1 Path Delay Test Pattern



Path Delay Testing Flow

PrimeTime generates the critical path information you need to input for a path delay ATPG test run as shown in [Figure 1](#).

Figure 1 Path Delay Test Flow

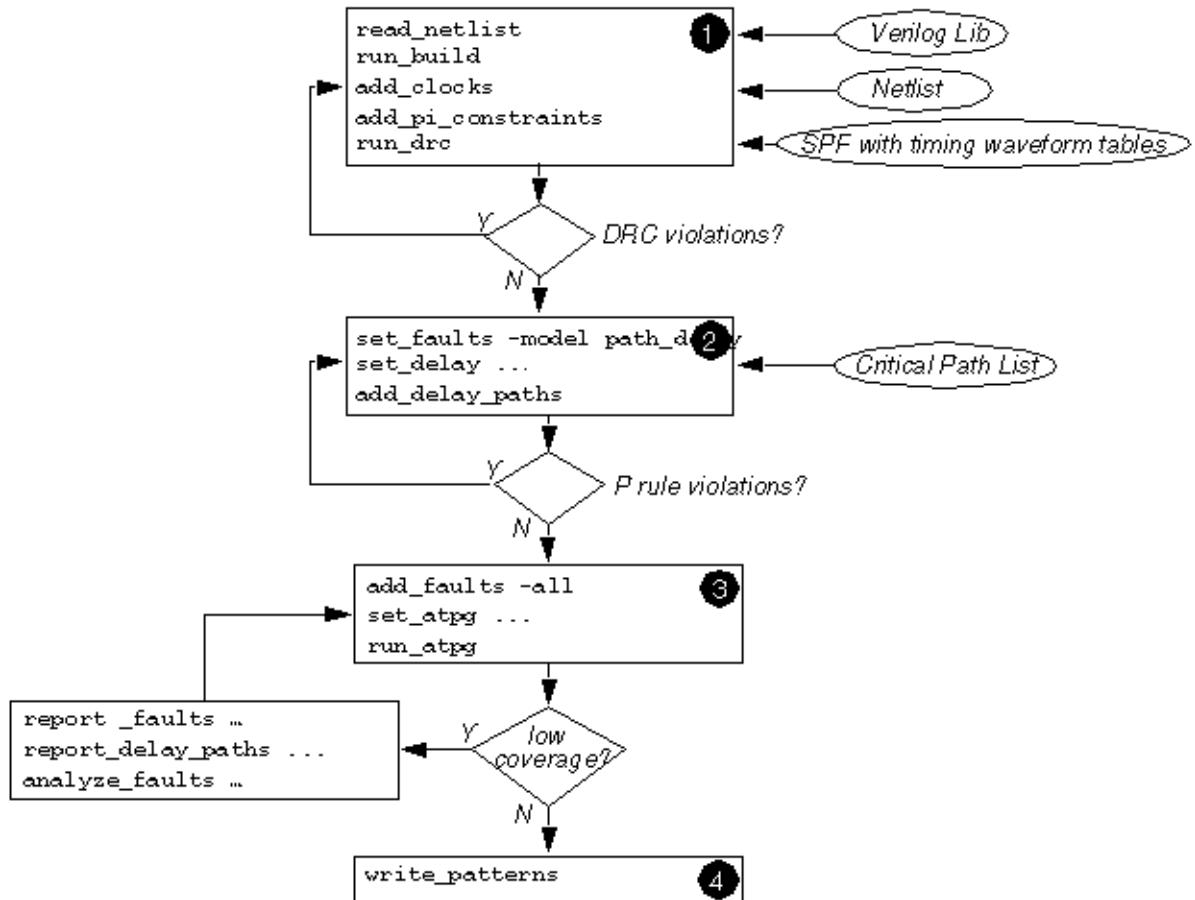


TetraMAX ATPG supports ATPG and fault simulation for scan-based path delay fault testing with the following features:

- Reads critical paths reported by PrimeTime
- Supports a comprehensive set of path (P) rules
- Most rule violations can be analyzed and debugged in the GSV
- Clock waveforms in the STL procedure file are checked to ensure they match static timing analysis conditions
- Identifies combinational false paths and other untestable paths
- Generates a full range of tests supporting both robust and non-robust path delay fault models

[Figure 2](#) shows the basic TetraMAX ATPG steps and checkpoints to generate an effective set of path delay tests.

Figure 2 Path Delay Test Generation Flowchart



Launch and capture events are pertinent only to transition and path delay fault environments. If the fault model is set to the default model (stuck), then the launch and capture events are likely to be dropped. TetraMAX ATPG will attempt to maintain this information when possible. However, because of the variety of flows and the ability to process patterns generated for one fault model under a different model (for instance, regrading transition patterns under a stuck model), care must be exercised if this information needs to be maintained. Before the `write_patterns` operation is executed in the file that reads-back the binary patterns, add the `set_faults -model transition` command. Then, the launch and capture events will remain across all outputs.

Obtaining Delay Paths

TetraMAX ATPG requires an input list of critical paths to target for path delay fault generation. TetraMAX ATPG can read an ASCII file containing the critical paths reported by a static timing analysis tool, such as PrimeTime, or you can specify these paths manually in an ASCII file.

To obtain a list of critical delay paths, use the `write_delay_paths` Tcl procedure, which is part of the `pt2tmax.tcl` file. This process is described in detail in the "Importing PrimeTime Path Lists" section.

A description of the ASCII file used to specify delay paths is in the "[Path Definition Syntax](#)" section.

For details on translating timing exceptions, see "[Specifying Timing Exceptions From an SDC File](#)".

Hold Time ATPG Test Flow

The TetraMAX DSMTTest option enables you to use hold time testing to perform test generation to detect critical path minimum delays. This option generates the most effective tests possible while providing the highest coverage of critical paths. TetraMAX ATPG also includes features to read, manage, and analyze paths from static timing analysis tools such as PrimeTime.

Most of the fault models supported by TetraMAX ATPG are intended to test maximum delays (or setup times), whether they are delay-based fault models (transition and dynamic bridging) or path-based fault models (path delay). Even the static fault models (stuck-at and bridging) are simulated so that the fault effect appears as a setup violation. The hold time fault model is different in that it tests minimum delays. In other respects, the hold time flow is very similar to the path delay ATPG flow

The hold time fault model is different in that it tests minimum delays. In other respects, the hold time flow is very similar to the path delay ATPG flow .

Note: You will need a Test-Fault-Max license to use the path delay fault or hold time fault testing features. This license is also checked out if you read an image that was saved with the fault model set to path delay.

The hold time ATPG test flow is the same as the path delay ATPG flow, except that instead of running the `set_faults -model path_delay` command, you need to specify the `set_faults -model hold_time` command. The `hold_time` argument specifies the ATPG and fault simulation commands to use the hold time fault model and must be specified before you add faults.

The standard hold time ATPG flow includes the following commands:

- `run_drc`
- `set_faults -model hold_time`
- `add_delay_paths hold_path_file`
- `add_faults -all`
- `run_atpg`

You can use normal reporting commands such as `report_summaries faults` and `report_delay_paths`. The fault types are reported as FTF (fast to fall) and FTR (fast to rise).

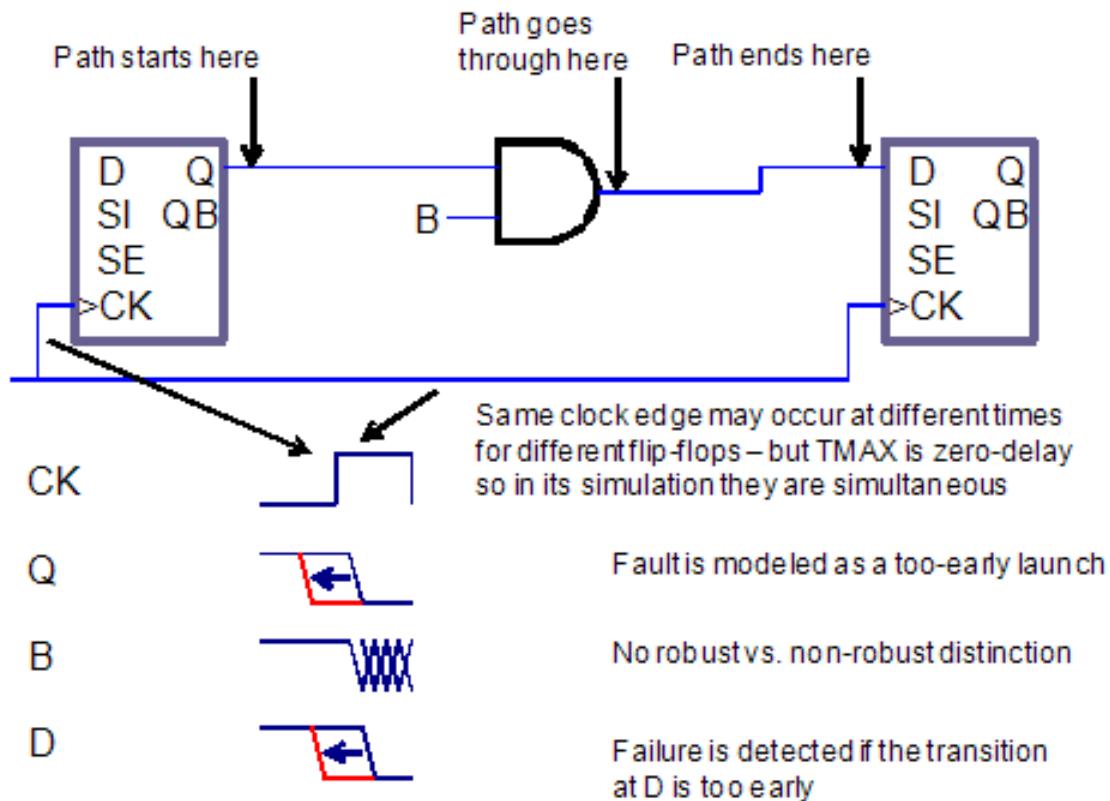
In the hold time ATPG test flow, all `set_delay` commands are ignored because the hold time path transition is launched and captured in a single clock cycle. Hold time faults are usually detected by the basic scan pattern type, although fast-sequential ATPG is also supported. Multiple-clock patterns are generated when the hold time path must be set up or when its effects are propagated through nonscan elements such as memories.

Note: You can usually reduce the pattern count by first fault-simulating the stuck-at patterns with the hold time fault model, and then using ATPG to create new patterns to detect the undetected paths.

All paths must start with a state element (DFF, DLAT, or memory) and must end with an edge-triggered state element (DFF or edge-triggered memory); only combinational gates can be situated between the starting and ending elements. The source and destination points must capture on the same edge of the same clock. If the source and destination points are clocked by different clocks, the clocks must be either synchronized internal clocks (see “[Specifying Synchronized Multi Frequency Internal Clocks](#)”) or equivalent external clocks (see the description of the `add_pi_equivalences` command in TetraMAX Online Help). If these conditions are not satisfied, the path is declared ATPG Untestable (fault status AN).

The edge information provided in the path file is only used for the source point of the path. If the path goes through XOR gates or multiple paths, then the polarity at the destination point and the path actually taken by the transition might differ from what was specified.

In the fault modeled by the TetraMAX fault simulator, the launching node makes its transition too early. The captured node is assumed to be on time, and all off-path inputs are also assumed to be on time. If these assumptions result in a 0/1 difference in the output, then the fault is detected. See the representations of a path delay test pattern in [Figure 1](#) and a hold time test pattern in [Figure 2](#).

Figure 2 Hold Time Test Pattern

Generating Path Delay Tests

The following sections describe how to generate path delay tests:

- [Flow for Generating Path Delay Tests](#)
- [Using set_delay Options](#)
- [Reading and Reporting Path Lists](#)
- [Analyzing Path Rule Violations](#)
- [Viewing Delay Paths](#)
- [Path Delay ATPG Options](#)
- [Internal Loopback and False/Multicycle Paths](#)
- [Creating At-Speed WaveformTables](#)
- [Maintaining At-Speed Waveform Table Information](#)
- [MUXClock Support for Path Delay Patterns](#)

Flow for Generating Path Delay Tests

The following steps show you how to generate a set of path delay tests:

1. Start TetraMAX ATPG.
2. Read in the libraries and netlists.
3. Build a circuit model.
4. Run DRC (you can use delay waveform tables in the STL procedure file):

```
run_drc filename.spf
```

5. Depending on the ATE functionality, set the delay testing options:

```
set_delay -nopi_changes  
set_delay -nopo_measures
```

6. Read in the delay paths:

```
add_delay_paths filename
```

7. Analyze any P rule errors or warnings.

Use the following command to remove any paths that were read:

```
remove_delay_paths pathname
```

8. Display a delay path (optional):

```
report_delay_paths path_name -display -pindata
```

9. Add path delay faults:

```
set_faults -model path_delay
```

10. Run ATPG:

```
run_atpg -auto
```

11. Analyze low path delay coverage (optional):

```
report_faults -class AU  
analyze_faults path_name -slow rise -display -verbose -fault_simulation
```

12. Write the path delay test patterns:

```
write_patterns patname.stil -format stil  
write_patterns patname.wgl -format wgl
```

Note: Many of the commands described in this flow have other options that you can use to adjust TetraMAX ATPG to your unique requirements.

Using `set_delay` Options

After passing DRC, and before reading in a list of critical paths, you can use the `set_delay` command to specify any options related to path delay testing.

Note that the launch cycle setting has no effect on full-sequential ATPG. TetraMAX ATPG uses either a last-shift or a system clock for the launch cycle. To prevent last-shift launch behavior, constrain the scan enable signal to its inactive value using the `add_pi_constraints` command.

Reading and Reporting Path Lists

After setting the delay options, you can read delay faults into TetraMAX ATPG using the `add_delay_paths` command. This command reads in a path delay definition file. You can remove paths from memory with the `remove_delay_paths` command. To display paths in text format, use the `report_delay_paths` command. By using the `-verbose` option, you can include in the report information regarding launch and capture clocks and nodes, transition direction of faults, fault status, and the vector in which detection took place.

Analyzing Path Rule Violations

You can analyze the P rule violations using the GSV. For example, to view additional information on P20 violations, enter the following commands:

```
report_violations P20  
analyze_violations P20-3
```

Viewing Delay Paths

You can use the `report_delay_paths path_name -display -pindata` command to report delay paths and view them in the GSV. The displayed data includes any path requirements (transitions and conditions) annotated to the wires of the design or primitive elements in the path.

Path Delay ATPG Options

Fast-sequential ATPG is the default for path delay tests and usually provides adequate coverage of most testable paths. In some cases, full-sequential ATPG can achieve slightly higher coverage. The recommended flow is to first generate path delay patterns with fast-sequential ATPG , then top off with full-sequential patterns, if they provide improvement. You can enable full-sequential ATPG using the `-full_seq_atpg` option of the `set_atpg` command. The following options can improve vector generation and pattern compression with full-sequential ATPG:

```
set_atpg -full_seq_abort_limit seq_max_remade_decs  
set_atpg -full_seq_time max_secs_per_fault  
set_atpg -full_seq_merge [ low | medium | high ]
```

If the fault report printed after ATPG indicates that some faults were aborted (undetected), you can increase the time limit beyond 10 seconds (the default), and rerun ATPG on the remaining faults. Raising the merge effort allows TetraMAX ATPG to generate fewer vectors for the same fault coverage. The default is to not merge patterns.

Internal Loopback and False/Multicycle Paths

You can generate transition and path delay tests while ensuring that you will not get tests that "loopback" through a bidirectional port or tests for false/multicycle paths that begin at a specific

start point. The following six commands implement this capability:

```
add_slow_bidis port_name | -all>
remove_slow_bidis port_name | -all>
report_slow_bidis
add_slow_cells instance_path | gate_id
remove_slow_cells instance_path | gate_id | -all
report_slow_cells
```

The `add_slow_bidis` command modifies the associated BUS primitives to output an X if any tristate driver (TSD) or switch (SW) primitives are not driving a Z onto the BUS primitive. The value observed on the primary inout (PIO) primitive continues to be the resolved value of the BUS primitive before this masking operation. If all TSD and SW primitives are driving a Z onto the BUS primitive, the BUS behavior is not modified. This includes the behavior if the PIO primitive is also driving a Z, or if there are weak input values.

An error message is issued if the `add_slow_bidis` command is specified for a port that is not an inout or does not exist. The `add_slow_bidis -all` command issues a message showing the number of ports modified.

The `add_slow_cells` command modifies the simulation behavior of DFF or DLAT cells in two ways:

- For Basic-Scan patterns, the DFF/DLAT gets loaded with an X if the adjacent scan cell (closer to the scan out) is being loaded with a different value (that is, if the last scan shift creates a transition on the DFF/DLAT output). The capture and unload behavior of the DFF/DLAT is not modified. When setting a scan cell value with this attribute, Basic-Scan ATPG also attempts to set the adjacent scan cell with this same value before pattern merging, if it has not already been set.
- For Fast-Sequential and Full-Sequential patterns, the DFF/DLAT outputs an X if data captured by a clock changes the state of the DFF/DLAT, or if a set/reset changes the state of the DFF/DLAT. The DFF or DLAT continues to output an X until the next load operation. However, the capture and internal state behavior is not modified and this internal state value, not an X, is observed by an unload operation. Full-sequential ATPG will continue to apply the “robust fill” algorithm before random fill. This decreases the probability that the launch clock creates a transition from scan cells feeding off-path inputs, including any with this attribute.

Creating At-Speed WaveformTables

Path delay tests are generated during both the fast-sequential and full-sequential test modes. These tests conform to user constraints through defined clocks and specified primary input constraints. The timing for these vectors adhere to one of several timing WaveformTables in the STIL procedure file.

If there are no additional waveform tables in the STL procedure file, then the default timing (`_default_WFT_`) is used for all path delay test vectors. However, special timing can be defined for the launch and capture events in ancillary timing waveform tables. These tables are as follows:

- `_launch_WFT_`
- `_capture_WFT_`
- `_launch_capture_WFT_`

When using generic capture procedures, the `allclock_launch`, `allclock_capture`, and `allclock_launch_capture` procedures are used. Each procedure calls a WFT specifically associated with it.

Each table can use different timing definitions for inputs, clocks, and output strobes. The path delay test vectors can use these timing definitions when applied to the device under test to detect faults defined in the path definition file.

The following example shows a `_capture_WFT_` timing WaveformTable in the context of a STL procedure file:

```
Timing {
    WaveformTable "_default_WFT_" {
        Period '100ns';
        Waveforms {
            "TxClk" { 01Z { '0ns' D/U/Z; } }
            "TxClk" { P { '0ns' D; '50ns' U; '80ns' D; } }
            "_default_In_Timing_" {01ZN {'0ns' D/U/Z/N; } }
            "_default_Out_Timing_" {X {'0ns' X; } }
            "_default_Out_Timing_" { HLT { '0ns' X; '4ns' H/L/T; } }
        }
    }
    WaveformTable "_capture_WFT_" {
        Period '20ns';
        Waveforms {
            "TxClk" { 01Z { '0ns' D/U/Z; } }
            "TxClk" { P { '0ns' D; '5ns' U; '10ns' D; } }
            "_default_In_Timing_" {01ZN {'0ns' D/U/Z/N; } }
            "_default_Out_Timing_" {X {'0ns' X; } }
            "_default_Out_Timing_" { HLT { '0ns' X; '4ns' H/L/T; } }
        }
    }
}
```

A path delay test cycle uses the same order of events as for other fault models:

- Force primary inputs
- Measure primary outputs (optional)
- Pulse a clock

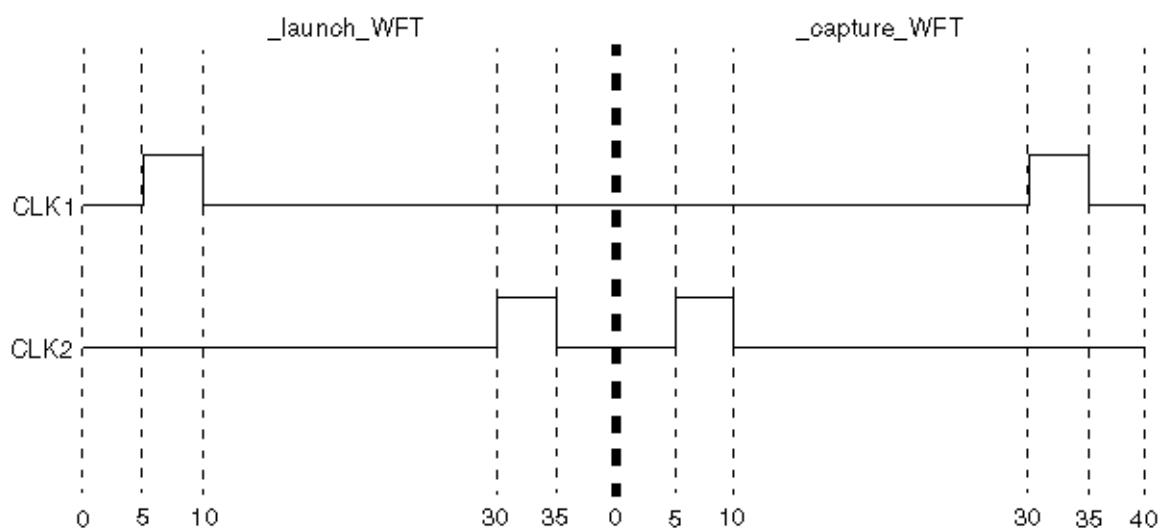
Given this order of events, one or two test cycles are required to launch and capture a path delay fault. For most paths, a two-cycle test is generated to apply a launch clock pulse and a capture clock pulse. However, a full-sequential delay fault requiring a launch on the rising (leading) edge of a clock and a capture on the falling (trailing) edge of the same clock generates a one-cycle test that uses the `"_launch_capture_WFT_"`. For a delay path fault test that requires a launch in one

clock domain and a capture in another clock domain, two vectors are generated, and thus use “`_launch_WFT_`” for the launch vectors, and “`_capture_WFT_`” for the capturing vector.

If two or more different at-speed frequencies need to be used for different clock domains within your design, you might consider the following example WaveformTable definition. This example shows two input clocks with their launch and capture timing defined (see [Figure 1](#)).

```
WaveformTable "_launch_WFT_" {
    Period '40ns';
    Waveforms {
        "CLK1" { 01Z { '0ns' D/U/Z; } }
        "CLK1" { P { '0ns' D; '5ns' U; '10ns' D; } }
        "CLK2" { 01Z { '0ns' D/U/Z; } }
        "CLK2" { P { '0ns' D; '30ns' U; '35ns' D; } }
        "_default_In_Timing_"{01ZN {'0ns' D/U/Z/N; } }
        "_default_Out_Timing_"{X {'0ns' X; } }
        "_default_Out_Timing_" { HLT { '0ns' X; '4ns' H/L/T; } }
    }
}
WaveformTable "_capture_WFT_" {
    Period '40ns';
    Waveforms {
        "CLK1" { 01Z { '0ns' D/U/Z; } }
        "CLK1" { P { '0ns' D; '30ns' U; '35ns' D; } }
        "CLK2" { 01Z { '0ns' D/U/Z; } }
        "CLK2" { P { '0ns' D; '5ns' U; '10ns' D; } }
        "_default_In_Timing_"{01ZN {'0ns' D/U/Z/N; } }
        "_default_Out_Timing_"{X {'0ns' X; } }
        "_default_Out_Timing_" { HLT { '0ns' X; '4ns' H/L/T; } }
    }
}
```

Figure 1 Two Different At-Speed Times



Maintaining At-Speed Waveform Table Information

The presence of launch and capture operations is pertinent only under transition and path delay environments. To ensure this information remains in a pattern set through various flows, such as importing patterns into TetraMAX ATPG (see “[Selecting the Pattern Source](#)”), specify the appropriate fault model for these patterns. See “Specifying Transition-Delay Faults” for transition patterns for the appropriate `set_faults -model` command.

MUXClock Support for Path Delay Patterns

Testing of internal paths in DSMTTest requires that the system clock be applied at-speed to the device under test. MUXClock, a common technique for applying the system clock at-speed, merges (or multiplexes) two patterns within a single, uniform cycle to create the at-speed clock. MUXClock is supported only for full-sequential ATPG, so you must use the `set_atpg -full_seq_atpg -nofast_path_delay` command.

For MUXClock vector formatting, two additional clock waveforms D (double) and E (early) need to be defined for the at-speed test. Definitions of the waveforms used during scan chain shifting and normal (slow) system cycles are contained in the STIL procedure file.

MUXClock is a single waveform table/timeset construct that eliminates switching waveform tables between the default path delay waveform tables for launch, capture, and launch_capture operations and reduces requirements on ATE to support this timing flexibility.

By overlapping both the launch and capture events in one tester cycle, it is possible for ATE timing accuracy to be higher than across multiple vectors. Also, it is possible to place the launch and capture events closer together in a single vector than normally permitted when separate vectors were required. This feature, however, requires testers to support flexible double-pulse definitions in STIL, and relies on MUX constructs in WGL that tie multiple tester channels together to generate a flexible double-pulse waveform.

The following formats are supported by the MUXClock technique:

- WGL (using the WGL “:mux” construct)
- STIL (using multiple pulsed waveforms P, E, and D)
- MUXClock is not supported with `clock_grouping`
- MUXClock is not supported with scan compression designs

Enabling MUXClock Functionality

The waveform table sections in the STL procedure file need to be modified to support MUXClock behavior for delay test vectors. The typical waveform table section specifies values that are applied during the scan shift and normal system tester cycles. Two additional waveform definitions are required to specify the at-speed clock.

Delay Test Vector Format

The following example shows a WaveformTable section for the MUXClock technique:

```
Timing {
    WaveformTable "_default_WFT_" {
```

```

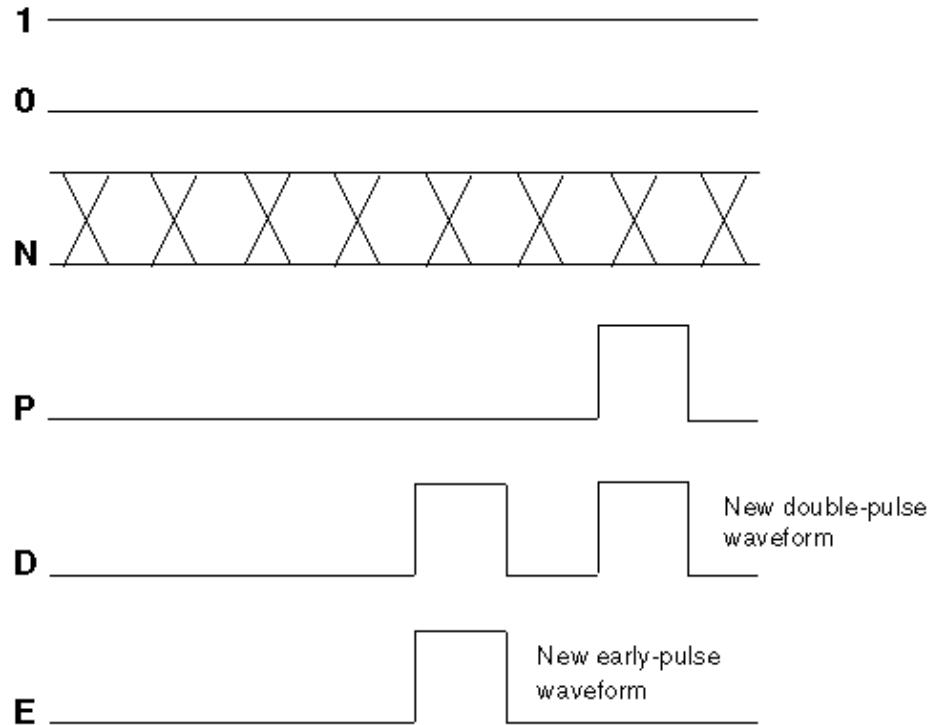
Period '100ns';
Waveforms {
    "all_inputs" { 0 { '0ns' D; } }
    "all_inputs" { 1 { '0ns' U; } }
    "all_inputs" { Z { '0ns' Z; } }
    "all_outputs" { X { '0ns' X; } }
    "all_outputs" { H { '0ns' X; '40ns' H; } }
    "all_outputs" { T { '0ns' X; '40ns' T; } }
    "all_outputs" { L { '0ns' X; '40ns' L; } }
    "CK" { P { '0ns' D; '75ns' U; '85ns' D; } }
    "CK" { D { '0ns' D; '45ns' U; '55ns' D; '75ns' U;
'85ns' D; } }
    "CK" { E { '0ns' D; '45ns' U; '55ns' D; } }
}
}
}

```

In the waveform table, all signals identified as clocks in the design must have two additional waveforms present. These waveforms use the WaveformCharacters D (double-pulse), and E (early-pulse). This brings the count of pulsed-waveforms for clocks up to 3: P, D, and E. These pulses have the following requirements:

- The edges of the E pulse must align with the edges of the first D pulse
- The edges of the P definition must align with the edges of the second D pulse

Also, the timing of all pulses, including the E pulse, must occur after the timing of the input edges and the output measures. In MUXClock mode, all path delay launch and capture operations are performed in a single cycle (described next); therefore, the timing of all events must follow the forcePI/measurePO/clock-pulse sequence. Because there is only one cycle, an option to define multiple cycles does not exist. Visually, the set of waveforms for an active-high clock to define an MUXClock operation appear similar to that shown in Figure 2

Figure 2 MUXClock: Active-High Clock Waveforms

When MUXClock waveforms have been defined, the WGL output will contain references to two WGL muxparts for each clock signal in the design. An example of this construct for the WGL signals and timeplate sections is as follows.

```
"CK" [ "CK_Epulse", "CK_Ppulse" ] :mux input;
-
-
-
timeplate "_default_WFT_" period 100ns
"CK_Ppulse" := input [0ps:D, 75ns:S, 85ns:D];
"CK_Epulse" := input [0ps:D, 45ns:S, 55ns:D];
-
-
-
```

Limitations of MUXClock Support for Path Delay Patterns

The following limitations apply to MUXClock support for path delay patterns:

- Output pattern files containing MUXClock waveforms are not yet readable in TetraMAX ATPG.

- Bidirectional clocks in a design are not supported in WGL output when MUXClock definitions are present. STIL output supports bidirectional clocks in the design.

ATPG Requirements to Support MUXClock

For MUXClock to function, the `set_delay` command options `-nopi_changes` and `-nopo_measures` must be used. In MUXClock mode, there can be no change of PI state or detectable PO information between the end-of-launch and the start-of-capture: the only event that can happen in the capture operation is a clock pulse.

Handling Untested Paths

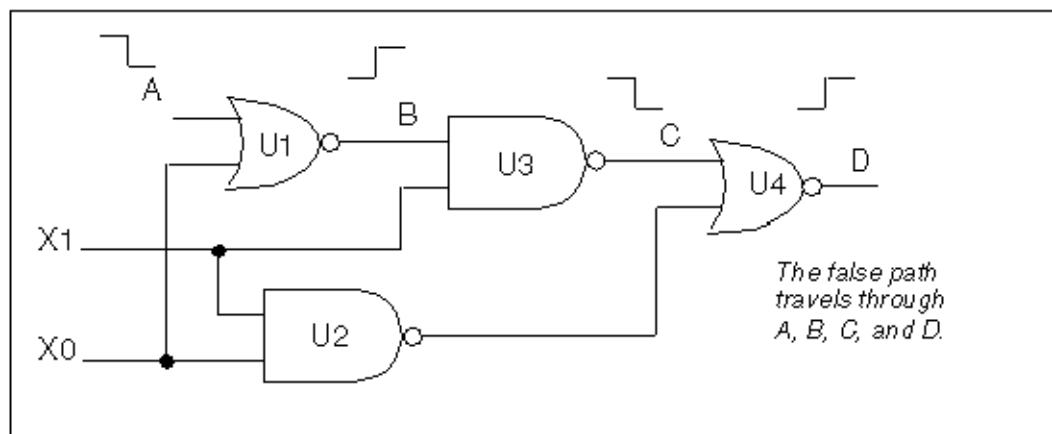
The following sections explain false and untestable paths, and describes how you can handle them:

- [Understanding False Paths](#)
- [Understanding Untestable Paths](#)
- [Reporting Untestable Paths](#)
- [Analyzing Untestable Faults](#)

Understanding False Paths

A false path might be caused by a portion of combinational logic that is configured so a path can never be fully exercised. In other words, the path can never propagate a high-to-low or low-to-high transition from the startpoint to the endpoint. [Figure 1](#) illustrates a false path, ABCD.

Figure 1 False Path Example



The transition cannot be propagated to output D because of a blockage created by the X0 pin driving U2 and subsequently U4. TetraMAX ATPG identifies组合ally false paths when reading paths and classifies the associated path delay fault as undetectable-redundant (UR). False paths will also be flagged with a P21 rule violation (on-path values not satisfiable).

Understanding Untestable Paths

TetraMAX DSMTTest might prove a path delay fault to be untestable for one of the following reasons:

- It is a sequentially false path. Such paths cannot be tested in a functional mode, because logic prevents the required state transitions.
- ATPG constraints or tester limitations might prevent some true paths from being tested. DSMTTest restrictions must adhere to all ATPG constraints.
- Redundant logic (for circuit speed) prevents a single path from being independently tested. Multiplier arrays are a good example of such circuits.

Note: If there are reconverging paths, you might want to use the `-allow_reconverging_paths` option to the `set_delay` command. The default is `-noallow_reconverging_paths`

- Paths that require multiple launch or capture events are not usually supported by TetraMAX ATPG and are declared untestable.
- Note:** Multicycle paths can often be tested if the appropriate clock timing is applied
- Other TetraMAX ATPG restrictions might cause paths to be declared untestable. These paths are usually flagged with a P-rule violation.
- Paths through RAMs or ROMs modeled with memory primitives are not supported by TetraMAX ATPG and are declared untestable.

Reporting Untestable Paths

A specific path delay fault might not be testable due to either a path rule violation or a failure of path delay ATPG to find a test for the path. You can generate a list of P rule violations using the `report_violations P` command. For analyzing undetectable and untestable paths, check the results of rules P19, P20, P21, P22, P23, and P24.

The following example shows how to review untestable paths after ATPG:

```
TEST-T> report_faults -class AU
str AN path8
str AN path9
```

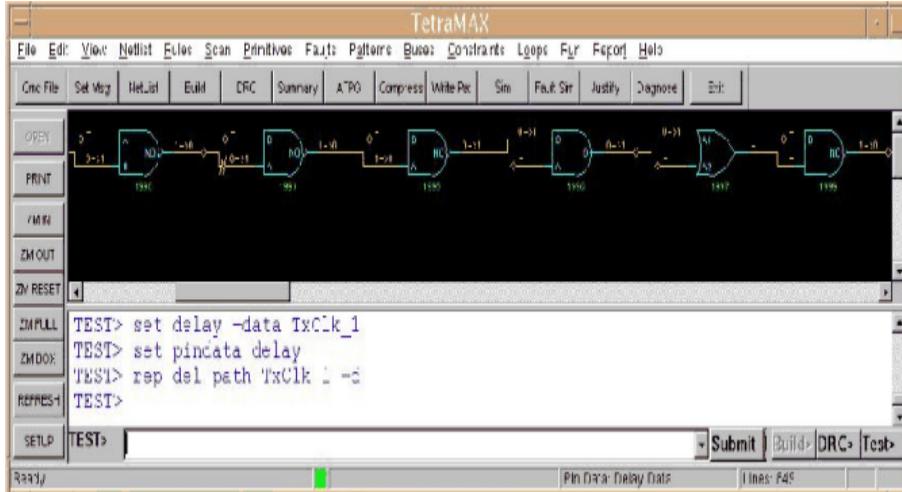
To display the delay for a particular path use the following command:

```
TEST-T> report_delay_paths path_name -verbose -display
```

Adding the `-display` option to the command displays the path in the GSV, where you can annotate ports with delay path data by selecting delay data from the pindata list in the Setup dialog box or by including the `-pindata` option.

[Figure 2](#) shows a path being displayed in the GSV.

Figure 2 Path Display Example

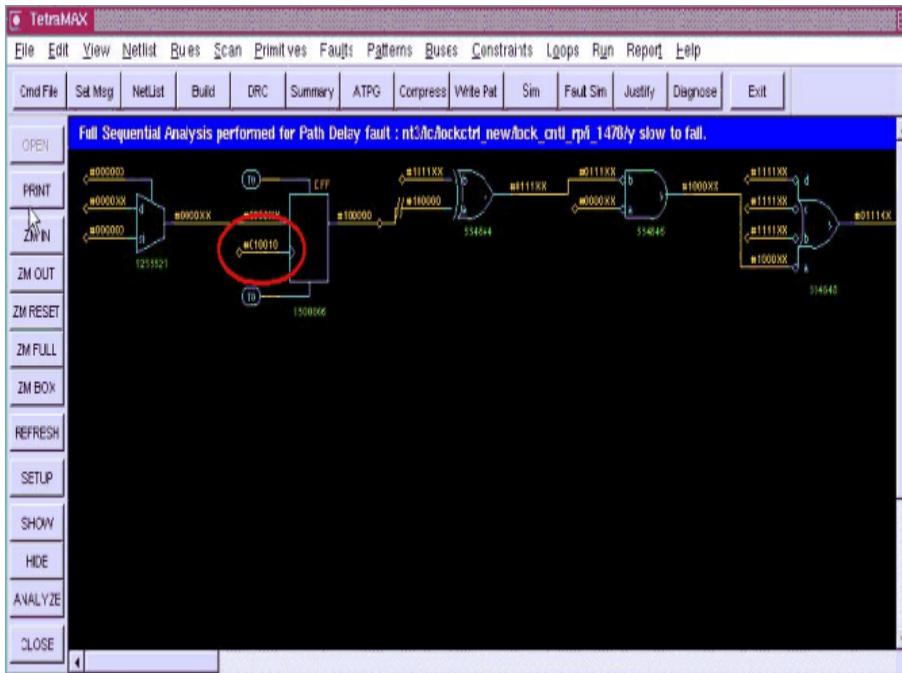


Analyzing Untestable Faults

If a fault has been classified as ATPG Untestable, you can use the `analyze_faults path_name -slow r | f` command to display the path in the GSV so you can analyze it. For example, [Figure 3](#) shows the displayed result of entering

```
analyze_faults CLK_0 -slow f -display
```

Figure 3 Analyze Untestable Faults Display



TetraMAX ATPG performs fault analysis for the specified path. An incremental approach to test generation is pursued in which the sensitization requirements for the path are attempted one node at a time, until a path node is added that causes a justification failure or a test is generated. With the `-display` option, pattern values for the last successful justification are shown in the GSV for the specified path.

TetraMAX Commands for Path Delay Fault Testing Example

In this example, the scan enable signal is constrained as in the transition fault testing using system clock launch. However, this step is not needed if the circuit can support last shift launch.

This example also uses commands specific to path delay testing such as the following:

- The `set_delay -mask_nontarget_paths` command, which ensures that TetraMAX ATPG does not generate expected values on multi cycle or false paths
- The `set_delay -relative_edge` command, which causes TetraMAX ATPG to inject both a slow-to-rise and a slow-to-fall fault for each path when you run the `add_faults -all` command

The following example also shows the pattern reporting commands that are unique to path delay testing:

```
read_netlist ckt.v

run_build_model test_ckt

set_delay -nopi_changes -nopo_measures # if needed
set_delay -mask_nontarget_paths
set_delay -common_launch_capture_clock # if needed
set_delay -relative_edge                 # if required

add_capture_masks dff0    # if needed
add_slow_cells dff1      # if needed
add_slow_bidi -all

add_pi_constraints 0 scan_enable

run_drc ckt.spf

add_delay_paths ckt.paths

set_faults -model path
add_faults -all

run_atpg -auto

report_patterns -all -path_delay # if required
report_patterns -all -slack   # if required

# You can optionally run the following command
analyze_faults path0 -slow r -verbose -display -fault_sim
```

22

Quiescence Test Pattern Generation

TetraMAX ATPG allows you to generate test patterns specifically targeted for quiescence, or IDDQ, testing. You can also verify IDDQ test patterns and choose IDDQ strobe points in existing patterns for maximum fault coverage.

The following topics describe the process for IDDQ test pattern generation:

- [Why Do IDDQ Testing?](#)
- [About IDDQ Pattern Generation](#)
- [Limitation](#)
- [Fault Models](#)
- [DRC Rule Violations](#)
- [Generating IDDQ Test Patterns](#)
- [Using IDDQ Commands](#)
- [IDDQ Bridging](#)
- [Design Principles for IDDQ Testability](#)

Why Do IDDQ Testing?

IDDQ testing can detect certain types of circuit faults in CMOS circuits that are difficult or impossible to detect by other methods. IDDQ testing, when used to supplement standard functional or scan testing, provides an additional measure of quality assurance against defective devices.

IDDQ testing detects circuit faults by measuring the amount of current drawn by a CMOS device in the quiescent state (a value commonly called “I_{ddQ}”). If the circuit has been designed correctly, this amount of current is extremely small. A significant amount of current indicates the presence of one or more defects in the device.

The following sections describe IDDQ testing in detail:

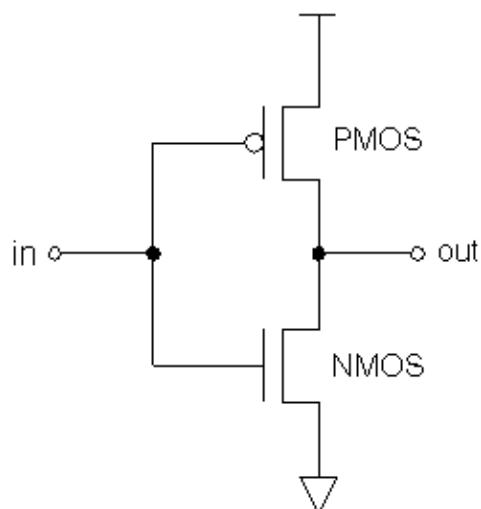
- [CMOS Circuit Characteristics](#)
 - [IDDQ Testing Methodology](#)
 - [Types of Defects Detected](#)
 - [Number of IDDQ Strobes](#)
-

CMOS Circuit Characteristics

An important characteristic of CMOS circuits is that they draw almost no current in the quiescent state. “Quiescent” means that the inputs are stable and the circuit is inactive. System designers sometimes take advantage of this characteristic by having a power down or sleep mode in which the device stops operating, but retains its internal state and memory contents, thus conserving battery charge while the device is idle.

[Figure 1](#) shows a schematic diagram of a typical CMOS inverter. The inverter has two MOS transistors, one NMOS and the other PMOS. The two transistor gates are tied together to make the inverter input, and the two drains are tied together to make the inverter output.

Figure 1 CMOS Inverter Schematic Diagram



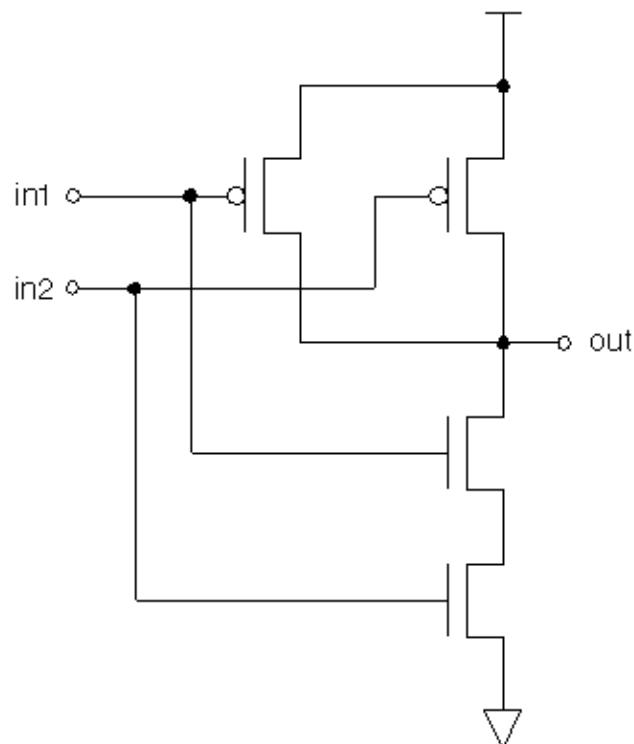
When the input is low, the upper transistor is on and the lower transistor is off, which pulls the output up to the supply voltage (VDD). When the input is high, the upper transistor is off and the lower transistor is on, which pulls the output to ground.

During a logic transition, a significant amount of current can flow while the capacitive load on the output node is charged up to VDD or discharged to ground. However, in the quiescent state, the only current that flows is the very small leakage current through the transistor that is off.

To ensure that no current flows in the quiescent state, every node must be pulled either low or high, and not allowed to float. For example, if the input of the inverter is allowed to float, the voltage could drift to an intermediate value, putting both transistors into a partially on state. This would allow a steady-state current to flow from VDD through the two transistors to ground.

A logical NAND gate uses multiple PMOS transistors in parallel at the top and multiple NMOS transistors in series at the bottom, as shown in [Figure 2](#). For each combination of input values, the power supply current is extremely small in the quiescent state because the path from VDD to ground is blocked by at least one off transistor.

Figure 2 CMOS NAND Gate Schematic Diagram

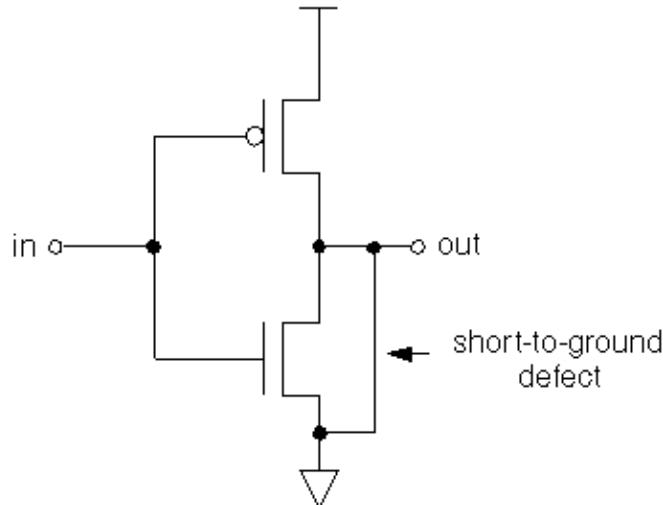


IDDQ Testing Methodology

IDDQ testing is different from traditional circuit testing methods such as functional or stuck-at testing. Instead of looking at the logical behavior of the device, IDDQ testing checks the integrity of the nodes in the design. It does this by measuring the current drain of the whole chip at times when the circuit is quiescent. Even a single defective node can easily cause a measurable amount of excessive current drain. In order to place the circuit into a known state, the IDDQ test sequence uses ATPG techniques to scan in data, but it does not scan out any data.

For example, consider the short-to-ground defect shown in [Figure 3](#). Depending on the controllability and observability characteristics of the defective node, this defect might be detectable as a stuck-at-0 fault using functional or scan testing.

Figure 3 Short-to-Ground Defect



With IDDQ testing, this defect can be detected even if the node is not observable. You only need to maintain the input of the inverter at logic 0, which turns on the upper transistor and places the output of the inverter at logic 1.

It is normal for current to flow during switching, but after the device has settled for a period of time, no more current should flow. At this point, an IDDQ strobe detects the excessive current drain through the upper transistor and the short to ground. The current drain of a single defect such as this can be orders of magnitude larger than the normal current drain of the entire device in the quiescent state.

Similarly, an IDDQ strobe can detect a short to VDD. For example, in the inverter circuit shown in [Figure 3](#), you only need to maintain the input of the inverter at logic 1, which turns on the lower transistor and places the output of the inverter at logic 0. After the device has settled, an IDDQ strobe detects the current drain through the short from VDD to the node and the lower transistor.

Types of Defects Detected

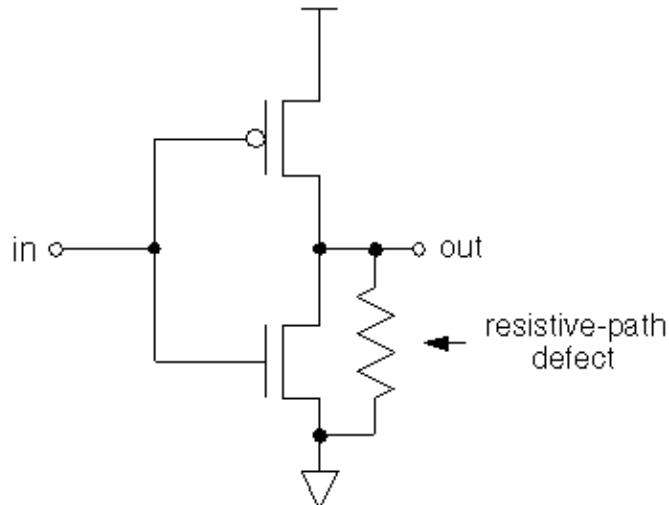
IDDQ testing can detect many kinds of circuit defects that are difficult or impossible to detect by functional or stuck-at testing, such as three-state enable nodes, redundant logic, high-resistance faults, scan chain control/data paths, undetectable faults, possibly detected faults, ATPG untestable faults, and bridging faults.

For example, consider the defect shown in [Figure 4](#), a resistive path to ground. This node might pass initial stuck-at testing, but fail after burn-in or during actual use by the customer. IDDQ testing can immediately detect this type of fault due to the excessive current drain when the node is at logic 1, even if the node is not observable by stuck-at testing.

IDDQ testing can partially or completely replace costly burn-in testing. Burn-in means testing the device using functional or scan testing, operating the device for a period of time under normal conditions, and then running the same tests to find any early failures in the lifetime of the device. IDDQ testing can detect many burn-in type defects.

IDDQ testing can also detect bridging faults. A bridging fault is a short between two different functional nodes in the design. An IDDQ strobe detects a fault of this type if one node is at logic 0 while the other is at logic 1.

Figure 4 Resistive Path to Ground



Number of IDDQ Strobes

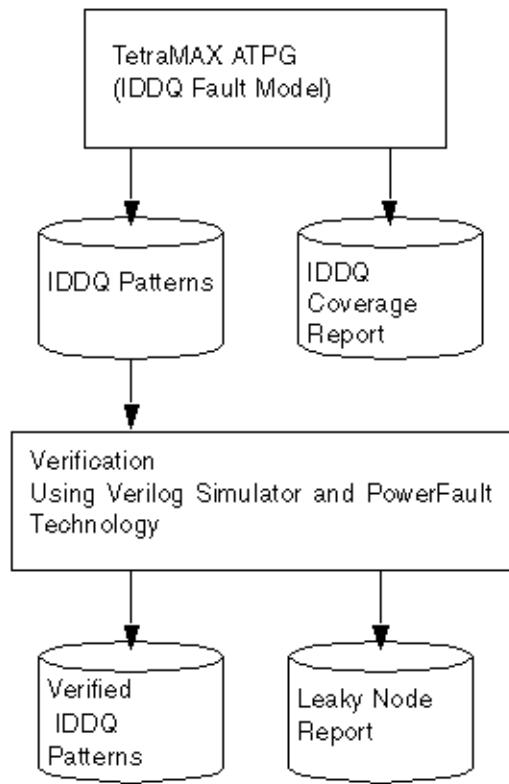
IDDQ testing can provide very high fault coverage with just a few strobes. The first IDDQ strobe typically detects half of all short-to-ground and short-to-VDD faults. IDDQ test patterns attempt to change or toggle as many nodes as possible in subsequent patterns to quickly increase fault coverage.

After the circuit nodes are forced to a known state, a certain amount of inactive time is required to allow the nodes to settle before the IDDQ measurement. The required settling time depends on the CMOS technology used and the required testing threshold. A tester time “budget” of 10 or 20 IDDQ strobes is typically allowed for testing each device. This number of strobes is usually enough to achieve satisfactory fault coverage.

About IDDQ Pattern Generation

[Figure 1](#) shows the IDDQ testing flow using TetraMAX ATPG test-pattern generation. The ATPG algorithm attempts to sensitize all IDDQ faults and apply IDDQ strobes to test all such faults. TetraMAX ATPG compresses and merges the IDDQ test patterns, just like ordinary stuck-at patterns.

Figure 1 IDDQ Testing Flow



While generating IDDQ test patterns, by default TetraMAX ATPG avoids any condition that could cause excessive current drain, such as strong or weak bus contention or floating buses.

TetraMAX ATPG generates an IDDQ test pattern and an IDDQ fault coverage report. It generates quiescent strobes by using ATPG techniques to avoid all bus contention and float states in every pattern it generates. The resulting test pattern has an IDDQ strobe for every ATPG test cycle. In other words, the output is an IDDQ-only test pattern.

After the test pattern has been generated, you can use PowerFault simulation to verify the test pattern for quiescence at each strobe. The simulation does not need to perform strobe selection or fault coverage analysis because these tasks are handled by TetraMAX ATPG. Refer to the *Test Pattern Validation User Guide* for details about PowerFault.

TetraMAX ATPG supports IDDQ testing in the following ways:

- It lets you generate test patterns that are targeted for IDDQ testing.
- It adds IDDQ verification and analysis capabilities into your Verilog simulator.

If you use the TetraMAX stuck-at model to generate standard test patterns, you can then use PowerFault technology to select the best strobe times in the resulting test patterns.

Note: An alternative approach is to use an existing set of stuck-at ATPG patterns and have the Verilog/PowerFault simulation select appropriate IDDQ strobe times from those patterns. This is described in section “Selecting Strobes in TetraMAX Stuck-At Patterns” in the *Test Pattern Validation User Guide*

IDDQ Limitation

Note the following limitation for IDDQ support:

- A Parallel Verilog or Parallel STIL testbench for scan compression mode patterns does not contain the iddq_capture annotations. A Serial Verilog for scan compression mode contains the iddq_capture annotations.

Fault Models

TetraMAX ATPG offers a choice of fault models: stuck-at, IDDQ, transition, bridging, and path delay faults. You specify the IDDQ fault model to generate test patterns specifically for IDDQ testing.

With an IDDQ fault model, TetraMAX ATPG does not attempt to observe the logical behavior of the device at the outputs. Instead, it tries to toggle as many nodes as possible into both states while avoiding conditions that violate quiescence. Any node defects can be detected by the excessive current drain that they cause. In this case, TetraMAX ATPG attempts to sensitize each node in the design, but does not try to propagate faults to the device outputs.

TetraMAX ATPG supports two IDDQ fault models:

- **Pseudo-Stuck-At Fault Model (the default)**

This fault model considers the functionality of each individual cell. It is similar to the standard stuck-at ATPG model, except that every cell output is considered observable by IDDQ testing. The fault site at a gate input requires sensitization and propagation to an output of the same gate (but not to an output of the device) to be given credit for IDDQ fault detection. In other words, to be considered detected, a fault must cause an incorrect value at the output of the cell.

- **Toggle Fault Model**

This fault model is a simple, net-only model that does not consider gate functionality. Each fault site only needs to have its state controlled to be given credit for IDDQ fault detection. The toggle model is less computationally intensive than the pseudo-stuck-at model, but it is not guaranteed to detect as wide a range of faults inside cells.

DRC Rule Violations

TetraMAX ATPG performs a wide range of test design rule checking (DRC) when you use the `run_drc` command. Some DRC rule violations indicate that your design might not be IDDQ testable or not fully modeled for IDDQ quiescence checking, or might require additional ATPG effort to achieve circuit quiescence. To help avoid DRC violations, follow the design guidelines in [Design Principles for IDDQ Testability](#).

To view a list of rule violations after you perform design rule checking, use the `report_rules` command. [Example 1](#) shows a typical DRC violation report.

Example 1 DRC Violation Report

```
TEST-T> report_rules -fail
rule  severity #fails description
----- -----
B6    warning     2 undriven module inout pin
B7    warning   178 undriven module output pin
B10   warning    32 unconnected module internal net
B13   warning     2 undriven instance input pin
S23   warning    64 unobservable potential TLA
S29   warning     1 invalid dependent slave operation
C3    warning    32 no latch transparency when clocks off
C6    warning     1 TE port captured data affected by new
capture
Z1    warning   289 bus contention ability check
Z2    warning   289 Z-state ability check
Z4    warning   360 bus contention in test procedure
```

[Table 1](#) lists TetraMAX ATPG design rule violations that warrant investigation if you plan to generate IDDQ test patterns.

Table 1 DRC Rule Violations and IDDQ Significance

Rule	Description, severity	Significance for IDDQ testing
B5	Undefined module referenced, error	Incomplete model; nonquiescent circuitry could be missing
B7	Undriven module output pin, warning	Possible floating net; could be just an unused net
B9	Undriven module internal net, warning	Possible floating net; could be just an unused net
B12	Undriven instance input pin, error	Likely to be a floating net
B18	Three-state and non-three-state drivers combined, warning	Might require more ATPG effort to avoid bus contention
N2	Unsupported construct, warning	Incomplete model; nonquiescent circuitry could be missing
Z1	Bus capable of contention, warning	Might require more ATPG effort to avoid bus contention

Table 1 DRC Rule Violations and IDDQ Significance (Continued)

Rule	Description, severity	Significance for IDDQ testing
Z2	Bus capable of holding Z state, warning	Might require more ATPG effort to avoid floating buses
Z3	Wire capable of contention, error	Likely to be a wired-net contention
Z7	Unable to prevent contention for circuit, error	ATPG cannot find nonquiescent circuit state
Z8	Unable to prevent contention for bus, warning	ATPG cannot avoid bus contention
X1	Sensitizable feedback path, warning	Possible circuit oscillation

For more information about TetraMAX ATPG design rule checking, see [Performing Test Design Rule Checking](#).

Generating IDDQ Test Patterns

The following sections describe how to generate IDDQ test patterns:

- [IDDQ Test Pattern Generation Flow](#)
- [Using the iddq_capture Procedure](#)
- [Off-Chip IDDQ Monitor Support](#)

IDDQ Test Pattern Generation Flow

The following steps show you how to generate IDDQ test patterns:

1. Set the fault type to IDDQ with the `set_faults` command.
2. Select the appropriate IDDQ fault model, either pseudo-stuck-at or toggle model, with the `set_iddq` command.
3. Create the fault list with the `add_faults` or `read_faults` command.
4. Set the maximum number of IDDQ strobes with the `set_atpg -patterns` command.
5. Run pattern generation with the `run_atpg` command.

For example, here is a typical IDDQ ATPG session:

```
TEST-T> set_faults -model iddq
TEST-T> set_iddq -toggle      # pseudo-stuck-at is the default
TEST-T> add_faults -all
```

```
TEST-T> set_atpg -patterns 20    # budget of 20 IDDQ strobes
TEST-T> run_atpg -auto_compression
```

The order of the steps is important. You cannot create the fault list until you have selected the IDDQ fault model.

After you generate the IDDQ test patterns, you can use PowerFault simulation technology to verify the patterns for quiescence. For more information, refer to the *Test Pattern Validation User Guide*.

If you generate stuck-at patterns and you want to use PowerFault to select IDDQ strobes from the pattern set, see “Selecting Strobes in TetraMAX Stuck-At Patterns” in the *Test Pattern Validation User Guide*.

Using the `iddq_capture` Procedure

When you create IDDQ patterns, TetraMAX ATPG defines a procedure, called `iddq_capture`, in the pattern output file. This procedure (shown in the following example) is used when an IDDQ measure is performed:

```
"iddq_capture" {
    W "_default_WFT_";
    F { "testmode"= 1; }
    V { "_pi"=\r379 # ; "_po"=\j \r276 X ; }
    IddqTestPoint;
    V { "_po"=\r276 # ; }
}
```

TetraMAX ATPG generates the default `iddq_capture` procedure when IDDQ test patterns are written and the input STL procedure file does not define an `iddq_capture` procedure. If you not define the `iddq_capture` procedure in the input STL procedure file, make sure you specify the `write_drc_file` command after the `write_patterns` command so the `iddq_capture` procedure is preserved in the output STL procedure file.

You should use the new STL procedure file in subsequent runs to provide the same `iddq_capture` procedure. Since default flows change in various releases, it is important to preserve the default behavior for these patterns. When WGL IDDQ patterns are written, V4 errors will occur if these patterns are read in a context that does not define the `iddq_capture` procedure. You can eliminate these problems in subsequent flows by saving the new STL procedure file with the complete set of procedures.

You can also define customized `iddq_capture` procedures in the STIL procedure file and pass them into the flow.

Off-Chip IDDQ Monitor Support

You can transfer information into off-chip IDDQ monitors as part of your IDDQ test data. Typically, an off-chip IDDQ monitor is an additional hardware unit placed physically adjacent to the device under test (DUT). The monitor is used to perform current measurements and typically has extra signals that you use to control when and how IDDQ measurements are performed.

Off-chip IDDQ monitors require two fundamental constructs to be supported at test. One construct is to support the definition of additional signals present on the monitors as part of the test flow. The second construct is the application of specific procedure calls at the IDDQ measurement points.

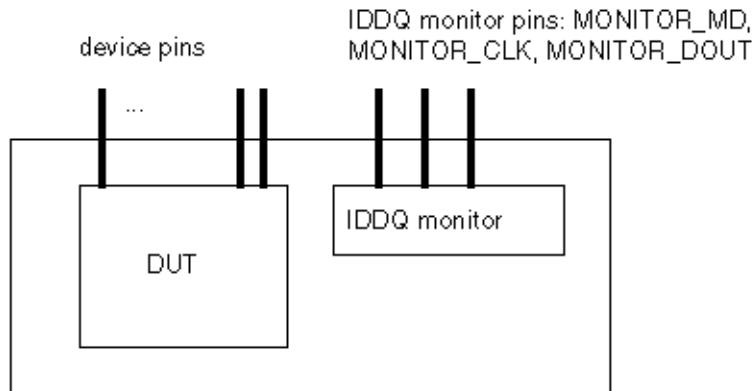
The following sections describe how to include off-chip IDDQ monitor signals in your testing:

- [Specifying Additional Signals in the Netlist](#)
- [Defining the iddq_capture Procedure to Support Additional Signals](#)

Specifying Additional Signals in the Netlist

Monitor signals are not part of the DUT nor are they part of the tester. They exist adjacent to the DUT on the loadboard or some location near the DUT for signal measurement integrity. While not part of the DUT, these signals are required because they must toggle during IDDQ testing. Define these signals in an additional hierarchical level that conceptually represents the DUT and off-chip IDDQ monitor as a single unit, as shown in [Figure 1](#). The only requirement in the flow is the presence of the additional monitor signals; a representation of the monitor itself is not required or expected.

Figure 1 Hierarchical Design With DUT and IDDQ Monitor



The following Verilog netlist shows how references to these signals could look, where the prefix `MONITOR_` is used on all the off-chip IDDQ monitor signals for easy identification:

```
// new top_module of design and MONITOR signals
module AAA_W_QSTAR ( MONITOR_MD, MONITOR_CLK, MONITOR_DOUT,
    ... other design signals ... );
    input MONITOR_MD, MONITOR_CLK ;
    output MONITOR_DOUT ;
    ...
    AAA_DUT ( ... other design signals ... );
endmodule; // AAA_W_MONITOR

// top design module
module AAA ( ... other design signals ... );
    ...

```

Defining the `iddq_capture` Procedure to Support Additional Signals

Using off-chip IDDQ monitors affects how you define the `iddq_capture` procedure because the DUT needs to be controlled in particular during the capture operation. You can expand the `iddq_capture` template to support operation of the additional IDDQ monitor pins. The `iddq_capture` procedure will vary depending on operations present to manipulate the IDDQ monitor or to return measurement data.

Because the monitor control signals are part of the netlist sent to TetraMAX ATPG, these signals need to be specified as “Fixed” signals in the flow for all other applications; that is, held at their inactive states except during IDDQ testing.

The following example represents an application where the IDDQ measurement/settling time is defined in a WaveformTable with minimal functionality, supporting only the maintenance of the input states during this period. There are no requirements on the name of this WaveformTable. The prefix `MONITOR_` is used on all the off-chip IDDQ monitor signals for easy identification.

```

Timing {
    WaveformTable "_default_WFT_" {
        Period '100ns';
        Waveforms {
            "_default_In_Timing_" { 0 { '0ns' D; } }
            "_default_In_Timing_" { 1 { '0ns' U; } }
            "_default_In_Timing_" { Z { '0ns' Z; } }
            "_default_In_Timing_" { N { '0ns' N; } }
            "_default_Clk0_Timing_" { P { '0ns' D; '50ns' U;
                '80ns' D; } }
            "_default_Out_Timing_" { X { '0ns' X; } }
            "_default_Out_Timing_" { H { '0ns' X; '40ns' H; } }
            "_default_Out_Timing_" { T { '0ns' X; '40ns' T; } }
            "_default_Out_Timing_" { L { '0ns' X; '40ns' L; } }
        }
    }
    WaveformTable "_IDDQ_MEASUREMENT_WFT_" {
        Period '100us';
        Waveforms {
            "_default_In_Timing_" { 0 { '0us' D; } }
            "_default_In_Timing_" { 1 { '0us' U; } }
            "_default_In_Timing_" { Z { '0us' Z; } }
            "_default_In_Timing_" { N { '0us' N; } }
            "_default_Out_Timing_" { X { '0us' X; } }
        }
    }
}
Procedures {
    "load_unload" {
        W "_default_WFT_";
        // establish inactive states on the monitor during
Shift
        V {"sdo"=X; "CLK"=0; "MONITOR_MD"=1; "MONITOR_CLK"=0; \

```

```

        "MONITOR_DOUT"=X;  }
Shift { V { "__si"=#; "__so"=#; "CLK"=P; } }
}
"capture*" {
// All Capture
Routines
Except iddq_capture
    // Hold monitor inactive
    F { "MONITOR_MD"=1; "MONITOR_CLK"=0; "MONITOR_DOUT"=X;
}
    W "_default_WFT_";
    V { ... }
}
"iddq_capture" {
    W "_default_WFT_";
    V { "_pi"=\r15 # ; "_po"=\j \r7 # ; }
    IddqTestPoint;
    W "_IDDQ_MEASUREMENT_WFT_";

    V { "MONITOR_MD"=0; "_out"=XXX ; } // Activate monitor
measurement
    W "_default_WFT_";
    V { "MONITOR_DOUT"=H; } // Detect successful
measurement (pass)
}
} // end Procedures

```

The following example merges the `_po` measure operation into the IDDQ measurement vector. It requires a more complete WaveformTable to support the measure operation on the outputs but reduces vector count in the `iddq_capture` procedure by one. Only the WaveformTable and `iddq_capture` changes are shown here. The prefix `MONITOR_>` is used on all the off-chip IDDQ monitor signals for easy identification.

```

Timing {
    WaveformTable "_IDDQ_MEASUREMENT_WFT_" {
        Period '100us';
        Waveforms {
            "_default_In_Timing_" { 0 { '0us' D; } }
            "_default_In_Timing_" { 1 { '0us' U; } }
            "_default_In_Timing_" { Z { '0us' Z; } }
            "_default_In_Timing_" { N { '0us' N; } }
            "_default_Out_Timing_" { X { '0us' X; } }
            "_default_Out_Timing_" { H { '0us' X; '98us' H; } }
            "_default_Out_Timing_" { T { '0us' X; '98us' T; } }
            "_default_Out_Timing_" { L { '0us' X; '98us' L; } }
        }
    }
}

Procedures {

```

```

"iddq_capture" {
    W "_default_WFT_";
    V { "_pi"=\r15 # ; "_out"= XXX ; }
    IddqTestPoint;
    W "_IDDQ_MEASUREMENT_WFT_";
    V { "MONITOR_MD"=0; "_po"=\r7 # ; } // Activate monitor
measurement
    W "_default_WFT_";
    V { "MONITOR_DOUT"=H; } // Detect successful measurement
(pass)
}
}

```

The following example maintains the current IDDQ test sequence with the addition of the extra cycles for the monitor's operation at the end. While consistent with current IDDQ constructs, this operation requires the most total cycles per IDDQ test. This construct can operate with a minimal measure WaveformTable, or a larger WaveformTable, depending on whether the outputs are masked in the monitor's measure cycle. Because no state changes are occurring, these outputs can remain in their previous measured state in the next two vectors (requiring a more complete WaveformTable), or can be masked (requiring less definitions in the monitor's measure WaveformTable). The prefix MONITOR_ is used on all the off-chip IDDQ monitor signals for easy identification.

```

Procedures {
    "iddq_capture" {
        W "_default_WFT_";
        V { "_pi"=\r15 # ; "_out"= XXX ; }
        IddqTestPoint;
        V { "_po"=\r7 # ; }
        W "_IDDQ_MEASUREMENT_WFT_";
        V { "MONITOR_MD"=0; } // Activate monitor measurement.
            // Note outputs still tested
        W "_default_WFT_";
        V { "MONITOR_DOUT"=H; } // Detect successful measurement
(pass)
}
}

```

Using IDDQ Commands

You can use the `set_faults` command to set the fault model. If you select the IDDQ fault model, you can use the `set_iddq` command to specify the quiescence constraints and toggle/no-toggle model type. The `add_atpg_constraints` command lets you set IDDQ-specific ATPG constraints on nodes in the design. These commands are described in the following sections:

- [Using the set_faults Command](#)
 - [Using the set_iddq Command](#)
 - [Using the add_atpg_constraints Command](#)
-

Using the `set_faults` Command

To generate IDDQ-only test patterns, use the `set_faults -model iddq` command. You can specify the quiescence constraints and toggle/no-toggle model with the `set_iddq` command.

To generate standard stuck-at test patterns, use the `set_faults -model stuck` command. This is the default model.

For the complete syntax and option descriptions, see the online help for the `set_faults` command.

Using the `set_iddq` Command

The `float`, `strong`, `weak`, and `write` options of the `set_iddq` command allow you to specify the conditions required for quiescence. TetraMAX will not generate a pattern that fails to meet an enabled restriction.

The assertive option `float`, `strong`, `weak`, or `write` means that the restriction is enforced. The restrictions minimize conditions that could cause excessive current drain, such as strong or weak bus contentions or floating buses. The negative option `nofloat`, `nostrong`, `noweak`, or `nowrite` means that the restriction is removed and the condition is allowed. By default, all the assertive options are in effect and all restrictions are enforced. To allow a condition for IDDQ test pattern generation, use the appropriate negative option.

By default, the individual restrictions operate in the following manner:

- The `float` restriction means that every BUS gate must not be at the Z state during an IDDQ measure.
- The `strong` restriction means that the IDDQ measure must be contention-free for strong drivers of BUS gates.
- The `weak` restriction means that BUS gates with weak inputs must not compete with other strong or weak BUS inputs during an IDDQ measure.
- The `write` restriction means that RAMs must not have an active write port during an IDDQ measure.

The `-atpg` or `-noatpg` option determines whether the test generator attempts to satisfy all the IDDQ constraints during pattern generation (`-atpg`), or only checks and discards patterns that fail to meet these constraints after completion of pattern generation (`noatpg`). The default setting is `-noatpg`.

The option `toggle` or `notoggle` option selects the type of IDDQ fault model. This selection is valid only if you have selected the IDDQ fault model with the `set_faults -model iddq` command. The default selection is `notoggle`, which selects the pseudo-stuck-at fault model.

To select the toggle model instead, use the `toggle` option. These two models are described in Pseudo-Stuck-At Fault Model.

Using the add_atpg_constraints Command

The `add_atpg_constraints` command lets you define constraints that apply during the generation of test patterns. For example, you can use this command to force a particular internal node to the value 1 at the clock-on time for all test patterns.

In this command, you specify an arbitrary name to identify the constraint, the value of the constraint (0, 1, or Z), and the place in the design where the constraint is to be applied. You can optionally specify when the constraint must be satisfied by using the `drc` or `iddq` option.

By default, the constraint must be satisfied only at clock-on time for test pattern generation.

Using the `drc` option means that the constraint must also be satisfied during DRC procedures and ATPG analyses.

Using the `iddq` option means that the constraint only has to be satisfied during IDDQ measure strobes, and only if the IDDQ fault model has been selected with the `set_faults -model iddq` command. An IDDQ measure strobe corresponds to the time in the tester cycle when outputs are measured, as specified by the `WaveformTable` block in the `run_drc` test protocol file.

IDDQ Bridging

You can use the IDDQ bridging fault model to generate additional patterns and increase the IDDQ coverage. This fault model, which is specified using the `set_faults -model iddq_bridging` command, behaves differently than the regular IDDQ fault model. Regular IDDQ fault model has two versions of the same model (Toggle or Pseudo-Stuck-At). The fault model for IDDQ bridging is only of type Toggle. This means that the fault site at a gate input does not require propagation to an output of the same gate to be given credit for IDDQ bridging fault detection.

In regular scan mode, unload values are written in the patterns to facilitate load/unload overlapping by the tester. However, capture is not in effect when using the IDDQ bridge fault model, and the unload values are exactly the same as the load values.

When the IDDQ bridging fault model is specified, the `set_iddq -toggle` command is invalid because only one version of the model is available.

The IDDQ bridging fault model is similar to the bridging fault model except that bridging faults are directly observed by an IDDQ strobe rather than by propagating the fault effect to a scan cell. The primary purpose of performing IDDQ bridging fault ATPG is to detect faults by inserting correct values into the fault nodes. As a result, there are no observation requirements. For example, to detect the IDDQ bridging fault named `ba0 node1 node2`, where the aggressor node is `node1` and victim node is `node2`, APTG sets the `node1` logic value to 0 and the `node2` logic value to 1.

The IDDQ bridging fault model uses the same fault codes as the bridging fault model. The existing `add_faults -node_file` and `-bridge_location` options are used to read net pairs and add the IDDQ bridging faults. The IDDQ bridging measurement criteria is adjusted by a separate `set_iddq` command, as is the case with the regular IDDQ fault model. During ATPG, the detection of one pair of bridges implies the detection of another pair. This behavior can be controlled using the `set_iddq -bridge_equivalence` command.

The `run_atpg` and `run_fault_sim` commands check out the Test-Fault-Max license.

The flow to generate IDDQ bridging patterns is as follows:

```
set_fault -model iddq_bridging  
# optional setting of measurement criteria  
set_iddq nofloat  
add_faults -node pair.txt  
run_atpg -auto  
write_patterns iddq_bridging.stil -format stil
```

Note the following limitations related to IDDQ bridging ATPG:

- The `analyze_faults` command is not supported when using the IDDQ bridging fault model.
- The strength-based optimizations used for the regular bridging fault model are not supported for the IDDQ bridging fault model.
- Full-sequential ATPG does not support the IDDQ bridging fault model.

Design Principles for IDDQ Testability

The following design principles apply to designing your circuits for IDDQ testability:

- [I/O Pads](#)
- [Buses](#)
- [RAMs and Analog Blocks](#)
- [Free-Running Oscillators](#)
- [Circuit Design](#)
- [Power and Ground](#)
- [Models With Switch/FET Primitives](#)
- [Connections](#)
- [IDDQ Design-for-Test Rule Summary](#)

Note: IDDQ testing and PowerFault simulation is more efficient and reliable if you follow these requirements. For details about PowerFault, see the *Test Pattern Validation User Guide*.

I/O Pads

Put I/O pads on a separate power rail, if possible. Then you can test the I/O and core logic separately as described in “Using PowerFault Technology” in the *Test Pattern Validation User Guide*.

If I/O pads and core logic share the same power rail, use I/O pads that have controllable pullups rather than passive pullups. This will allow the pullups to be gated out during IDDQ testing.

Slew control for I/O pins must be disabled or I/O pins must be put on a separate rail. If I/O pads and core logic share the same power rail, all DC paths from power to ground (such as slew control) must be disabled during IDDQ testing. There are two strategies to achieve this:

- Use controllable pullups/pulldowns so that they can be gated out during IDDQ testing. This is the preferred method.
 - Drive pads so that pullups/pulldowns not active (for example, drive a pad with a pullup to VDD). Have the testbench drive pads that have both pullups and pulldowns to VDD (or to VSS if you are measuring ISSQ).
-

Buses

Use fully multiplexed bus drivers so that only one driver can be active at a time. Furthermore, always drive a bus if possible, as described in the “Using PowerFault Technology” in the *Test Pattern Validation User Guide*.

If buses cannot always be driven, gate buses at the receivers as described in “Using PowerFault Technology” in the *Test Pattern Validation User Guide*.

If buses can't be driven or gated, use keeper latches. Model keeper latches structurally as described in “Using PowerFault Technology” in the *Test Pattern Validation User Guide*.

Avoid internal pullups and pulldowns. If possible, either drive or gate a bus to prevent it from floating. If pullups and pulldowns must be used, model them structurally as described in “Using PowerFault Technology” in the *Test Pattern Validation User Guide*.

Avoid tri1, tri0, wor, and wand wire types. Use pullup/pulldown primitives instead.

RAMs and Analog Blocks

Check your databook to make sure you do not hardwire your RAM into a high-current state. RAMs and analog blocks that have high-current states require either a sleep mode or a separate power supply.

If your chip uses a sleep mode for RAM or analog blocks, prevent IDDQ strobing when the blocks are not in sleep mode by doing one of the following,

- Avoid invoking the `strobe_try` command when the chip is in a high-current state.
- Use the `disallow` command to tell PowerFault when RAM or analog blocks are in high-current states.

If RAM or analog blocks are on a separate power rail and PowerFault reports them as leaky, use the `allow` command to have PowerFault ignore them.

Free-Running Oscillators

Avoid free-running oscillators, if possible, because they draw current. If you must use a free-running oscillator, disable it during IDDQ testing, or put the affected circuitry on a separate power rail. Use the `disallow` command to tell PowerFault when the oscillator is running.

Circuit Design

To prevent current drain through the substrate, connect the bulk node for n-type transistors to VSS and the bulk node for p-type transistors to VDD.

Avoid degraded voltages. For example, avoid using an NMOS transistor to serve as a pass gate.

Avoid circuits that put the gate and drain or source nodes of a transistor in the same transistor group.

Avoid circuits that create control loops among transistor groups. Obviously, control loops must exist to implement flip-flops and latches, but using certain flip-flop and scan chain design rules can make bridging faults more testable.

Avoid circuits that use charge sharing or charge retention. Bridging faults within dynamic (domino) logic cells are difficult to detect with IDDQ testing. Furthermore, the output voltage of dynamic logic cells might degrade during an IDDQ measurement, causing the inputs to the following static logic block to float.

Power and Ground

Declare supply0, supply1, tri0, and tri1 nets fed in from the testbench so that they have the same type in the DUT. For example, if tbench.VDD1 is a supply1 net in the testbench and it is connected to tbench.dut.vdd1, make sure that tbench.dut.vdd1 is also declared as a supply1 net.

If you are using Verilog-XL, do not use cell ports for VSS/VDD. Use a local supply0 or supply1 net, or a 'b0 or 'b1 constant to connect terminals and ports inside cells to VSS/VDD.

If you are using VCS, connect terminals and ports to supply0 or supply1 nets instead of using 'b0 or 'b1 constants.

Models With Switch/FET Primitives

Try to limit switch modeling to three-state cells.

Use user-defined primitives (UDPs) or standard logic gates to build models for multiplexers, flip-flops, and latches.

Avoid `tran`, `tranif0`, and `tranif1` primitives. Instead, use `cmos`, `nmos`, and `pmos` primitives.

Avoid having channels of switch primitives in series extend between module scopes.

Do not pass three-state values (Zs) through switches or field effect transistors (FETs). If a net can take on a three-state value, make the receivers (loads) strength-restoring gates, not switch primitives.

Connections

Maintain cell-level hierarchy and avoid creating a very large cell containing many Verilog primitives at the same level. Limit each bottom-level cell to a few hundred primitives at most. (Most ASIC libraries have only a few primitives per bottom-level cell.)

Do not use continuous assignments to connect nets to nets.

Do not use continuous assignments to implement three-state drivers.

Do not use mismatched drivers to model latches and flip-flops. If possible, use UDPs.

Do not connect registers directly to gate terminals. Connect registers to wires (via continuous assignments or module ports), and then connect the wires to gate terminals.

All internal buses should have gate loads. Each internal bus should fan out to at least one gate input, instead of fanning out to only behavioral statements (such as continuous assignments and event control for `always` blocks).

PowerFault is most accurate at identifying leaky states when used with gate-level models and libraries. Avoid using RTL models because they might not contain enough structural information to allow identification of floating nodes and drive contention.

IDDQ Design-for-Test Rule Summary

The following design-for-test (DFT) rules summarize the design principles for IDDQ testing:

1. Define an IDDQ test mode signal that does not contend with the scan test mode signal.
2. Use separate power rails for the I/O and core modules.
3. Use fully complementary, fully static CMOS.
4. Use separate power rails for analog and nonstatic CMOS modules.
5. Use separate power rails for unknown or otherwise IDDQ-untestable cores.
6. For RTL modules, specify any known input conditions and sequences that cause internal contention. Use ATPG constraints or IDDQ `allow` or `disallow` statements (or both).
7. For RTL modules, specify any known input conditions and sequences that cause internal floating.
8. Use transistors to enable and disable pullups and pulldowns. Disable them with the IDDQ test mode signal.
9. Using the IDDQ test mode signal, disable three-state and bidirectional outputs that require pullups or pulldowns.
10. Each internal three-state nets requires one of the following: a bus holder, one-hot enable logic, or logic to gate off all bits of the bus except for the least significant using the IDDQ test mode signal.
11. Each compiled SRAM or ROM requires one of the following: 100 percent CMOS circuitry, a separate power rail, or a defined condition controlled by the IDDQ test mode signal that guarantees quiescence.
12. Do not allow SRAM and DRAM outputs to go to the Z state unless a bus holder is present on the output.
13. Each compiled data path cell must either be 100 percent static CMOS or allow quiescence control with the IDDQ Test Mode signal.
14. Do not allow any unconnected module or cell inputs.

Additional System-on-a-Chip Rules

The following rules apply to system-on-a-chip (SOC) applications:

1. Each core must have a test isolation mode. Each core must not be affected by other cores or user-defined logic, and must not affect other cores or user-defined logic. Each core must not be allowed or required to propagate contention or float conditions.
2. All cores and user-defined logic sharing a power rail must be quiescent during the time each core is being IDDQ-tested.
3. It must be possible to stop the clock. The core must have a bypass clock signal from the tester (a primary I/O).

23

Transition-Delay Fault ATPG

The transition-delay fault model is used to generate test patterns to detect single-node slow-to-rise and slow-to-fall faults. For this model, TetraMAX ATPG launches a logical transition upon completion of a scan load operation, and a pulse on capture clock procedure is used to observe the transition results.

The following topics describe how to use the transition-delay fault model:

- [Using the Transition-Delay Fault Model](#)
- [Specifying Transition-Delay Faults](#)
- [Pattern Generation for Transition-Delay Faults](#)
- [Pattern Formatting for Transition-Delay Faults](#)
- [Specifying Timing Exceptions From an SDC File](#)
- [Slack-Based Transition Fault Testing](#)

Note: You need a Test-Fault-Max license to use the transition-delay fault ATPG feature. This license is also checked out if you read an image that was saved with the fault model set to transition.

Using the Transition-Delay Fault Model

The transition-delay fault model is similar to the stuck-at fault model, except that it attempts to detect slow-to-rise and slow-to-fall nodes, rather than stuck-at-0 and stuck at-1 nodes. A slow-to-rise fault at a defect means that a transition from 0 to 1 on the defect does not produce the correct results at the maximum operating speed of the device. Similarly, a slow-to-fall fault means that a transition from 1 to 0 on a node does not produce the correct results at the maximum operating speed of the device.

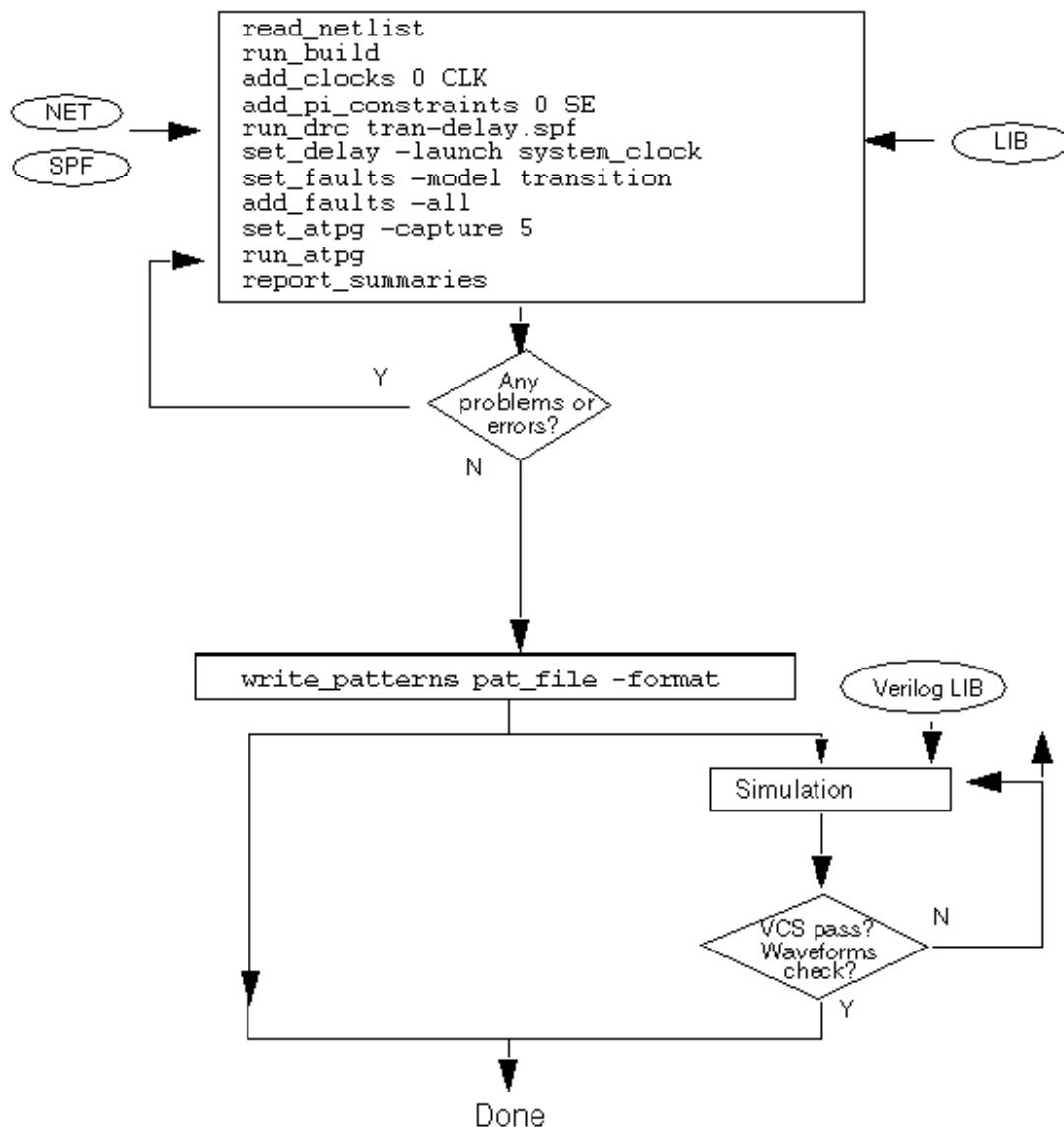
To detect a slow-to-rise or slow-to-fall fault, the ATPG process launches a transition with one clock edge and then captures the effect of that transition with another clock edge. The amount of time between the launch and capture edges should test the device for correct behavior at the maximum operating speed.

The following sections describe how to use the transition-delay fault model:

- [Transition-Delay Fault ATPG Flow](#)
 - [Transition-Delay Fault ATPG Timing Modes](#)
 - [STIL Protocol for Transition Faults](#)
 - [Creating Transition Fault Waveform Tables](#)
 - [DRC for Transition Faults](#)
 - [Limitations of Transition-Delay Fault ATPG](#)
-

Transition-Delay Fault ATPG Flow

The ATPG process for transition-delay faults is similar to the process for stuck-at faults. Figure 1 shows the typical steps for performing transition-delay fault ATPG.

Figure 1 Transition-Delay Fault Test Flow

Transition-Delay Fault ATPG Timing Modes

TetraMAX ATPG transition-delay fault ATPG supports several ATPG modes for applying transition-delay tests. You select the required mode with the `set_delay -launch_cycle`

command. The following modes are supported:

- **Launch-On Shift (LOS)** — Specified by the `last_shift` option, TetraMAX ATPG launches a logic value in the last scan load cycle when the scan enable is active, that is, in scan-shift mode. It exercises target transition faults and then captures new logic values in a system clock cycle when the scan enable is inactive, that is, in capture mode. Figure 2 shows the clock and scan enable timing for this mode.
- **System Clock** — Specified by the `system_clock` option (the default ATPG mode for transition-delay faults), TetraMAX ATPG launches a logic value using a normal system clock. It exercises target transition faults and then captures the new logic values with a subsequent system clock. Figure 3 shows the clock and scan enable timing for this mode.
- **Launch-On Extra Shift (LOES)** — Specified by the `extra_shift` option, TetraMAX ATPG launches a logic value based on one more shift than launch on shift mode. This ensures that all clock domains receive their last scan shift before the internally-controlled capture clock pulse. Unlike launch-on shift mode, launch-on extra shift mode does not place additional timing requirements on an on-chip clocking controller.
- **Any** — Specified by the `any` option, TetraMAX ATPG attempts launch-on shift mode first, and then goes to launch-on capture or launch-on extra shift, depending on the pipelined SE constraint, or goes to both modes if it's unconstrained. .

The following sections explain some of the key characteristics of the timing modes:

- [Launch-On Shift Mode Versus System Clock Launch Mode](#)
- [Using Launch-On Extra Shift Timing](#)

Launch-On Shift Mode Versus System Clock Launch Mode

One of the major differences between launch-on shift mode and system clock mode is that for the launch-on shift mode, the scan enable signal must switch between a launch and capture cycle, which might not be possible depending on the design and cycle time. For details, see “DRC for Transition Faults”.

Figure 2 and Figure 3 show the clock waveform pertaining to launch on shift mode and system clock mode for a typical target transition fault that is between registers. If the target fault is between primary inputs and registers, or if the target fault is between registers and primary outputs, then you can expect just one clock pulse, either launch or capture, or no clock pulse.

Figure 2 Last Shift Launch Timing

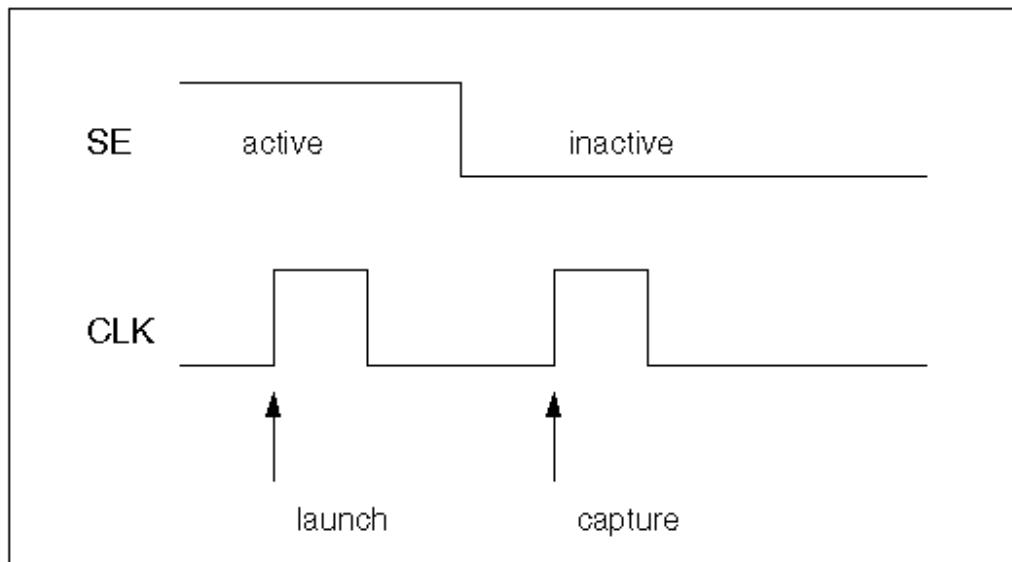
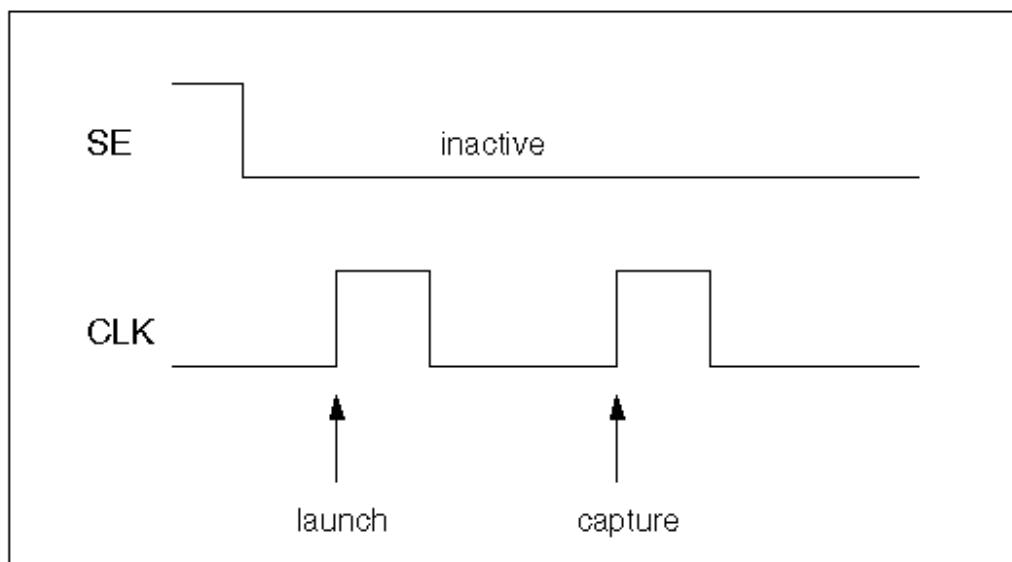


Figure 3 System Clock Launch Timing



Launch-on shift mode generates only basic-scan patterns, using a single capture procedure between scan load and scan unload.

By default, when using the `run_atpg -auto_compress` command for the system clock mode, TetraMAX ATPG uses a highly optimized two-clock ATPG process that has some features of both the basic-scan and fast-sequential engines. The patterns generated by this

process are only two clock cycles long and listed as Fast Sequential patterns in the TetraMAX ATPG pattern summary.

To use system clock mode, fast-sequential ATPG must be enabled before starting the ATPG process. You enable fast-sequential ATPG and specify its effort level with the `-capture_cycles` option of the `set_atpg` command. For details, see “Using the `set_atpg` Command”. The two-clock process, used with the `run_atpg -auto_compress` command by default, will automatically set the `-capture_cycles` option to 2.

If there is a need for more than two capture cycles — for example, if there are memories in the circuit — you can set the capture cycles to a number larger than 2 before issuing the `run_atpg -auto_compress` command. In this case, TetraMAX ATPG will first run the optimized two-clock process for all the faults that can be detected in two capture cycles and then run fast-sequential ATPG with the larger number of capture cycles for any remaining undetected faults.

Launch-On Extra Shift Timing

Launch-on extra shift mode provides many advantages of launch-on shift mode without requiring an exotic on-chip clocking controller. However, a small but significant percentage of transition faults (which represent valid functional paths) cannot be detected by launching on either the last shift or an extra shift. When using LOES, you also need to perform a top-off ATPG run using launch-on capture (LOC).

Because the LOES test application waveforms are similar to those used for LOC, you can create an SPF for LOES based on the SPF used for LOC. The constraint on the LOSPipelineEnable signal is the only parameter you need to change. If the pipeline scan enable logic was created by DFT Compiler, the LOSPipelineEnable signal type appears in the dc_shell script, or it defaults to a port named `global_pipe_se`. This parameter should be a PI constraint and a scan enable for LOC and LOES; each mode is set to a different value. To specify the LOSPipelineEnable signal in the SPF, set LOSPipelineEnable in an F (for Fixed) block in each capture procedure, then add the setting to the first V statement in the load_unload procedure, and set it to the same value at the end of the `test_setup` macro. The LOSPipelineEnable port should be set to its active state (1 by default) for LOES and to its inactive state for LOC.

The LOSPipelineEnable signal is unconstrained in the SPF written from DFT Compiler. Instead of editing the SPF, use the `add_pi_constraints` command to constrain it to 1 for LOES or to 0 for LOC.

When a LOS protocol is modified to make a protocol for LOES, you also must change the `load_unload` procedure. The LOS `load_unload` procedure has an extra V statement, with `"_pi" = #;` and `"_po" = #;` values, following the Shift loop; this V statement must be removed for LOES. You should also carefully check the waveform table usage and launch procedures since they can be very different for LOS compared to any other application.

The only additional command option used for LOES is the `-launch_cycle extra_shift` option of the `set_delay` command. This option should be set before the `run_drc` command. It directs the DRC process to perform different clock matrix checking for the launch cycle, since no disturbance originates from the functional data inputs of the scan registers. Instead, the disturbance comes only from the scan inputs. The `-launch_cycle extra_shift` option might result in better coverage.

LOES can be run with the `set_delay -launch_cycle system_clock` command. It is possible to generate patterns without constraining the LOSPipelineEnable signal so that the LOES and LOC patterns are intermingled. Both of these should be avoided because LOES patterns generated in this way uses pessimistic clock disturbances, which results in more patterns.

Transition delay fault ATPG should be run first with LOES, followed by a second run on the undetected faults with LOC. In comparative testing, this configuration resulted in higher coverage with fewer patterns than running ATPG with only LOC. An example of this flow is as follows:

Example 1 Typical Flow for Using Launch-On Extra Shift Mode

```
set_delay -launch_cycle extra_shift

# Other delay settings are the same for LOES and LOC
set_delay -common_launch_capture_clock -nopi_changes
add_po_masks -all
run_drc design_with_loes.spf -patternexec Internal_scan
set_faults -model transition
run_atpg -auto
write_patterns design_with_loes.stil -format stil
write_faults design_with_loes.faults -all
drc -force

# Prepare to change the LOSPipelineEnable constraint value
remove_pi_constraints -all
set_delay -launch_cycle system_clock
set_delay -common_launch_capture_clock -nopi_changes
add_po_masks -all
run_drc design_with_loc.spf -patternexec Internal_scan
set_faults -model transition
#Use -retain_code so the redundant faults do not need to be
identified again
read_faults design_with_loes.faults -retain_code

# Many faults that are AU for LOES can be detected by LOC
update_faults -reset_au
run_atpg -auto
write_patterns design_with_loc.stil -format stil
```

Note the following:

- ATPG in LOES mode uses the two-clock optimized ATPG engine. Fast-sequential patterns, and full-sequential patterns (if enabled), may also be generated. This is the same as with LOC, but different from LOS, which uses the basic-scan ATPG engine.

- The fault coverage achieved by LOES ATPG changes with scan chain reordering, even when run in uncompressed scan mode.
- For stuck-at testing, the LOSPipelineEnable port should be constrained to its inactive state (0 by default) for more efficient pattern generation.

See Also

Using Load_Unload for [Last Shift-Launch Transition](#)

STIL Protocol for Transition Faults

By default, TetraMAX ATPG generates a capture procedure consisting of the following events:

- Force PI
- Measure PO
- Pulse Clock

You can insert a force PI or measure PO event between a launch and capture cycle. However, this has a negative impact on the overall quality of a transition test because an extra time delay is added between launch and capture. Therefore, it is recommended that you use a single-event capture procedure containing only the pulse clock event.

If there are scan postamble vectors (vectors that follow the scan shift in the load_unload procedure) in the STL procedure file, the extra time delay for the postamble is inserted between the launch and capture cycle in the last_shift mode. The extra time delay for last_shift negatively impacts the overall quality of the test, but will not affect test quality for system_clock mode. If such a scan postamble exists in the last_shift or any mode during the ATPG process (when you execute the run_atpg command), a warning message is reported (M237).

There can be primary inputs initialized to known values in the scan load and unload procedure of a STL procedure file. This can cause faults between primary inputs and registers to be ATPG untestable in the last_shift mode.

See Also

[Creating Generic Capture Procedures](#)

Creating Transition Fault Waveform Tables

For transition fault delay paths, you can control the clock speed with different waveform tables: one for the load_unload procedure “_default_WFT_” and one for the capture procedure “_fast_WFT_” (as shown in the following example).

```
Timing {
  WaveformTable "_default_WFT_" {
    Period '100ns';
    Waveforms {
      "all_inputs" {01Z {'0ns' D/U/Z;}}
      "all_bidirectionals" {01XZ {'0ns' D/U/X/Z;}}
    }
  }
}
```

```

    "all_bidirectionals" {THL {'0ns' X; '40ns' T/H/L;}}
    "all_outputs" {X {'0ns' X;}}
    "all_outputs" {HLT {'0ns' X; '40ns' H/L/T;}}
    "Pixel_Clk" {P {'0ns' D; '45ns' U; '55ns' D; } }
}
}

WaveformTable "_fast_WFT_" {
    Period '20ns';
    Waveforms {
        "all_inputs" {01Z {'0ns' D/U/Z;}}
        "all_bidirectionals" {01XZ {'0ns' D/U/X/Z;}}
        "all_bidirectionals" {THL {'0ns' X; '8ns' T/H/L;}}
        "all_outputs" {X {'0ns' X;}}
        "all_outputs" {HLT {'0ns' X; '8ns' H/L/T;}}
        "Pixel_Clk" {P {'0ns' D; '9ns' U; '11ns' D; } }
    }
}
}

```

The following example shows a load_unload procedure defined in the STIL procedure file for the preceding “_default_WFT_” waveform table:

```

"load_unload" {
W "_default_WFT_";
Shift { W "_default_WFT_";
    V { "BPCICLK"=P; "Pixel_Clk"=P; "Test_mode"=1; "nReset"=1;
        "test_sei"=0; "_so"=###; "_si"=###; }
    }
}

```

The following example shows a three-event capture procedure (the default) followed by its recommended single-event capture procedure for the “_fast_WFT_” waveform table in the previous example:

```

"capture_Pixel_Clk" {
    W "_default_WFT_";
    F { "CSC_test_mode"=0; "Test_mode"=1; }
    "forcePI": V { "_pi"=\r587 # ; "_po"=\j \r101 X ; }
    "measurePO": V { "_po"=\r101 # ; }
    "pulse": V { "Pixel_Clk"=P; "_po"=\j \r101 X ; } }

"capture_Pixel_Clk" {
    W "_fast_WFT_";
    F { "CSC_test_mode"=0; "Test_mode"=1; }
    V { "_pi"=\r587 # ; "_po"=\r101 # ; "Pixel_Clk"=P; } }

```

Notice that the three pattern events ForcePI, MeasurePO, and PulseClock are in separate vectors in the first capture procedure, but have been combined into a single vector in the second capture procedure.

There is another way to do waveform timing for transition fault testing in TetraMAX ATPG. TetraMAX ATPG allows the use of special waveform tables for at-speed testing; both transition fault testing and path delay fault testing. There are separate waveform tables for the clock cycle in which a transition is launched (`_launch_WFT_`), for the clock cycle in which a transition is captured (`_capture_WFT_`), and for cycles in which a transition is both launched and captured (`_launch_capture_WFT_`).

The use in transition fault testing is different from the use in path delay testing. For path delay testing, these special waveform tables are always used. If they are not present in the STIL procedure file, they are first created and then used.

The difference for transition faults is that these special waveform tables must be present in the STIL procedure file to be used; TetraMAX ATPG will not create them for transition fault ATPG.

Several additional options for timing support are available. For information about waveform tables, see [“Defining Basic Signal Timing”](#). To get more details about specialized timing support for both transition and path delay environments, see the following sections:

- [Pattern Formatting for Transition-Delay Faults](#)
- [Creating At-Speed WaveformTables](#)
- [MUXClock Support for Transition Patterns](#)

See Also

[Generating Generic Capture Procedures](#)

DRC for Transition Faults

In the `last_shift` mode, the scan enable signal must have a transition between launch and capture. In the `system_clock` mode, the scan enable signal must be inactive between launch and capture, so the `add_pi_constraints` command (or a constraint in the STIL procedure file) must be used to set the scan enable signal to inactive. Otherwise, you might get patterns in the `system_clock` mode with the scan enable signal switching between launch and capture. This transition fault ATPG requirement does not normally apply to stuck-at ATPG.

Note: For at-speed ATPG, the ScanEnable, Set, and Reset signals should not pulse during capture because they are typically slow signals.

You can use the `-clock port_name` option of the `set_drc` command to enable a specific clock and to disable other clocks in a design. This option can be useful for transition-delay fault ATPG if you want to target only those faults that can be launched and captured from a specific clock (for example, to prevent skew between different clock domains). However, this option works only in the `last_shift` mode. In the `system_clock` mode, you can use the `add_pi_constraints` command to disable the clocks that you do not want to be used.

Limitations of Transition-Delay Fault ATPG

The following limitations apply to transition-delay fault ATPG:

- For a target fault between a register and an output, only a launch clock is needed to test. In TetraMAX ATPG, an output strobe occurs before a clock pulse (when using a single-cycle capture procedure). This adds an extra capture cycle without a clock pulse just to strobe an output, which might negatively affect the overall quality of the transition-delay fault test. For this type of fault to be tested effectively, an output strobe after a clock pulse (end of cycle measure) should be used, which is not supported in the current release.
- For pattern formatting, the FAST_MUXCLOCK (also called MUXClock) technique is not supported unless you set the options:

- `set_faults -model_transition`
- `set_delay -launch_type system_clock`
- `set_delay -nopi_changes`
- `add_po_masks -all`
- `set_atpg -capture_cycles > 1`

These constraints are necessary to generate patterns appropriate for MUXClock operation. The FAST_CYCLE technique is not supported in the `system_clock` mode if the launch and capture clock are the same.

- The Verilog testbench written out by TetraMAX ATPG only supports a single period value for all cycle operations. This implies that a single waveform table can be taken into account when writing out the testbench. The delay waveform tables are not supported with the Verilog testbench. The flow is to write out the STIL vectors and then use the Verilog DPV PLI with VCS. Refer to the *Test Pattern Validation User Guide*.

Specifying Transition-Delay Faults

To start the transition-delay fault ATPG process, you need to select the transition fault model with the `set_faults` command. Then you can add faults to the fault list using the `add_faults` or `read_faults` command. You can select all fault sites, a statistical sample of all fault sites, or individually specified fault sites for the fault list.

The following sections show you how to specify transition-delay faults:

- [Selecting the Fault Model](#)
- [Adding Faults to the Fault List](#)
- [Reading a Fault List File](#)

Selecting the Fault Model

You select the transition fault model using the `set_faults -model transition` command. You can change the fault model during a TetraMAX ATPG session so that patterns produced with one fault model can be fault-simulated with another fault model. To do this, you need to remove faults and use the `set_patterns external` command before the fault simulation run.

The three available transition-delay fault ATPG modes (`last_shift`, `system_clock`, and `any`) can be selected by the `-launch_cycle` option to the `set_delay` command. The

default is the `system_clockmode`. This option selection is valid only if the transition model is selected with the `-model transition` option. In the `any` mode, TetraMAX ATPG will do the following:

- Attempt to detect all faults using `last_shift` mode
 - Apply `system_clock` mode to target faults left undetected by the `last_shift` mode
-

Adding Faults to the Fault List

The `add_faults` command adds stuck-at or transition faults to fault sites in the design. The faults added to the fault list are targeted for detection during test pattern generation.

To add a specific transition fault to the design, use the `pin_pathname -slow` option and specify R, F, or RF to add a slow-to-rise fault, a slow-to-fall fault, or both types of faults, respectively. To add faults to all potential fault sites in the design, use the `-all` option.

The following steps show you how to add a statistical sample of all faults to the fault list:

1. Add all possible transition faults.

```
add_faults -all
```

2. Remove all but the required percentage of transition faults (10 percent in this example).

```
remove_faults -retain_sample 10
```

Reading a Fault List File

To read a list of faults from a file, use the `read_faults` command.

A fault file can be read into TetraMAX ATPG and should have the format shown in the following example. Each node of the design has two associated transition faults: slow-to-rise (str) and slow-to-fall fault (stf). Attempting to read a fault list containing stuck-at fault notation (sa1 and sa0) results in an “invalid fault type” error message (M169).

```
str    NC    /TOP/EMU_FLK/SYNTOP_GG/NR1/A
stf    NC    /TOP/EMU_FLK/SYNTOP_GG/U123/Z
```

Pattern Generation for Transition-Delay Faults

The TetraMAX ATPG commands for transition fault ATPG are the same as the commands for stuck-at fault ATPG. You should be aware of how the command options affect the operation of ATPG for the transition-delay fault model.

The following sections describe the various TetraMAX ATPG commands used for transition-delay fault ATPG:

- [Using the set_atpg Command](#)
- [Using the set_delay Command](#)
- [Using the run_atpg Command](#)
- [Pattern Compression for Transition Faults](#)

- [Using the report_faults Command](#)
 - [Using the write_faults Command](#)
-

Using the set_atpg Command

The `set_atpg` command sets the parameters that control the ATPG process.

The `-merge` option is effective in reducing a number of transition patterns in the `last_shift` mode.

You can enable Fast-Sequential ATPG by using the command `set_atpg -capture_cycles d`, where `d` is a nonzero value. However, the `last_shift` mode is based strictly on the Basic-Scan ATPG engine. Therefore, when you use `run_atpg` in the `last_shift` mode, TetraMAX ATPG uses Basic-Scan ATPG only and generates Basic-Scan patterns, even if Fast-Sequential ATPG has been enabled. No warning or error message is reported to indicate that Fast-Sequential ATPG has been skipped.

The `system_clock` mode is based on Fast-Sequential ATPG engine. If Fast-Sequential ATPG is not enabled when you use `run_atpg` in the `system_clock` mode, TetraMAX ATPG reports an error message (M236).

When you enable Fast-Sequential ATPG with the `set_atpg -capture_cycles d` command, you must set the effort level `d` to at least 2 for the `system_clock` mode. If you try to set it to 1 in the `system_clock` mode, the `set_atpg` command returns an “invalid argument” error. For full-scan designs, you can set the effort level `d` to 2. For partial-scan designs, a number greater than 2 might be necessary to obtain satisfactory test coverage.

Using the set_delay Command

The `set_delay` command determines whether the primary inputs are allowed to change between launch and capture.

The default setting is `-pi_changes`, which allows the primary inputs to change between launch and capture. With this setting, slow-to-transition primary inputs can cause the transition test to be invalid.

The `-nopi_changes` setting causes all primary inputs to be held constant between launch and capture, thus preventing slow-to-transition primary inputs from affecting the transition test. This setting is useful only in the `system_clock` mode. The `-nopi_changes` characteristic must be set before you use the `run_atpg` command.

The `-nopi_changes` option causes an extra unclocked tester cycle to be added to each generated transition fault or path delay pattern. The use of a `set_drc -clock -one_hot` command might interfere with the addition of this unclocked cycle and is not recommended for use when the `-nopi_changes` option is in effect.

The primary outputs can still be measured between launch and capture. To mask all primary outputs, use the `add_po_masks -all` command.

Using the run_atpg Command

The `run_atpg` command starts the ATPG process. The `-auto_compression` option should be used.

The `-auto_compression` option works for transition-delay fault ATPG, but it is not as effective as it is for stuck-at. Also, when you use the `-auto_compression` option, you must enable the appropriate ATPG mode using the `-capture_cycles` option of the `set_atpg` command for the transition ATPG mode in effect: Basic-Scan ATPG for the `last_shift` mode, or Fast-Sequential ATPG for the `system_clock` or any mode.

The `run_atpg` command has three additional ATPG options: `basic_scan_only`, `fast_sequential_only`, and `full_sequential_only`. Under normal conditions, you should not attempt to use these options to start transition-delay fault ATPG. If you do so, be aware of the following cases:

- If you use the Full-Sequential ATPG engine with transition faults, you should be aware that its behavior is not controlled by `set_delay -launch_cycle` command options. If you want to avoid last-shift launch patterns and generate only system-clock launch patterns with the Full-Sequential engine, you must constrain all scan enable signals to their inactive values. Conversely, if you want to generate only last-shift launch patterns and avoid all system-clock launch patterns, you should be aware that there is no way to guarantee that you will get only last-shift launch patterns with the Full-Sequential engine. Even those last-shift launch patterns that it might generate will not be identical in form to those generated by the Basic-Scan ATPG engine.
- The `basic_scan_only` and `fast_sequential_only` options work for transition-delay fault ATPG when used correctly: `basic_scan_only` for the `last_shift` mode, or `fast_sequential_only` for the `system_clock` mode. If you use the wrong command option, no patterns are generated and no warning or error message is reported.

Pattern Compression for Transition Faults

Dynamic pattern compression specified by the `set_atpg` command works for transition faults as it does for stuck-at faults.

Using the report_faults Command

The `report_faults` command provides various types of information on the faults in the design.

You can use the `-slow` option to report a specific transition fault.

The fault classes for transition-delay fault ATPG are the same as for stuck-at ATPG. There are no specific fault classes that apply only to transition-delay faults. The faults classified as DI (Detected by Implication) before the ATPG process for transition-delay fault ATPG are the same as for stuck-at ATPG.

The total number of transition faults in a design is the same as the total number of stuck-at faults.

Using the `write_faults` Command

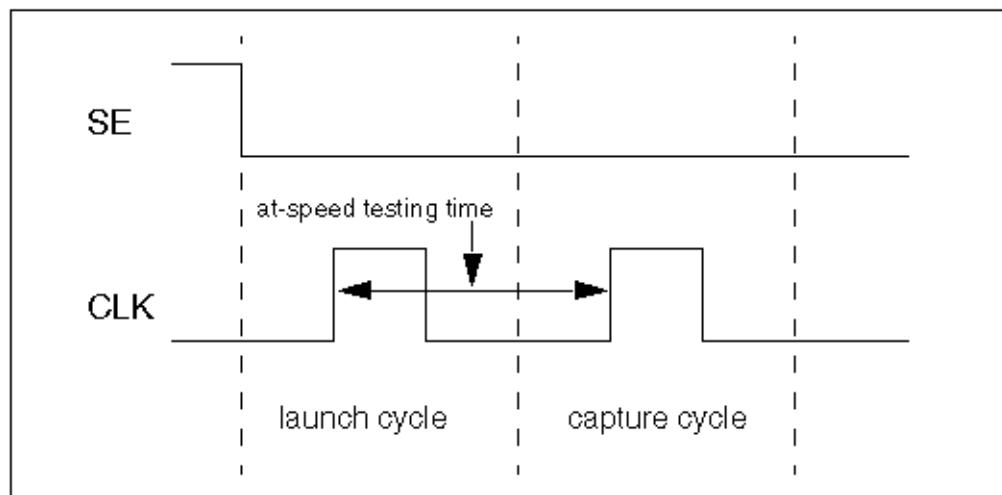
The `write_faults` command writes fault data to an external file. The file can be read back in later to specify a future fault list.

You can use the `-slow` option to write out a specific transition fault to a file.

Pattern Formatting for Transition-Delay Faults

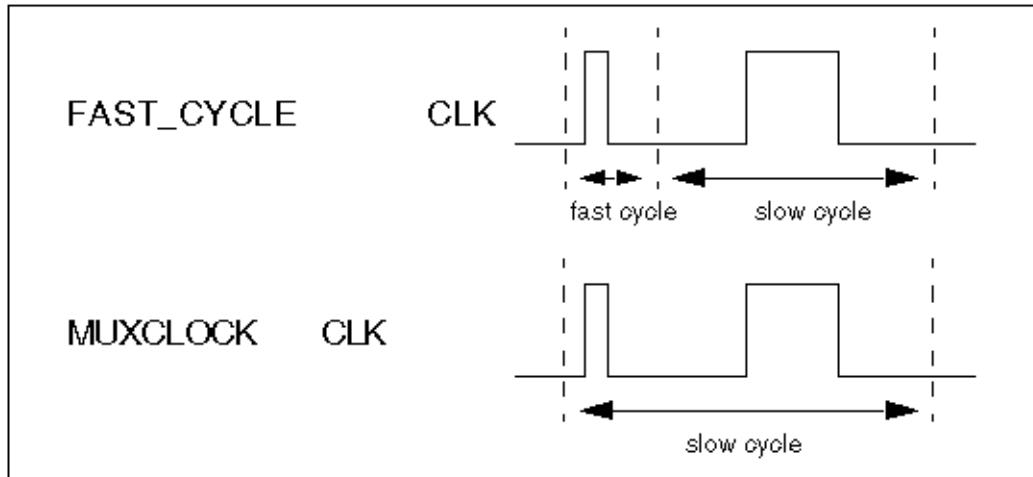
For a transition test to be effective, the time delay between launch and capture should be an at-speed value, as illustrated in [Figure 1](#).

Figure 1 At-Speed Transition Test Timing



A fast tester might be able to generate two clock pulses (cycles) at the required at-speed value without dynamic cycle-time switching or other special timing formatting. For these testers, TetraMAX can generate ready-for-tester transition patterns.

A slow tester might need dynamic cycle-time switching or a special timing format to test a transition fault at the required at-speed value. In general, two pattern formatting techniques are available for slow testers. In TestGen terminology, these two techniques are called `FAST_CYCLE` and `MUXClock`. [Figure 2](#) illustrates these techniques.

Figure 2 Pattern Formatting Techniques

Using the FAST_CYCLE technique, the cycle time is switched dynamically from fast time to slow time. Using the MUXClock technique, two tester timing generators are logically ORed to produce two clock pulses in one cycle.

The FAST_CYCLE technique is supported for the following cases:

- The `last_shift` mode is being used. The waveform format of the scan load and scan unload procedure in a STL procedure file can be different from that of a capture clock procedure.
- The `system_clock` mode is being used and the launch clock is different from the capture clock. In this case, each capture clock procedure can have its own waveform format.

The FAST_CYCLE operation is supported by defining specific WaveformTables to apply to the launch and capture vectors. The constructs necessary to support the creation of these test cycles are the same constructs used for path delay test generation. See the section "[Creating At-Speed WaveformTables](#)". The constructs presented in that section are also used to identify the launch and capture timing for transition-delay tests.

The testgen FAST_MUXCLOCK operation is supported by defining TetraMAX MUXClock constructs. However, to apply MUXClock behavior to transition tests requires the following set of options to be specified when transition tests are developed:

- `set_faults -model_transition`
- `set_delay -launch_cycle system_clock`
- `set_delay -nopi_changes`
- `add_po_masks -all`
- `set_atpg -capture_cycles > 1`

These options will support the creation of patterns that might merge the launch and capture operations into a single test vector necessary to support MUXClock application. To create MUXClock-based patterns use the same constructs defined for MUXClock path delay definitions. For information on these constructs, see [Generating Path Delay Tests](#).

MUXClock Support for Transition Patterns

The following limitations apply to MUXClock support for transition patterns:

- Output pattern files containing MUXClock waveforms are not yet readable in TetraMAX ATPG.
- Bidirectional clocks in a design are not supported in WGL output when MUXClock definitions are present. STIL output supports bidirectional clocks in the design.
- MUXclock is not supported with clock_grouping. To disable multiple clocks and dynamic clocking and use only a single clock for launch and capture, exclude these commands from your command file:
`#set_delay -common_launch_capture_clock
#set_delay -noallow_multiple_common_clocks`
- MUXClock is not supported with scan compression designs

Specifying Timing Exceptions From an SDC File

TetraMAX ATPG can read timing exceptions directly from an SDC (Synopsys Design Constraints) file. You can use an SDC file written by PrimeTime or create one independently, but it must adhere to standard SDC syntax. This section describes the flow associated with reading an SDC file. Note that this flow is supported only in Tcl mode.

The following sections describe how to specify timing exceptions from an SDC file:

- [Reading an SDC File](#)
- [Interpreting an SDC File](#)
- [How TetraMAX Interprets SDC Commands](#)
- [Controlling Clock Timing, ATPG, and Timing Exceptions for SDC](#)
- [Reporting SDC Results](#)

Note: The following limitation applies to SDC support in TetraMAX ATPG:

- Multicycle 1 paths cannot be used. In some applications, a `set_multicycle_path` command is used for one set of paths, but is followed by another `set_multicycle_path` command — with a `path_multiplier` of 1 on a subset of these paths. This is used to set that subset back to single-cycle timing. TetraMAX ATPG does not support this usage.

Reading an SDC File

You use the `read_sdc` command to read in an SDC file. Note that you must be in DRC mode (after you successfully run the `run_build_model` command, but before running the `run_drc` command) to use the `read_sdc` command.

Note the following:

- The SDC commands cannot be entered on the command line; they must be specified in an SDC file and can only be executed via the `read_sdc` command.
- The input SDC file must contain only SDC commands — not arbitrary PrimeTime commands. Constraints files comprised of arbitrary PrimeTime commands interspersed with SDC commands are unreadable. If the SDC file can be read into PrimeTime using its `read_sdc` command, then it can be read into TetraMAX ATPG. If it must be read into PrimeTime using the `source` command, then it cannot be read into TetraMAX ATPG. PrimeTime can write SDC, and this output is valid as SDC input for TetraMAX ATPG.

Interpreting an SDC File

To control how TetraMAX ATPG interprets an SDC file, you can specify the `set_sdc` command. This command will only work if you specify it before the `read_sdc` command. As is the case with the `read_sdc` command, you must be in DRC mode to use the `set_sdc` command.

Note that the `set_sdc` command settings are cumulative; this command might be run multiple times to prepare for a `read_sdc` command. If multiple `read_sdc` commands are required, you can also specify the `set_sdc` command before each `read_sdc` command to specify its verbosity and instance.

How TetraMAX Interprets SDC File Commands

TetraMAX ATPG creates timing exceptions for transition delay testing based on a set of SDC (Synopsys Design Constraints) file commands. Note that not all SDC commands are used for this purpose, however they all must be specified in an SDC file.

The following list describes the set of SDC commands that are used by TetraMAX ATPG, and how they are interpreted:

`set_false_path` -- This command creates a timing exception for a false path according to the specified from, to, or through points. TetraMAX ATPG does not distinguish between edges; this means, for example, that `-rise_from` is interpreted the same as `-from`.

`set_multicycle_path` -- This command creates a timing exception for a multicycle path according to the specified from, to, or through points. TetraMAX ATPG does not distinguish between edges. This means, for example, that `-rise_from` is interpreted the same as `-from`. Setup path multipliers of 1 are ignored.

`create_clock` and `create_generated_clock` -- For both of these timing exception commands, the `-name` argument and the `source_objects` are used to identify either the clock or the generated clock sources. Clocks must be traced to specific registers so there are some support limitations. “Virtual clock” definitions without a `source_object` are ignored. Multiple clocks that are defined with the same `source_objects` cannot be distinguished from each other. Note that clocks defined in the SDC file are only used to identify timing exceptions and only clocks defined by the TetraMAX ATPG `add_clocks` command or in the STIL protocol are used for pattern generation.

`set_disable_timing` -- This command creates a timing exception by disabling timing arcs between the specified points. TetraMAX ATPG does not support library cells in the `object_list`.

`set_case_analysis` -- This command is used to assist in tracing clocks to specific registers. Only static logic values (0 or 1, not rising or falling) are supported. This information is used based on the value set by the `set_drc -sdc_environment` command (see the “Controlling Clock Tracing” section for details).

`set_clock_groups` -- This command creates a timing exception that specifies exclusive or asynchronous clock groups between the specified clocks. It works only if the `-asynchronous` switch is used and the `-allow_paths` switch is not used. All other usages are ignored.

See Also

[How TetraMAX Processes Setup and Hold Violations](#)

Controlling Clock Timing, ATPG, and Timing Exceptions for SDC

You can control the following processes related to reading timing exceptions for a Synopsys design constraints (SDC) file:

- **Controlling Clock Timing**

You can control clock tracing using the `set_sdc -environment` command.

- **Controlling ATPG Interpretation**

In some cases, you might want to treat multicycle paths below a certain number as if they are single-cycle paths. To do this, use the `set_delay -multicycle_length <N>` command.

Based on this option, all `set_multicycle_path` exceptions with numbers of **N** or less are ignored. The default is to treat all multicycle paths of length 2 or greater as exceptions.

- **Controlling Timing Exceptions Simulation for Stuck-at Faults**

You can use the `set_simulation[-timing_exceptions_for_stuck_at | -notiming_exceptions_for_stuck_at]` command to control timing exceptions simulation for stuck-at faults. The default is `-notiming_exceptions_for_stuck_at`.

Reporting SDC Results

There are several ways you can report SDC results. You can report specific types of results using the `report_sdc` command (note that this command can only be run in TEST mode).

In addition, you can use the `report_settings_sdc` command to report the current settings specified by the `set_sdc` command.

A pindata type related to SDC is available. You can control the display of this pindata type by running the `set_pindata -sdc_case_analysis` command:

The format of the data is N/M (N is the case analysis setting from the SDC, and M is the TetraMAX constraint value). Unconstrained values are printed as X. You can also specify the display of this pindata type directly from the GSV Setup menu.

Slack-Based Transition Fault Testing

As geometries shrink, it is increasingly important to identify small delay defects. Standard transition fault testing is insufficient for detecting small delay defects because it focuses only on finding the simplest and shortest paths.

TetraMAX ATPG uses a special slack-based mode of transition fault testing to identify small delay defects. When this mode is activated, TetraMAX ATPG generates a specific set of transition fault tests that systematically identify the longest paths.

Using this testing methodology, you can extract slack data from PrimeTime, read this data into TetraMAX ATPG, and use various TetraMAX ATPG commands, command options, and flows related to testing small delay defects.

TetraMAX ATPG uses a single ATPG run to generate tests for both slack-based transition faults and regular transition faults.

The following sections describe the slack-based transition fault testing process:

- [Basic Usage Flow](#)
 - [Special Elements of Slack-Based Transition Fault Testing](#)
 - [Limitations](#)
-

Basic Usage Flow

The basic flow for slack-based transition fault testing includes the following steps:

- [Extracting Slack Data from PrimeTime](#)
- [Utilizing Slack Data in the TetraMAX Flow](#)
- [Command Support](#)

Extracting Slack Data from PrimeTime

TetraMAX ATPG uses a specific set of timing data extracted from PrimeTime. To obtain this information, you use the `report_global_slack` PrimeTime command to extract slack data for all pins from PrimeTime.

The sequence of commands is shown in the following example:

```
pt_shell> set timing_save_pin_arrival_and_slack TRUE
pt_shell> update_timing
pt_shell> report_global_slack -max -nosplit > <global_slack_file>
```

Note: The `-max` option is used with the `report_global_slack` command because PrimeTime considers setup margins to be "max" and hold margins to be "min". In this case, setup margins are required, so use the `-max` option to extract the minimum setup slacks.

The output format shown in the following example:

Max_Rise	Max_Fall	Point
4 .65	4 .40	SE_FMUL10/OP0_L0_reg_23/_Q
*	*	SE_FMUL10/OP0_L0_reg_23/_SE
-0 .82	-0 .80	SE_PEO/U16112/A1

Note: A * character is used instead of INFINITY.

Utilizing Slack Data in the TetraMAX Flow

After producing a slack data file, use the `read_timing` command to read this data into TetraMAX ATPG. Make sure you specify this command after entering DRC or TEST mode (after a successfully running the `run_build_model` or `run_drc` commands).

When TetraMAX ATPG reads a slack data file, it uses a set of slack-based transition fault testing processes to construct a pattern for the target fault. If TetraMAX ATPG does not read the slack file, regular transition-delay ATPG is performed.

How TetraMAX Integrates Slack Data

During ATPG, TetraMAX ATPG selects the first available fault from the list of target faults. It then uses the available slack data for the selected fault, and attempts to construct a delay test that makes use of the longest available sensitizable path. Secondary target faults for that same

pattern might not have their longest testable path sensitized because some values were already set in the test for the primary target fault. Faults that are detected only by fault simulation without being targeted by test generation are not necessarily be detected along a long path. However, the fault simulator will use the slack data to determine the size of defect that could be detected at that fault site by the pattern.

For maximum efficiency, ATPG typically targets the easiest solution. This means that transition faults are more likely detected along the shorter paths or paths with larger slack. Fault simulation and ATPG increase the efficiency by accounting for transition faults that are randomly detected by the tests generated for the targeted fault. Those transition faults detected only by fault simulation represent a large fraction of the detected faults and are usually detected along paths with slacks that are random with respect to all the paths on which the faults could be detected.

Command Support

Table 1 lists the key commands available to help validate the flow and pattern content.

Table 1: Key TetraMAX ATPG Commands for Slack-Based Transition Fault Testing

Command	Description
<code>read_timing file_name [-delete]</code>	Reads in minimum slack data in the defined format and optionally deletes previous data
<code>report_timing instance_name -all -max_gates number</code>	Reports pin slack data accepted by TetraMAX ATPG
<code>set_pindata slack</code>	Sets the displayed pindata type to show slack data
<code>set_delay [-noslackdata_for_atpg -slackdata_for_atpg]</code>	Turns on and off the slack-based transition fault testing function during ATPG. If slack data exists, the default is the <code>-slackdata_for_atpg</code> option.
<code>set_delay [-noslackdata_for_faultsim -slackdata_for_faultsim]</code>	Turns on and off the slack-based transition fault testing function during fault simulation. If slack data exists, the default is the <code>-slackdata_for_faultsim</code> option.
<code>set_delay -max_tmgn <float defect%></code>	Defines the cutoff for faults of interest for slack-based transition fault testing generation. Faults with minimum slacks larger than the <code>-max_tmgn</code> parameter are targeted by the normal transition fault ATPG algorithm rather than by the slack-based algorithm.

Table 1: Key TetraMAX ATPG Commands for Slack-Based Transition Fault Testing (Continued)

Command	Description
<code>set_delay -max_delta_per_fault float</code>	Sets a level between the longest path and the path on which the fault is detected. Full detection is still credited, and the fault is dropped from further consideration. The default is zero (full credit is given only when detection is on the minimum slack path).
<code>report_faults [-slack tmgn [integer float]]</code>	Reports a histogram of faults based on the minimum slack numbers read in by the <code>read_timing</code> command. This histogram is either fixed in the number of buckets or fixed in the slack interval between two consecutive buckets. The fixed number of buckets is specified by an integer and the fixed bucket interval is specified with a float. The default is an integer of 10.
<code>report_faults -slack tdet [integer float]</code>	Reports a histogram of faults based on the slack numbers for the detection path for each fault (detection slacks). This histogram is either fixed in number of buckets or fixed in the slack interval between two consecutive buckets. The fixed number of buckets is specified by an integer and the fixed bucket interval is specified with a float. The default is an integer of 10.
<code>report_faults -slack delta [integer float]</code>	Reports a histogram of faults based on the difference between detection slacks and minimum slacks. The reported histogram is either fixed in number of buckets or fixed in the slack interval between two consecutive buckets. The fixed number of buckets is specified by an integer and the fixed bucket interval is specified with a float. The default is an integer of 10.

Table 1: Key TetraMAX ATPG Commands for Slack-Based Transition Fault Testing (Continued)

Command	Description
report_faults -slack effectiveness	Reports a measure of the effectiveness of the slack-based transition fault set. The measure varies from 0 percent (no faults of interest with detection slacks smaller than the <code>-max_tmgn</code> parameter) to 100 percent (all faults of interest detected on the minimum-slack path).
report_faults -slack sdql	Reports the SDQL (Statistical Delay Quality Level) value for the pattern set.
set_delay -sdql_coefficient [A B C D E]	Specifies the values to be used in either the power function or the exponential function when computing the SDQL number. If you do not specify this option, the defaults of A, B, C, D, and E are 1, 1, 0, 0, and infinity.
set_delay -sdql_histogram -sdql_power_function -sdql_exponential_function	Specifies the type of probability distribution function used in computing the SDQL.

Special Elements of Slack-Based Transition Fault Testing

This section describes some of the unique characteristics related to slack-based transition fault tests. These special elements are described in the following topics:

- [Allowing Variation From the Minimum-Slack Path](#)
- [Defining Faults of Interest](#)
- [Reporting Faults](#)

Allowing Variation From the Minimum-Slack Path

When creating slack-based transition fault tests, the transition fault test generator targets the path with the minimum slack for the primary target fault. As with regular transition fault ATPG, there might be secondary target faults that are targeted following the successful generation of a test for the primary target fault.

Many faults detected during fault simulation are likely not be targeted faults for the test generator. You need to decide if you are willing to accept a test that detects a transition fault, or a

test that detects the fault along a path with small slack. To specify the type of test you are willing to accept, use the `set_delay -max_delta_per_fault` command.

If you are unwilling to accept a fault unless it has been detected along the path that has the absolute smallest slack for the fault, you can use a setting of 0 for the `-max_delta_per_fault` parameter (the default setting). If you want to accept any test that comes within 0.5 time units of the minimum slack for the fault, set `-max_delta_per_fault` to 0.5. This allows you to control when faults can be dropped from simulation in a slack-based transition fault ATPG run.

When a fault is detected with a slack that exceeds the minimum slack by more than the `-max_delta_per_fault` parameter, the fault goes into a special sub-category of Detected (DT). This sub-category is called Transition Partially-detected (TP). A fault that has gone into the TP category might continue to be simulated in hopes of getting a better test for the fault.

A fault detected with a slack equal to or smaller than the `-max_delta_per_fault` parameter is placed in the DS category normally used for detected transition faults. A DS category fault is always dropped from further simulation.

Specifying the `-max_delta_per_fault 0` option likely produces the highest quality test set. However, this specification also likely produces the longest runtimes and the largest test sets. The `-max_delta_per_fault` setting allows you to choose an acceptable trade-off point for test set quality versus runtime and test set size.

Defining Faults of Interest

You can specify how small the slack needs to be for TetraMAX ATPG to target the fault with the slack-based transition fault test generation algorithm. If you specify the `set_delay -max_tmgn` command, the test generator uses the slack-based algorithm to target only those faults with a slack smaller than the number specified by the `-max_tmgn` parameter. All faults not designated as “faults of interest” are targeted by the normal transition fault test generation algorithm. All faults are fault-simulated even if they are not designated as “faults of interest” targeted for test generation.

You can use the `report_faults -slack tmgn` command to examine the distribution of slacks to determine a reasonable value of the `-max_tmgn` option. This command prints a histogram of the slack values that are read in by the `read_timing` command. You can also use the `report_faults -slack tmgn` command to specify how many categories are included in this report.

Reporting Faults

Slack-based transition fault tests are applied to paths with a smaller slack than those typically activated in regular transition fault test generation. The `report_faults` command includes several options that facilitate the examination of this data:

- The `-slack tdet` option prints a histogram that shows the slack of the detection paths. You can compare this data directly against the output of the `-slack tmgn` option to see how close TetraMAX ATPG got to the minimum slack paths.
- The `-slack delta` option clearly shows the slack of the detection paths. The reporting histogram associated with this option is based on the difference between the slacks for

the detection paths and the minimum slack read from the slack data file. A distribution heavily skewed toward the zero end of the continuum indicates a highly successful slack-based transition fault test generation.

- The `-slack effectiveness` option reports a measure of delay effectiveness based on how close the slacks for the fault detection paths came to the minimum slacks. If every fault defined to be of interest is detected on its minimum slack path, the delay effectiveness measure would be 100 percent. If no faults of interest are detected on paths that have slack smaller than the `-max_tmgn` parameter used to define faults of interest, the delay effectiveness measure is 0 percent.

The output of the `report_faults -all` command also includes additional fields related to the slack-based transition fault testing. For more information, see the "Slack-Based Transition Fault Format" section of the "Understanding the report_faults Output" topic in TetraMAX Help.

Limitations

The following limitations currently apply to slack-based transition fault testing:

- [Engine and Flow Limitations](#)
- [ATPG Limitations](#)
- [Limitations in Support for Bus Drivers](#)

Engine and Flow Limitations

Last-shift launch, two-clock, and fast-sequential transition fault testing are supported. There is currently no support for Full-Sequential mode.

ATPG Limitations

There are two limitations for slack-based transition ATPG:

- **Second Smallest Slack**

If the path with the smallest slack for a given fault is untestable, TetraMAX ATPG begins normal back-tracking in an attempt to find a test along some other path. There is no guarantee that the second path TetraMAX ATPG will try is the path with the second smallest slack. For now, the only guarantee is that the first path tried is the path with the smallest slack.

- **Test Might End Prematurely at PO**

When propagating a fault effect along the minimum-slack propagation path, the fault effect might propagate to a primary output. If this occurs, and the fault can be considered detected at the primary output, TetraMAX ATPG will stop trying to propagate the fault effect along the minimum-slack path. This can produce fault detection on a path with larger slack than required.

In this case, the detection slack is measured accurately and will reflect the detection along the path with greater slack to the primary output. This is not normally a problem for transition fault test generation, because the `-nopopo_measures` option is commonly set

for transition faults. If that option is set, then the fault cannot be detected at a primary output so the propagation along the minimum-slack path will continue uninterrupted.

Limitations in Support for Bus Drivers

Full slack-based transition fault testing support is not available for BUS drivers. TetraMAX ATPG does not choose the minimum slack path when back-tracing through a bus driver if that path goes through the enable input. TetraMAX ATPG always chooses the path through the data input to the driver. This limitation applies to test generation only. The detection slack and the slack delta are accurately reported in all cases.

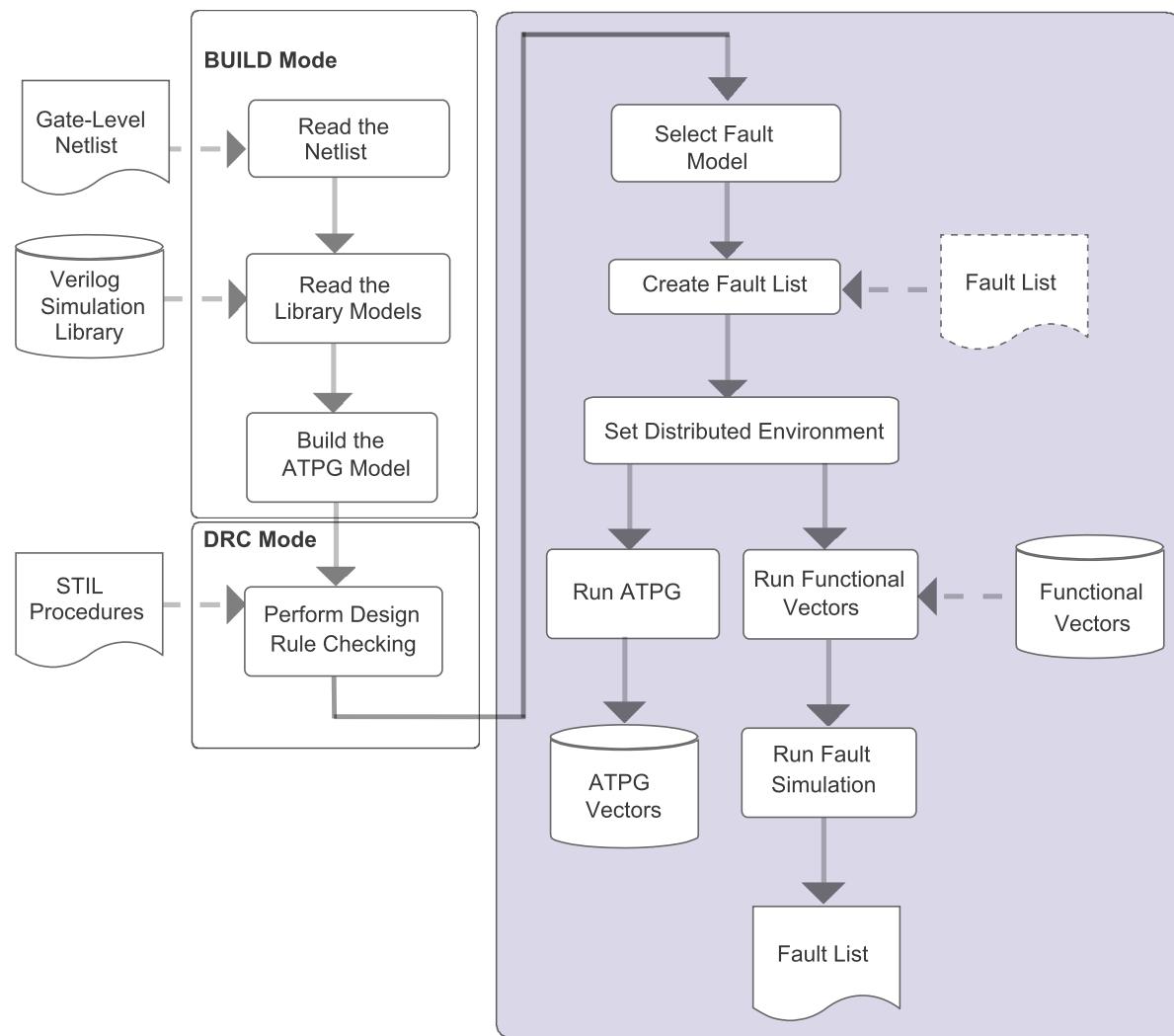
24

Running Distributed ATPG

TetraMAX distributed ATPG launches multiple slave processes on a computer farm or on several standalone hosts. The slave processes read an image file and execute a fixed command script. ATPG distributed technology does not differentiate between multiple CPUs and separate workstations. Each slave requires as much memory as a single CPU TetraMAX run. Distributed ATPG offers both scalability and runtime improvement.

The following topics describe how to set up and run distributed ATPG:

- [Checking Your Environment for Distributed Processing](#)
- [Machine Access and Setup for Distributed ATPG](#)
- [Preparing to Run Distributed Processing](#)
- [Setting Up the Distributed Environment](#)
- [Setting Up the Distributed Environment With Load Sharing](#)
- [Verifying Your Environment for Distributed ATPG](#)
- [Starting Distributed ATPG](#)
- [Starting Distributed Fault Simulation](#)
- [Debugging Distributed ATPG Issues](#)
- [Distributed ATPG Limitations](#)

Figure 1 Distributed Processing Flow

Checking Your Environment for Distributed Processing

Perform the following tasks to make sure you can run ATPG distributed processing:

1. Verify that you can use the UNIX `rsh` command to log in on each slave machine without having to supply a password, as shown in the following example:

```
rsh <machine_name>
```

See "[Machine Access and Setup](#)."

2. Verify that each slave machine has an identical search path to TetraMAX ATPG, as shown in the following example:

```
rsh <machine_name> which tmax
```

See "[Machine Access and Setup](#)" below.

3. If you are using LSF for launching slaves, find out from your system administrator what queues or special options you need to use. The machine that you run as master should be a valid LSF submit host. You can verify that it is an LSF host by trying a sample LSF job.

```
/path/to/bsub -q hw-vlsi tmax -shell -version
```

4. If you are using GRD for launching the slave, ask your system administrator if you need to use any special project name or queue. The machine that you run as master should be a valid GRID submit host. You can verify that it is a grid host by trying a sample grid job.

```
/path/to/qsub $SYNOPSYS/bin/tmax -shell -version
```

Machine Access and Setup for Distributed ATPG

Distributed ATPG makes use of the UNIX `rsh` command to login and launch distributed processing on slave machines. To setup your system:

1. Make sure that you have network access to each slave machine. You should successfully perform step 2 described previously, which verifies that you can log in on each slave machine without having to supply a password.
If you are prompted for a password, create a `.rhosts` file in your home directory. Add to that file the names of the slave machines in a list format. This list categorizes the slave machines as "trusted hosts."
Another technique is to put a single "+" entry (without the quotation marks) in your `.rhosts` file.
If you are still not able to `rsh` to a machine after creating a `.rhosts` file, refer to the manpage for `rsh` or talk to your System Administrator or contact your IT group.
Inability to use `rsh` will result in a M316 error message when you attempt to run distributed ATPG.
2. Make sure TetraMAX ATPG is set up identically on each slave machine. You should successfully do step 3 above, which verifies that each slave machine accesses TetraMAX ATPG through the same network path. If this is not the case
3. There might be something in your `.cshrc` file (or equivalent) might be preventing the machine from locating TetraMAX ATPG on the network. This could happen if you have

tty, stty, or interactive statements in your .cshrc file. To debug this, add echo statements to your .cshrc file to determine where the search fails.

4. If you cannot fix your .cshrc file or you are not setting the path to TetraMAX ATPG through your .cshrc (or equivalent file), you can use the `-script` option of the `set_distributed` command to pass a script for setting up TetraMAX ATPG on the slave. Here's an example csh/sh/tcsh script:

```
#!/bin/csh -f
setenv SYNOPSYS /tools/tmax/v-2003.12
set path ($SYNOPSYS/bin $path)
tmax $*
```

The inability to set the search path to TetraMAX ATPG on the slave machines causes a M315 error message when you attempt to run distributed ATPG.

Preparing to Run Distributed Processing

Before running distributed processing, you need to build the design model and run DRC.

Example Script:

```
BUILD-T> read_netlist Libs/*.v -delete -library -noabort
BUILD-T> run_build_model top_level
DRC-T> set_drc top_level.spf
DRC-T> run_drc
```

For details, see "Building the ATPG Model" and "Running DRC."

You also need to select the fault model and create the fault list. Note that N-detect ATPG and fault simulation are not supported for distributed ATPG.

The following is a summary of the support for distributed ATPG and the various fault models:

- The stuck-at fault model is supported by basic-scan, fast-sequential, and full-sequential ATPG
- The path delay fault model is supported by fast-sequential, and full-sequential ATPG
- The transition fault model is supported by basic-scan, fast-sequential, and full-sequential ATPG
- The IDDQ fault model is supported by basic-scan and fast-sequential ATPG
- The bridging fault model is supported by basic-scan and fast-sequential ATPG

The following example scripts are for selecting fault models for distributed processing:

Example 1:

```
TEST-T> set_faults -model stuck
TEST-T> add_nofaults top_level/module1/sub_mod
TEST-T> add_faults -all
```

Example 2:

```
TEST-T> set_faults -model transition
TEST-T> set_delay -nopi_change -nopo_measure
```

```
TEST-T> set_delay -launch last_shift
TEST-T> read_faults specFaultList.flt
```

Setting Up the Distributed Environment

Defining the working directory is the first step in setting up the environment for distributed processing. The working directory stores all the files required for exchanging data between the various machines, including the log files. This directory must be accessible by each machine involved in the distributed process and they must be able to read from and write to this directory, as shown in the following example:

```
TEST-T> set_distributed -work_dir /home/dist/work_dir
```

The working directory must be specified using an absolute path name starting from the root of the system. Relative paths are not supported in the current TetraMAX ATPG release. If you do not specify a working directory, the current directory is used as the default work directory.

After you set the working directory, you can populate the distributed processors list; for example:

```
TEST-T> add_distributed_processors zelda nalpari
      Arch: sparc-64, Users: 22, Load: 2.18 2.14 2.17
      Arch: sparc-64, Users: 1, Load: 1.45 1.41 1.40
```

The following commands help you maintain this list:

- `add_distributed_processors`
- `remove_distributed_processors`
- `report_distributed_processors`

For every machine, you automatically get the type of platform (Architecture), as well as the number of users currently logged on that machine and the processor load. You can add as many distributed processes as you want on one machine. However, you should know in advance the number of processors on that machine in order not to start more distributed processes than the number of available processors. Even if it is technically possible, the various processes would have to share time on the processors; thus you will not be able to take full advantage of the parallelization.

TetraMAX supports heterogeneous machine architectures (sparcOS5, Linux, and HP-UX). For example,

```
TEST-T> add_distributed_processors proc1_sparcOS5 proc2_Linux \
      proc3_HPUX
```

You can visualize the current list of machines in the list of distributed processors with the `report_distributed_processors` command; for example:

```
TEST-T> report_distributed_processors
      Working directory ==> "/remote/dtg654/atpg/dfs" (32bit)
      -----
      MACHINE: zelda [ARCH: sparc-64]
      MACHINE: nalpari [ARCH: sparc-64]
      -----
```

You get both the name of the machine and its architecture. If you see the same machine name several times, this means that several distributed processes were launched on this machine. The working directory is also displayed in the report along with the type of files in use (32- or 64-bit). This type of file is automatically determined by the master machine. If the master machine is a 32-bit machine, then distributed processes will have to be 32-bit also. If the master is a 64-bit machine, then everything has to follow 64-bit conventions.

You might want to remove some machines from this list (for example because of an overloaded machine). In this case, you can use the `remove_distributed_processors` command; for example:

```
TEST-T> remove_distributed_processors zelda
TEST-T> report_distributed_processors
Working directory ==> "/remote/atpg/dfs" (32bit)
-----
MACHINE:    nalpari [ARCH: sparc-64]
-----
```

You can use the `report_settings distributed` command to get a list of the current timeout and shell settings; for example:

```
BUILD-T> report_settings distributed
distributed =      shell_timeout=30, slave_timeout=100,
                  print_stats_timeout=30, verbose=-noverbose,
                  shell=rsh;
```

Setting Up the Distributed Environment With Load Sharing

TetraMAX supports the load sharing facility (LSF) and GRID network management tools. When you are using load sharing, jobs are submitted to a queue instead of to specific machines. The load sharing system manager then decides on which machine the job is started. This allows you to maximize the usage efficiency of your network.

To populate the distributed processor list, you need to use the `add_distributed_processors` command to specify the absolute path to the LSF and GRID submission executables (`bsub`), as well as the number of slaves to be spawned and additional options. For using LSF to launch the slaves, all of these options must be specified. For using GRID to launch the slaves, all of these options must be specified as well as the `-script` option of the `set_distributed` command. If you do not have any additional options to pass to `bsub`, you can pass empty options using `-options " "`. For descriptions of these options, see the online help for the `add_distributed_processors` command.

Note the following example:

```
BUILD-T> add_distributed_processors \
-lsf /u/tools/LSF/mnt/2.2-glibc2/bin/bsub -nslaves 4 \
-options "-q lb0202"
BUILD-T> report_distributed_processors
Working directory ==> "/remote/dtgnat/Distributed"
(32bit)
```

```
-----***-----
MACHINE    **lsf** [ARCH:      linux]
MACHINE    **lsf** [ARCH:      linux]
MACHINE    **lsf** [ARCH:      linux]
MACHINE    **lsf** [ARCH:      linux]
-----***-----
```

Notice that instead of getting some distributed processors in the report, you see ****lsf****. This is because no job has been started yet, and thus, no distributed processor has been assigned to the job. After you issue the `run_atpg -distributed` command (or the `run_fault_sim -distributed` command), four jobs are assigned to four distributed processors. However, this is transparent to you.

You cannot remove only one distributed processor from the list when you are using the LSF environment. If you simply want to change the current number of distributed processors in the pool, you have to issue a new `add_distributed_processors` command with the correct value for the `-nslaves` option. Every time you issue an `add_distributed_processors` command under the LSF environment, it overrides the previous definition of your distributed processor list. Here is an example:

```
BUILD-T> add_distributed_processors \
-lsf /u/tools/LSF/mnt/2.2-glibc2/bin/bsub -nslaves 3 \
-options "-q lb0202"
BUILD-T> report_distributed_processors
Working directory ==> "/remote/dtgnat/Distributed"
(32bit)
-----***-----
MACHINE    **lsf** [ARCH:      linux]
MACHINE    **lsf** [ARCH:      linux]
MACHINE    **lsf** [ARCH:      linux]
-----***-----
```

Verifying Your Environment

Each time TetraMAX ATPG starts a distributed process, it issues the `tmax` command. As a consequence, be sure you have the `tmax` script directly accessible from each distributed processor machine. Do not alias this script to another name or TetraMAX ATPG will not be able to spawn distributed processes. The safest approach is to have the path to this script added in your PATH environment variable. For example:

```
setenv SYNOPSYS /softwares/synopsys/2004.12
set path = ( $SYNOPSYS/bin $path )
```

For a discussion about the use of the `SYNOPSYS_TMAX` environment variable, see “Specifying the Location for TetraMAX Installation”.

Remote Shell Considerations

If you are running distributed processing by directly running the host names, TetraMAX ATPG relies on the `rsh` (`remsh` for HP platforms) UNIX command to start a process on a distributed

processor machine. This command is very sensitive to the user environment, so you could experience some problems because of your UNIX environment settings. In case you get an error message while adding a distributed processor, refer to the message list at the end of this document to find out the reason and some advice on how to solve the problem.

You need to have special permissions to start a distributed process with a `rsh` (or `remsh`) command. In a classical UNIX installation, those permissions are given by default, however your system administrator might have changed them. If you experience any issue with starting slaves and suspect it is due to this command, enter the following:

```
rsh distributed_processor_machine "tmax -shell"
```

If you get an error message, it is related to your local UNIX environment. Contact your system administrator to solve this issue.

Tuning Your .cshrc File

You should pay special attention to what you put in your `.cshrc` file. Avoid putting commands that exercise the following behavior:

- Are interactive with the user (that is, the system asking the user to enter something from the keyboard). Because you will not have the ability to answer (distributed processes are transparent to the user), it is likely that the process will halt waiting for an answer to a question you will never get.
- Require some GUI display. Your `DISPLAY` environment variable will not point to your master machine when the `rsh` (or `remsh`) command starts a distributed process. As a consequence, the system will put the task “tty output stopped mode” on the distributed processor machine, and your master will wait for a process that has quit running.

If you have any trouble using the `add_distributed_processors` command, you might want to have a dedicated `.cshrc` file for running your distributed tasks. A basic configuration should help you get through these issues.

Checking the Load Sharing Setup

If you are planning to run distributed ATPG with load-sharing software, make sure you have the following information available to you.

- Path to the load sharing application (`bsub` for LSF and `qsub` for GRID)
- Required options (like project name or queue name)

The TetraMAX ATPG session for the master must be run on a machine that is a valid submit host capable of submitting jobs to load sharing software.

Starting Distributed ATPG

Distributed ATPG works in a similar way as distributed fault simulation. You simply have to add the `-distributed` switch on the `run_atpg` command line to trigger a distributed job; for example:

```
TEST-T> run_atpg -distributed -auto
```

The master process sends the fault information to the distributed processors in a collapsed format. Thus, all reports refer to the collapsed fault list. Note that the reported faults will not add up: there is a difference between collapsed and non-collapsed faults. The master only sends active faults.

Each slave contains all the faults. In this case, the fault list is not split; only the ATPG process is split. The slave log files try to report uncollapsed faults, but since they only receive collapsed faults information, the number reported is actually collapsed faults.

[Example 1](#) shows how the master collapse fault list correlates to the slaves uncollapsed fault list. In this case, 1715195 is the key number of faults that appear in both reports. Note that even though the slave file reports the faults as uncollapsed, the faults are actually the collapsed list from the master.

Example 1 Comparing the Master Collapse Fault List to the Slaves Uncollapsed Fault List**From master log file:**

```

report_faults -summary -collapse
    Collapsed Stuck Fault Summary Report
-----
fault class          code #faults
-----
Detected             DT   802240
Possibly detected    PT   0
Undetectable         UD   34404
ATPG untestable      AU   127879
Not detected         ND   1715195
-----
total faults          2679718
test coverage        30.33%
-----
run_atpg -auto -dist
Master: Saving image of session for slaves ...
Master: Spawning the slaves ...
Master: Starting distributed process with 3 slaves ...
Slaves: About to get licenses ...
Slaves: About to restore master's session ...
Master: Removing temporary files ...
Master: Sending 1715195 faults to slaves ...
Master: End sending faults. Time = 14.00 sec.

```

From slave log file:

```

run_atpg -auto
*****
* NOTICE: The following DRC violations were previously *
* encountered. The presence of these violations is an *
* indicator that it is possible that the ATPG patterns *
* created during this process might fail in simulation. *
*
* Rules: C8
*****
ATPG performed for stuck fault model using internal pattern
source.
-----
---
#patterns #patterns #faults #ATPG
faults test process
simulated eff/total detect/active red/au/abort coverage CPU
time
-----
---
Begin deterministic ATPG: #uncollapsed_faults=1715195, abort_

```

```
limit=10...
```

If you have some vectors in the external buffer before starting distributed ATPG, they are automatically transferred into the internal buffer. The new vectors created during distributed ATPG are added to this existing set of vectors. After the run is complete, you will have both the external vectors and the ATPG created vectors in the internal pattern buffer. If you do not want to merge those sets, you have to clean the external buffer before starting distributed ATPG by issuing a `set_patterns -delete` command.

As the following transcript shows, TetraMAX ATPG starts the various processes and issues some informational messages to keep you informed at the beginning of the run. A warning or error message is issued if TetraMAX ATPG cannot proceed. Then, TetraMAX ATPG starts generating the vectors.

```
run_atpg -auto -distributed
Master: Saving image of session for slaves ...
Master: Spawning the slaves ...
Master: Starting distributed process with 2 slaves ...
Slaves: About to get licenses ...
Slaves: About to restore master's session ...
Master: Removing temporary files ...
Master: Sending 5918 faults to slaves ...
Master: End sending faults. Time = 1.00 sec.
-----
#patterns #collapsed faults test process
      total    inactive/active coverage   CPU time
-----
Compressor unload adjustment completed:
#patterns_adjusted=241,
#patterns_added=0,
CPU time=1.00 sec.

      Uncollapsed Stuck Fault Summary Report
-----
fault class          code #faults
-----
Detected             DT     8109
Possibly detected    PT      0
Undetectable         UD     10
ATPG untestable     AU     28
Not detected         ND     11
-----
total faults          8158
test coverage        99.52%
fault coverage        99.40%
ATPG effectiveness   99.87%
-----
      Pattern Summary Report
-----
#internal patterns      243
-----
```

Where:

#patterns total = The total number of patterns generated up to this point.
#collapsed faults inactive = The number of collapsed faults already processed by TetraMAX ATPG.

#collapsed faults active = The number of collapsed faults not yet processed.
process CPU time = The time consumed up to this point.

At the end of the pattern generation process, TetraMAX ATPG automatically prints a summary for the faults and the vectors.

Note:

When using the `set_atpg -patterns max_patterns` command, for some designs that run quickly, the master sends a signal to stop the slaves. However, because of a network delay for this signal, the pattern count is already met; therefore the pattern count might already have exceeded the limit.

Saving Results

The following sample script shows you how to save results:

```
TEST-T> set_faults -summary verbose
TEST-T> report_faults -summary
TEST-T> report pattern -summary
TEST-T> write_faults final.flt -all -collapsed -compress gzip -
replace
TEST-T> write_patterns final_pat.bin.gz \
-format binary -compress gzip -replace
TEST-T> write_patterns final_patv -format verilog_single_file -
replace
```

Distributed Processor Log Files

When you run distributed ATPG or distributed fault simulation, the tool creates a log file in the work directory for each slave. The name of this log file is derived from the name of the master log file appending a number to it. For example, if the master log file is defined with a `set_messages log run.log -replace` command, a command that indicates you are running distributed ATPG with four slaves, the log files that are created would be called "run.log.1," "run.log.2," "run.log.3," and "run.log.4."

The tool creates the slave log files to give you visibility to the activity happening on the slaves.

Note that if you run distributed ATPG multiple times in the same session, the slave log files are overwritten by each run. If you want to prevent the slave log files from being overwritten, you can either save a copy or redefine the work directory by issuing a `set_distributed -work_dir` command before starting a new distributed run.

Starting Distributed Fault Simulation

After the functional vectors are read into the TetraMAX external pattern buffer, you simply need to add the `-distributed` option to the `run_fault_sim` command to trigger the parallelization of the process; for example:

```
TEST-T> run_fault_sim -distributed
Master: Saving patterns for slaves ...
Master: Saving image of session for slaves ...
Master: Spawning the slaves ...
Master: Starting distributed process with 2 slaves .

..
Slaves: About to get licenses ...
Slaves: About to restore master's session ...
Slaves: About to read in patterns ...
Master: Removing temporary files ...
Master: Sending 98 faults to slaves ...
Master: End sending faults. Time = 0.00 sec.

-----
#patterns #collapsed faults test process
simulated inactive/active coverage CPU time
----- ----- ----- -----
Fault simulation completed: #patterns=32
Uncollapsed Path_delay Fault Summary Report
-----
fault class code #faults
----- -----
Detected DT 61
  detected_by_simulation DS (2)
  detected_robustly DR (59)
Possibly detected PT 0
Undetectable UD 0
ATPG untestable AU 33
  atpg_untestable-not_detected AN (33)
Not detected ND 4
  not-controlled NC (4)
-----
total faults 98
test coverage 62.24%
fault coverage 62.24%
ATPG effectiveness 95.92%
-----
Pattern Summary Report
-----
#internal patterns 0
#external patterns (pat.bin) 32
```

```
#full_sequential_patterns 32
```

Events After Starting A Distributed Run

First, the master machine writes an image of the database in the working directory. This image is a binary file containing everything the distributed processors need to know to process the fault simulation. This file can be rather large, because it is based on the size of your design, so as soon as the database is read by the slaves, it is deleted from the disk. Next, the distributed processes are started (see the message in the example report shown in the next section). If something goes wrong at this step (problem with starting the slave processors), TetraMAX ATPG will notify you and stop.

After the distributed machines read the database, they are in the same state as the master with respect to the information about the design. The fault list is then split among the various processors and they all start to run concurrently. Whenever a slave processor finishes its job, it sends some information back to the master machine and then it shuts down. If any of the slaves unexpectedly dies during the process, the master machine will detect it and that process stops. An error message is issued to notify you. After every slave processor finishes, the master machine computes the fault coverage and prints out the final results.

Interpreting Distributed Fault Simulation Results

The transcript that follows shows the relevant information displayed during distributed fault simulation.

```
TEST-T> run_fault_sim -distributed
Master: Saving patterns for slaves ...
Master: Saving image of session for slaves ...
Master: Spawning the slaves ...
Slaves: About to get licenses ...
Slaves: About to restore master's session ...
Slaves: About to read in patterns ...
Master: Removing temporary files ...
Master: Sending 98 faults to slaves ...
Master: End sending faults. Time = 0.00 sec.
-----
#patterns #collapsed faults test process
simulated inactive/active coverage CPU time
----- ----- ----- -----
----- ----- ----- -----
Fault simulation completed: #patterns=32
Uncollapsed Path_delay Fault Summary Report
-----
fault class code #faults
----- -----
Detected DT 61
detected_by_simulation DS (2)
detected_robustly DR (59)
Possibly detected PT 0
Undetectable UD 0
```

```

ATPG untestable           AU      33
atpg_untestable-not_detected AN    (33)
Not detected             ND      4
not-controlled           NC    (4)
-----
total faults              98
test coverage            62.24%
fault coverage            62.24%
ATPG effectiveness        95.92%
-----
Pattern Summary Report
-----
#internal patterns          0
#external patterns (pat.bin) 32
#full_sequential patterns   32
-----
```

Where:

#patterns simulated = The number of patterns simulated

#collapsed faults inactive = The number of faults TetraMAX ATPG has already processed

Debugging Distributed ATPG Issues

You might encounter the following issues when setting up to run distributed ATPG:

- If you specify the host directly, verify your environment based on information provided in "Checking your Environment" above. If you are using the -script option of the set_distributed command and your command file contains lines similar to this:

```
set_distributed -script /home/pjaini/datpg_setup
add_distributed_processors yosemite goldengate
```

Enter something similar to the following:

```
rsh yosemite /home/pjaini/datpg_setup -shell -version
```

This should return the proper version of TetraMAX ATPG and you should not see any additional messages.

- For GRID, assume your command entries look like the following to launch the slave processors:

```
set distributed processor -script /user/larry/tmax_launch
add distributed processor -grd "/hw/tools/grid/qsub" -nslaves 5 -options "-P hw-rush"
```

Then, in an xterm window, enter:

```
% /hw/tools/grid/qsub -P hw-rush /user/larry/tmax_launch -version -shell
```

Check to make sure your GRID has been launched. You should see a message from GRID indicating your job has been submitted and the TetraMAX version string is printed wherever your GRID job output would normally appear. The standard output from a GRID

job could be sent to you via email or stored in some file in your home directory, depending on your GRID settings.

If the job does not launch, find the set of options (that is, project name, queue name, and so forth) required for launching a GRID job.

- For LSF, assume your command entry looks like the following to launch the slave processors:

```
add_distributed_processors -lsl "/path/to/bsub" -nslaves 5 -options "-q bnrmal"
```

Then, in an xterm window, enter:

```
% /path/to/bsub -q normal tmax -version -shell
```

Check to make sure your LSF job has been launched. Your indication for this is an email notifying you that the job has ran and the TetraMAX version string prints in the output.

Note that some LSF setups might mandate the use of certain queues or project names.

- Debugging a "Master: Couldn't start the daemon ..." message. There are two conditions where this message could occur:

1. The job was never launched. To check that the job launched, enter the following when you see the "Master: Spawning the slaves.." message:

For GRID:

```
qstat -u <username>
```

For LSF:

```
bjobs -u <username>
```

If you do not get an indication that the job is running, check to make sure you can launch the jobs from xterm window as described above.

2. Jobs were launched, but are not scheduled to be executed yet. If you see that the jobs have been launched, try increasing the slave setup timeout with a `set_distributed -slave_setup` command entry.

A less likely possibility is an error with the read/write image process. Prior to running distributed ATPG, write out the image file (for example, `write_image save.img -viol`), and then read the image in a new TetraMAX session (for example, `read_image save.img`). If you get an error message when you read the image in this new session, contact Synopsys Support with a testcase.

- Received "Error: At least one slave died" message. This message indicates that one of slave processors terminated in the middle of pattern generation. Possible causes are:

- a slave machine was rebooted or restarted
- a slave process was explicitly killed
- a slave process ran out of memory

Distributed ATPG Limitations

The following limitations are associated with distributed ATPG:

- The `-analyze_untestable_faults` option of the `set_atpg` command is not supported.
- N-detect ATPG and fault simulation are not supported.

25

Persistent Fault Model Support

TetraMAX ATPG supports a variety of fault models that abstractly represent real-world defects. Supported models include the stuck-at, transition, path delay, bridging, dynamic bridging, and IDDQ models. These models are implemented in a serial manner, which means that only one model is active at any time.

When you change fault models, TetraMAX ATPG flushes the current fault list from memory, along with any internal patterns, and starts again from scratch.

The single-fault model approach is inefficient in terms of pattern count and runtime. A set of transition patterns, for example, will also detect a certain number of stuck-at faults; but transition ATPG does not recognize them. Before you run stuck-at ATPG, you can reduce the stuck-at pattern count by fault-grading the stuck-at fault list against the transition patterns to prune previously detected faults. The stuck-at pattern reduction can be quite significant. Conversely, when you generate transition and stuck-at patterns in isolation, you waste time and patterns generating tests for stuck-at faults that were already detected by the transition patterns.

Persistent fault model support helps you manage multiple fault model flows easily by providing an automated way for TetraMAX ATPG to perform the following operations:

- [Persistent Fault Model Overview](#)
- [Direct Fault Crediting](#)
- [Persistent Fault Model Operations](#)
- [Example Commands Used in Persistent Fault Model Flow](#)

Persistent Fault Model Overview

The persistent fault model flow is enabled by the `set_faults -persistent_fault_models` command.

This flow enables the following behaviors:

- The fault list for the active fault model is saved in a cache. When you return to an inactive fault model, the saved faults are restored. When path delay faults are preserved, the delay paths are also preserved.
- The `report_faults -summary` command prints the total number of faults and the test coverage for each inactive fault model.
- All other fault-oriented commands continue only to affect the active fault model. This includes, but is not limited to, the following commands: `run_atpg`, `run_fault_sim`, `write_faults`, `read_faults`, `add_faults`, and `remove_faults`.
- The fault lists are preserved when you switch to DRC mode. You can't interact with the lists in DRC mode, but they will still be available when you return to TEST mode. ATPG untestable faults (AU) are automatically reset for any fault list that was in the cache during DRC mode.
- Faults detected in the transition fault model are credited as equivalent stuck-at detects without fault simulation. This is activated by the `update_faults -direct_credit` command.

See Also

[IDQ Testing](#)

Persistent Fault Model Operations

The following sections describe the primary processes associated with the persistent fault model flow:

- [Switching Fault Models](#)
- [Working With Internal Pattern Sets](#)
- [Manipulating Fault Lists](#)
- [Reporting Persistent Fault Models](#)

See Also

[Working with Fault Lists](#)

[What Are Fault Models?](#)

Switching Fault Models

You can set a different fault model in the persistent fault model flow, even if you have faults in the active fault model, as shown in [Example 1](#):

Example 1 Example Commands Used for Switching Fault Models

```
set_faults -persistent_fault_models
set_faults -model transition
add_faults -all
run_atpg -auto
(Transition faults exist)
set_faults -model bridging
```

Note: In this case, if you do not set the `-persistent_fault_models` option, TetraMAX ATPG will issue an M106 error.

Working With Internal Pattern Sets

The internal pattern set, and generated patterns in general, are preserved even if the fault model is changed in the persistent fault model flow. This means you can run fault simulation with an alternate fault model, as shown in [Example 2](#).

Example 2 Running Fault Simulation With an Alternative Fault Model

```
set_faults -persistent_fault_models
set_faults -model stuck
add_faults -all
set_faults -model transition
add_faults -all
run_atpg -auto
set_faults -model stuck
(Transition fault patterns are preserved as internal pattern set)
update_faults -direct_credit
run_fault_sim
```

If you need to change primary input (PI) constraints or the STL procedure file, you still need to return to DRC mode. After you are in DRC mode, the saved internal pattern set is deleted.

Manipulating Fault Lists

The following topics explain how to manipulate fault lists:

- [Automatically Saving Fault Lists](#)
- [Automatically Restoring Fault Lists](#)

- [Removing Fault Lists](#)
- [Adding Faults](#)

Automatically Saving Fault Lists

A fault list is automatically saved in the cache as an inactive fault model when a fault model is changed or when you go back to DRC mode. [Example 3](#) shows how to save fault lists automatically.

Example 3 Automatically Saving Fault Lists

```
set_faults -persistent_fault_models
add_pi_constraint 1 mem_bypass
run_drc compression.spf
set_faults -model stuck
add_faults -all
set_faults -model transition
(Stuck-at fault list is saved)
add_faults -all
run_atpg -auto
drc -f
(Transition fault list is saved)
remove_pi_constraints -all
add_pi_constraint 0 mem_bypass
run_drc compression.spf
```

Automatically Restoring Fault Lists

A fault list is automatically restored from the cache as an active fault model when a fault model is reactivated or before exiting DRC mode. See [Example 4](#).

Example 4 Automatically Restoring Fault Lists

```
set_faults -persistent_fault_models
add_pi_const 1 mem_bypass
run_drc compression.spf
set_faults -model stuck
add_faults -all
set_faults -model transition
(Stuck-at fault list is saved)
add_faults -all
run_atpg -auto
drc -f
(Transition fault list is saved)
remove_pi_constraints -all
add_pi_const 0 mem_bypass
run_drc compression.spf
(Transition fault list is restored)
run_atpg -auto
set_faults -model stuck
(Transition fault list is saved)
(Stuck-at fault list is restored)
```

```
update_faults -direct_credit
run_fault_sim
```

Note: The process of automatically restoring fault lists is equivalent to executing the command `read_faults fault.list -retain_code`. Therefore, when you return to DRC mode and change the STL procedure file, you might see some minor differences in the fault summaries obtained before DRC.

Removing Fault Lists

There are several different ways to remove fault lists:

- Use the command `remove_faults -all` to remove a fault list on an active fault model.
- Use the command `set_faults -nopersistent_fault_models` to delete all inactive fault lists from the cache.
- When you return to BUILD mode, all fault lists are automatically removed.

Adding Faults

There are some precautions you need to take when adding faults. Even when the command `set_faults -persistent_fault_models` is enabled, faults cannot be added when an internal pattern set is present. [Figure 1](#) shows the actual situations that are considered when adding faults.

Figure 1 Process For Adding Faults

Patterns Generated By:		Followed By:	
Fault Model	Process	Fault Model	Process
Transition	ATPG	Stuck-at	Direct fault credit
Any	ATPG	Any other	Fault simulation
Any	ATPG	Any other	ATPG

Note that the processes described in the “Followed by” column always require a fault list. However, if an internal pattern is present in any process in the “Patterns Generated By” column, you cannot add faults.

If you try to add faults to a different model when an internal pattern set is present, TetraMAX ATPG will issue an M104 error.

[Example 5](#) shows an example flow for adding faults.

Example 5 Typical Flow For Adding Faults

```

set_faults -persistent_fault_models
set_drc compression.spf
run_drc
set_faults -model stuck
add_faults -all
set_faults -model transition
add_faults -all
run_atpg -auto
set_faults -model stuck
(You can't add faults here because internal pattern set is
present)
update_faults -direct_credit
run_fault_sim

```

Reporting Persistent Fault Models

When the persistent fault model flow is enabled and inactive fault models are present, TetraMAX ATPG prints additional information when any of the following conditions exist:

- TetraMAX ATPG exits the DRC process
- The ATPG process is completed
- The `report_summaries` command is executed

[Example 6](#) shows an example of an Uncollapsed Stuck Fault Summary Report.

Example 6 Typical Uncollapsed Stuck Fault Summary Report

Uncollapsed Stuck Fault Summary Report

fault class	code	#faults
Detected	DT	2000576
detected_by_simulation	DS	(1610727)
detected_by_implementation	DI	(389849)
Possibly detected	PT	0
Undetectable	UD	1331
undetectable-unused	UU	(504)
undetectable-tied	UT	(491)
undetectable-blocked	UB	(295)
undetectable-redundant	UR	(41)
ATPG untestable	AU	18985
atpg_untestable-not_detected	AN	(18985)
Not detected	ND	13052
not-controlled	NC	(565)
not-observed	NO	(12487)
total faults		2033944
test coverage		98.42%

Inactive Fault Summary Report		
fault model	total faults	test coverage
Transition	1841908	96.46%

This report shows inactive faults list information. These numbers can be changed using the `set_faults -report [-collapsed | -uncollapsed]` command.

When you execute direct fault crediting using the `update_faults -direct_credit` command, you will see shorter reports that show how many faults are credited to DS, DI and NP. These faults are also generated by the `set_faults -report [-collapsed | -uncollapsed]` command. [Example 7](#) shows an example report using this command.

Example 7 Report Created Using the update_faults -direct_credit Command

```
update_faults -direct_credit
# 15597 stuck-at faults were changed to DS from the inactive
transition fault list.
# 0 stuck-at faults were changed to DI from the inactive
transition fault list.
# 0 stuck-at faults were changed to NP from the inactive
transition fault list.
```

Direct Fault Crediting

The persistent fault model flow supports direct fault crediting. To understand how this works, consider an example slow-to-rise (STR) transition fault. A pattern that detects this fault on a particular node must control that node from a 0 to a 1 and observe the result in a specified amount of time. To detect a stuck-at-0 (SA0) on the same node, only the 1 needs to be observed, and the timing is irrelevant. Thus, any slow-to-rise detection can be detected as a stuck-at-0 detection without actually simulating the transition patterns.

Direct fault crediting is enabled by running the `update_faults -direct_credit` command.

This command automatically reads back the transition fault list if it is in the cache, and it credits the following fault models:

- Dynamic bridging (victim only) faults to transition delay faults
- Dynamic bridging faults to static bridging faults
- Dynamic bridging (victim only), static bridging (victim only), and transition delay faults to stuck-at faults

[Example 1](#) shows a typical direct fault crediting flow.

Example 1 Typical Direct Fault Crediting Flow

```
set_faults -persistent_fault_models
set_faults -model bridging
```

```

add_faults -node_file nodes.txt
run_atpg -auto
set_faults -model transition
# (transition fault patterns are preserved as internal pattern
set)
add_faults -all
run_atpg -auto
set_faults -model bridging
# update_faults -direct_credit
# (Transition fault detections can't be credited to bridging
faults, so fault simulation is necessary)
run_fault_sim

```

[Example 2](#) shows an example log of applying direct credit with four fault models.

Example 2 Applying Direct Credit to Stuck-at Faults

```

set_faults -model stuck
81568 faults moved to the inactive bridging fault list.
419252 stuck faults moved to the active fault list.
3126 stuck AU faults were reset.
update_faults -direct_credit
112790 stuck faults were changed to DS from the inactive dynamic_
bridging fault list.
0 stuck faults were changed to DI from the inactive dynamic_
bridging fault list.
0 stuck faults were changed to NP from the inactive dynamic_
bridging fault list.
10121 stuck faults were changed to DS from the inactive bridging
fault list.
0 stuck faults were changed to DI from the inactive bridging fault
list.
0 stuck faults were changed to NP from the inactive bridging fault
list.
203578 stuck faults were changed to DS from the inactive
transition fault list.
0 stuck faults were changed to DI from the inactive transition
fault list.
0 stuck faults were changed to NP from the inactive transition
fault list.

```

[Table 1](#) describes the direct fault crediting process.

Table 1 Direct Fault Crediting Process

Transition Fault Status	Existing Stuck-at Fault Status	Updated Stuck-at Fault Status
DS	Not DS	DS
DI	Not DS	DI

Table 1 Direct Fault Crediting Process (Continued)

Transition Fault Status	Existing Stuck-at Fault Status	Updated Stuck-at Fault Status
TP (small delay defect)	Not DS	DS
AP	Not DT or AP	NP
NP	Not DT or AP	NP

If the `-persistent_fault_models` option is not enabled, you can apply direct crediting to stuck-at faults by using `-external` option if you have transition fault list. [Example 3](#) is a script example that uses this method:

Example 3 Script Example Using the -external Option

```
run_drc compression.spf
set_faults -model stuck
add_faults -all
update_faults -direct_credit -external transition.flt
run_atpg -auto
```

See Also

[Fault Categories and Classes](#)

Example Commands Used in Persistent Fault Model Flow

The following example shows the commands used in a typical persistent fault model flow:

```
read_netlist des_unit
run_build_model des_unit
set_delay -launch system_clock
#
#Activate persistent fault model feature
#
set_faults -persistent_fault_models
#
# Model=Transition memory bypass=No OCC=Yes
#
add_pi constraints 0 memory_bypass
run_drc des_unit.spf -patternexec comp
set_faults -model stuck
add_faults -all
set_fault -model transition
```

```
add_faults -all
run_atpg -auto
write_patterns trans_bp0_occ1.bin -format binary
set_faults -model stuck
#
# Credit transition detections to stuck-at faults.
#
update_faults -direct_credit
#
# Optional step to increase fault coverage
# run_fault_sim
#
drc -force
remove_pi_constraints -all
remove_clocks -all
#
# Model=Stuck-at memory bypass=Yes OCC=No
#
add_pi_constraints 1 memory_bypass
run_drc des_unit.spf -patternexec comp_occ_bypass
set_fault -model stuck
run_atpg -auto
write_patterns stuck_bp1_occ0.bin -format binary
drc -force
remove_pi_constraints -all
remove_clocks -all
#
# Model=Stuck-at memory bypass=No OCC=No
#
add_pi_constraints 0 memory_bypass_mode
run_drc des_unit.spf -patternexec comp_occ_bypass
set_faults -model stuck
run_atpg -auto
write_patterns stuck_bp0_occ0.bin -format binary
```

26

Using TetraMAX and DFTMAX Ultra Compression

DFTMAX Ultra compression is an advanced test compression technology that delivers the optimal quality of results as measured by test time, data volume, design area, congestion, and time to implementation.

TetraMAX ATPG has built-in knowledge of DFTMAX Ultra compression and its pattern decompression and compression technology. Using a design netlist and a STIL procedure file, TetraMAX ATPG generates a set of test patterns specifically intended for the DFTMAX Ultra test mode.

TetraMAX ATPG is the only ATPG tool that supports pattern generation for designs using DFTMAX Ultra compression. The TetraMAX version must be the same as or later than the DFTMAX Ultra version used for the design.

The following sections describe how to use TetraMAX ATPG with DFTMAX Ultra compression:

- [Generating Patterns for DFTMAX Ultra Designs](#)
- [Manipulating Patterns for DFTMAX Ultra](#)
- [High Resolution Pattern Flow for DFTMAX Ultra Chain Diagnostics](#)
- [Test Validation and VCS Simulation for DFTMAX Ultra Designs](#)
- [Limitations for Using DFTMAX Ultra](#)

26 Generating Patterns for DFTMAX Ultra Designs

The process for generating patterns for DFTMAX Ultra designs is similar to the standard TetraMAX ATPG flow described in the "[Basic ATPG Design Flow](#)" section. You must use either serial STIL or parallel STIL patterns generated by TetraMAX ATPG. DFTMAX Ultra compression does not accept any other pattern format.

The following sections describe how to generate patterns specifically for a DFTMAX Ultra design:

- [Pattern Types Required by DFTMAX Ultra](#)
 - [Script Example for Generating Patterns for DFTMAX Ultra](#)
-

Pattern Types Required by DFTMAX Ultra

TetraMAX ATPG generates two types of STIL test patterns that can be used by DFTMAX Ultra compression:

- **Serial STIL**— These patterns are used for both scan testing and simulation of full scan testing. The test patterns are applied to the device for testing on the ATE and are used for simulating the entire test procedure, including serial scan-in data, decompression of the scan-in data, launch and capture, compression of the scan-out data, and serial scan-out data.
- **Parallel STIL**— These patterns are used for fast simulation of the launch and capture phases of scan testing. The decompressed test patterns are loaded directly into the scan chains in parallel and bypasses the serial scan-in and scan-out parts of the simulation.

You can write serial or parallel STIL patterns with or without the unified STIL flow. However, to use the unified STIL flow, you must explicitly specify the `-unified_stil_flow` option. The following example writes STIL patterns using the unified STIL flow:

```
TEST-T> write_patterns patterns.stil -format stil \
      -unified_stil_flow
```

For details on generating serial and parallel STIL patterns, see the "[Writing ATPG Patterns](#)" section.

After you simulate and validate the results of the test procedure using several test patterns, you can skip these patterns in future runs. You can then selectively simulate the launch and capture segments using additional test patterns loaded in parallel.

Script Example for Generating Patterns for DFTMAX Ultra

The following script is an example of a TetraMAX pattern generation session for a chip that uses DFTMAX Ultra compression:

```
##### USER INPUTS AND DFTMAX ULTRA OUTPUT FILES #####
set TOP_MODULE_NAME top_module_name
set NETLIST_FILES1 netlist_files1
set NETLIST_FILES2 netlist_files2
set LIBRARY_FILES1 library_files1
set LIBRARY_FILES2 library_files2
set BUILD_CONSTRAINTS_FILE build_constraints_file
set DRC_CONSTRAINTS_FILE drc_constraints_file
set STL_procedure_file spf_file
set LOG log_file
setenv SYNOPSYS path_to_tool_installation

##### BUILD SETTINGS #####
set_messages -level expert -log $LOG -replace
report_version -full
build -force
set_faults -pt_credit 0
set_faults -summary verbose
set_rules N2 warning
set_rules B12 warning
set_rules B5 warning
set_faults -atpg_effectiveness
set_atpg -verbose
set_netlist -redefined_module last
read_netlist $NETLIST_FILES1
read_netlist $NETLIST_FILES2
read_netlist $LIBRARY_FILES1 -library
read_netlist $LIBRARY_FILES2 -library
source -echo $BUILD_CONSTRAINTS_FILE
run_build_model $TOP_MODULE_NAME

##### DRC SETTINGS #####
source -echo $DRC_CONSTRAINTS_FILE
set_faults -model stuck
run_drc $STL_procedure_file

##### RUN ATPG #####
add_nofaults -module .*COMPRESSOR.*
add_faults -all
run_atpg -auto_compression
```

```
run_simulation -remove_padding_patterns  
write_patterns ultra_ser.stil -serial -format stil  
write_patterns ultra_par.stil -parallel -format stil -nounified_  
stil_flow
```

Manipulating Patterns for DFTMAX Ultra

You can use the `update_streaming_patterns` command to modify or remove ATPG-generated patterns for use in DFTMAX Ultra compression. In some cases, the order in which you specify this command depends on whether you are using internal or external patterns.

The following topics describe how to use the `update_streaming_patterns` command to manipulate ATPG-generated patterns:

- [Controlling the Peak and Average Power During Shifting](#)
 - [Increasing the Maximum Shift Length of Patterns](#)
 - [Optimizing Padding Patterns](#)
 - [Removing and Reordering Patterns](#)
-

Controlling the Peak and Average Power During Shifting

You can use the `-load_scan_in` option of the `update_streaming_patterns` command to control the peak and average power during shifting. This option enables you to specify certain scan-in pins to maintain a constant value during the shift operations. For example, if you specify a value of 0 for the test1 scan-in pin, the pattern is modified so that all test1 pins maintain a constant 0 value during load shifting.

You can specify values for as many scan-in pins as required using the Tcl list syntax. The `-load_scan_in` option reduces overall power consumption during shifting, and it can be used for both internal and external patterns. However, this option also causes some coverage loss and simulation mismatches might occur if you specify scan-in pins connected to OCC chains.

The following example modifies internal patterns for the `test_si1` and `test_si3` pins after running ATPG:

```
TEST-T> run_atpg -auto
TEST-T> update_streaming_patterns -load_scan_in \
    {test_si1 0 test_si3 1}
```

The next example updates external patterns created during ATPG with the specified values of the `test_si4` and `test_si7` scan-in pins:

```
TEST-T> set_patterns -external pat.stil
TEST-T> update_streaming_patterns -load_scan_in \
    {test_si4 1 test_si7 1}
```

Increasing the Maximum Shift Length of Patterns

You can use the `-max_shifts` option of the `update_streaming_patterns` command to specify the maximum shift length, which enables you to increase the size of internal and external patterns from the optimal value set by TetraMAX ATPG. You can use this option before an

ATPG run so the generated patterns use the specified shift length or you can apply it to external patterns. The `-max_shifts` option prevents overshifting and makes the pattern shift lengths equal across different blocks, which assures correct pattern porting.

You can apply this option in an initial session to increase the shift length of the patterns, then use these same patterns in another session by writing and reading them back again. For subsequent sessions, make sure you set the pattern shift length to the same value set in the previous session. Otherwise, you will see errors and simulation mismatches.

The following example uses the `-max_shifts` option before an initial ATPG run:

```
read_image design.img
update_streaming_patterns -max_shifts 300
run_atpg -auto
write_patterns pat.stil -format stil -serial -replace
```

The following example applies the `-max_shifts` option in a second session using external patterns:

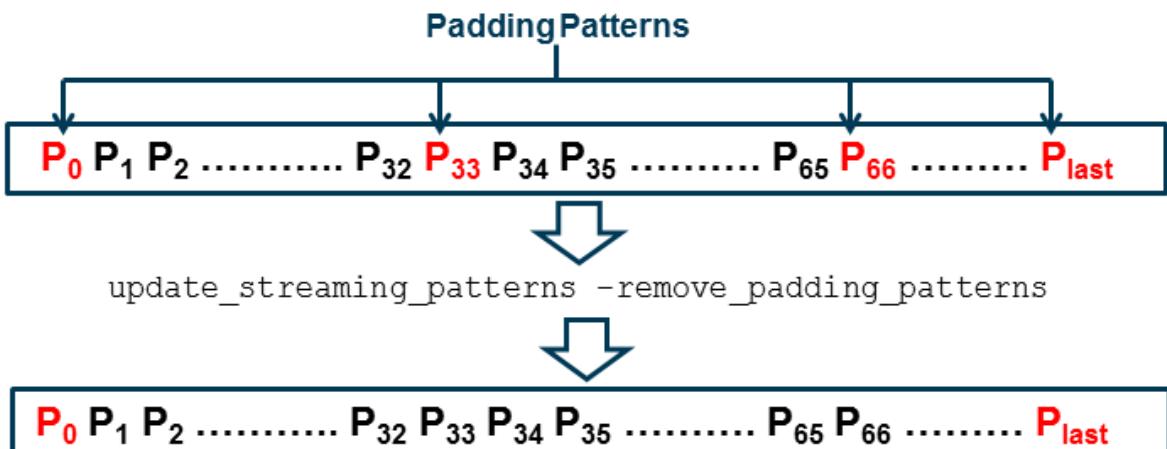
```
read_image design.img
update_streaming_patterns -max_shifts 320
set_patterns -external pat.stil
```

Optimizing Padding Patterns

When you use DFTMAX Ultra compression technology, the MUX control bits for each test pattern are loaded by the previous pattern. During ATPG, patterns are created in groups. For the first pattern in a group, TetraMAX ATPG prepends a padding pattern that loads the required MUX control bits. During the later stages of pattern generation, TetraMAX ATPG searches for patterns that do not incrementally detect new faults and removes those patterns from the pattern set. This process also introduces padding patterns.

You can use the `-remove_padding_patterns` option of the `update_streaming_patterns` command to optimize padding patterns by removing all padding patterns except for the first and last padding pattern.

Figure 1 Optimizing Padding Patterns



Performing Padding Pattern Optimization

The following commands optimize padding patterns for an internal pattern set generated by ATPG:

```
run_atpg ...
update_streaming_patterns -remove_padding_patterns
write_patterns ...
```

The following commands optimize padding patterns for an external pattern set:

```
set_patterns -external stil_file_name
update_streaming_patterns -remove_padding_patterns
write_patterns ...
```

Removing and Reordering Patterns

Use the `-remove` and `-insert` options of the `update_streaming_patterns` command to remove and reorder individual patterns and blocks of patterns.

To remove one or more patterns, specify a list using the `-remove` option of the `update_streaming_patterns` command.

The following command removes patterns 3 and 9:

```
update_streaming_patterns -remove {3 9}
```

You can specify blocks of patterns by providing the first and last pattern numbers of the block as a sublist inside the pattern removal list. You can mix individual pattern numbers and pattern blocks in the list.

The following command removes patterns 5 through 7:

```
update_streaming_patterns -remove {{5 7}}
```

The following command removes patterns 3, 5 through 7, and 9:

```
update_streaming_patterns -remove {3 {5 7} 9}
```

To remove patterns and reinsert them at a different location in the pattern set, use both the `-insert` and `-remove` options of the `update_streaming_patterns` command. For each pattern number or block provided in the removal list, specify the pattern number at which the removed patterns should be reinserted in the insertion list. For removed patterns that you do not want reinserted, specify an insertion pattern value of X. Note that all removal pattern numbers are pre-manipulation values.

The following command removes pattern 3, and reinserts patterns 4 through 6 at pattern 0:

```
update_streaming_patterns -remove {3 {4 6}} -insert {X 0}
```

You can issue multiple the `update_streaming_patterns` command using the `-remove` and `-insert` options. Each command resequences the patterns and pattern numbers after

performing the specified pattern manipulations. Subsequent `update_streaming_patterns` commands must refer to the resequenced pattern numbers.

You can reorder chain test patterns, stuck-at patterns, and transition fault patterns. However, you cannot perform reordering operations that mix these pattern types. If you mix pattern types, an error is reported, as shown in the following example:

```
Error: Pattern 2 and 13 are not of same type. Reordering is
possible between patterns of same type only
Transition patterns are 6 to 99
Chain test patterns are 1 to 5
```

High Resolution Pattern Flow for DFTMAX Ultra Chain Diagnostics

You can use the standard TetraMAX chain diagnostics flow for DFTMAX Ultra chain failures. If a failing part has multiple chain defects, an additional flow is available to create high resolution patterns that can more accurately identify failing scan cells when there are multiple failing scan chains. There are four basic steps to this flow:

1. Use chain diagnostics to identify defective scan chains
 2. Create high resolution patterns for the defective scan chains
 3. Retest the failing part with the high resolution patterns
 4. Rerun diagnostics using the high resolution patterns
-

Identifying Defective Chains

Your first step is to perform an initial diagnostics run to identify a set of defective chains based on the chain test failures reported in the failure log file. To do this, specify the `-streaming_report_chains_only` option of the `run_diagnosis` command, as shown in the following example:

```
run_diagnosis failure_log_file.log -streaming_report_chains_only
chain_fail_report.txt
```

Generating High Resolution Patterns

After you have identified the defective chains, generate a set of high resolution patterns that identify the failing flip-flops in the defective chains.

To do this, apply the `add_chains_masks` command to the entire production pattern set (including the chain test patterns and other logic patterns). You then use these patterns to

generate a new set of failure log files that are used to identify the defective flip-flops. The following example shows this process:

```
set_patterns -external full_pattern_set.stil  
add_chain_masks -filename chain_fail_file.txt -diagnosis -external  
write_patterns high_resolution_set.stil -format stil -external
```

Rerunning Diagnostics Using the High Resolution Patterns

Using the newly generated high resolution patterns, you need to retest the failing part and collect the new fail data. You can then rerun diagnostics using the failure log file generated from the high resolution patterns. You don't need to use any specific options in the run_diagnosis command for DFTMAX Ultra compression designs, as shown in the following example:

```
run_diagnosis high_res_pat_failure_log_file.log -verbose
```

Flow Example

```
read_image image_file.dat  
  
# Step 1  
## Run diagnostics to identify defective chains  
run_diagnosis high_res_pat_failure_log_file.log -streaming_report_  
chains_only chain_fail_list  
set_patterns -delete  
  
# Step 2  
## Read full pattern test file  
set_patterns -external full_pattern_set.stil  
  
## Use add_chain_masks command to generate high resolution  
patterns  
add_chain_masks -external -filename chain_fail_list -diagnosis  
  
## Write the patterns  
write_patterns high_resolution_set.stil -format stil -external  
  
# Step 3  
## Retest the failing part with the high resolution patterns and  
collect the fail data  
  
# Step 4  
set_patterns -external high_resolution_set.stil  
  
## Main diagnosis run using log file generated using high  
resolution patterns  
run_diagnosis high_res_pat_failure_log_file.log -verbose
```

See Also

[Generating Patterns for DFTMAX Ultra Designs](#)
[Optimizing Padding Patterns](#)
[VCS Simulation for DFTMAX Ultra Designs](#)

Test Validation and VCS Simulation for DFTMAX Ultra Designs

You can perform test pattern validation for a DFTMAX Ultra design using MAX Testbench and then run a VCS simulation to validate the test protocol and test patterns.

For more information, see the "Using MAX Testbench" in the *Test Pattern Validation User Guide*.

The validation process for a DFTMAX Ultra design uses a serial STIL file or a parallel STIL file.

Limitations for Using DFTMAX Ultra

The following ATPG requirements and limitations apply to DFTMAX Ultra compression:

- Path delay fault testing is supported only for fast-sequential ATPG.
- The `write_patterns` command normally writes out a unified STIL file by default, which uses a single STIL file for both serial and parallel simulation. You can perform serial or parallel simulation using the unified STIL flow. However, to use this flow for DFTMAX Ultra designs, you must explicitly specify the `-unified_stil_flow` option to write out the STIL pattern files.
- Parallel test validation using NShift is not supported. However, NShift can still be validated using serial test patterns. NShift is a feature that simulates the last “N” shifts of the scan operation so that nonscan flip-flops in the design are initialized to known values.
- The `run_justification` command is not supported.
- The following options of the `set_drc` command are not supported:
 - `-lockup_after_compressor` | `-nolockup_after_compressor`
 - `-pipeline_in_compressor` | `-nopipeline_in_compressor`
- The `-per_pin_limit` option of the `set_diagnosis` command is not supported.

27

Troubleshooting

The following sections describe troubleshooting tips and techniques:

- [Reporting Port Names](#)
- [Reviewing a Module Representation](#)
- [Rerunning Design Rule Checking](#)
- [Troubleshooting Netlists](#)
- [Troubleshooting STIL Procedures](#)
- [Analyzing the Cause of Low Test Coverage](#)
- [Completing an Aborted Bus Analysis](#)

Reporting Port Names

To verify the names of top-level ports, you can obtain a list of the inputs, outputs, or bidirectional ports for the top level of the design using these commands:

```
DRC-T> report_primitives -pis
DRC-T> report_primitives -pos
DRC-T> report_primitives -pios
DRC-T> report_primitives -ports
```

To obtain the names of ports for any specific module, use the following command:

```
DRC-T> report_modules module_name -verbose
```

[Example 1](#) shows a verbose report produced by the `report_modules` command. The names of the pins are listed in the Inputs and Outputs sections.

Example 1 Verbose Module Report

module name	tot(i/ o/ io)	inst	refs(def'd)	used	pins
INC4	11(5/ 6/ 0)	10	1	(Y)	1
Inputs	:	A0 () A1 () A2 () A3 () CI ()			
Outputs	:	S0 () S1 () S2 () S3 () CO () PR ()			
PROP1	:	and conn=(O:PROP I:A0 I:A1 I:A2 I:A3)			
HADD0S	:	xor conn=(O:S0 I:A0 I:CI)			
HADD1S	:	xor conn=(O:S1 I:A1 I:C0)			
HADD2S	:	xor conn=(O:S2 I:A2 I:C1)			
HADD3S	:	xor conn=(O:S3 I:A3 I:C2)			
HADD0C	:	and conn=(O:C0 I:A0 I:CI)			
HADD1C	:	and conn=(O:C1 I:A1 I:C0)			
HADD2C	:	and conn=(O:C2 I:A2 I:C1)			
CARRYOUT	:	and conn=(O:CO I:PROP I:CI)			
buf9	:	buf conn=(O:PR I:PROP)			

Reviewing a Module Representation

To review the internal representation of a module definition, you will need to specify the `report_modules` command with the name of the module and the `-verbose` option. Alternatively, you can use the `run_build_model` command and specify the name of the module as the top-level design.

You might want to review the internal representation of a library module in TetraMAX ATPG if errors or warnings are generated by the `read_netlist` command. For example, suppose that you use the `read_netlist` command to read in the module `csdff`, whose truth table definition is shown in [Example 1](#), and the command generates the warning messages shown in [Example 2](#).

Example 1 Truth Table Logic Model

```

primitive csdff (Q, SDI, SCLK, D, CLK, NOTIFY);
  output Q; reg Q;
  input SDI, SCLK, D, CLK, NOTIFY;
  table
    // SDI SCLK D  CLK  NR   : Q-  : Q+
    // --- --- --- --- : --- : --- : ---
    ? 0 0 (01) ? : ? : 0 ; // clock D=0
    ? 0 1 (01) ? : ? : 1 ; // clock D=1
    0 (01) ? 0 ? : ? : 0 ; // scan clock SDI=0
    1 (01) ? 0 ? : ? : 1 ; // scan clock SDI=1

    ? 0 * 0 ? : ? : - ; // hold
    * 0 ? 0 ? : ? : - ;
    ? 0 ? 0 ? : ? : - ;
    ? 0 ? (?0) ? : ? : - ;
    ? (?0) ? 0 ? : ? : - ;
    ? 0 ? ? * : ? : x ; // force to X
  endtable
endprimitive

```

Example 2 Read Netlist Showing Warnings

```

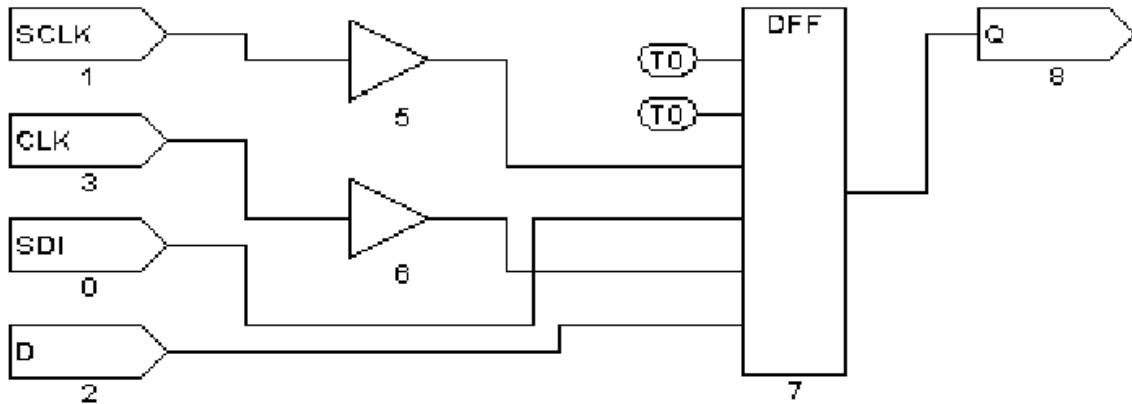
BUILD-T> read_netlist csdff.v
Begin reading netlist ( csdff.v )...
Warning: Rule N15 (incomplete UDP) failed 64 times.
Warning: Rule N20 (underspecified UDP) failed 2 times.
End parsing Verilog file test.v with 0 errors;
End reading netlist: #modules=1, top=csdff, #lines=25,
CPU_time=0.01 sec

```

To review the model:

- Execute the `run_build` command:
- 1. BUILD-T> **run_build_model csdff**
- 2. Click the SHOW button in the graphical schematic viewer (GSV) toolbar, and from the pop-up menu, choose ALL.

A schematic similar to [Figure 1](#) appears, allowing you to examine the ATPG model.

Figure 1 Module Showing Correct Interpretation

Do not be concerned if the schematic shows extra buffers. During the model building process, TetraMAX ATPG inserts these buffers wherever there is a direct path to a sequential device from a top-level port. These buffers are not present in instantiations of the module in the design.

Rerunning Design Rule Checking

The file specified in the `run_drc` command is read each time the design rule checking (DRC) process is initiated. You can quickly test any changes that you make to this file by issuing another `run_drc` command, as follows:

```
DRC-T> run_drc specfile.spf
# pause here for edits to DRC file
TEST-T> drc -force
DRC-T> run_drc
```

Troubleshooting Netlists

The following tips are for troubleshooting problems TetraMAX ATPG might encounter while reading netlists:

- For severe syntax problems, start troubleshooting near the line number indicated by the TetraMAX error message.
- Focus on category N rules; these cover problems with netlists.
- To see the number of failures in category N, execute the `report_rules n -fail` command.
- To see all violations in a specific category such as N9, execute the `report_violations n9` command.
- To see violations in the entire category N, execute the `report_violations n`

command.

- Netlist parsing stops when TetraMAX ATPG encounters 10 errors. To increase this limit, execute the `set_netlist -max_errors` command.
- When reading multiple netlist files using wildcards in the `read_netlist` command, to determine which file had a problem, reread the files with the `-verbose` option and omit the `-noabort` option.
- Extract the problematic module definition, save it in a file, and attempt to read in only that file.
- Consider the effect of case sensitivity on your netlist, and explicitly set the case sensitivity by using the `-sensitive` or `-insensitive` option with the `read_netlist` command.
- Consider the effect of the hierarchical delimiter. If necessary, change the default by using the `-hierarchy_delimiter` option of the `set_build` command. Then reread your netlists.

Troubleshooting STIL Procedures

Problems in the procedures defined in the STIL procedure file can be either syntax errors or DRC violations. Syntax errors usually result in a category V (vector rule) violation message, and TetraMAX ATPG reports the line number near the violation.

The following sections describe how to troubleshoot STIL procedures:

- [Opening the STIL Procedure File](#)
- [STIL load_unload Procedure](#)
- [STIL Shift Procedure](#)
- [STIL test_setup Macro](#)
- [Correcting DRC Violations by Changing the Design](#)

Opening the STIL Procedure File

To fix the problem, open the STIL procedure file with an editor, make any necessary changes, and use the `run_drc` command again to verify that the problem was corrected. For detailed descriptions and examples of the STIL procedures, see [“STIL Procedure Files”](#).

A general tip for troubleshooting any of the STIL procedure file procedures is to click the ANALYZE button in the GSV toolbar and select the applicable rule violation from the Analyze dialog box. TetraMAX ATPG draws the gates involved in the violation and automatically selects an appropriate pin data format for display in the schematic. To specify a particular pin data format, click the SETUP button and select the Pin Data Type in the Setup dialog box. For more information on pin data types, see [“Displaying Pin Data”](#).

STIL load_unload Procedure

When you analyze DRC violations TetraMAX ATPG encountered during the `load_unload` procedure, the GSV automatically sets the pin data type to `Load`. With the `Load` pin data type, strings of characters such as `10X11{ }11` are displayed near the pins. Each character corresponds to a simulated event time from the vectors defined in the `load_unload` procedure. The curly braces indicate where the `Shift` procedure is inserted as many times as necessary. Thus, the last value before the left curly brace is the logic value achieved just before starting the `Shift` procedure. The values following the right curly brace are the simulated logic values between the last `Shift` procedure and the end of the `load_unload` procedure.

The following guidelines are for using the `load_unload` procedure:

- Set all clocks to their off states before the `Shift` procedure.
- Enable the scan chain path by asserting a control port (for example, `scan_enable`).
- Place any bidirectional ports that operate as scan chain inputs into input mode.
- Place any bidirectional or three-state ports that operate as scan chain outputs into output mode, and explicitly force the ports to Z.
- Set all constrained ports to values that enable shifting of scan chains.
- Place all bidirectional ports into a non-latching input mode if this is possible for the design.

STIL Shift Procedure

When you analyze DRC violations encountered during the `Shift` procedure, the GSV automatically sets the displayed pin data type to `Shift`. In the `Shift` pin data type, logic values such as `010` are displayed. Each character represents a simulated event time in the `Shift` procedure defined in the STIL procedure file.

The following guidelines are for the test cycles you define in the `Shift` procedure:

- Use the predefined symbolic names `_si` and `_so` to indicate where scan inputs are changed and scan outputs are measured.
- If you want to save patterns in Waveform Generation Language (WGL) format, describe the `Shift` procedure using a single cycle.
- Remember that state assignments in STIL are persistent for a multicycle `Shift` procedure. Therefore, when you place a `CLOCK=P` to cause a pulse, that setting continues to cause a pulse until `CLOCK` is turned off (`CLOCK=0` for a return-to-zero port, or `CLOCK=1` for a return-to-one port).

[Example 1](#) shows a `Shift` procedure that contains an error. The first cycle of the shift applies `MCLK=P`, which is still in effect for the second cycle. As the `Shift` procedure is repeated, both `MCLK` and `SCLK` become set to P, which unintentionally causes a pulse on each clock on each cycle of the `Shift` procedure.

Example 1 Multicycle Shift Procedure With a Clocking Error

```
"load_unload" {
    V { MCLK      = 0; SCLK      = 0; SCAN_EN     = 1; }
```

```

Shift {
  V { _si=##; _so=##; MCLK=P; }
  V { SCLK=P; }    // PROBLEM: MCLK is still on!
}
}

```

[Example 1](#) shows the same `Shift` procedure with correct clocking. As the `Shift` procedure is interactively applied, `MCLK` and `SCLK` are applied in separate cycles. An additional `SCLK=0` has been added after the `Shift` procedure, before exiting the `load_unload`, to ensure that `SCLK` is off.

Example 1 Multicycle Shift Procedure With Correct Clocking

```

"load_unload" {
  V {
    MCLK = 0; SCLK = 0; SCAN_EN = 1;
  }
  Shift {
    V { _si=##; _so=##; MCLK=P; SCLK=0; }
    V {                   MCLK=0; SCLK=P; }
  }
  V { SCLK=0; }
}

```

[Example 2](#) shows the same `Shift` procedure converted to a single cycle. The procedure assumes that timing definitions elsewhere in the test procedure file for `MCLK` and `SCLK` are adjusted so that both clocks can be applied in a non-overlapping fashion. Thus, the two clock events can be combined into the same test cycle.

Example 2 Multicycle Shift Converted to a Single Cycle

```

"load_unload" {
  W "TIMING";
  V { MCLK = 0; SCLK = 0;; SCAN_EN = 1; }
  Shift {
    V { _si=##; _so=##; MCLK=P; SCLK=P; }
  }
  V { MCLK=0; SCLK=0; }
}

```

STIL test_setup Macro

When you analyze DRC violations encountered during the `test_setup` macro, the graphical schematic viewer automatically sets the displayed pin data type to Test Setup. In the Test Setup pin data type, logic values in the form `XX1` are displayed. Each character represents a simulated event time in the `test_setup` macro defined in the STL procedure file.

The following rules are for the test cycles you define in the `test_setup` macro:

- Force bidirectional ports to a Z state to avoid contention.
- Initialize any constrained primary inputs to their constrained values by the end of the procedure.
- Pulse asynchronous set/reset ports or clocking in a synchronous set/reset only if you want to initialize specific nonscan circuitry.
- Place clocks and asynchronous sets and resets at their off states by the end of the procedure. Note that it is not necessary to stop Reference clocks (including what DFT Compiler refers to as ATE clocks). All other clocks still must be stopped.

Correcting DRC Violations by Changing the Design

If you cannot correct a DRC violation by adjusting one of the STL procedure file procedures, defining a primary input constraint, or changing a clock definition, the violation is probably caused by incorrect implementation of ATPG design practices, and a design change might be necessary. Note that a design can be testable with functional patterns and still be untestable by ATPG methods.

If you have scan chains with blockages and you cannot determine the right combination of primary input constraints, clocks, and STL procedure file procedures, the problem might involve an uncontrolled clock path or asynchronous reset. Try dropping the scan chain from the list of known scan chains. This will increase the number of nonscan cells and decrease the achievable test coverage, but it might let you generate ATPG patterns without a design change.

If you still cannot correct the violation, you must make a design change. Examine the design along with the design guidelines presented in the ["Working With Design Netlists and Libraries"](#) section to determine how to change your design to correct the violation.

Analyzing the Cause of Low Test Coverage

When test coverage is lower than expected, you should review the AN (ATPG untestable), ND (not detected), and PT (possibly detected) faults, and refer to the following sections:

- [Where Are the Faults Located?](#)
- [Why are the Faults Untestable or Difficult to Test?](#)
- [Using Justification](#)

Where Are the Faults Located?

To find out where the faults are located, choose Faults > Report Faults to access the Report Faults window, which displays a report in a separate window. Alternatively, you can use the `report_faults` command with the `-class` and `-level` options.

The following command generates a report of modules that have 256 or more AN faults:

```
TEST-T> report_faults -class an -level 4 256
```

[Example 1](#) shows the report generated by this command. The first column shows the number of AN faults for each block. The second column shows the test coverage achieved in each block. The third column shows the block names, organized hierarchically from the top level downward.

Example 1 Fault Report of AN Faults Using the Level Option

```
TEST-T> report_faults -class AN -level 4 256
#faults  testcov  instance name (type)
-----  -----
 22197  91.70%  /spec_asic  (top_module)
 2630   83.00%  /spec_asic/born (born)
 2435   28.00%  /spec_asic/born/fpga2 (fpga2)
   788   5.35%   /spec_asic/born/fpga2/avge1 (avge)
 1647   3.28%   /spec_asic/born/fpga2/avge2 (yavge)
 5226   0.00%   /spec_asic/dac (dac)
 5214   0.00%   /spec_asic/dac/dual_port (dual_port)
11098   66.46%  /spec_asic/video (video)
11098   66.24%  /spec_asic/video/decipher (vdp_cyphr)
11027   60.00%  /spec_asic/video/decipher/dpreg (dpreg)

 426   96.97%  /spec_asic/gex (gex)
 260   93.89%  /spec_asic/gex/fifo (gex_fifo)
 799   94.56%  /spec_asic/vint (vint)
 798   54.29%  /spec_asic/vint/vclk_mux (vclk_mux)
1514   94.80%  /spec_asic/crtc_1 (crtc)
 476   96.79%  /spec_asic/crtc/crtc_sub (crtc_sub)
 465   94.20%  /spec_asic/crtc/crtc_sub/attr (attr)
1004   77.68%  /spec_asic/crtc/crap (crap)
```

The report shows that the two major contributors to the high number of AN faults are the following hierarchical blocks:

- /spec_asic/dac/dual_port (with 5,214 AN faults and 0.00 percent test coverage)
- /spec_asic/video/decipher/dpreg (with 11,027 faults and 60.00 percent test coverage)

You can also review other classes of faults and combinations of classes of faults by using different option settings in the `report_faults` command.

Why Are the Faults Untestable or Difficult to Test?

To find out why the faults cannot be tested, you can use the `analyze_faults` command or the `run_justification` command.

The following example uses the `analyze_faults` command to generate a fault analysis summary for AN faults:

```
TEST-T> analyze_faults -class an
```

[Example 2](#) shows the resulting fault analysis summary, which lists the common causes of AN faults. In this example, the three major causes are constraints that interfered with testing (7,625

faults), blockages as a secondary condition of constraints (5,046 faults), and faults downstream from points tied to X (1,500 faults). As with the `report_faults` command, you can specify other classes of faults or multiple classes.

Example 2 Fault Analysis Summary of AN Faults

```
TEST-T> analyze_faults -class an
Fault analysis summary: #analyzed=13398, #unexplained=257.
7625 faults are untestable due to constrain values.
5046 faults are untestable due to constrain value blockage.
11 faults are connected to CLKPO.
11 faults are connected to DSLAVE.
210 faults are connected to TIEX.
233 faults are connected to TLA.
129 faults are connected to CLOCK.
50 faults are connected to TS_ENABLE.
26 faults are connected from CLOCK.
128 faults are connected from TLA.
1500 faults are connected from TIEX.
114 faults are connected from CAPTURE_CHANGE.
```

To see specific faults associated with each classification cause (for example, to see a specific fault connected from TIEX), use the `-verbose` option with the `analyze_faults` command.

The following command generates an AN fault analysis report that gives details of the first three faults in each cause category:

```
TEST-T> analyze_faults -class an -verbose -max 3
```

You can redirect this report to a file by using the output redirection option:

```
TEST-T> analyze_faults -class an -verb > an_faults_detail.txt
```

You can examine each fault in detail by using the `analyze_faults` command and naming the specific fault. For example, the following command generates a report on a stuck-at-0 fault on the module /gcc/hclk/U864/B:

```
TEST-T> analyze_faults /gcc/hclk/U864/B -stuck 0
```

[Example 3](#) shows the result of this command. The report lists the fault location, the assigned fault classification, one or more reasons for the fault classification, and additional information about the source or control point involved.

Example 3 Fault Analysis Report of a Specific Fault

```
Fault analysis performed for /gcc/hclk/U864/B stuck at 0 \
  (input 2 of MUX gate 58328).
Current fault classification = AN \
  (atpg_untestable-not_detected).
```

```
Connection data: to=DSLAVE
Fault is blocked from detection due to constrained values.
  Blockage point is gate /gcc/hclk/writedata_reg0 (91579).
```

For additional examples, see "[Example: Analyzing an AN Fault](#)".

Using Justification

The `run_justification` command provides another troubleshooting tool. Use it to determine whether one or more internal points in the design can be set to specific values. This analysis can be performed with or without the effects of user-defined or ATPG constraints.

If there is a specific fault that shows up in an NC (not controlled) class, you can use the `run_justification` command to determine which of the following conditions applies to the fault:

- The fault location can be identified as controllable if TetraMAX ATPG is given more CPU time or a higher abort limit and allowed to continue.
- The fault location is uncontrollable.

In [Example 4](#), the `run_justification` command is used to confirm that an internal point can be set to both a high and low value.

Example 4 Using run_justification

```
TEST-T> run_justification -set /spec_asic/gex/hclk/U864/B 0
Successful justification: pattern values available in pattern 0.
Warning: 1 patterns rejected due to 127 bus contentions (ID=37039,
pat1=0) . (M181)

TEST-T> run_justification -set /spec_asic/gex/hclk/U864/B 1
Successful justification: pattern values available in pattern 0.
Warning: 1 patterns rejected due to 127 bus contentions (ID=37039,
pat1=0) . (M181)
```

For additional examples of the `run_justification` command, see [“Checking Controllability and Observability”](#).

Completing an Aborted Bus Analysis

During the DRC analysis, TetraMAX ATPG identifies the multidriver nets in the design and attempts to determine whether a pattern can be created to do the following:

- Turn on multiple drivers to cause contention.
- Turn on a single driver to produce a noncontention state.
- Turn all drivers off and have the net float.

TetraMAX ATPG automatically avoids patterns that cause contention. However, it is important to determine whether each net needs to be constantly monitored. The more nets that must be monitored, the more CPU effort is required to create a pattern that tests for specific faults while avoiding contention and floating conditions.

When TetraMAX ATPG successfully completes a bus analysis, it knows which nets must be monitored. However, if a bus analysis is aborted, nets for which analysis was not completed are

assumed to be potentially problematic and therefore need to be monitored. Usually, increasing the ATPG abort limit and performing an `analyze_buses` command completes the analysis, allowing faster test pattern generation.

For an example of interactively performing a bus analysis, see "[Analyzing Buses](#)".

A

Test Concepts

When you perform manufacturing testing, you ensure high-quality integrated circuits by screening out devices with manufacturing defects. You can thoroughly test your integrated circuit when you adopt structured design-for-test (DFT) techniques. The DFT techniques currently supported by TetraMAX ATPG consist of internal scan (both full scan and partial scan) and boundary scan. This appendix covers the background you need to understand these techniques.

The following sections of this appendix describe test concepts:

- [Why Perform Manufacturing Testing?](#)
- [Understanding Fault Models](#)
- Scan Cell Types
- [Coverage Calculations](#)
- [What Is Internal Scan?](#)
- [What Is Boundary Scan?](#)

Why Perform Manufacturing Testing?

Functional testing verifies that your circuit performs as it was intended to perform. For example, assume you have designed an adder circuit. Functional testing verifies that this circuit performs the addition function and computes the correct results over the range of values tested. However, exhaustive testing of all possible input combinations grows exponentially as the number of inputs increases. To maintain a reasonable test time, you must focus functional test patterns on the general function and corner cases.

Manufacturing testing verifies that your circuit does not have manufacturing defects by focusing on circuit structure rather than functional behavior. Manufacturing defects include the following problems:

- Power or ground shorts
- Open interconnect on the die caused by dust particles
- Short-circuited source or drain on the transistor caused by metal spike-through

Manufacturing defects might remain undetected by functional testing yet cause undesirable behavior during circuit operation. To provide the highest-quality products, development teams must prevent devices with manufacturing defects from reaching the customers. Manufacturing testing enables development teams to screen devices for manufacturing defects.

A development team usually performs both functional and manufacturing testing of devices.

Understanding Fault Models

A manufacturing defect has a logical effect on the circuit behavior. An open connection can appear to float either high or low, depending on the technology. A signal shorted to power appears to be permanently high. A signal shorted to ground appears to be permanently low. Many of these manufacturing defects can be represented using the industry-standard stuck-at fault model. Other faults can be modeled using the IDDQ, or quiescent current fault model.

The following sections describe fault models:

- [Stuck-At Fault Models](#)
- [Detecting Stuck-At Faults](#)
- [Using Fault Models to Determine Coverage](#)
- [IDDQ Fault Model](#)
- [Fault Simulation](#)
- [Automatic Test Pattern Generation](#)
- [Translation for the Manufacturing Test Environment](#)

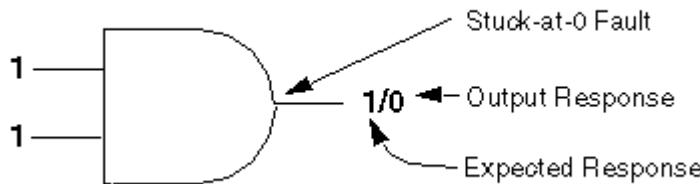
Stuck-At Fault Models

The stuck-at-0 model represents a signal that is permanently low regardless of the other signals that normally control the node. The stuck-at-1 model represents a signal that is permanently high

regardless of the other signals that normally control the node.

For example, [Figure 1](#) shows a two-input AND gate that has a stuck-at-0 fault on the output pin. Regardless of the logic level of the two inputs, the output is always 0.

Figure 1 Stuck-at-0 Fault on Output Pin of 2-input AND Gate



Detecting Stuck-At Faults

The node of a stuck-at fault must be controllable and observable for the fault to be detected.

A node is controllable if you can drive it to a specified logic value by setting the primary inputs to specific values. A primary input is an input that can be directly controlled in the test environment. A node is observable if you can predict the response on it and propagate the fault effect to the primary outputs where you can measure the response. A primary output is an output that can be directly observed in the test environment.

To detect a stuck-at fault on a target node, you must perform the following steps:

- Control the target node to the opposite of the stuck-at value by applying data at the primary inputs.
- Make the node's fault effect observable by controlling the value at all other nodes affecting the output response, so the targeted node is the active (controlling) node.

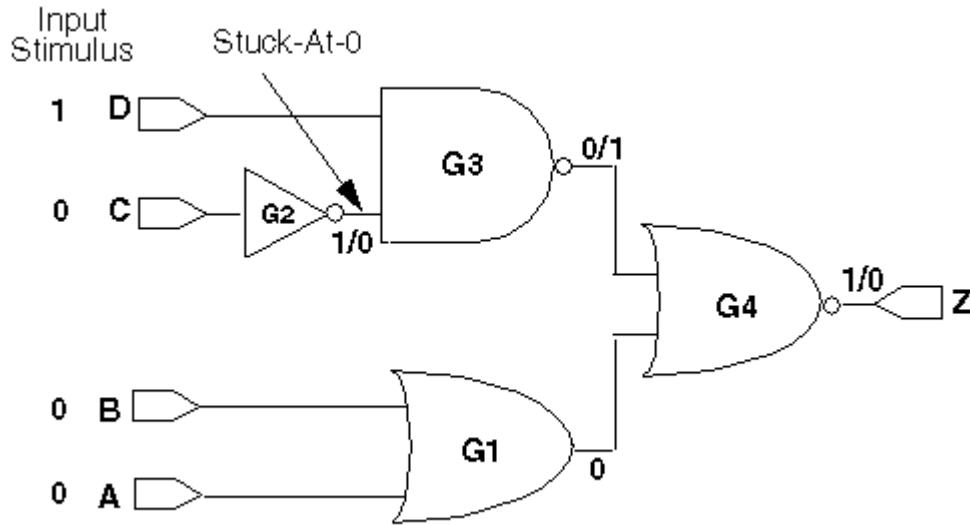
The set of logic 0s and 1s applied to the primary inputs of a design is called the input stimulus. The set of resulting values at the primary outputs, assuming a fault-free design, is called the expected response. The set of actual values measured at the primary outputs is called the output response.

If the output response does not match the expected response for a given input stimulus, the input stimulus has detected the fault. To detect a stuck-at-0 fault, you need to apply an input stimulus that forces that node to 1. For example, to detect a stuck-at-0 fault at the output of the two-input AND gate shown in [Figure 1](#), you need to apply a logic 1 at both inputs. The expected response for this input stimulus is logic 1, but the output response is logic 0. This input stimulus detects the stuck-at-0 fault.

This method of determining the input stimulus to detect a fault uses the single stuck-at fault model. The single stuck-at fault model assumes that only one node is faulty and that all other nodes in the circuit are good. This type of model greatly reduces the complexity of fault modeling and is technology independent.

In a more complex situation, you might need to control all other nodes to ensure observability of a particular target node. [Figure 2](#) shows a circuit with a detectable stuck-at-0 fault at the output of cell G2.

Figure 2 Simple Circuit With Detectable Stuck-At Fault



To detect the fault, you need to control the output of cell G2 to logic 1 (the opposite of the faulty value) by applying a logic 0 value at primary input C. To ensure that the fault effect is observable at primary output Z, you need to control the other nodes in the circuit so that the response value at primary output Z depends only on the output of cell G2.

For this example, you can accomplish the following goals:

- Apply a logic 1 at primary input D so that the output of cell G3 depends only on the output of cell G2. The output of cell G2 is the controlling input of cell G3.
- Apply logic 0s at primary inputs A and B so that the output of cell G4 depends only on the output of cell G2.

Given the input stimuli of A = 0, B = 0, C = 0, and D = 1, a fault-free circuit produces a logic 1 at output port Z. If the output of cell G2 is stuck-at-0, the value at output port Z is a logic 0 instead. Thus, this input stimulus detects a stuck-at-0 fault on the output of cell G2.

This set of input stimuli and expected response values is called a test vector. Following the process previously described, you can generate test vectors to detect stuck-at-1 and stuck-at-0 faults for each node in the design.

Using Fault Models to Determine Test Coverage

One definition of a design's testability is the extent to which that design can be tested for the presence of manufacturing defects, as represented by the single stuck-at fault model. Using this definition, the metric used to measure testability is test coverage. For a precise explanation of how test coverage is calculated in TetraMAX ATPG, see [“Test Coverage”](#).

For larger designs, it is not feasible to analyze the test coverage results for existing functional test vectors or to manually generate test vectors to achieve high test coverage results. Fault simulation tools determine the test coverage of a set of test vectors. ATPG tools generate manufacturing test vectors. Both of these automated tools require models for all logic elements

in your design to calculate the expected response correctly. Your semiconductor vendor provides these models.

IDDQ Fault Model

For CMOS circuits, an alternative testing method is available, called IDDQ testing. IDDQ testing is based on the principle that a CMOS circuit does not draw a significant amount of current when the device is in the quiescent (quiet, steady) state. The presence of even a single circuit fault, such as a short from an internal node to ground or to the power supply, can be detected by the resulting excessive current drain at the power supply pin. IDDQ testing can detect faults that are not observable by stuck-at fault testing.

For the IDDQ testing, the ATPG process uses an IDDQ fault model rather than a stuck-at fault model. The generated test patterns only need to control internal nodes to 0 and 1 and comply with quiescence requirements. The patterns do not need to propagate the effects of faults to the device outputs. The ATPG tool attempts to maximize the toggling of internal states and minimize the number of patterns needed to control each node to both 0 and 1 for IDDQ testing.

TetraMAX ATPG has an optional IDDQ pattern generation/verification capability called IddQTest. It uses the following criteria for IDDQ pattern generation:

- No current should flow through resistors.
- There must not be contention on any bus or node.
- No nodes can be allowed to float. A floating node could cause some CMOS transistors to turn on and draw current.
- RAM modules must be disabled so that they do not draw any current.

For more information on IddQTest, see the *Test Pattern Validation User Guide*.

Fault Simulation

Fault simulation determines the test coverage of a set of test vectors. It performs several logic simulations concurrently: one simulation represents the fault-free circuit (a good machine simulation) and several simulations represent the circuits containing single stuck-at faults (a faulty machine simulation). Fault simulation detects a fault each time the output response of the faulty machine is a non-X value and is different from the output response of the good machine for a given vector.

Fault simulation determines all faults detected by a test vector. By fault simulating the test vector that is generated to detect the stuck-at-0 fault on the output of G2 in [Figure 2](#), it is apparent that this vector also detects the following single stuck-at faults:

- Stuck-at-1 on all pins of G1 (and ports A and B)
- Stuck-at-1 on the input of G2 (and port C)
- Stuck-at-0 on the inputs of G3 (and port D)
- Stuck-at-1 on the output of G3
- Stuck-at-1 on the inputs of G4
- Stuck-at-0 on the output of G4 (and port Z)

You can generate manufacturing test vectors by manually generating test vectors and then fault-simulating them to determine the test coverage. For large or complex designs, however, this process is time consuming and often does not result in high test coverage.

Automatic Test Pattern Generation

ATPG generates test patterns and provides test coverage statistics for the generated pattern set. The difference between test vectors and test patterns is defined in "[What Is Internal Scan?](#)". For now, consider the term "test vector" to be the same as "test pattern."

ATPG for combinational circuits is well understood; it is usually possible to generate test vectors that provide high test coverage for combinational designs.

Combinational ATPG tools can use both random and deterministic techniques to generate test patterns for stuck-at faults. By default, TetraMAX ATPG only uses deterministic pattern generation; using random pattern generation is optional.

During random pattern generation, the tool assigns input stimuli in a pseudo-random manner, then fault-simulates the generated vector to determine which faults are detected. As the number of faults detected by successive random patterns decreases, ATPG can change to a deterministic technique.

During deterministic pattern generation, the tool uses a pattern generation process based on path-sensitivity cone timepts to generate a test vector that detects a specific fault in the design. After generating a vector, the tool fault-simulates the vector to determine the complete set of faults detected by the vector. Test pattern generation continues until all faults either have been detected or have been identified as undetectable by the process.

Because of the effects of memory and timing, ATPG is much more difficult for sequential circuits than for combinational circuits. It is often not possible to generate high test coverage test vectors for complex sequential designs, even when you use sequential ATPG. Sequential ATPG tools use deterministic pattern generation algorithms based on extended applications of the path-sensitivity cone timepts.

Structured DFT techniques (for example, internal scan) simplify the test pattern generation task for complex sequential designs, resulting in higher test coverage and reduced testing costs. For more information about internal scan techniques, see "[What Is Internal Scan?](#)".

Translation for the Manufacturing Test Environment

To test for manufacturing defects in your chips, you need to translate the generated test patterns into a format acceptable to the automated test equipment (ATE). On the ATE, the logic 0s and 1s in the input stimuli are translated into low and high voltages to be applied to the primary inputs of the device under test. The logic 0s and 1s in the output response are compared with the voltages measured at the primary outputs. For combinational ATPG, one test vector corresponds to one ATE cycle.

You might use more than one set of test vectors for manufacturing testing. The term "test program" means the collection of all test vector sets used to test a design.

What Is Internal Scan?

Internal scan design is the most common DFT technique and has the greatest potential for high test coverage. The principle of this technique is to modify the existing sequential elements in the design to support serial shift capability, in addition to their normal functions; and to connect these elements into serial chains to make, in effect, long shift registers.

Each scan chain element can operate like a primary input or primary output during ATPG testing, greatly enhancing the controllability and observability of the internal nodes of the device. This technique simplifies the pattern generation problem by effectively dividing complex sequential designs into fully isolated combinational blocks (full-scan design) or semi isolated combinational blocks (partial-scan design).

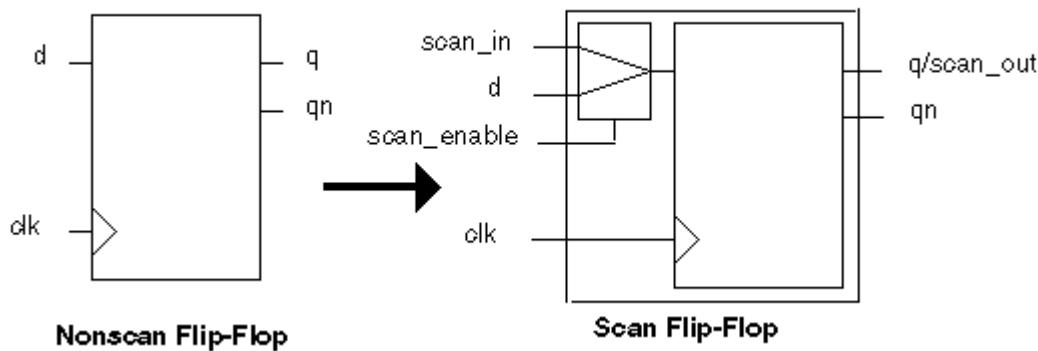
The following sections describe internal scan:

- [Example](#)
 - [Applying Test Patterns](#)
 - [Scan Design Requirements](#)
 - [Full-Scan Design](#)
 - [Partial-Scan ATPG Design](#)
-

Example

[Figure 1](#) shows an example of the multiplexed flip-flop scan style, where a D flip-flop has been modified to support internal scan by the addition of a multiplexer. Inputs to the multiplexer are the data input of the flip-flop (d) and the scan-input signal (scan_in). The active input of the multiplexer is controlled by the scan-enable signal (scan_enable). Input pins are added to the cell for the scan_in and scan_enable signals. One of the data outputs of the flip-flop (q or qn) is used as the scan-output signal (scan_out). The scan_out signal is connected to the scan_in signal of another scan cell to form a serial scan (shift) capability.

Figure 1 D Flip-Flop With Scan Capability



The modified sequential cells are chained together to form one or more large shift registers. These shift registers are called scan chains or scan paths. The sequential cells connected in a

scan chain are scan controllable and scan observable. A sequential cell is scan controllable when you can set it to a known state by serially shifting in specific logic values. ATPG tools treat scan controllable cells as pseudo-primary inputs to the design. A sequential cell is scan observable when you can observe its state by serially shifting out data. ATPG tools treat scan-observable cells as pseudo-primary outputs of the design.

Most semiconductor vendor libraries include pairs of equivalent nonscan and scan cells that support a given scan style. One special test cell is a scan flip-flop that combines a D flip-flop and a multiplexer. You can also implement the scan function of this special test cell with discrete cells, such as the separate flip-flop and multiplexer shown in [Figure 1](#).

Adding scan circuitry to a design usually has the following effects:

- Design size and power increases slightly because scan cells are usually larger than the nonscan cells they replace, and the nets used for the scan signals occupy additional area.
- Design performance (speed) decreases marginally because of changes in the electrical characteristics of the scan cells that replace the nonscan cells.
- Global test signals that drive many sequential elements might require buffering to prevent electrical design rule violations.

The effects of adding scan circuitry vary depending on the scan style and the semiconductor vendor library you use. For some scan styles, such as level-sensitive scan design (LSSD), introducing scan circuitry produces a negligible local change in performance.

The Synopsys scan DFT synthesis capabilities fully optimize for the user's design rules and constraints (timing, area, and power) in the context of scan. These scan synthesis capabilities are available in DFT Compiler, the Synopsys test-enabled synthesis configuration. For information about how DFT Compiler minimizes the impact of inserting scan logic in your design, see the *DFT Compiler Scan User Guide*.

For scan designs, an ATPG tool generates input stimuli for the primary inputs and pseudo-primary inputs and expected responses for the primary outputs and pseudo-primary outputs. The set of input stimuli and output responses is called a test pattern or scan pattern. This set includes the data at the primary inputs, primary outputs, pseudo-primary inputs, and pseudo-primary outputs.

A test pattern represents many test vectors because the pseudo-primary-input data must be serialized to be applied at the input of the scan chain, and the pseudo-primary-output data must be serialized to be measured at the output of the scan chain.

Applying Test Patterns

Test patterns are applied to a scan-based design through the scan chains. The process is the same for a full-scan or partial-scan design.

Scan cells operate in one of two modes: parallel mode or shift mode. In the multiplexed flip-flop scan style shown in [Figure 1](#), the mode is controlled by the scan_enable pin. When the scan_enable signal is inactive, the scan cells operate in parallel mode; the input to each scan element comes from the combinational logic block. When the scan_enable signal is asserted, the scan cells operate in shift mode; the input comes from the output of the previous cell in the scan chain (or from the scan input port, if it is the first chain element). Other scan styles work similarly.

The target tester applies a scan pattern in the following sequence:

1. Select shift mode by setting the scan-enable port. This test signal is connected to all scan cells.
2. Shift in the input stimuli for the scan cells (pseudo-primary inputs) at the scan input ports.
3. Select parallel mode by disabling the scan-enable port.
4. Apply the input stimuli to the primary inputs.
5. Check the output response at the primary outputs after the circuit has settled and compare it with the expected fault-free response. This process is called parallel measure.
6. Pulse one or more clocks to capture the steady-state output response of the nonscan logic blocks into the scan cells. This process is called parallel capture.
7. Select shift mode by asserting the scan-enable port.
8. Shift out the output response of the scan cells (pseudo-primary outputs) at the scan output ports and compare the scan cell contents with the expected fault-free response.

Scan Design Requirements

You achieve the best test coverage results when all nodes in your design are controllable and observable. Adding scan logic to your design enhances its controllability and observability. The rules governing the controllability and observability of scan cells are called test design rules.

Controllability of Sequential Cells

For sequential cells, design rules require that all state elements can be controlled, by scan or other means, to required state values from the boundary of the design. These requirements are primarily involved with the shift operations in scan test.

In an ideal full-scan design, the scan chain contains all state elements, the circuit is fully controllable, and any circuit state can be achieved.

Using a partial-scan methodology, not all state elements need to be in the scan chain. As long as the nonscan state elements can be brought to any required state predictably through sequential operation, the circuit remains sufficiently controllable.

Observability of Sequential Cells

For sequential cells, test design rules require predictable capture of the next state of the circuit and visibility at the boundary of the design. In the context of scan design, you can ensure that sequential cells are observable if you successfully clock the scan cells in the circuit, and then shift their state to the scan outputs.

The following operations define circuit observability:

1. Observe the primary outputs of the circuit after scan-in.
Normally, this does not involve DFT and does not present problems.
2. Reliably capture the next state of the circuit.
If the functional operation is impaired, unpredictable, or unknown, the next state is unknown. This unknown state makes at least part of the circuit unobservable.
3. Extract the next state through a scan-out operation.

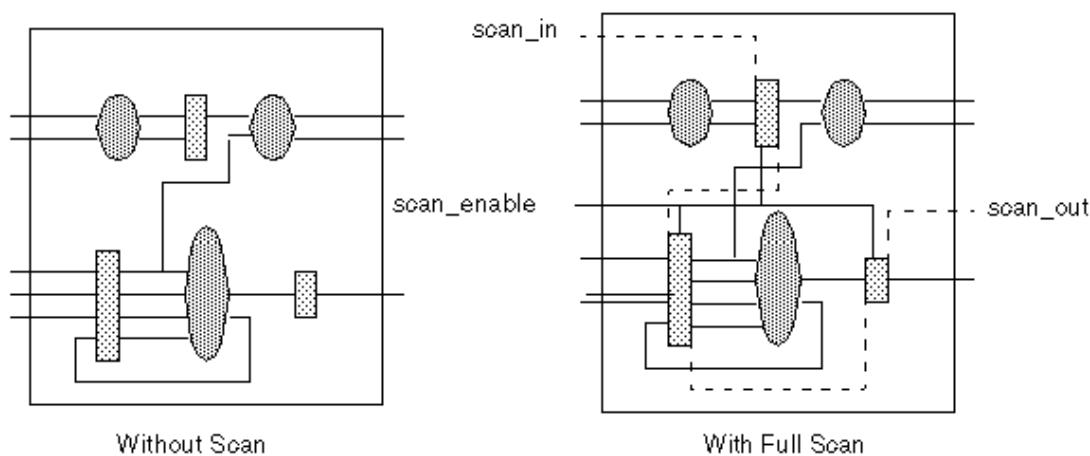
This process is similar to scan-in. The additional requirement is that the shift registers pass data reliably to the output ports.

Full-Scan Design

With a full-scan design technique, all sequential cells in the design are modified to perform a serial shift function. Sequential elements that are not scanned are treated as black box cells (cells with unknown function).

Full scan divides a sequential design into combinational blocks as shown in [Figure 2](#). Ovals represent combinational logic; rectangles represent sequential logic. The full-scan diagram shows the scan path through the design.

Figure 2 Scan Path Through a Full-Scan Design



Through pseudo-primary inputs, the scan path enables direct control of inputs to all combinational blocks. The scan path enables direct observability of outputs from all combinational blocks through pseudo-primary outputs. You can use the efficient combinational capabilities of TetraMAX ATPG to achieve high test coverage results on a full-scan design.

Partial-Scan ATPG Design

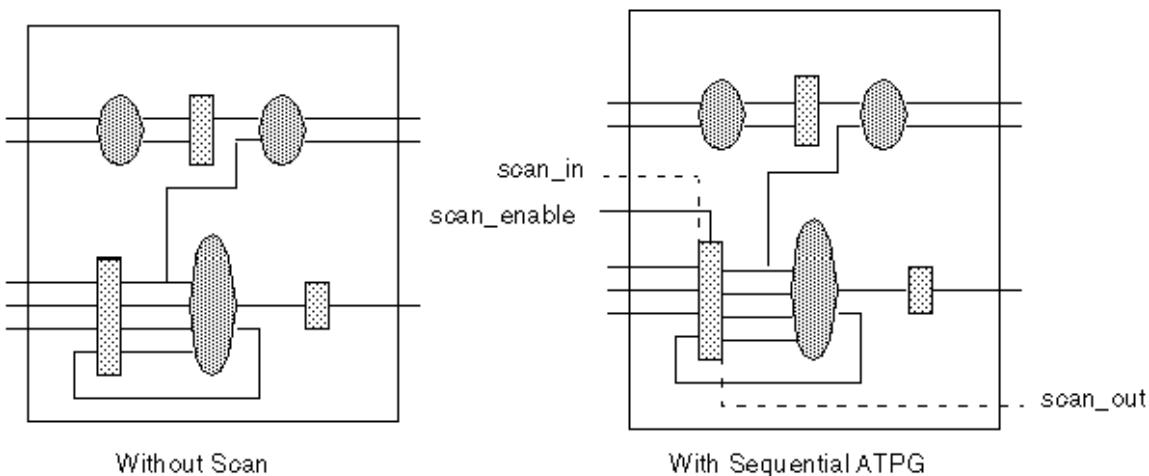
With a partial-scan design technique, the scan chains contain some, but not all, of the sequential cells in the design. A partial-scan technique offers a tradeoff between the maximum achievable test coverage and the effect on design size and performance.

The default ATPG mode of TetraMAX ATPG, called Basic-Scan ATPG, performs combinational ATPG. To get good test coverage in partial-scan designs, you need to use Fast-Sequential or Full-Sequential ATPG. The sequential ATPG processes perform propagation of faults through nonscan elements. For more information, see ["ATPG Modes"](#).

Partial scan divides a complex sequential design into simpler sequential blocks as shown in [Figure 3](#). Ovals represent combinational logic; rectangles represent sequential logic. The partial-

scan diagram shows the scan path through the design after sequential ATPG has been performed.

Figure 3 Scan Path Through a Partial-Scan Design



Typically, a partial-scan design does not allow test coverage to be as high as for a similar full-scan design. The level of test coverage for a partial-scan design depends on the location and number of scan registers in that design, and the ATPG effort level selected for the Fast-Sequential or Full-Sequential ATPG process.

What Is Boundary Scan?

Boundary scan is a DFT technique that simplifies printed circuit board testing using a standard chip-board test interface. The industry standard for this test interface is the *IEEE Standard Test Access Port and Boundary Scan Architecture* (IEEE Std 1149.1).

The boundary-scan technique is often referred to as JTAG. JTAG is the acronym for Joint Test Action Group, the group that initiated the standardization of this test interface.

Boundary scan enables board-level testing by providing direct access to the input and output pads of the integrated circuits on a printed circuit board. Boundary scan modifies the I/O circuitry of individual ICs and adds control logic so the input and output pads of every boundary scan IC can be joined to form a board-level serial scan chain.

The boundary-scan technique uses the serial scan chain to access the I/O ports of chips on a board. Because the scan chain comprises the input and output pads of a chip's design, the chip's primary inputs and outputs are accessible on the board for applying and sampling data.

Boundary scan supports the following board-level test functions:

- Testing of the interconnect wiring on a printed circuit board for shorts, opens, and bridging faults
- Testing of clusters of non-boundary-scan logic
- Identification of missing, misoriented, or wrongly selected components

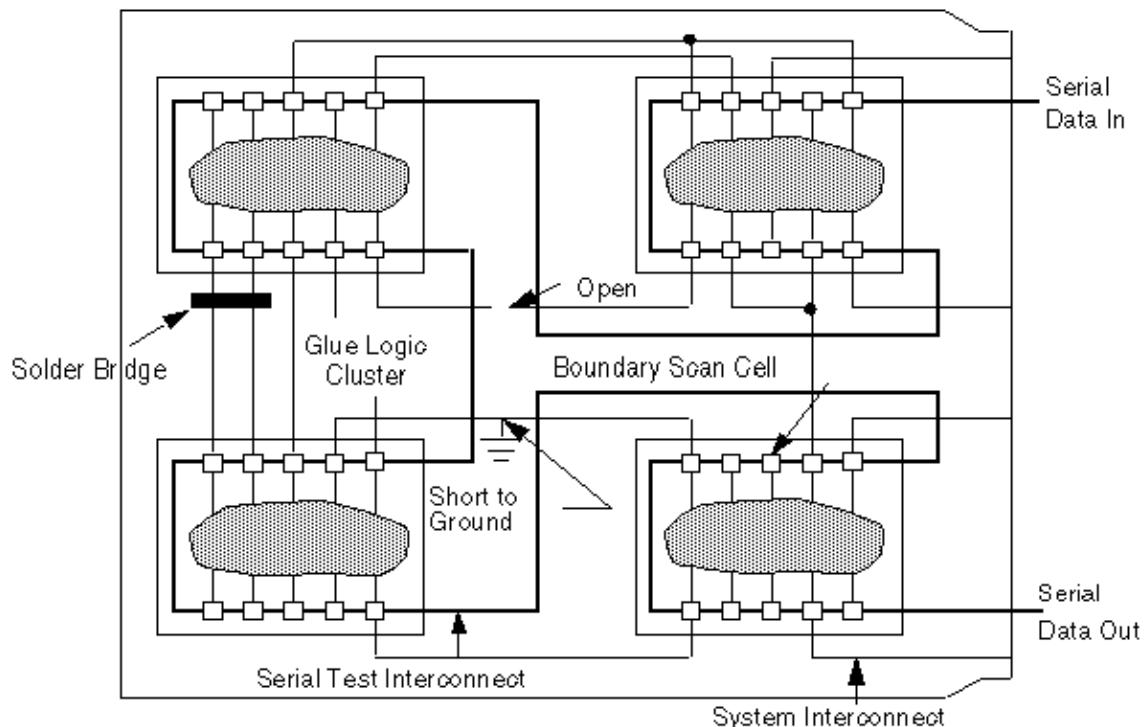
- Identification of fixture problems
- Limited testing of individual chips on a board

Note:

Although boundary scan addresses several board-test issues, it does not address chip-level testability. To provide testability at both the chip and board level, combine chip-test techniques (such as internal scan) with boundary scan.

[Figure 1](#) shows a simple printed circuit board with several boundary scan ICs and illustrates some of the failures that boundary scan can detect.

Figure 1 Board Testing With IEEE Std 1149.1 Boundary Scan



B

AATPG Design Guidelines

This appendix presents some design guidelines to facilitate successful ATPG and suggests sources of extra ports for test I/O. The design topics are discussed first in textual form. Next, selected design guidelines are illustrated with schematics. Finally, concise checklists for the design guidelines and port suggestions provide you with a quick reference as you implement your design.

This appendix contains the following sections:

- [AATPG Design Guidelines](#)
- [Checklists for Quick Reference](#)

ATPG Design Guidelines

This section provides guidelines for ATPG testing and offers suggestions for identifying ports to use for test I/O. The provided guidelines are not exhaustive, but if implemented, they can prevent many problems that commonly occur during ATPG testing.

Guidelines are provided for the following design entities:

- [Internally Generated Pulsed Signals](#)
 - [Clock Control](#)
 - [Pulsed Signals to Sequential Devices](#)
 - [Multidriver Nets](#)
 - [Bidirectional Port Controls](#)
 - [Clocking Scan Chains: Clock Sources, Trees, and Edges](#)
 - [Protection of RAMs During Scan Shifting](#)
 - [RAM and ROM Controllability During ATPG](#)
 - [Pulsed Signal to RAMs and ROMs](#)
 - [Bus Keepers](#)
-

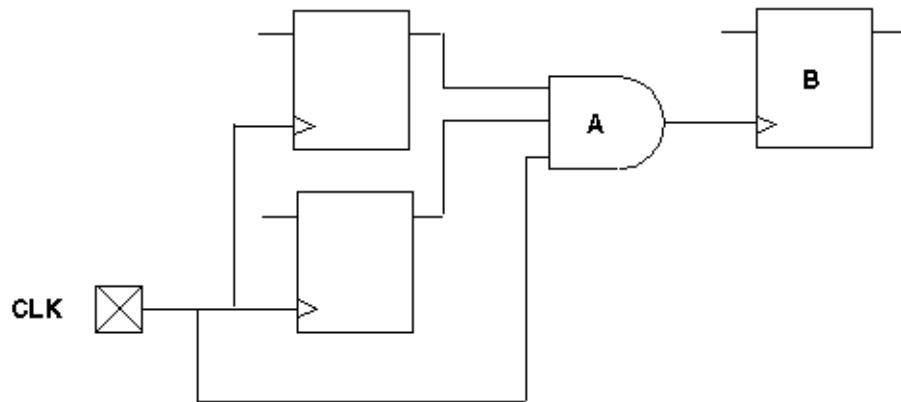
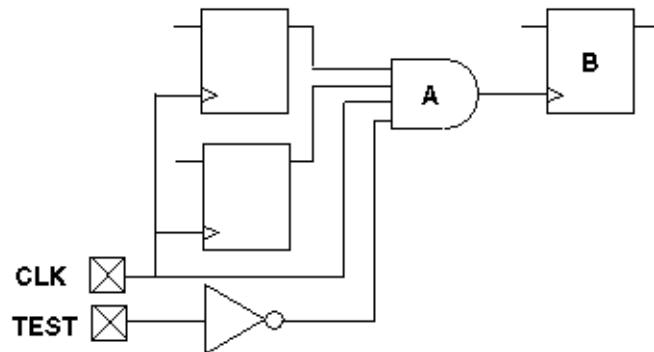
Internally Generated Pulsed Signals

While TetraMAX ATPG is in ATPG test mode, ensure that clocks and asynchronous set or reset signals come from primary inputs. Your design should not include internally generated clocks or asynchronous set or reset signals.

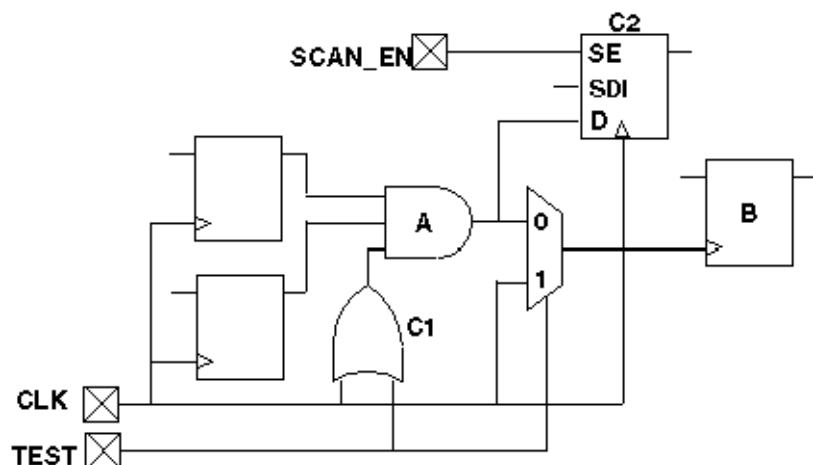
Do not use clock dividers in ATPG test mode. If your design contains clock dividers, bypass them in ATPG test mode. A scan chain must shift one bit for one scan clock. Use the TEST signal to control the source of the internal clocks, so that in ATPG test mode you can bypass the clock divider and source the internal clocks from the primary CLK output.

Do not use gated clocks such as the one shown in [Figure 1](#) in ATPG test mode. If your design contains clock gating, constrain the control side of the gating element while in ATPG test mode.

[Figure 2](#) and [Figure 3](#) show two solutions. In [Figure 2](#), the TEST input blocks the path from CLK to register B. However, B cannot be used in a scan chain.

Figure 1 Gated Clock: A Problem**Figure 2 Gated Clock: Solution 1**

In [Figure 3](#), the TEST input controls a MUX that changes the clock source for register B. Optionally adding gates C1 and C2 provides observability for the output of gate A; otherwise, gate A is unobservable and all faults into A are ATPG untestable.

Figure 3 Gated Clock: Solution 2

Do not use phase-locked loops (PLLs) as clock sources in ATPG test mode. If your design contains PLLs, bypass the clocks while in ATPG test mode.

Do not use pulse generators in ATPG test mode, such as the one shown in [Figure 4](#). If your design contains pulse generators, bypass them using a MUX with the select line constrained to a constant value or shunted with AND or OR logic so that the pulse generators do not pulse while in ATPG test mode, as shown in Solution 1 and Solution 2 in [Figure 5](#).

In Solution 1, the TEST input disables the pulse generator. However, using this solution, any sequential elements that use N2 as a clock source no longer have a clock source. In Solution 2, the TEST input multiplexes out the original pulse and replaces it with access from a top-level input port.

Figure 4 Pulse Generators: A Problem

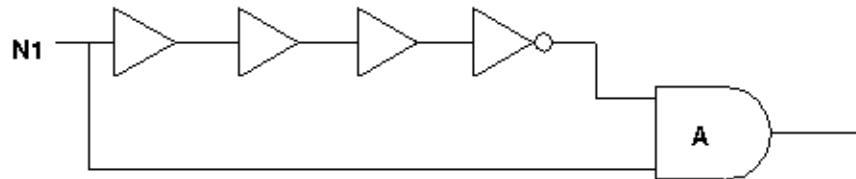
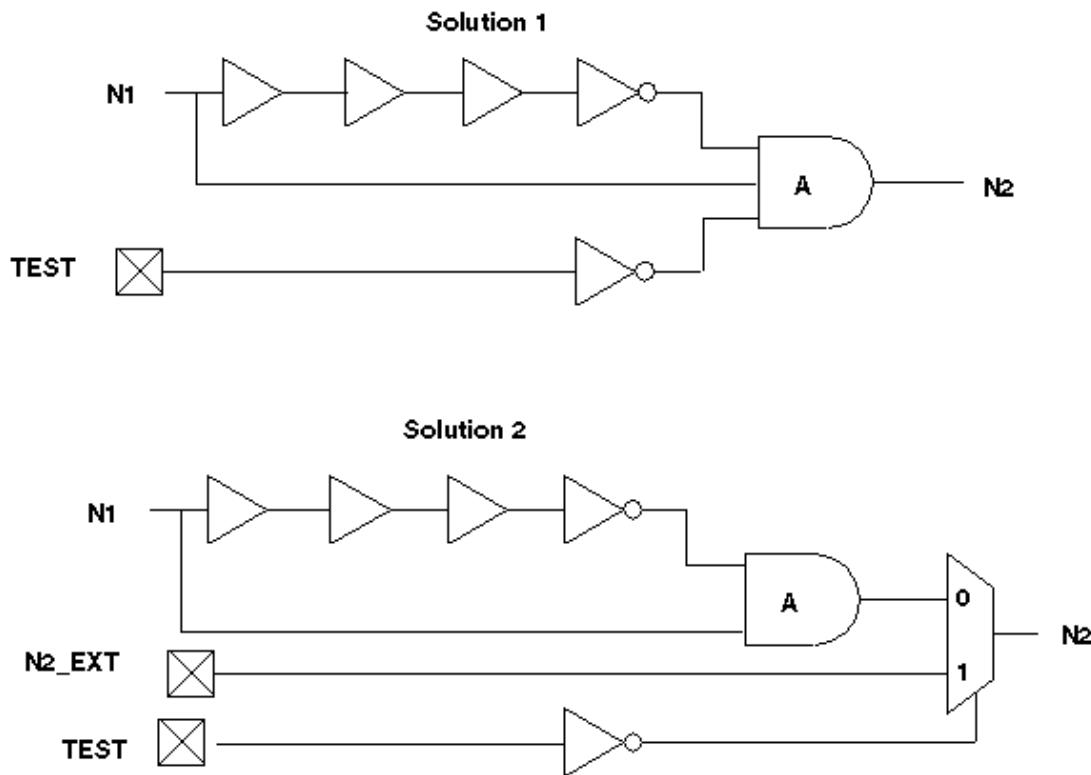


Figure 5 Pulse Generators: Two Solutions



Do not use a power-on reset circuit in ATPG test mode. A power-on reset circuit is essentially an uncontrolled internal clock source that operates when the power is initially applied to the circuit. See [Figure 6](#).

To prevent a power-on reset circuit from operating during test, you can perform either of the following steps:

Use the test mode control signal to multiplex the power-on reset signal so that it comes from an existing reset input or some other primary input during test. See [Figure 7](#).

Use the test mode control signal to block the power-on reset source so that it has no effect during test. See [Figure 8](#).

The first of these two methods is usually better because it is less likely to cause a reduction in test coverage.

Figure 6 Power-On Reset Circuit Configuration to Avoid



Figure 7 Power-On Reset Circuit Test Method 1

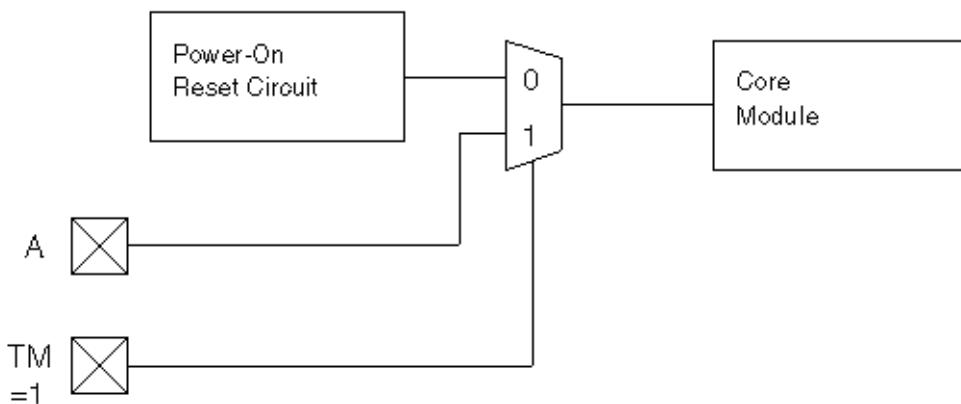
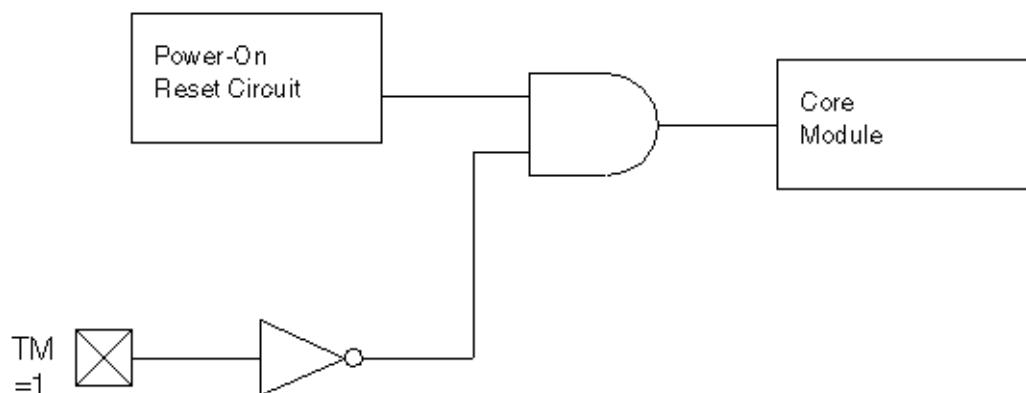


Figure 8 Power-On Reset Circuit Test Method 2



Clock Control

While TetraMAX ATPG is in ATPG test mode, provide complete control of clock paths to scan chain flip-flops. The clock/set/reset paths to scan chain elements must be fully controlled.

If a clock passes through a MUX, constrain the select line of the MUX to a constant value while in ATPG test mode.

If a clock passes through a combinational gate, constrain the other inputs of the gate to a constant value while in ATPG test mode. See [Figure 9](#) and [Figure 10](#).

Pass clock signals directly through JTAG I/O cells without passing through a MUX, unless the MUX control can be constrained. This typically involves using a special JTAG input cell. [Figure 11](#) shows a JTAG input cell with a MUX through which the signal passes; it is difficult to hold the MUX control constant. [Figure 12](#) shows a modified JTAG input cell that has no MUX in the path.

Avoid using bidirectional clocks or asynchronous set or reset ports while in ATPG test mode. If your design supports bidirectional clocks or asynchronous set or reset ports, force them to operate as unidirectional ports while in ATPG test mode. See [Figure 13](#) and [Figure 14](#).

Figure 9 Problem: Clock Paths Through Combinational Gates

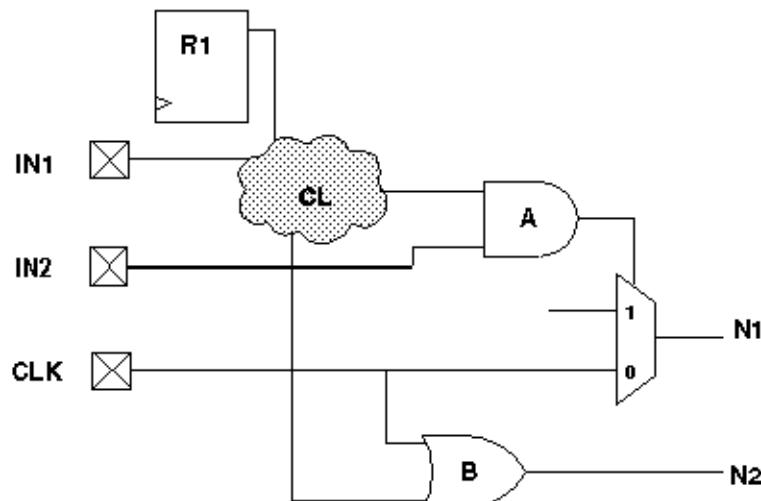


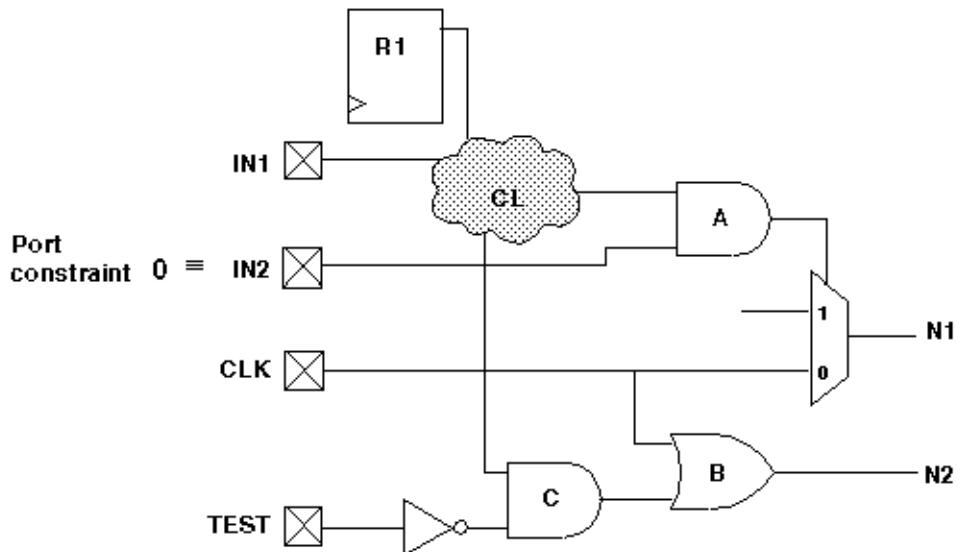
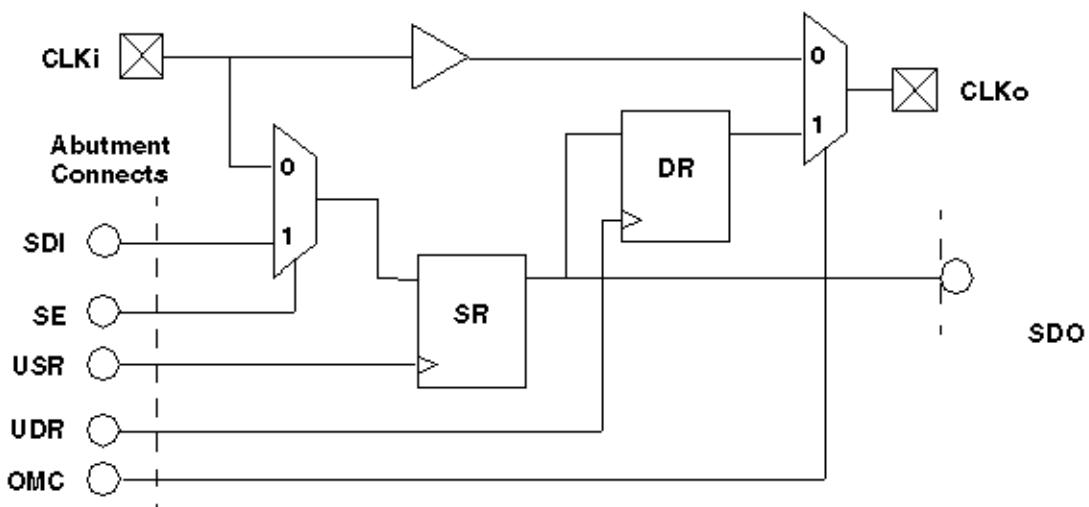
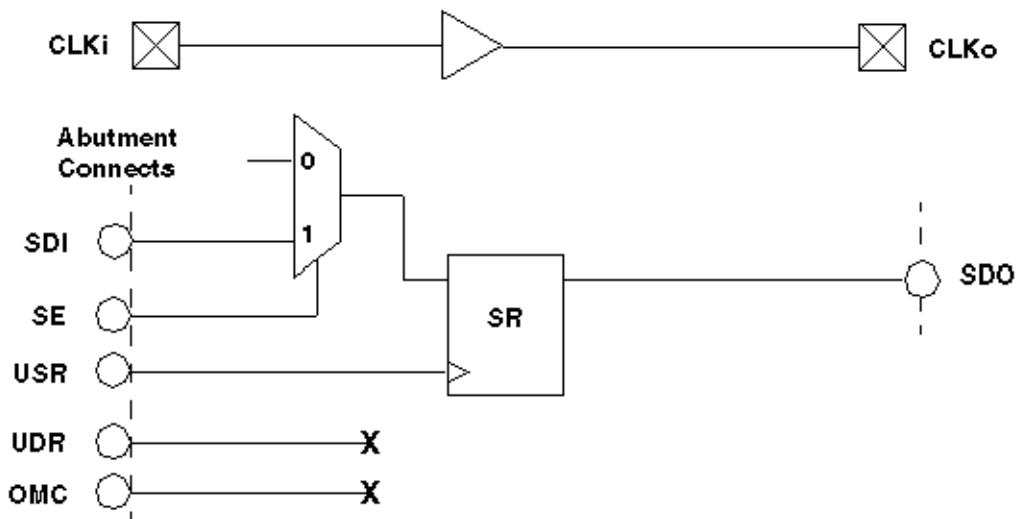
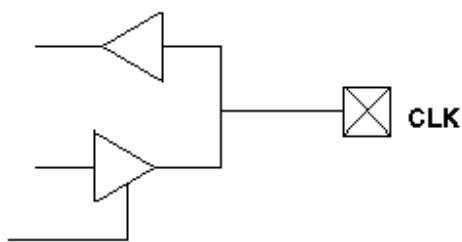
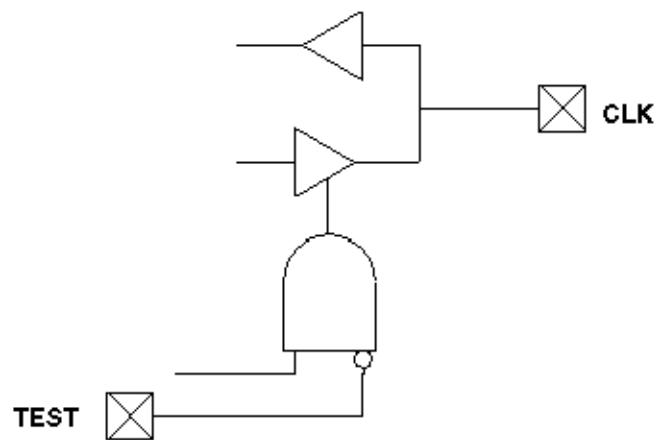
Figure 10 Solution: Clock Paths Through Combinational Gates**Figure 11 Problem: Clock/Set/Reset Inputs and JTAG I/O Cells**

Figure 12 Solution: Clock/Set/Reset Inputs and JTAG I/O Cells**Figure 13 Problem: Bidirectional Clock/Set/Reset****Figure 14 Solution: Bidirectional Clock/Set/Reset**

Pulsed Signals to Sequential Devices

While TetraMAX ATPG is in ATPG test mode, do not allow an open path from a pulsed input signal (clock, asynchronous set/reset) to the data input of a sequential device.

- Do not allow a path from a pulsed input to both the data input and clock of the same flip-flop while TetraMAX ATPG is in ATPG test mode. As shown in [Figure 15](#), the value of the data captured cannot be determined in the absence of timing analysis. If your design contains such a path, then while in ATPG test mode, shunt the path to either the data or clock pin with AND or OR logic, or with a MUX, as shown by Solution 1 and Solution 2 in [Figure 16](#). In Solution 1, a controllable top-level input is used to replace the path of the clock/set/reset into the combinational cloud. In Solution 2, the TEST input blocks the path of the clock/set/reset into the combinational cloud so that it does not pass the clock pulse while in ATPG test mode.
- Do not allow a path from a pulsed input to both the data input and the asynchronous set or reset input of the same flip-flop while TetraMAX ATPG is in ATPG test mode. If your design contains such a path, while in ATPG test mode, shunt the path to either the data pin or the set/reset pin with AND logic, OR logic, or a MUX.

Figure 15 Problem: Sequential Device Pulsed Data Inputs

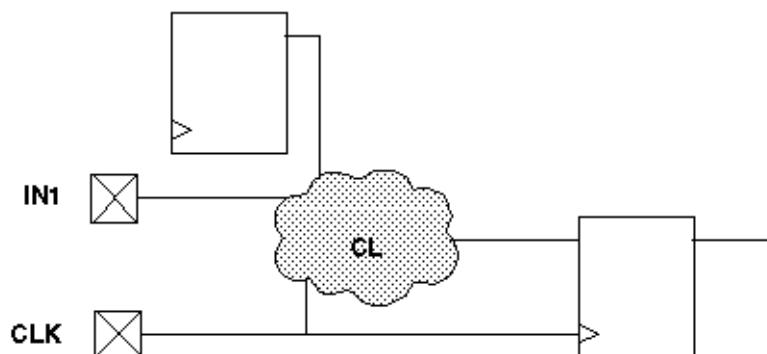
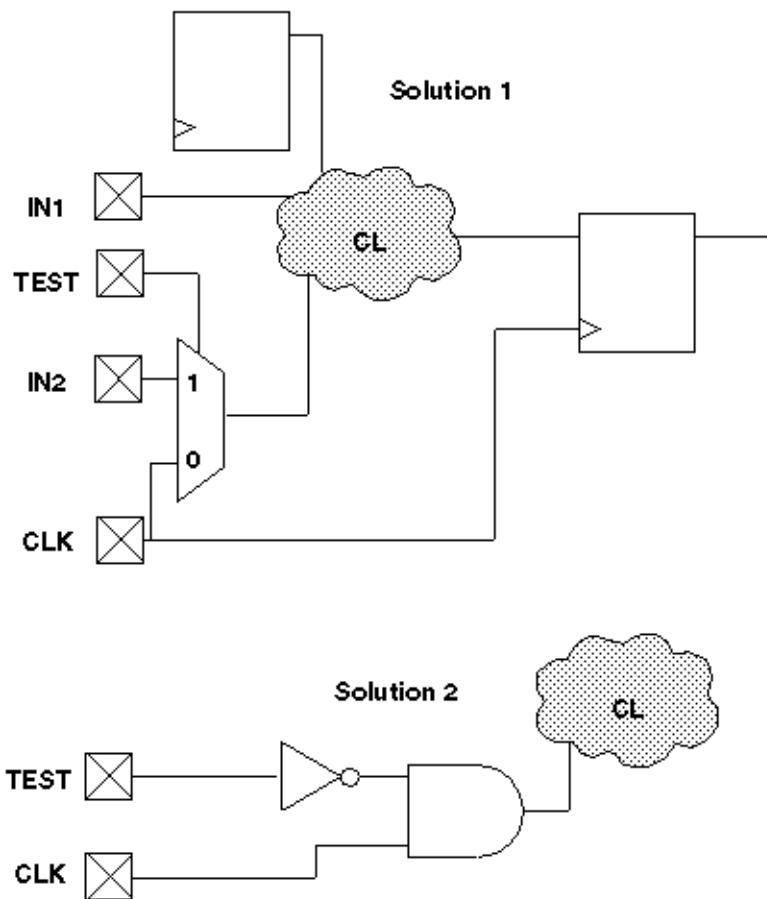


Figure 16 Solution: Sequential Device Pulsed Data Inputs

Multidriver Nets

For multidriver nets, ensure that exactly one driver is enabled during the shifting of scan chains in ATPG test mode. Plan this guideline into your design. For most designs with multidriver nets, there is danger of internal driver contention because shifting a scan chain has a random effect on the design state. See [Figure 17](#).

Here are two methods for satisfying this design guideline:

- Have a primary input port that acts as a global override on internal driver enable signals in ATPG test mode, disabling all but one driver of the net and forcing that driver to an on state, as shown in [Figure 18](#). This primary input port should be asserted during the scan chain load and unload operation. This design guideline is supported by DFT Compiler and is the default behavior of DFT Compiler.
- Use deterministic decoding on the driver enables. Use a 1-of- n logic to ensure that only one driver is enabled at all times and that at least one driver is enabled at all times, as shown in [Figure 19](#). Deterministic decoding might not be appropriate for some designs.

For example, for a design with hundreds of potential drivers, a 1-of- n decoder would be too large or would add too much delay to the circuit.

Figure 17 Problem: Multidriver Nets

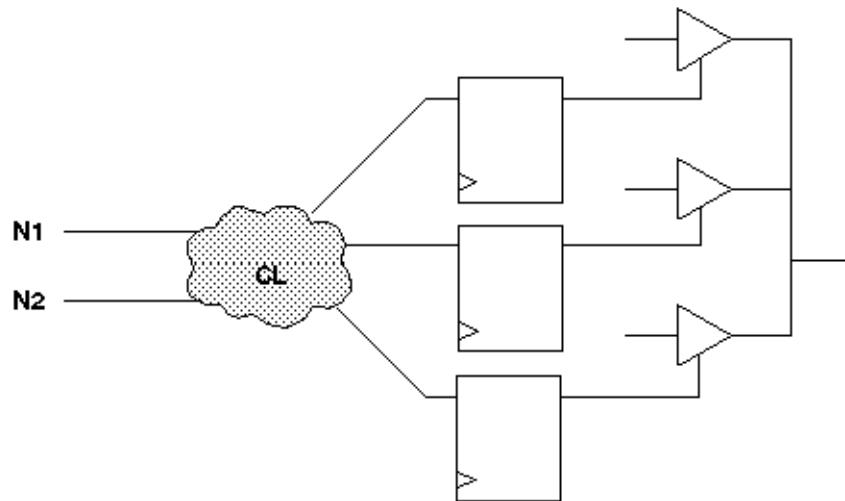


Figure 18 Multidriver Nets: Global Override Input

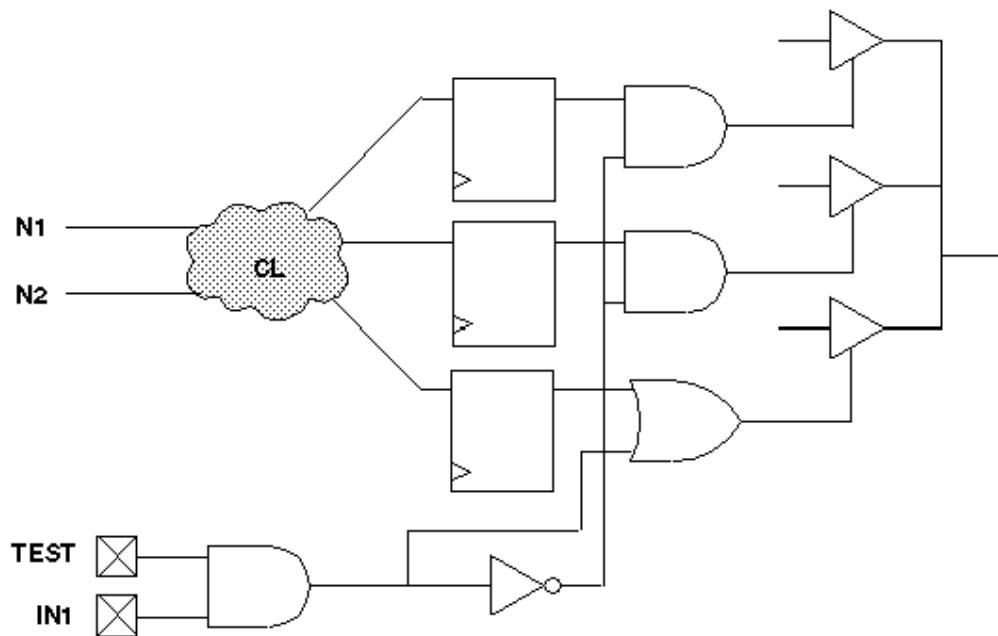
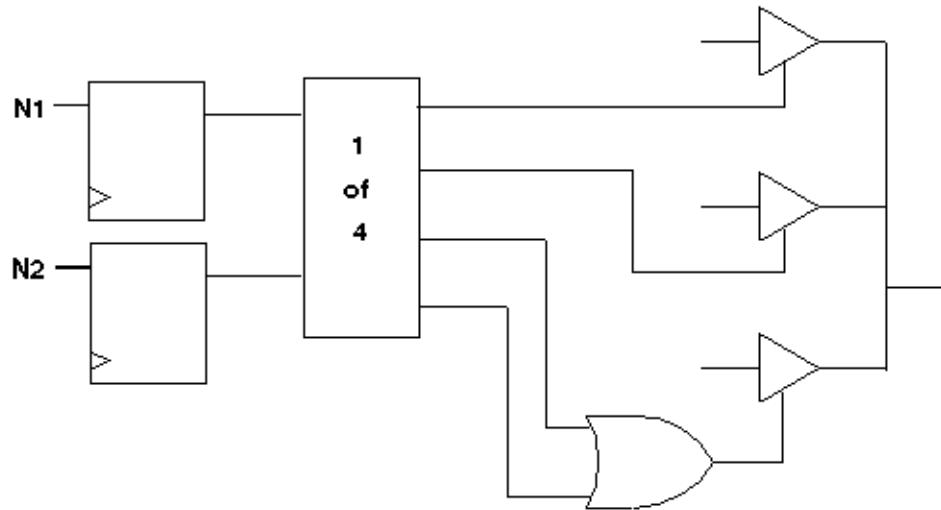


Figure 19 Multidriver Nets: Deterministic Decoding

Bidirectional Port Controls

Force all bidirectional ports to input mode while shifting scan chains in ATPG test mode, using a top-level port as control. See [Figure 20](#) and [Figure 21](#). In [Figure 21](#), TEST controls the disabling logic and SCAN_EN ensures that the scan chain outputs are turned on.

The top-level port is often tied to a scan enable control port. However, there are advantages to performing this function on a different port, if extra ports are available, because keeping the control of the bidirectional ports separate from the scan enable gives the ATPG process more flexibility in generating patterns.

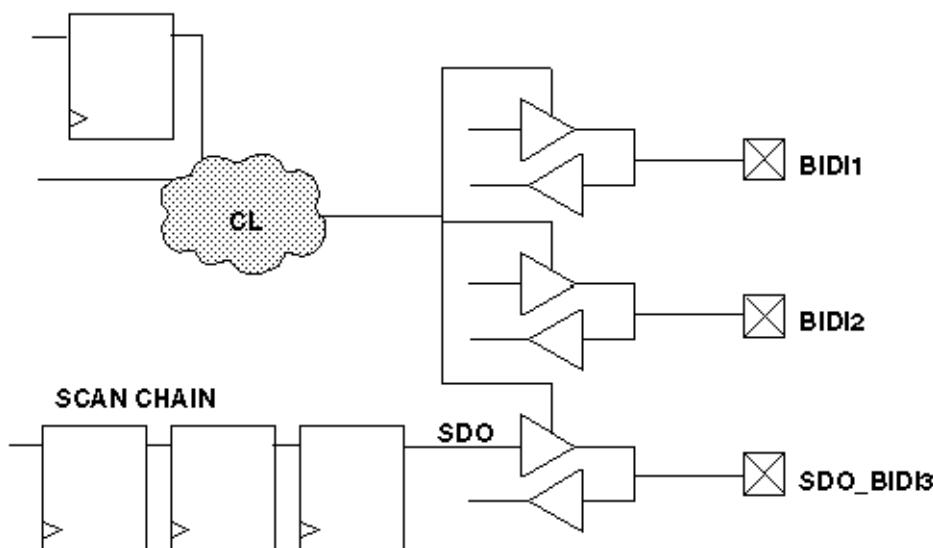
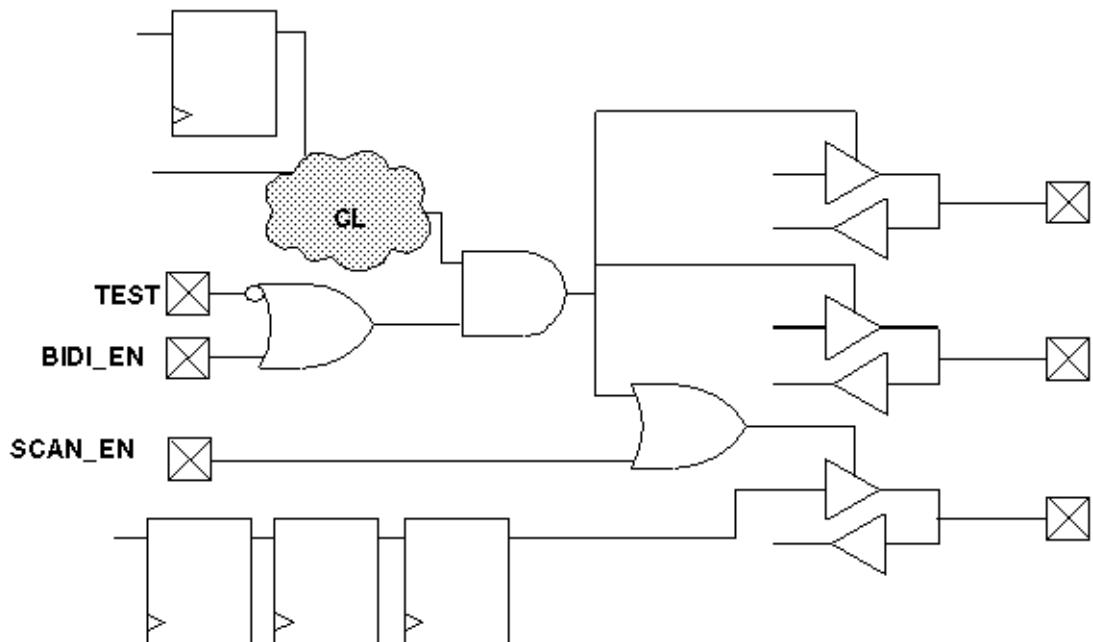
Figure 20 Problem: Bidirectional Port Controls

Figure 21 Solution: Bidirectional Port Controls

If you follow this guideline along with Guideline 3, you can easily ensure that no internal or I/O contention can occur during scan chain load/unload operations.

This guideline is supported by DFT Compiler (using a single pin) and is the default behavior.

Exception

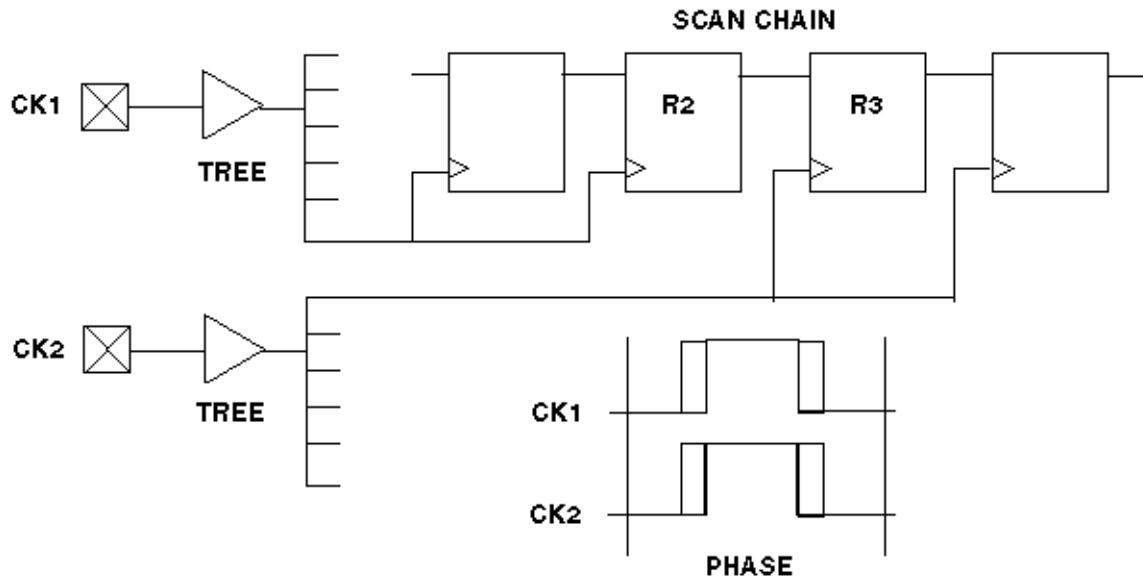
Force scan chain outputs that use bidirectional or three-state ports into an output mode while shifting scan chains in ATPG test mode, using a top-level port (usually SCAN_ENABLE), as shown in [Figure 21](#).

This guideline is the exception to Guideline 5 and is automatically supported by DFT Compiler if you specify a bidirectional port for use as a scan chain output.

Clocking Scan Chains: Clock Sources, Trees, and Edges

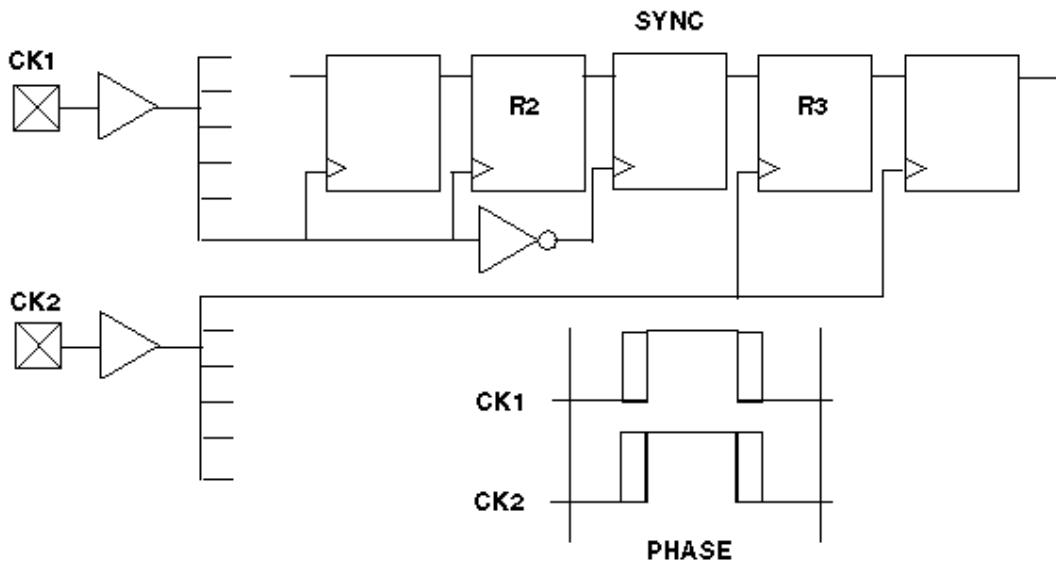
Use a single clock tree to clock all flip-flops in the same scan chain. If the design contains multiple clock trees, insert resynchronization latches in the scan data path between scan cell flip-flops that use different clock sources.

In [Figure 22](#), the two clock sources can cause a race condition. For example, if CK1 leads CK2 because of jitter or differences in clock tree delays, then R2 clocks before R3. Because R2's output is changing while R3 is clocking, a race condition results.

Figure 22 Problem: Multiple Clock Trees

In [Figure 23](#), there is a resynchronization register or latch (SYNC) between R2 and R3, which is clocked by the opposite phase of the clock used for R2.

This design guideline is supported by DFT Compiler and is the default behavior of DFT Compiler.

Figure 23 Solution: Multiple Clock Trees

Clock Trees

Treat each clock tree as a separate clock source in designs that have a single clock input port but multiple clock tree distributions.

Sometimes a design has a single clock input port but uses multiple clock tree distributions to produce “early” and “standard” clocks, as shown in [Figure 24](#). Under these conditions, treat each clock tree as a separate clock source. Insert resynchronization latches between scan cells where the clock source switches from one clock tree to another, as shown in [Figure 25](#).

This design style is supported by DFT Compiler but is not the default method of DFT Compiler.

Figure 24 Problem: Single Clock With Multiple Clock Trees

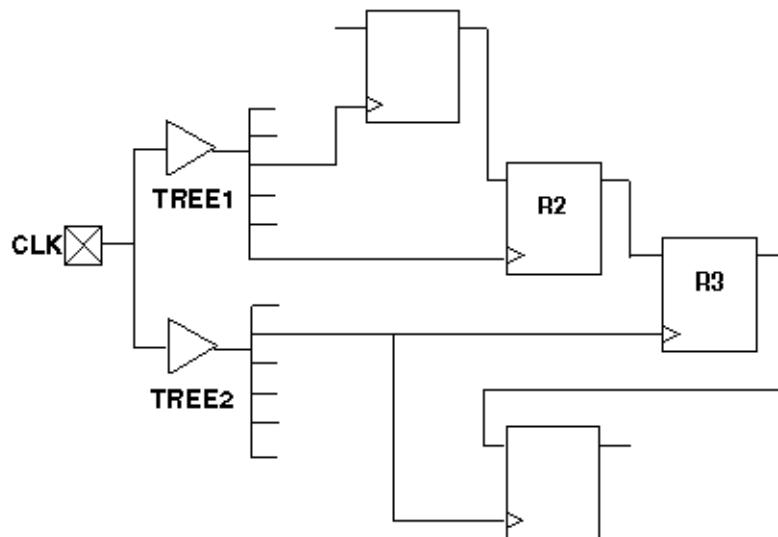
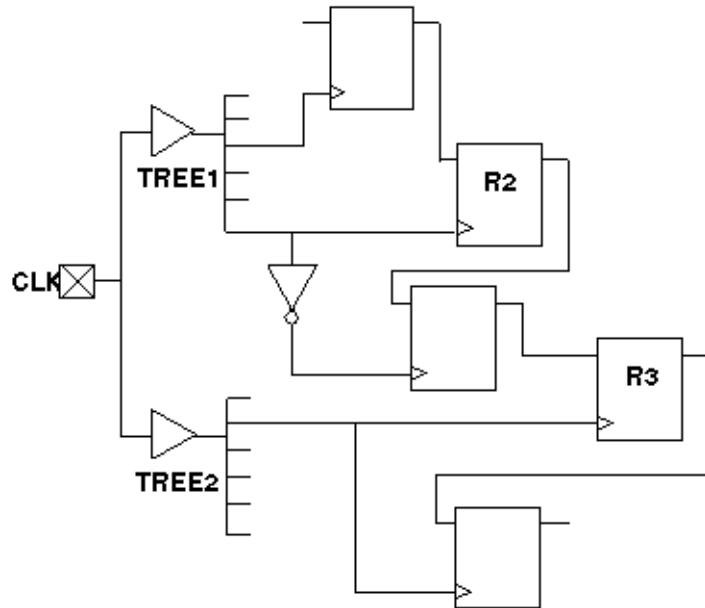


Figure 25 Solution: Single Clock With Multiple Clock Trees

Clock Flip-Flops

If possible, clock all flip-flops on the same scan chain on the same clock edge. If this is not possible, then group together all flip-flops that are clocked on the trailing clock edge and place them at the front of the scan chain (closest to the scan chain input); and group together all flip-flops that are clocked on the leading clock edge and place them closest to the scan chain output.

In [Figure 26](#), B1 and B2 are always loaded with the same data as A1 and A4, respectively, during scan chain loading, because they are clocked on the trailing edges. Thus, parts of the circuit that require A1 and B1 (or A4 and B2) to have opposite values are untestable.

In [Figure 27](#), the scan chain registers are ordered so that all of the trailing-edge cells are grouped together at the front of the scan chain. B1 and B2 can be set independently of A1 and A4.

This design guideline is automatically implemented by DFT Compiler if you allow it to mix clock edges on a scan chain.

Figure 26 Problem: Mixed Clock Edges on a Scan Chain

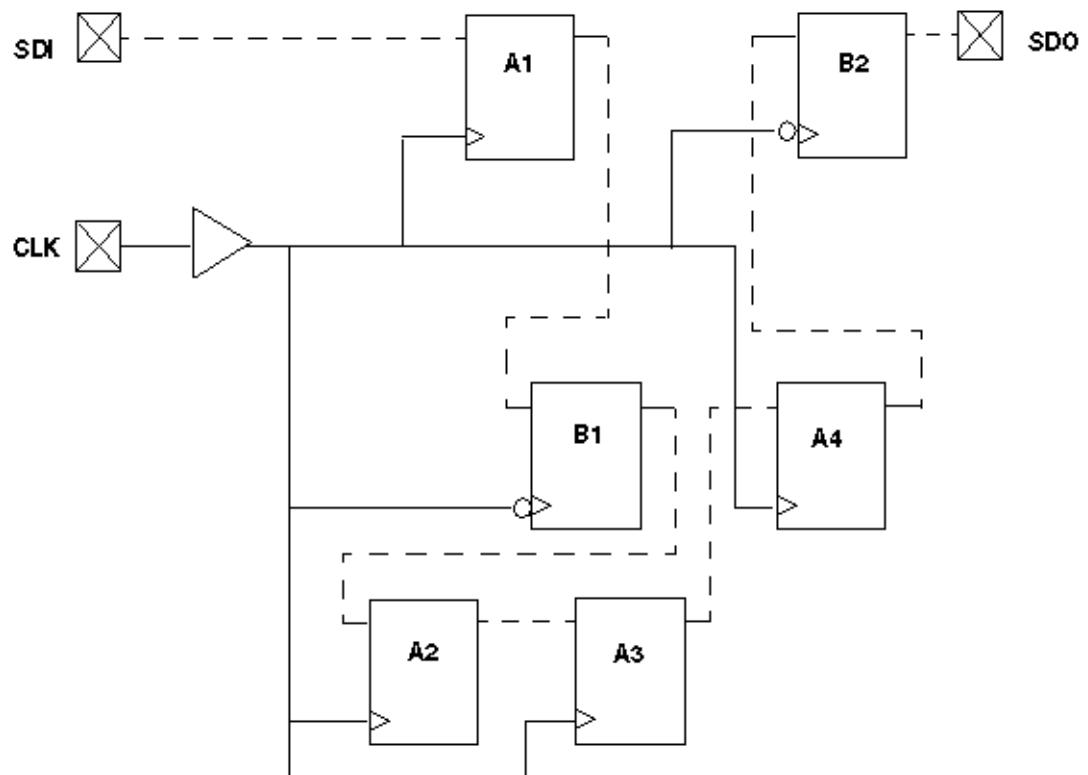
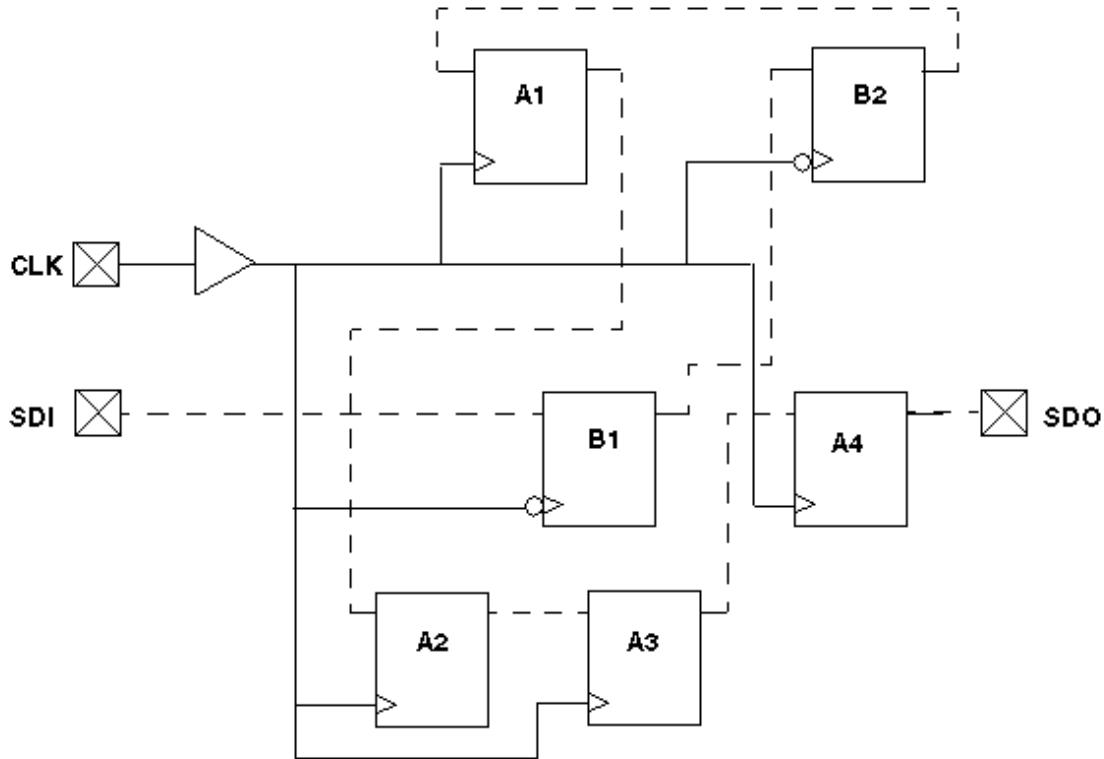


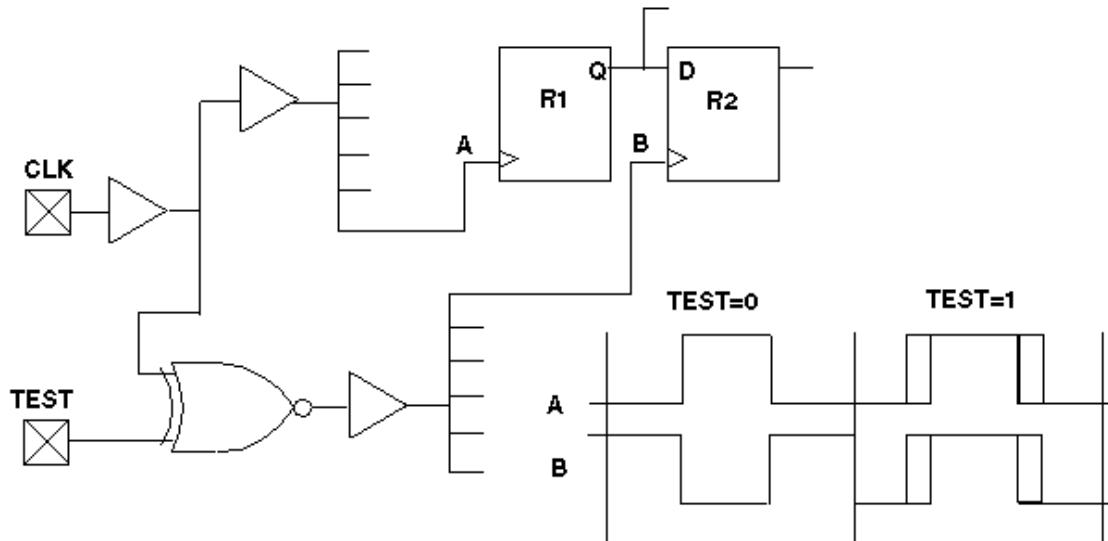
Figure 27 Solution: Mixed Clock Edges on a Scan Chain

XNOR Clock Inversion and Clock Trees

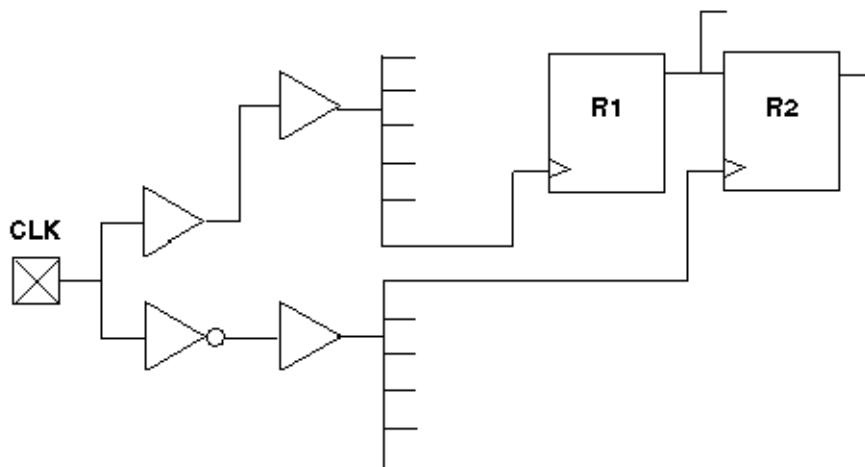
Do not mix XNOR clock inversion techniques and clock trees.

A common design technique when both edges of the same clock are used for normal operation of scan chain flip-flops is to use an XNOR in place of an INV to form the opposite clock polarity. Then, in test mode, the XNOR can be switched from an inverter into a buffer.

This technique is not advisable unless you can analyze the timing during test mode to ensure that no timing violations can occur during the application of any clocks. While in normal operation, there are essentially two clock zones of opposite phase. The phasing of the two clocks is such that reasonable timing is achieved between flip-flops that are on opposite phases of the clock. When one of the clocks is no longer inverted, two clock tree distributions are driven by the same-phase signal, resulting in timing-critical configurations in ATPG mode that do not exist in normal functional mode. See [Figure 28](#).

Figure 28 Problem: XNOR Clock Inversion and Clock Trees

To prevent this problem, replace the XNOR gate with an inverter, as shown in [Figure 29](#). If you need the XNOR function, use it locally in the vicinity of the affected gates, rather than on the input side of a clock tree.

Figure 29 Solution: XNOR Clock Inversion and Clock Trees

Protection of RAMs During Scan Shifting

To protect RAMs from random write cycles, disable the RAM write clock or write enable lines while shifting scan chains in ATPG test mode.

In ATPG test mode, RAMs must remain undisturbed by random write cycles while the scan chains are being shifted. You can accomplish this by disabling the write clock or write enable line

to each data write port during ATPG test mode. Often, the SCAN_ENABLE control is used for this function, coupled with an AND or OR gate, as appropriate.

However, to also achieve controllability over the write port, use a separate top-level input other than SCAN_ENABLE. The RAM write control is usually used as a pulsed port (RZ/RO), while the SCAN_ENABLE is a constant value (NRZ/NRO). Trying to achieve both simultaneously usually presents problems that can be avoided by using separate ports.

RAM and ROM Controllability During ATPG

If you want controllability of RAMs and ROMs for ATPG generation, connect their read and write control pins directly to a top-level input during ATPG test mode. This is most conveniently accomplished by using a MUX, which switches control from an internal to a top-level port. Multiple RAMs can share the same control port for the write port.

[Figure 30](#), if the registers are in scan chains, random patterns that occur while loading and unloading scan chains are written to the RAMs. Thus, the RAM contents are unknown and treated as X.

[Figure 31](#), MUX controls activated by TEST mode bring the write control signals up to the top-level input ports.

For achieving controllability and higher test coverage, direct control of write ports is more important than control of read ports.

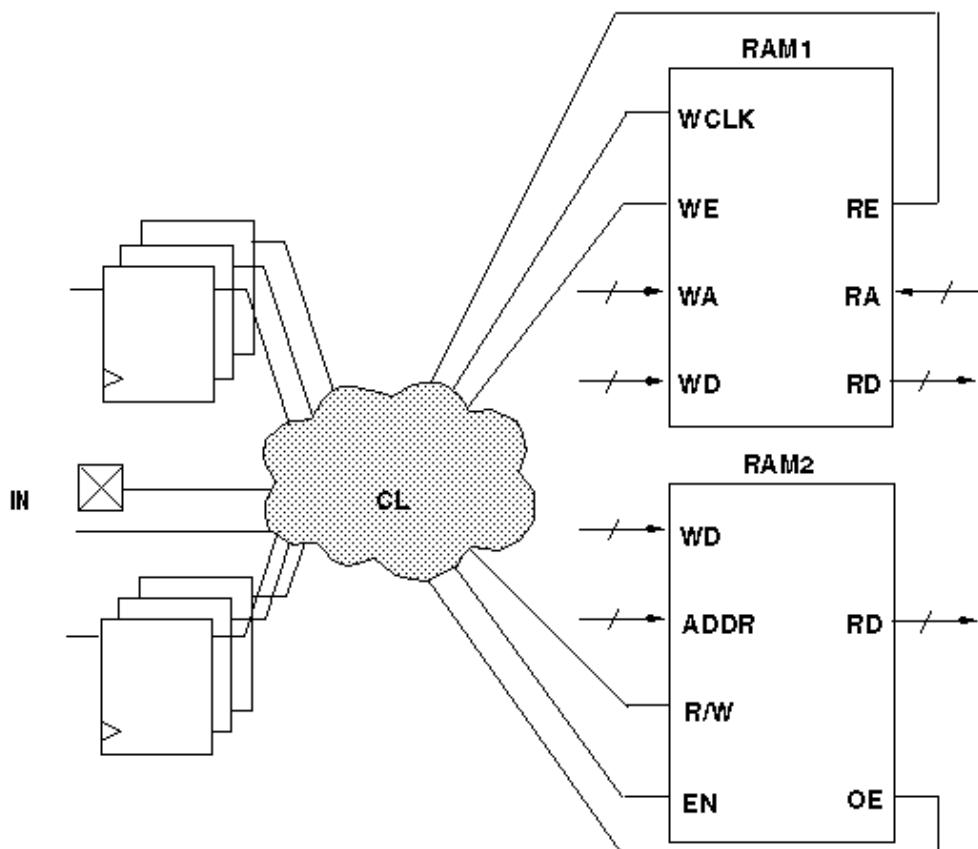
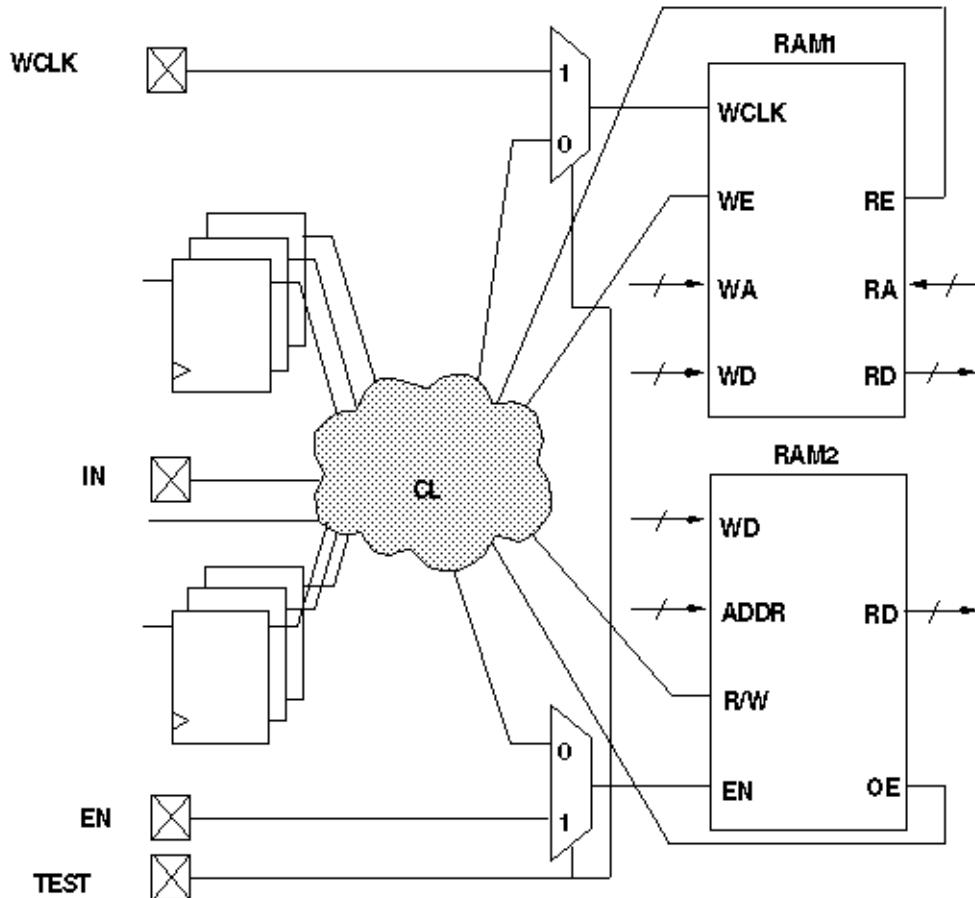
Figure 30 Problem: RAM/ROM Control

Figure 31 Solution: RAM/ROM Control

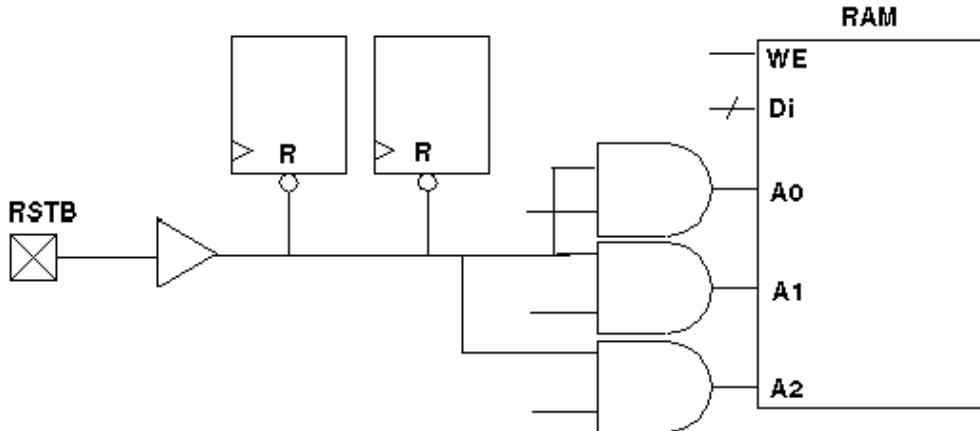
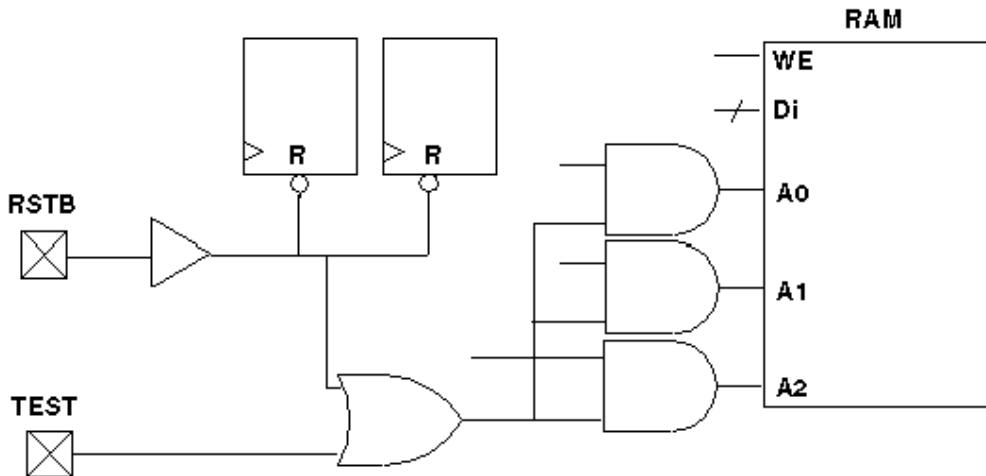
Pulsed Signal to RAMs and ROMs

Do not allow an open path from a pulsed signal to a data, address, or control input of a RAM or ROM (except read/write control) while in ATPG test mode.

If a combinational path exists from a defined clock or asynchronous set or reset port to a data, address, or control pin of a RAM or ROM, the ATPG process treats the memory device as filled with X. The exceptions are the read clock and write clock signals, which are operated in a pulsed fashion but should not be mixed with a defined clock.

In [Figure 32](#), the address or data inputs are coupled with a clock/set/reset port, so their values are not constant while capture clocks are occurring elsewhere in the design. The result is that RAM read and write data cannot be determined; Xs are used instead.

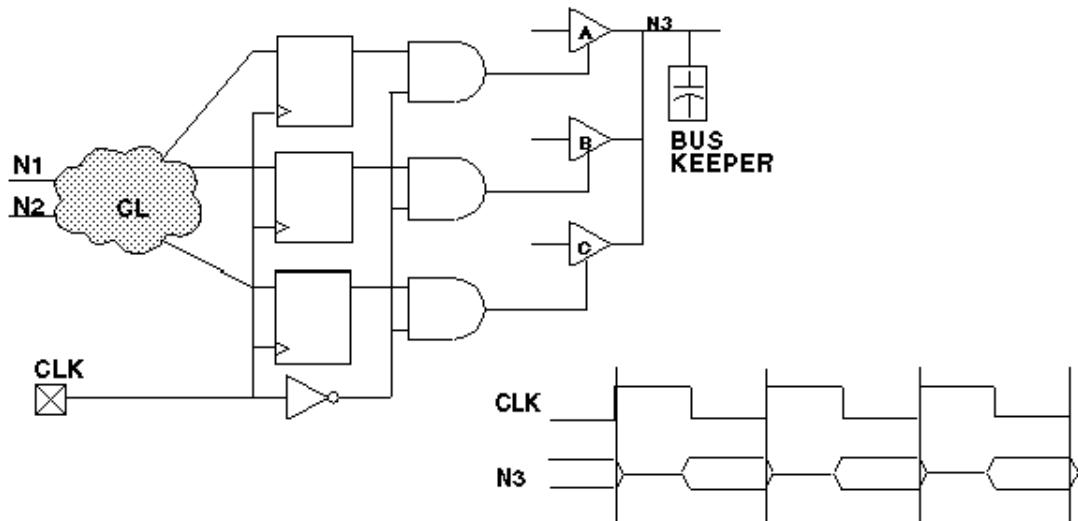
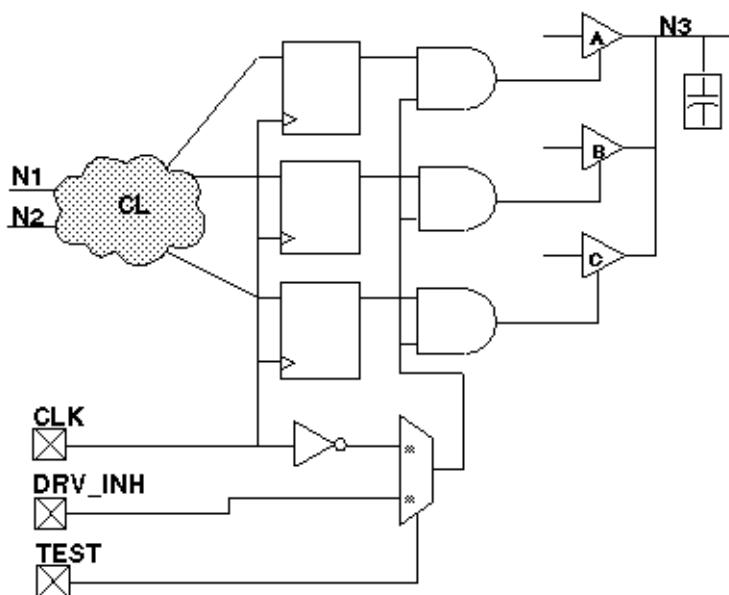
In [Figure 33](#), the TEST input disables pulsed paths during ATPG test mode.

Figure 32 Problem: RAMs/ROMs and Pulsed Signals**Figure 33 Solution: RAMs/ROMs and Pulsed Signals**

Bus Keepers

While in ATPG test mode, do not allow a combinational gate path from any pulsed port to drive the enable controls of three-state drivers that contribute to a multidriver net, as illustrated in [Figure 34](#). In [Figure 35](#), the TEST input redirects the control to a top-level port, and the port is constrained to a value that does not affect the driver enables. Then, on the tester and during simulation, the port is driven with the same signal as that on the original CLK port.

A common practice is to gate all internal driver enables with some phase of a clock so that all drivers are off during the first half of each cycle and one driver is on during the second half. This practice solves some contention problems that occur during the transition of one driver off to another driver on, but it renders bus keeper usage impossible in ATPG test mode.

Figure 34 Problem: Bus Keepers**Figure 35 Solution: Bus Keepers**

Non-Z State on a Multidriver Net

When using bus keepers, ensure a non-Z state on a multidriver net by the end of the `load_unload` procedure.

Non-Clocked Events

When using bus keepers, do not allow the non-clocked events that occur before the system capture clock to disturb the multidriver net.

The system capture cycle should not disturb the multidriver net, at least not until after the clock/set/reset pulse, unless a change on a primary input enables one of the drivers and drives a known value on the net.

When you use a bus keeper, you expect it to retain the last value driven on the bus. Therefore, you do not need to design the driver enable controls so that one driver is always on. However, if the DRC analysis of the bus keeper finds violations, the beneficial effects possible with a bus keeper are ignored.

When no driver is enabled on the multidriver net, the bus assumes a Z or X state. When a Z passes through some other internal gate, it becomes an X; thus, an internal source generates and propagates a multitude of X states to observe points (for example, output ports and scan cells), which must be masked off in the ATPG patterns. There is a significant increase in the number of pattern bits that the tester must mask off; thus, you can obtain patterns that are legal and generate high test coverage but are unusable on many testers because of the excessive number of compare masks required.

In [Figure 32](#), the address or data inputs are coupled with a clock/set/reset port, so their values are not constant while capture clocks are occurring elsewhere in the design. The result is that RAM read and write data cannot be determined; Xs are used instead.

In [Figure 33](#), the TEST input disables pulsed paths during ATPG test mode.

Figure 32 Problem: RAMs/ROMs and Pulsed Signals

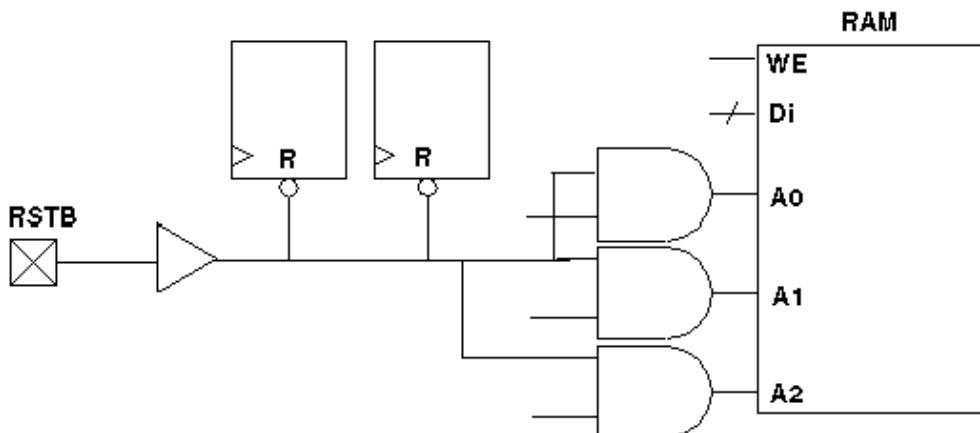
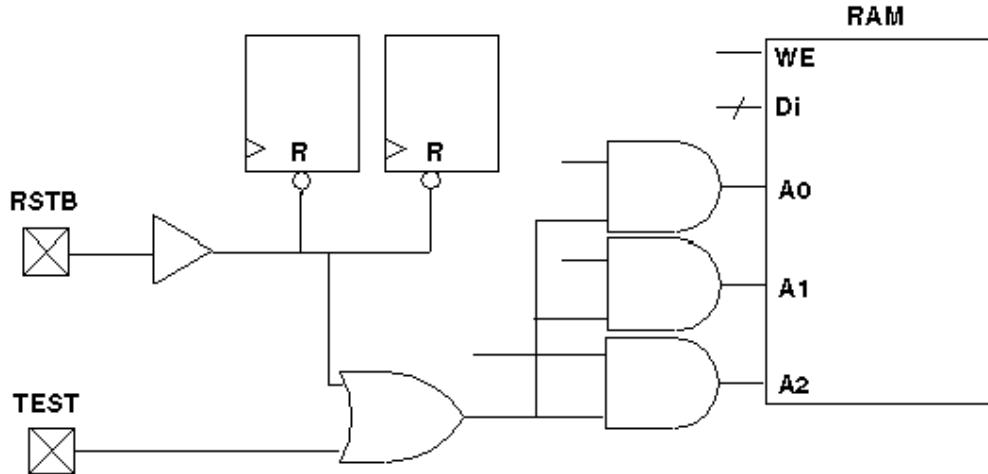


Figure 33 Solution: RAMs/ROMs and Pulsed Signals

Bus Keepers

While in ATPG test mode, do not allow a combinational gate path from any pulsed port to drive the enable controls of three-state drivers that contribute to a multidriver net, as illustrated in [Figure 34](#). In [Figure 35](#), the TEST input redirects the control to a top-level port, and the port is constrained to a value that does not affect the driver enables. Then, on the tester and during simulation, the port is driven with the same signal as that on the original CLK port.

A common practice is to gate all internal driver enables with some phase of a clock so that all drivers are off during the first half of each cycle and one driver is on during the second half. This practice solves some contention problems that occur during the transition of one driver off to another driver on, but it renders bus keeper usage impossible in ATPG test mode.

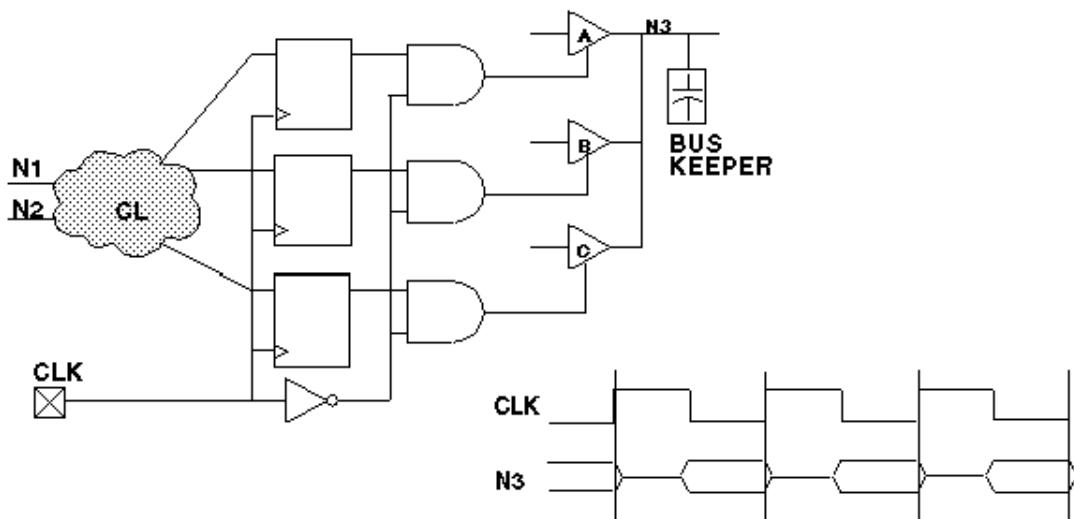
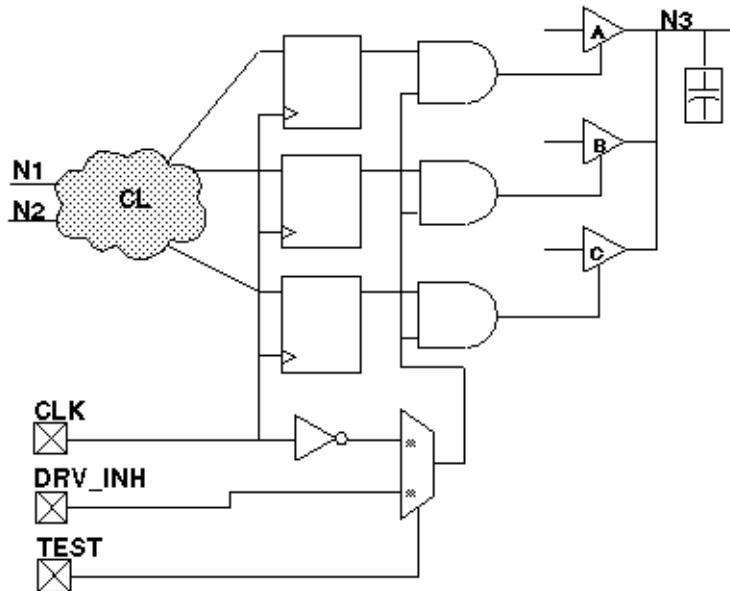
Figure 34 Problem: Bus Keepers

Figure 35 Solution: Bus Keepers

Non-Z State on a Multidriver Net

When using bus keepers, ensure a non-Z state on a multidriver net by the end of the `load_unload` procedure.

Non-Clocked Events

When using bus keepers, do not allow the non-clocked events that occur before the system capture clock to disturb the multidriver net.

The system capture cycle should not disturb the multidriver net, at least not until after the clock/set/reset pulse, unless a change on a primary input enables one of the drivers and drives a known value on the net.

When you use a bus keeper, you expect it to retain the last value driven on the bus. Therefore, you do not need to design the driver enable controls so that one driver is always on. However, if the DRC analysis of the bus keeper finds violations, the beneficial effects possible with a bus keeper are ignored.

When no driver is enabled on the multidriver net, the bus assumes a Z or X state. When a Z passes through some other internal gate, it becomes an X; thus, an internal source generates and propagates a multitude of X states to observe points (for example, output ports and scan cells), which must be masked off in the ATPG patterns. There is a significant increase in the number of pattern bits that the tester must mask off; thus, you can obtain patterns that are legal and generate high test coverage but are unusable on many testers because of the excessive number of compare masks required.

Checklists for Quick Reference

This section provides checklists of the design guidelines and port redefinition suggestions. Use the following checklists as a convenient reminder as you implement your design:

- [ATPG Design Guideline Checklist](#)
 - [Ports for Test I/O Checklist](#)
-

ATPG Design Guideline Checklist

Follow these guidelines during ATPG test mode:

1. Ensure that clocks and asynchronous set/reset signals come from a primary input.
 1. Do not use clock dividers.
 2. Do not use gated clocks.
 3. Do not use phase-locked loops (PLLs) as clock sources.
 4. Do not use pulse generators.
2. Provide complete control of clock paths to scan chain flip-flops.
 1. If a clock passes through a MUX, constrain the select line of the MUX to a constant value.
 2. If a clock passes through a combinational gate, constrain the other inputs of the gate to a constant value.
 3. Pass clock signals directly through JTAG I/O cells without passing through a MUX, unless the MUX control can be constrained.
 4. Avoid using bidirectional clocks and asynchronous set/reset ports.
3. Do not allow an open path from a pulsed input signal (clock or asynchronous set/reset) to a data-capture input of a sequential device.
 1. Do not allow a path from a pulsed input to both the data input and the clock of the same flip-flop.
 2. Do not allow a path from a pulsed input to both the data input and the asynchronous set or reset input of the same flip-flop.
4. For multidriver nets, ensure that only one driver is enabled during the shifting of scan chains.
5. Force all bidirectional ports to input mode while shifting scan chains, using a top-level port as control.
6. Force scan chain outputs that use bidirectional or three-state ports into output mode while shifting scan chains, using a top-level port (usually SCAN_ENABLE).
7. Use a single clock tree to clock all flip-flops in the same scan chain.
8. Treat each clock tree as a separate clock source in designs that have a single clock input port but multiple clock tree distributions.
9. Use the same clock edge for all flip-flops in the same scan chain.

10. Do not mix XNOR clock inversion techniques and clock trees.
 11. To protect RAMs from random write cycles, disable RAM write clock or write enable lines while shifting scan chains.
 12. Connect RAM and ROM read and write control pins directly to a top-level input while in ATPG test mode.
 13. Do not allow an open path from a pulsed signal to a RAM's or ROM's data, address, or control inputs (except read/write control).
 14. Do not allow a combinational gate path from any pulsed port to drive the enable controls of three-state drivers that contribute to the multidriver net.
 15. When using bus keepers, ensure a non-Z state on multidriver nets by the end of the scan chain load/unload.
 16. When using bus keepers, do not allow the nonclocked events that occur before the system capture clock to disturb the multidriver net.
-

Ports for Test I/O Checklist

Follow these port usage guidelines:

1. A port that already feeds the input of a flip-flop in a scan chain is the best port to redefine as a scan chain input.
2. A port that already comes directly from the output of a flip-flop in a scan chain is the best port to redefine as a scan chain output.
3. A three-state output can be redefined as a bidirectional port and used in input mode while TEST is asserted.
4. A standard output can be redefined as a bidirectional port and used in input mode while TEST is asserted.
5. An input port that feeds directly into the input of a flip-flop in a scan chain can be redefined as a shared port.
6. An output port that comes directly from the output of a flip-flop in a scan chain can be redefined as a bidirectional port and used in input mode while TEST is asserted.
7. An output port that is a derived clock or pass-through clock can be redefined as a bidirectional port and used in input mode while TEST is asserted.
8. An input port that has a small amount of fanout before entering a flip-flop is a good choice to be redefined for use as a test-related input while TEST is asserted.

C

Importing Designs From DFT Compiler

This appendix provides a brief overview of how to take a design from DFT Compiler to TetraMAX ATPG to generate ATPG patterns.

Before you begin, you should be aware of the differences between DFT Compiler and TetraMAX ATPG in the treatment of your design. These are the main differences:

- Bidirectional port timing
- Ordering of events within an ATE cycle
- Latch models
- Support for DFT Compiler commands in TetraMAX ATPG

These differences are explained in detail in the section “Exporting a Design To TetraMAX ATPG” in the *DFT Compiler Scan User Guide* (provided with the DFT Compiler tool).

Before you import a design from DFT Compiler to TetraMAX ATPG, you need to ensure that the design has valid scan chains and does not have design rule violations. These are the steps to import the design:

1. Before doing any work with DFT Compiler (including scan insertion), set the test timing variables to the values specified by your ASIC vendor.
2. Identify the netlist format that you are exporting to TetraMAX ATPG, using the `test_stil_netlist_format` environment variable.
3. To guide netlist formatting, set the environment variables that affect how designs are written out.
4. If you want to pass capture clock group information to TetraMAX ATPG, set the `test_stil_multiclock_capture_procedures` variable to true, and use the `check_test (not check_scan)` command in the next step.
5. Check for design rule violations and fix any violations.
6. Write out the netlist.
7. Write out the STL procedure file.

After you perform these steps, you can read in the design with TetraMAX ATPG. For more information on performing these steps, see the section “Exporting a Design To TetraMAX ATPG” in the *DFT Compiler Scan User Guide*.

D

Utilities

The following sections of this appendix describe the utility programs supplementing TetraMAX ATPG:

- [Ltran Translation Utility](#)
- [Generating PrimeTime Constraints](#)
- [Converting Timing Violations Into Timing Exceptions](#)

Ltran Translation Utility

When you use the Write Patterns dialog box or the `write_patterns` command to write patterns in the FTDL, TDL91, TSTL2, or WGL_FLAT format, TetraMAX ATPG invokes a separate translation process called Ltran. This translation process runs independently in a new window. You can optionally launch Ltran in the shell mode or use an Ltran configuration file to control the output format.

The following sections provide basic information on starting Ltran in the shell mode, and specifying and modifying Ltran configuration files:

- [Ltran in the Shell Mode](#)
- [FTDL, TDL91, TSTL2 Configuration Files](#)
- [Understanding the Configuration File](#)
- [Configuration File Syntax](#)

Note: If there is a problem with your Ltran installation, the following error message might appear:

```
sh: /stil2wgl.sh: No such file or directory
```

In this case, you should make sure the following files are in your `$installation/$platform/syn/ltran` directory:

- Ltran
- Ltran.sh
- gzip stil2wgl
- stil2wgl.sh
- vread.bin
- vread3.bin
- vread5.bin

If these files are not in this directory, you should go to the SolvNet Download Center, obtain the download instructions for TetraMAX ATPG, and perform the installation.

Ltran in the Shell Mode

Ltran launches in the shell mode one of two ways:

- If you have not set the `DISPLAY` environment variable (which is common when you use a telnet session)
- If you have set the `LTRAN_SHELL` environment variable to 1

When using Ltran in the shell mode, the execution of TetraMAX ATPG stops until Ltran finishes. This is different than the xterm version that kept TetraMAX ATPG running and allowed parallel Ltran runs; for example, if you try writing files with the `-split` option of the `write_patterns` command, which causes the intermediate file created and passed to the external translator to

use the STIL pattern format (the default is to use WGL as the intermediate file). Now, each Ltran runs sequentially.

Since this is a third-party interface, any output from Ltran in GUI or shell mode might appear in the UNIX transcript in which TetraMAX ATPG was started, but that output will not be captured in the TetraMAX log file.

Linux platforms need the path to the xterm executables. These are located in the /usr/X11R6/bin directory. After you add this to your search path, you can write out TDL91, TSTL2, and FTDL pattern formats from the TetraMAX Linux shell. This is especially true if you receive a "sh : xterm: command not found" message.

FTDL, TDL91, and TSTL2 Configuration Files

If you select FTDL, TDL91, or TSTL2 as the format in the Write Patterns dialog box or in the `write_patterns` command, a separate Ltran translation process is executed. This process begins as an independent operation in the new window. You can perform other tasks while the translation process is carried out.

The Write Patterns dialog box and `write_patterns` command optionally let you specify an Ltran configuration file to be used for controlling the output format. If you do not specify a configuration file, a default file is used from the following directory:

`$SYNOPSYS/auxx/syn/ltran`

For a discussion about the use of the `SYNOPSYS_TMAX` environment variable, see "[Specifying the Location for TetraMAX Installation](#)".

The configuration files contained in this directory are:

- `stil2ftdl` : STIL to FTDL
- `stil2tdl91` : STIL to TDL91
- `stil2tstl2` : STIL to TSTL2
- `wgl2ftdl` : WGL to FTDL
- `wgl2tdl91` : WGL to TDL91
- `wgl2tstl2` : WGL to TSTL2

By default, when you use the `write_patterns` command and specify FTDL, TDL91, and TSTL2 as the pattern format, the pattern generator first generates patterns in an intermediate STIL format file. Ltran then translates the STIL patterns to the target format using the conversion parameters specified in the `stil2ftdl`, `stil2tdl91`, or `stil2tstl2` configuration file.

Note: You can also use the `-wgl` option to generate an intermediate WGL format file, and Ltran will use the provided WGL-related configuration files. However, in most cases, writing a STIL intermediate file is much faster than writing WGL file; this can save minutes of time for large pattern files.

The default files are adequate for most translations. However, you can modify a number of fields in the files to customize the output results. These user-editable fields are part of the simulator command and are identified with comments in the Ltran configuration files. All of these fields are optional and can be commented out with curly braces "{}". Most of these fields provide a way to specify header information in the output file, as summarized in the sections that follow.

To use a customized configuration file, copy one of the existing files to your own local directory, and then edit your copy to adjust the user-editable fields and controls. In the Write Patterns dialog box or `write_patterns` command, specify the name of your modified configuration file.

Understanding the Configuration File

Each configuration file contains two mandatory command blocks (`OVF_BLOCK` and `TVF_BLOCK`) and one optional command block (`PROC_BLOCK`).

The commands in the mandatory command block `OVF_BLOCK` describe the format of data in the original vector file. The commands in the mandatory command block `TVF_BLOCK` provide instructions for formatting vectors in the target vector file. The commands in the optional command block `PROC_BLOCK` describe other processing required to translate the data in the original vector file into the target vector file.

The configuration file structure can be summarized as shown in [Example 1](#).

Example 1 Translation Configuration File Structure

```

OVF_BLOCK
    BEGIN
        OVF_BLOCK_COMMANDS
    END
PROC_BLOCK {Optional}
    BEGIN
        PROC_BLOCK_COMMANDS
    END
TVF_BLOCK
    BEGIN
        TVF_BLOCK_COMMANDS
    END
END

```

The configuration file is not case-sensitive. Pin names retain their case in the translation to the target vector file. Pin names can contain any printable ASCII characters (but not spaces), including any of the following characters:

, ; < > [] { } () = \ & | @

For the full syntax of the `OVF_BLOCK`, `PROC_BLOCK`, and `TVF_BLOCK` command blocks, see ["Configuration File Syntax"](#).

Customizing the FTDL Configuration File

For FTDL output, the `write_patterns` command uses the `wgl2ftdl` configuration file. You can customize the configuration file by editing the following parameters:

- `-AUTO_GROUP`

This optional switch tells the `write_patterns` command to algorithmically identify similar signals and group them in the FTDL output file.

Revision number

- `REVISION = "0001", { edit "0001" as required }`

- Designer name**
- DESIGNER = "Designer", { edit "Designer" as required }
- Test vector function**
- TNAME = "FUNC", { edit "FUNC" as required }
- Test vector name**
- CNAME = "TEST", { edit "TEST" as required }
- Date of design file creation**
- DATE = "99/10/05" ; { edit DATE as required }

Customizing the TDL91 Configuration File

For TDL91 output, the `write_patterns` command uses the `wgl2tdl91` configuration file. You can customize the configuration file by editing the following parameters:

- Library**
- LIBRARY_TYPE = "Library", { edit "Library" as required }
- Customer**
- CUSTOMER = "Customer", { edit "Customer" as required }
- Part number**
- TI_PART_NUMBER = "PartNum", { edit "PartNum" as required }
- Pattern set name**
- PATTERN_SET_NAME = "SetName", { edit "SetName" as required}
- Pattern set type**
- PATTERN_SET_TYPE = "SetType", { edit "SetType" as required}
- Revision number**
- REVISION = "1.00", { edit REVISION as required }
- Date of design file creation**
- DATE = "10/5/2009" ; { edit DATE as required }

You can also do the following:

- Specify the following general Ltran configuration commands:
 - `-AUTO_GROUP` — Enables Ltran to algorithmically identify similar signals and group them in the TDL_91 pattern output file.
 - `SD_PORT = "SD"` — Enables you to specify a port name to be added to the end of each scan cell name to form the scan cell shift input pin name. The default port name is "SD". If you set this to a null string, then no text is added.

Reference your custom configuration file when creating patterns with the `write_patterns` command. Exclude the scan chain test when writing TDL91 patterns, for example:

- TEST-T> `write_patterns <pattern_file_name> -format td191 \ -config_file spec_CUSTOM_FILE -exclude chain_test -replace`
- When translating STIL/WGL files an additional flag can be set in the TABULAR_FORMAT statement which instructs Ltran to look for Header information at the beginning

of the STIL/WGL file and pass it through to the TDL_91 output file. This flag is -TDL91_INFO and is used as follows:

```
TABULAR_FORMAT stil -cycle, -scan, -include_cells, -TDL91_INFO ;
;
```

OR

```
TABULAR_FORMAT wgl -cycle, -scan, -include_cells, -keep_
annotations,
-TDL91_INFO ;
```

Note that this is only applicable to translations from STIL/WGL files generated by TetraMAX ATPG to TDL_91 format.

Customizing the TSTL2 Configuration File

For TSTL2 output, the `write_patterns` command uses the `wgl2tstl2` configuration file. You can customize this configuration file by editing the following parameters:

- Title

```
TITLE = "TITLE",      { edit "TITLE" as required }
```

- Function Test

```
FUNCTEST = "FC1"      { edit "FC1" as required }
```

- Scale

```
scale 1000;
```

Place the `scale` statement in the `PROC_BLOCK` section of the `stil2tstl2` or `wgl2tstl2` configuration file. The `scale 1000` statement in the previous example adjusts the scaling and resolution from the default, in nanoseconds (ns), to picoseconds (ps).

For example, take a signal defined as follows:

```
"rst" { P { '0ps' U; '50006ps' D; '52400ps' U; } }
"rst" { P { '0ps' U; '50001ps' D; '52600ps' U; } }
"rst" { P { '0ps' U; '45000ps' D; '55000ps' U; } }
```

With the scale value set to 1000, the TSTL2 output is as follows:

```
TIMESET(2) NP, 50006, 2394 ;
TIMESET(2) NP, 50001, 2599 ;
TIMESET(2) NP, 45000, 10000 ;
```

Additional Controls

In addition to the simulator adjustments just described, most of the configuration files have two Ltran controls that you can use to further customize the format of the pattern output files:

- `rename_bus_pins`
- `header nn`

If these controls are supported, they appear commented out by default but can be activated by removing the curly braces "{}" surrounding them.

This is the syntax of the `rename_bus_pins` control:

```
rename_bus_pins $bus$vec;
```

The `rename_bus_pins` control flattens bused signal names. With this command, a bus signal name like `bus[5]` becomes `bus5`. The form of the mapped name can be controlled by changing the `busvec` string. For example:

```
rename_bus_pins $bus$_$vec_;
```

This example maps `bus[5]` into `bus_5_`.

The `header` control tells Ltran to place the names of signals in a vertical list as comments above their column position in the vectors. This control has the following syntax

```
header nn;
```

where `nn` is an integer that specifies how often to repeat the pin header listing, expressed as a number of lines.

Configuration File Syntax

The following sections describe the syntax of the statements in the `OVF_BLOCK`, `PROC_BLOCK`, and `TVF_BLOCK` command blocks.

OVF_BLOCK Statements

`AUX_FILE [=] "filename";`

Used to specify an auxiliary file for some canned readers.

`BEGIN_LINE [=] n;`

Used to define the line number in the OVF file at which VTRAN should begin processing vectors.

`BEGIN_STRING [=] "string";`

Used to define a unique text string in the OVF file after which VTRAN should begin processing vectors.

`BIDIRECTS [=] pin_list;`

Defines the names and order of pins in the OVF file that are bidirectional.

`BUSFORMAT radix; or BUSFORMAT pin_list = radix;`

Specifies the radix of buses in the OVF file.

`CASE_SENSITIVE;`

Allows there to be more than one signal with the same name spelling but differing only in case of letters in the name.

`GROUP n [=] pin_list;`

Together with the `$gstatesn` keyword, it tells VTRAN how the pin states are organized.

`INPUTS [=] pin_list;`

Defines the names and order of input pins in OVF file.

`MAX_UNMATCHED [=] n [verbose]:`

Specifies the number of, and information contained in, warnings for lines in the OVF file that does not a `format_string`.

`ORIG_FILE [=] "filename";`

Used to specify the OVF file name to be translated.

`OUTPUTS [=] pin_list;`

Defines the names and order of output pins in the OVF file.

```
SCRIPT_FORMAT [=] "format#1" [, . ."format#n"] ;
```

Format descriptors for User-Programmed reader.

```
TABULAR_FORMAT [=] "format #1" [, . ."format#n"] ;
```

Format descriptors for User-Programmed reader.

```
TERMINATE TIME [=] n; or
```

```
TERMINATE LINE [=] m; or
```

```
TERMINATE STRING [=] "string";
```

Defines where in the OVF to stop processing, at a certain time, line number or when a string is reached.

```
WAVE_FORMAT [=] "format #1" [, . ."format#n"] ;
```

Format descriptors for User-Programmed reader.

```
WHITESPACE [=] 'a','b', 'c', . . . , 'n';
```

Defines characters in the OVF file that are to be treated as though they are space (they are ignored).

PROC_BLOCK Statements

```
ADD_PIN pinname = state1 [WHEN expr=state2, OTHERWISE state3];
```

Tells VTRAN to add a new pin to the TVF file, and allows you to define the state of this pin.

```
ALIGN_TO_CYCLE [-warnings] cycle pin_list @ time, . . . ,  
pin_list  
@ time ;
```

Vectors can be mapped to a set of cycle data, the state of each pin in a given cycle is determined by its state at a specified strobe time in the OVF file.

```
ALIGN_TO_STEP [-warnings] step [offset];
```

Forces a minimum time resolution in the TVF file.

```
AUTO_ALIGN [-warnings] cycle;
```

Collapses print-on-change data in the OVF file to cycle data by computing strobe points from information given in the PINTYPE commands.

```
BIDIRECT_CONTROL pin_list = dir WHEN expr = state ;
```

Separates input data from output data on bidirectionals under control of a pin state or logical combination of pin states.

```
BIDIRECT_CONTROL pin_list = direction @ time ;
```

Separates input data from output data on bidirectionals based upon when the state transitions occur.

```
BIDIRECT_STATES INPUT state_list, OUTPUT state_list ;
```

Separates input data from output data on bidirectionals where unique state characters identify pin direction.

```
CYCLE [=] n;
```

Specifies the time step between vectors in the OVF when the format of the vectors does not include a time stamp.

```
DISABLE_VECTOR_FILTER;
```

Disables filtering of redundant vectors.

```
DONT_CARE 'X';
```

Defines the character state to which output pins should be set outside of their check windows.

```
EDGE_ALIGN pinlist @ rtime [,ftime] [xtime];
```

Modifies pin transition times by snapping them to predefined positions within each cycle.

```
EDGE_SHIFT pinlist @ rtime [,ftime] [,xtime];
```

Modifies pin transition times by shifting them by fixed amounts.

```
MASK_PINS [mask_character ='X'] [pin_list] @ t1, t2 [-CYCLE]
; or
MASK_PINS [mask_character ='X'] [pin_list] @ CONDITION expr =
state ;
```

Masks the state of specified pins to the mask_character within the time range between t1 and t2, or when a specified logic condition exists on other pins.

```
MERGE_BIDIRECTS state_list ; or
MERGE_BIDIRECTS rules = n ;
```

Merges the input and output state information of a bidirectional pin to a single pin after it has been split and processed.

```
PINTYPE pintype pin_list @ start1 end1 [start2, end2] ;
```

Defines the behavior and timing to be applied to input or output pins during translation.

```
POIC;
```

Specifies that vectors in the OVF file should be translated to the TVF only when at least 1 input pin has changed in the vector.

```
SCALE [=] nn;
```

Linearly expands or reduces the time line of the OVF. Happens before any timing modifications.

```
STATE_TRANS [=] [dir] 'from1'-'>'to1', . . . ;
```

Tells VTRAN not to incorporate pin timing and behavior into the vectors themselves.

```
SEPARATE_TIMING;
```

Defines a mapping from pin states in the OVF file to states in the TVF file.

```
STATE_TRANS_GROUP pin_list = 'from1'-'>'to1', . . . ;
```

Supplements the STATE_TRANS command by providing state translations on an individual pin or group basis.

```
TIME_OFFSET [=] n ;
```

When reading the vectors from the OVF file, the time stamp can be offset by an arbitrary amount.

TVF_BLOCK Statements

```
ALIAS ovf_name = tvf_name, . . . ; or
ALIAS "ovf_string"="tvf_string";
```

Provides a way to change the names of pins listed in the OVF file, for listing in the TVF file.

```
BIDIRECTS [=] pin_list;
```

Defines the names and order of pins to be listed in the TVF file which are bidirectional.

```
BUSFORMAT radix; or
BUSFORMAT pin_list = radix;
```

Specifies the radix of buses in the TVF file.

```
COMMAND_FILE [=] "filename";
```

Specifies the name of a separate output command file for the target simulator, in addition to the vector data file.

`DEFINE_HEADER [=] "text string";`

Inhibits the automatic generation of headers and replaces it with a custom text string.

`HEADER [=] n;`

Causes a vertical list of the pin names to appear as comments in the TVF every *n* vector lines.

`INPUTS [=] pin_list ;`

Defines the names and order of pins to be listed in the TVF file which are inputs.

`INPUTS_ONLY;`

Causes only input and the input versions of bidirectional pins to be listed in the TVF.

`LOWERCASE;`

Forces all pin names in the TVF file to use lowercase letters.

`OUTPUTS [=] pin_list ;`

Defines the names and order of pins to be listed in the TVF file that are outputs.

`OUTPUTS_ONLY;`

Causes only output and the output versions of bidirectional pins to be listed in the TVF file.

`RENAME_BUS_PINS format;`

Provides a way of globally modifying all bus names in the TVF file.

`RESOLUTION [=] n;`

Specifies the resolution of time stamps in the output vector file (*n* = 1.0, 0.1, 0.01 or 0.001).

`SCALE [=] nn ;`

Linearly scales all times to the TVF file.

`SIMULATOR [=] name [param_list];`

Defines the target vector file format to be compatible with the simulator named.

`STOBE_WIDTH [=] n;`

Used with several of the simulator interfaces to define the width of an output strobe window.

`SYSTEM_CALL ". . .text . . . ";`

Upon completion of translating vectors from the OVF file to the TVF file, VTRAN sends this text string to the system just before termination.

`TARGET_FILE [=] "filename";`

Specifies the name of the output file.

`TITLE [=] "title";`

Specifies a special character string to be placed in the header of certain simulator vector files.

`UPPERCASE;`

Forces all pin names in the TVF to be listed with uppercase letters.

Generating PrimeTime Constraints

You can use the `tmax2pt.tcl` script to generate PrimeTime constraints for performing static timing analysis of a design under test. This script extracts relevant data and creates a PrimeTime script that constrains the design in test mode.

Although this flow simplifies the process of performing static timing analysis with PrimeTime, it is no substitute for the experienced user to validate timing analysis. See the *PrimeTime*

Fundamentals User Guide and the *PrimeTime Advanced Timing Analysis User Guide* for these details.

The following sections describe how to generate PrimeTime Constraints:

- [Input Requirements](#)
 - [Starting the Tcl Command Parser Mode](#)
 - [Setting Up TetraMAX](#)
 - [Making Adjustments for OCC Controllers](#)
 - [Performing an Analysis for Each Mode](#)
 - [Implementation](#)
-

Input Requirements

The TetraMAX input data requirements are:

- Netlists
- Library
- STIL procedure file
- Tcl command script for `build`, `run_drc` commands, and so on.
- An image file can only be used if it is written using the command `write_image - netlist_data`.

The PrimeTime input data requirements are:

- Netlists
- Technology library (.db files)
- Command scripts to read design, link, and so on
- Timing models
- Layout data (for example, SDF)

Starting the Tcl Command Parser Mode

To use this flow, you must run the tool in Tcl command parser mode, which is the default mode for TetraMAX ATPG starting with the C-2009.06 release.

The command files must be in Tcl format and not in the native format. You can use the TetraMAX command translation script, `native2tcl.pl` to convert native mode TetraMAX command scripts into Tcl command scripts. For instructions on how to download this script, see [“Converting TetraMAX Command Files to Tcl”](#).

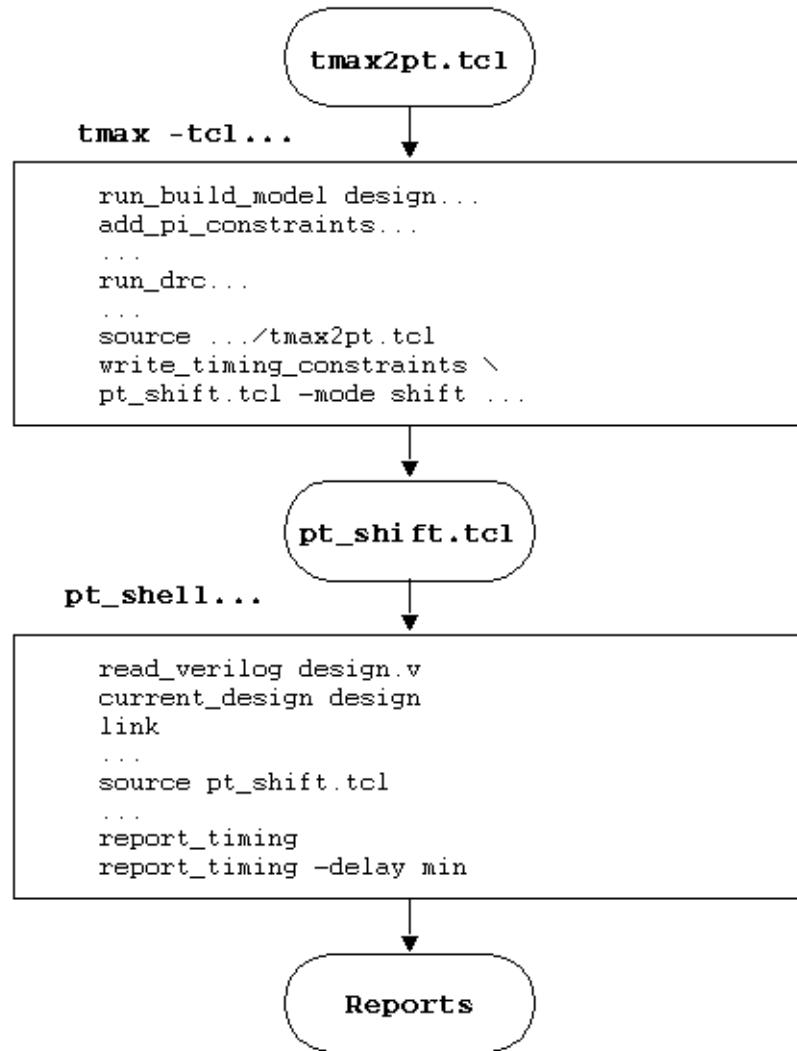
Setting Up TetraMAX

The normal flow of configuring the design and TetraMAX ATPG for ATPG is required. However, ATPG does not have to be run. After you run DRC and set the configuration, TetraMAX ATPG has enough data to support generating the PrimeTime script.

The `tmax2pt.tcl` script is located in the `$SYNOPSYS/auxx/syn/tmax` directory. This script must be sourced from TetraMAX ATPG (see [Figure 1](#)); for example:

```
TEST-T> source $env(SYNOPSYS)/auxx/syn/tmax/tmax2pt.tcl
```

Figure 1 Shift Mode Analysis Example



The `write_timing_constraints` procedure is part of the `tmax2pt.tcl` script. Use this procedure to create a PrimeTime Tcl script. For example:

```
TEST-T> write_timing_constraints pt_shift.tcl -mode shift
...
```

The syntax for the `write_timing_constraints` procedure is as follows:

```
write_timing_constraints output_pt_script_file
[-debug]
[-man]
[-mode shift | capture | last_shift | update]
```

```

[-no_header]
[-only constrain_scanouts]
[-replace]
[-wft wft_name | default | launch | capture| launch_capture
[-wft wft_name | default | launch | capture | launch_capture] ]
[-unit ns | ps]

```

Argument	Description
-debug	Writes additional debug data into the output file. This is useful if you are attempting to modify this script.
-man	Displays a detailed description of the <code>write_timing_constraints</code> options.
-mode <i>mode_name</i>	Specifies the mode in which to perform timing analysis.
-no_header	Suppresses header information in the output file. This is useful for comparing the results of different versions.
-only constrain_scanouts	Sets output delay constraints only on scanout ports. By default, all outputs are constrained. This option is only compatible with the <code>-mode shift</code> option.
-replace	Overwrites the output PrimeTime script file, if it exists.
-wft <i>wft_name</i>	Specifies the WaveformTable as defined in the STIL protocol file from which the timing data is gathered. If well-known WFT names are defined, they can be abbreviated as follows: default (_default_WFT_), launch (_launch_WFT_), capture (_capture_WFT_), launch_capture (_launch_capture_WFT_). This option can be specified two times, if necessary.
-unit <i>unit</i>	Specifies ps (picoseconds) if the protocol uses ps. The default is ns (nanoseconds).

The `write_timing_constraints` procedure and options accept abbreviations.

The `mode_name` can be either `shift`, `capture`, `last_shift`, or `update`. Shift mode uses the constraints from the `load_unload` procedure and configures the design to analyze timing during scan chain shifting. Capture mode (the default) uses constraints from the `capture` procedures and configures the design to analyze timing during the capture cycles. Last_shift mode analyzes the timing of the last shift cycle and the subsequent capture cycle. This is

normally used for analyzing the last shift launch transition pattern timing. Update mode analyzes the timing of the last shift cycle, capture cycle, and first shift cycle to determine the timing of the DFTMAX Ultra cache registers or the DFTMAX shift power groups control chain latches.

The `-wft` option causes the timing used for the analysis to be specified separately from the mode specification. The argument to the `-wft` option must be a valid WaveformTable in the SPF. Well-known WFT names can be abbreviated. You can use the `-wft` option one time or twice in a single command. If two WFTs are specified, two cycles are timed. The first WFT is used for the first cycle timing and the second WFT is used for the second cycle. Two-cycle analysis is done by superimposing two cycles, offset by a period, for each clock. The default WFT name is `._default_WFT_`.

You should call the `write_timing_constraints` procedure for each mode. A separate script is created for each mode, and sourced in PrimeTime during separate sessions.

The following examples show the usage of the `-mode` and `-wft` options.

To validate shifting:

```
-mode shift -wft _slow_WFT_
```

To validate stuck-at capture cycles:

```
-mode capture -wft default
```

To validate system clock launch capture cycles for transition faults:

```
-mode capture -wft launch -wft capture
```

To validate the timing between shift and capture for transition faults:

```
-mode last_shift -wft default -wft _fast_WFT_
```

Note the following:

- The 'force PI' and 'measure PO' times are relative to virtual clocks in PrimeTime. The 'force PI' virtual clock rises at 0, and the 'measure PO' clock falls at the earliest PO measure time. Input and output delays are specified relative to these clocks.
- For the two WFT modes, all the clock ports will have two superimposed clocks representing the two cycles that need to be analyzed.
- The end-of-cycle measures produce cycle times of double the normal cycles to account for the expansion of vectors into multiple vectors.
- You should carefully review the generated PrimeTime script to ensure the static timing analysis configuration is as expected.
- In PrimeTime, the flow of setting up the design does not change. The design, SDF, parasitics, and so forth are read. Next, the script generated by `write_timing_constraints` in TetraMAX ATPG is sourced in PrimeTime; for example:

```
pt_shell> source pt_shift.tcl
```

Making Adjustments for OCC Controllers

If you source the script written by the `write_timing_constraints` procedure inside the `pt_shell`, and an internal clock source (for example, `OCC_controller_clock_root`) is included, the following message is echoed:

```
TMAX2PT WARNING: Internal clock OCC_controller_clock_root timing  
is defaulted  
Adjust this timing to correct values before checking.
```

In this case, the script written by the `write_timing_constraints` procedure does not include all of the information required to perform the clock gating check in PrimeTime. The clock gating check is important and should be done for both maximum and minimum timing.

The following steps show you how to create a clock gating check script from the script written by the `write_timing_constraints` procedure:

1. Locate the `create_clock` commands for each OCC clock, and change the `source_object` to the PLL source for the OCC.
2. In each corresponding `create_generated_clock` command, change the `-source` argument to match the PLL source.
3. Add the following command to the clock gating check script:

```
set_clock_gating_check -high OCC_clock_inst
```

In this case, `OCC_clock_inst` is the instance name (without the pin name) of the OCC clock source. **Note:** This step is required for OCC controllers that use multiplexors or combinational gating. However, you must skip this step for OCC controllers that use integrated clock-gating latches, since they already have clock gating checks defined for them in the library.

4. Add the following commands to enable the clock gating check to verify the slow (shift) clock gating in addition to the fast (capture) clock gating:

```
remove_case_analysis scan_enable  
set_false_path -from scan_enable -to [get_clocks OCC_clock]
```

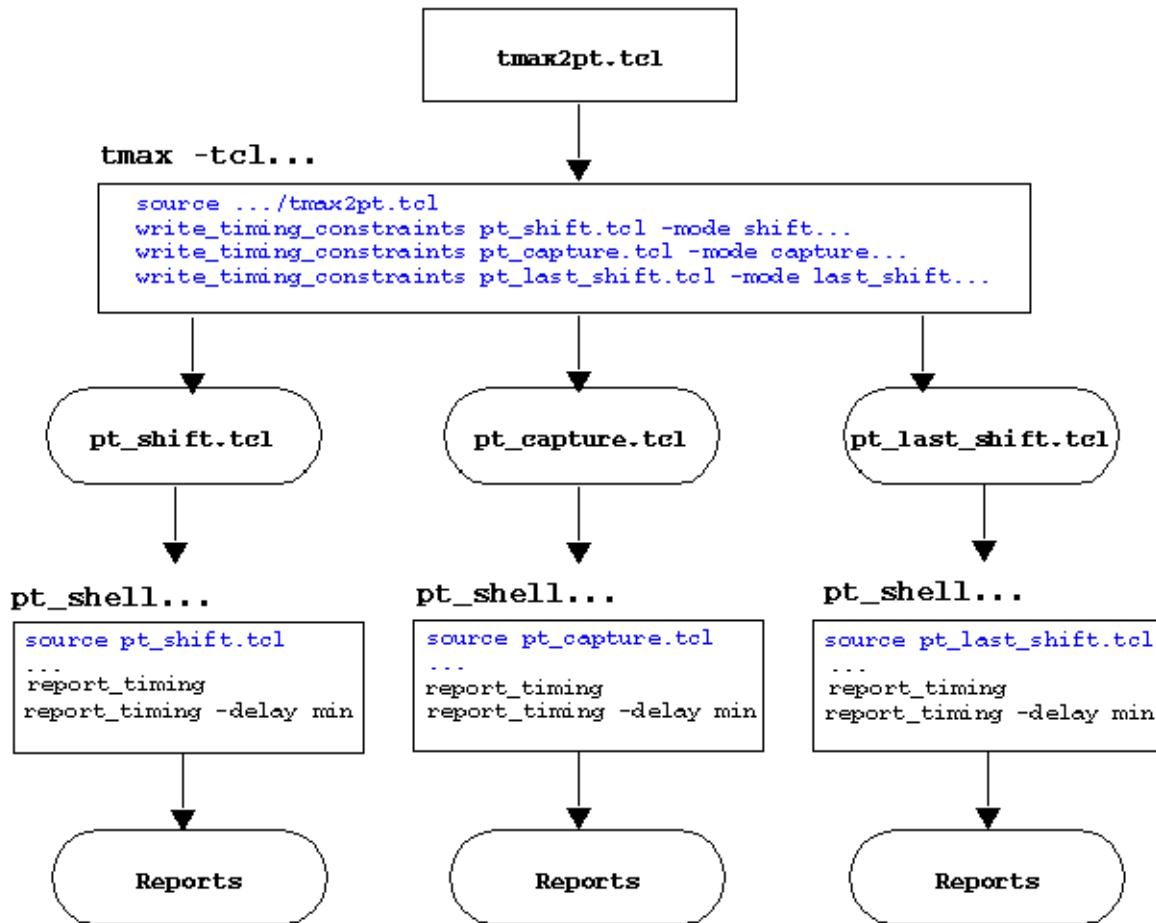
In this case, `OCC_clock` refers to all OCC clocks defined by the `create_clock` or `create_generated_clock` commands.

After you make these changes, the clock gating check is performed when you run the `report_timing` command. For more discussion of static timing analysis with OCC clocks, see [SolvNet Article #022490: "Static Timing Analysis Constraints for On-Chip Clocking Support."](#)

Performing an Analysis for Each Mode

As discussed previously, the flow involves performing a separate timing analysis for each mode. This is illustrated in [Figure 2](#). Example usages for various common modes are given in the following paragraphs.

Figure 2 Analysis of Three Modes Flow Example



To analyze timing for when the scan chain is *shifting*, the following is suggested:

```
TEST-T> write_timing_constraints pt_shift.tcl -mode shift \
-wft wft_name
```

You must select the WaveFormTable defined in the STL procedure file to be used during the shift cycle and specify it by using the `-wft` option.

For capture cycles for stuck-at faults, the following usage is suggested:

```
TEST-T> write_timing_constraints pt_capture.tcl \
-mode capture -wft <wft_name>
```

You must select the WaveFormTable defined in the STL procedure file to be used during the capture cycles and specify it by using the `-wft` option.

For analysis of transition fault patterns using *system clock launch*, the following usage is suggested:

```
TEST-T> write_timing_constraints pt_trans_sys_clk.tcl \
-mode capture -wft <launch_wft_name> -wft <capture_wft_name>
```

You must select the WaveFormTables defined in the STL procedure file to be used for the launch and capture cycles and specify them by using the `-wft` option. The first WFT is used as the launch WFT and the second WFT is used as the capture WFT. Launch-capture can be done in the same way as the stuck-at capture analysis above, with the WFT being the `launch_capture`.

For analysis of transition fault timing for *last-shift launch*, the following usage is suggested:

```
TEST-T> write_timing_constraints pt_last_shift.tcl \
    -mode last_shift -wft <shift_wft_name> -wft <capture_wft_name>
```

You must select the WaveFormTables defined in the STL procedure file to be used for the shift and capture cycles and specify them by using `-wft` option. The first WFT is used in the launch cycle and the second WFT is used in the capture cycle. Constraints are specified only as `set_case_analysis` if both cycles have the same TetraMAX ATPG constraints. Exceptions, such as `false_path`, are specified only for the capture cycle. You should check that scan-enable transitioning in the second cycle meets the setup time for the capture clock in the second cycle. The same mode can time both the shift to capture transition, and the capture to shift transition.

You can use the `-mode update` option to analyze timing for the DFTMAX Ultra cache registers or the DFTMAX shift power groups control chain latches. The suggested usage is as follows:

```
TEST-T> write_timing_constraints pt_update.tcl -mode update
```

With this mode, the constraints file defines either a `$dftmax_ultra_cache_cells` variable or a `$spcc_cache_cells` variable, depending on the compression architecture. In PrimeTime, use this variable to check the cache register timing, as shown in the following example:

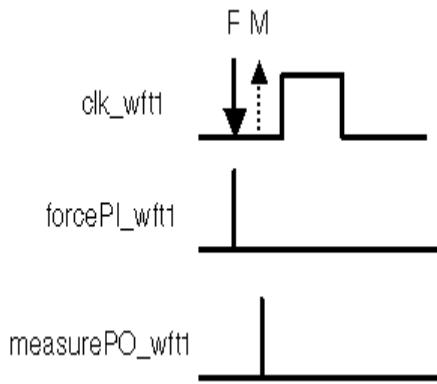
```
pt_shell> report_timing -to $dftmax_ultra_cache_ \
    cells -delay min_max
pt_shell> report_timing -from $dftmax_ultra_cache_cells \
    -delay min_max
```

This analysis mode is intended only for analyzing the DFTMAX Ultra cache registers or the DFTMAX shift power groups control chain latches. You should still perform full-design analysis with constraints generated for the shift and capture modes.

If you are generating separate constraints for transition and stuck-at timing, you do not need to do this for update mode because the stuck-at timing is the worst case for updating the cache registers due to the shorter capture cycle. If on-chip clocking is used, the constraints should not be used for any other purpose than checking timing to and from the variables, because the clock definitions must be modified in this case.

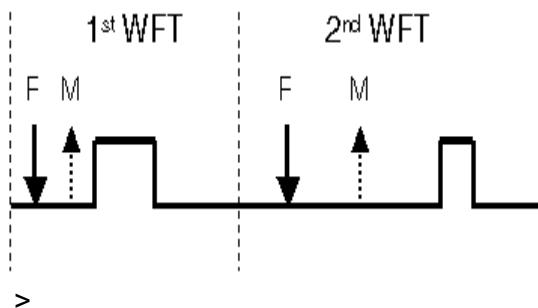
Implementation

The timing waveforms for clocks and signals reflect what is used on the tester. Input and output timing are relative to virtual clocks with prefixes "forcePI" and "measurePO" (see [Figure 3](#)). These clocks are impulse clocks with 0 percent duty cycles. The forcePI virtual clocks pulse at the beginning of the cycle. The measurePO clocks pulse at the earliest measure PO time. The timing data is taken from the STL procedure file.

Figure 3 Waveforms Used For Timing

Each PI, PO, and PIO is listed individually, because each can have a separate input or output delay. Also, each clock is individually listed.

For delay test timing analysis, a single clock net can have clock waveforms that vary due to different waveform tables. For example, the waveform might change between the last shift cycle and the capture cycle. PrimeTime has some facilities to handle this situation. This involves superimposing two clock cycles on top of each other, offset by the period of the first cycle. Each cycle will have its own set of forcePI and measurePO virtual clocks. This is shown in [Figure 4](#). The WFTs used are based on the order specified.

Figure 4 Superimposed Cycles For Two WFTs

Note:

The `create_generated_clocks` command is used to allow clock reconvergence pessimism reduction to work on the two pulses. PrimeTime version Z-2006.12 or later should be used. (Earlier versions report spurious paths between master clocks in addition to the paths between the generated clocks.) The PTE-075 warnings reported by PrimeTime version Z-2006.12 are spurious and can be ignored.

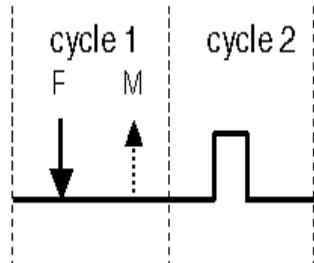
Note that the analysis with superimposed clocks is specific to the two cycles specified. They do not cover other cycles, such as setup and propagation cycles around the launch and capture cycles of a system clock launch pattern.

The `write_timing_constraints` script attempts to apply a minimal set of timing exceptions to aid accurate timing analysis. For the `last_shift` mode, false and multicycle paths

from the capture cycle are used. Case analysis exceptions are applied for multiple cycle modes only if both cycles have the same PI constraints during ATPG.

For capture cycles of end-of-cycle measures, the waveforms are expanded into a two-cycle period to adjust for the expansion of each capture vector into multiple vectors (see [Figure 5](#)). Shift cycles remain single cycle.

Figure 5 End of Cycle Pattern Expansion



Converting Timing Violations Into Timing Exceptions

Timing exceptions can negatively impact ATPG efficiency. You can ensure ATPG efficiency by using only those timing exceptions that apply directly to the current ATPG environment and ignoring others that are irrelevant to tester timing. The following flow describes how to convert timing violations into timing exceptions:

1. Set up TetraMAX ATPG in the appropriate mode.
2. Follow the steps described in "[Generating PrimeTime Constraints](#)."
3. Use the `write_exceptions_from_violations` PrimeTime Tcl procedure described in this section.
4. Read the results into TetraMAX ATPG using the `read_sdc` command. (**Note:** Don't read the results of step 2 at this time, since they are implicit in the TetraMAX ATPG setup and will reduce efficiency if applied again.)

The `write_exceptions_from_violations` procedure is part of the `$SYNOPSYS/auxx/syn/tmax/pt2tmax.tcl` script. To use this Tcl script, source it into `pt_shell`, set up the timing environment, and then run it. This script converts all timing violations into timing exceptions, then applies them to ensure that timing is clean. When the timing isn't clean, the newly found timing violations are converted. Each update check conversion process is considered an iteration. The new exceptions are written in SDC format.

The syntax for the `write_exceptions_from_violations` procedure is as follows:

```
write_exceptions_from_violations
  [-output filename]
  [-specific_start_pin]
```

```

[-max_iterations number]
[-delay_type <max | min | min_max>]
[-full_update_timing]
[-pba]
[-slack crit_slack]
[-man]

```

Argument	Description
<code>-output filename</code>	Writes the output to a specific file name. The default is <code>tmax_exceptions.sdc</code> .
<code>-specific_start_pin</code>	Writes separate exceptions for different outputs of a violating cell. The default is one exception per startpoint cell. This switch can improve the efficiency of the timing exceptions on some designs, especially designs in which some memory paths violate timing but other paths with the same memories do not. However, this analysis requires multiple iterations, which can dramatically increase the runtime.
<code>-max_iterations number</code>	Iterates the specified <i>number</i> of times before placing blanket exceptions on endpoints to ensure that timing is met. The default is 40.
<code>-delay_type <max min min_max></code>	Specifies which violations to convert to timing exceptions. The <code>max</code> argument converts setup time violations to exceptions. The <code>min</code> argument converts hold time violations to exceptions. The <code>min_max</code> argument (the default) converts both setup and hold time violations to exceptions.
<code>-full_update_timing</code>	Forces a full timing update for the second iteration, and all later iterations, of the update check conversion process. You should use this option when violating paths cause excessive runtime during timing updates.
<code>-pba</code>	Runs timing analysis using the "Path" mode of path-based analysis. In most cases, this option reduces the number of violating paths. However, this additional analysis affects the runtime.

`-slack crit_slack` Sets the minimum non-violating slack. The critical slack can be positive or negative. The default is 0.0. You can use this option to reduce the number of timing exceptions when testing several paths with small timing violations. In this case, use a small negative number as the critical slack. This option can be used for other purposes since the critical slack can be positive or negative.

`-man` Prints the syntax message.

E

STIL Language Support

The following sections of this appendix provide a brief overview of the Standard Test Interface Language (STIL), and identifies how TetraMAX ATPG uses the STIL constructs.

- [STIL Overview](#)
- [TetraMAX ATPG and STIL](#)
- [STIL Conventions in TetraMAX](#)
- [IEEE Std. 1450.1 Extensions Used in TetraMAX](#)
- [Elements of STIL Not Used by TetraMAX](#)

STIL Overview

The STIL language is an emerging standard for simplifying the number of test vector formats that automated test equipment (ATE) vendors and computer-aided engineering (CAE) tool vendors must support.

As an emerging standard, STIL is evolving with additional standardization efforts. TetraMAX ATPG makes use of both the current STIL standard (IEEE Std. 1450-1999 Standard Test Interface Language (STIL) for Digital Test Vectors), and the IEEE Std. 1450.1 Design Extensions. Many of the extensions were developed in support of TetraMAX ATPG users and subsequently proposed to the IEEE Std. 1450.1 working group. Both of these efforts are detailed in the following sections:

- [IEEE Std. 1450-1999](#)
 - [IEEE Std. 1450.1 Design Extensions to STIL](#)
-

IEEE Std. 1450-1999

The Standard Test Interface Language (STIL) provides an interface between digital test generation tools and test equipment. The following defines a test description language:

- Facilitates the transfer of digital test vector data from CAE to ATE environments
- Specifies pattern, format, and timing information sufficient to define the application of digital test vectors to a DUT
- Supports the volume of test vector data generated from structured tests

STIL is a representation of information needed to define digital test operations in manufacturing tests. STIL is not intended to define how the tester implements that information. While the purpose of STIL is to pass test data into the test environment, the overall STIL language is inherently more flexible than any particular tester. Constructs might be used in a STIL file that exceed the capability of a particular tester. In some circumstances, a translator for a particular type of test equipment might be capable of restructuring the data to support that capability on the tester; in other circumstances, separate tools might operate on that data to provide that restructuring.

The STIL language can be used for defining the test protocol input and the pattern input and output. STIL test protocol input is used for various design rule checking (and tester rules checking) and drives the test generation process. ATPG-generated STIL patterns might be structured such that intra cycle timing, cyclization of test and raw data are separated into, respectively, Timing, Procedure,s and Pattern Blocks. This structure simplifies various rules checking, maintenance, and pattern mapping for system-on-chip testing.

To understand more about STIL, refer to the IEEE Std. 1450.0-1999 Standard Test Interface Language (STIL) for Digital Test Vectors. For general information about the STIL standard, click the Executive Overview link on the STIL home page at <http://grouper.ieee.org/groups/1450/index.html>.

IEEE Std. 1450.1 Design Extensions to STIL

TetraMAX ATPG makes use of several IEEE Std. 1450.1 Design Extensions to support both test program definition and internal tool behaviors. Many of the extensions were developed in support of TetraMAX ATPG users and subsequently proposed to the 1450.1 working group. While these extensions are used by TetraMAX ATPG, they are not generated or present when stil99-compliant patterns are written, as described in the next section. The presence of 1450.1 extensions allows for a more flexible definition of STIL data. Without these constructs, STIL is more restrictive in its application, requiring complete regeneration when certain expected constructs are modified, which in turn can lead to a usage environment that is less flexible and is more likely to fail than an environment with the 1450.1 extensions present.

The documentation for these extensions is being developed by the IEEE working group and is expected to go to ballot during 2002. After ballot, the document is available through the normal IEEE channels. Until the ballot is complete, copies might be obtained by contacting the working group. See this IEEE web site for information on this development effort:<http://grouper.ieee.org/groups/1450/dot1/index.html>

TetraMAX ATPG and STIL

TetraMAX ATPG uses STIL in several different contexts. Design information may be provided to TetraMAX ATPG through the STIL procedure file . TetraMAX ATPG supports a subset of STIL syntax for input to describe scan chains, clocks, constrained ports, and pattern/response data as part of the STL procedure file definitions. Complete test sets may be written out in STIL format. Also, Tester Rules Checking is provided through STIL-formatted files.

In constructing a STL procedure file, you can define the minimum information needed by TetraMAX ATPG. However, any STIL files written as TetraMAX output contain an expanded form of the minimum information and may also contain pattern/response data that the ATPG process produces. TetraMAX ATPG reads and writes in STIL, so one time a STIL file is generated for a design, TetraMAX ATPG can read it again at a later time to recover the clock/constraint/chain data, the pattern/response data, or both.

TetraMAX ATPG can read some constructs that it does not generate. For example, an external pattern source can be read into TetraMAX ATPG for fault simulation but it cannot be written out with the same constructs as were read in.

The generation of the 1450.1 extensions is controlled by the `-stil` or `-stil99` option to the `write_patterns` command. When the `-stil99` option is used, then only standard IEEE-1450 syntax is used without, of course, the benefit of the added functionality enabled by the extensions. Full flexibility and robust STIL definitions are supported via the `-stil` option.

STIL Conventions in TetraMAX

STIL supports a very flexible data representation. TetraMAX has defined conventions in the use of STIL to represent data in a uniform manner yet maintain this flexibility. These conventions are discussed in the following sections:

- [Use of STIL Procedures](#)
 - [Context of Partial Signal Sets in Procedure Definitions](#)
 - [Use of STIL SignalGroups](#)
 - [WaveFormCharacter Interpretation](#)
-

Use of STIL Procedures

When possible, TetraMAX generates calls to STIL procedures from the pattern body. STIL procedures are used in general by TetraMAX because a STIL procedure is self-contained; that is, the state of all signals used in a procedure is established and maintained only during that procedure execution. On return at the end of a procedure, the state of the signals is restored to the values they maintained before the call. This is in contrast to the STIL macro construct, where the final state of these signals must be returned and applied in the patterns before continuing to process STIL data.

By using STIL procedures, the sequence of pattern operations are insensitive to the procedure operations. If macros were used, the next set of pattern activity would be based on information returned from the last macro operation. Also, the execution/behavior of a STIL macro might be different depending on the value of the signals present at the start of the macro, whereas the behavior of the procedure is always the same. The effort both to start a macro with the current state at the call, and to return the right information to the calling context at the return of macro adds significant processing overhead of STIL data when macros are present.

Also, procedure constructs are defined by the STIL standard to be maintained through processing. Macro constructs are defined by the STIL standard to be expanded or “flattened” during processing, and are defined to not be present after processing. While specific processing environments might be able to maintain macro constructs, in general (and to follow the STIL specification) macros would be processed-out or “in-lined” by tools reading STIL data, while procedure constructs would remain in a processed stream.

Finally, because procedures are defined as standalone constructs, it is possible to manipulate the contents of a STIL procedure (within certain constraints such as not changing the functionality of that procedure), to manipulate the procedure without concern of affecting the rest of the pattern operation. While this can be done with some macros as well, because macro behavior is not constrained to the execution of that macro, changes inside a macro can affect data in the rest of the pattern set.

Context of Partial Signal Sets in Procedure Definitions

Another consideration of TetraMAX’s application of procedures is its ability to define values for only those signals used in the procedure. While the capture procedures reference all signals in the design (generally through application of the _pi and _po groups), the `load_unload` procedure can leverage a partial signal set context.

The `load_unload` procedure requires establishing values only on the signals necessary to support the scan-shift operation on the design. In addition, TetraMAX supports the definition of a `load_unload` procedure in the STIL procedure file that references signals later in the procedure (for example, during the shift block) that might not have been assigned a value by the first Vector of the procedure. These capabilities allow maximum flexibility in interpreting the

load_unload operation during test generation. When STIL test patterns are generated by TetraMAX, the `load_unload` procedure is “completed” to contain all signals used in the procedure, in the first Vector (or Condition) of that procedure, to create a standards-compliant definition. However, unspecified signals that do not affect the scan-shift operation will not be present in this procedure.

The STIL standard defines that unused signals in a procedure are assigned DefaultState values when the procedure is called. This is a valid state for these signals, because they cannot affect the procedure operation. This is not a requirement of the standard, however (requirements contain the word “shall”), and TetraMAX leverages the flexibility of an incomplete definition for generating other test formats, in particular WGL.

TetraMAX uses the flexibility of deferred and unspecified signal assignment in procedure definitions to maintain the last assigned state on these signals for WGL generation. This option of maintaining the last-assigned-state generates a WGL test program that minimizes transitions on signals at test, particularly transitions that don’t affect the test behavior and might have other adverse effects.

In test contexts where STIL is being applied and procedure operations are being “expanded” or “in-lined” in the final test program, it might be valuable to consider the “default” handling of unused signals in the procedure to allow generation of a test that behaves similarly to the TetraMAX-generated WGL test.

Use of STIL SignalGroups

TetraMAX makes use of STIL SignalGroups to simplify creation of STIL protocol information. The STIL procedures file might be a complete set of information for certain sections of the final STIL file, or it might be an incomplete file completed by TetraMAX when the final test file is generated.

SignalGroups are used in this context to simplify referencing to sets of signals, without needing to define these signal collections for a specific design, which simplifies STIL procedure file creation. In order to support this operation, however, TetraMAX must assume certain naming conventions. Also, the grouping conventions that TetraMAX supports relate directly to the operations that are performed by ATPG operations to generate test sequences.

By using STIL SignalGroups, the output pattern data generated by TetraMAX is more compact than the equivalent by-Signal constructs, and the output format is more general. For example, it can be easier to modify a signal name when that name occurs only inside the SignalGroups, than if that signal reference is used throughout the pattern data.

TetraMAX defines two primary groups, “_pi” and “_po”, which contain an overlapping set of InOut (bidirectional) Signal references. While this can be confusing in some situations, by maintaining these groups this way, the context of test generation as performed internally in TetraMAX is maintained in the output patterns. This supports direct correspondence of the test set with the internal operations performed by TetraMAX, which in turn reduces confusion between TetraMAX-based analysis of test behaviors and the actual information present in the test. However, this information can only be maintained completely through the use of P14501 extensions.

When only IEEE Std. 1450-1999 constructs are used, some loss of information can be expected in STIL programs, causing test programs to be written in a way that is dependent on the presence of constructs used. For example, bidirectional signal behavior, supported by complete

representation of both the input behavior and the output behavior in the _pi and _po groups, respectively, allows for the potential modification of the capture procedures without changing the pattern data, in the P14501 context. This opportunity is much more limited under IEEE Std. 1450 constructs, as the pattern data might be written dependent on the capture procedure constructs, and the capture procedures might not be changed without changing the functionality of the pattern.

WaveFormCharacter Interpretation

TetraMAX supports a fixed context for WaveFormCharacter (WFC) interpretation in STIL data. A minimum set of requirements are validated against the waveforms associated with these WFCs to avoid undue constraints and support test behaviors as necessary. See [Table 1](#) for a description of the WFC interpretations supported by TetraMAX.

Table 1 Supported WaveFormCharacter Interpretations

WFC	Interpretation
0	Drive-low during the waveform.
1	Drive-high during the waveform.
Z	Drive-inactive (typically implemented on ATE as a driver-off operation) during the waveform.
N	Drive-unknown during the waveform. This waveform, if used in the patterns, can be mapped to any of the drive operations above without affecting the test.
P	Drive an active pulse during the waveform. The pulse may be either a high-going pulse or a low-going pulse as appropriate for the type of clock. This waveform is supported only for signals identified as clocks in the design. In path-delay contexts, the timing of this pulse must match the second pulse of the D waveform, if the D waveform is defined.
D	Drive two pulses during the waveform. This definition is supported only for path-delay MUX operation.
E	Drive an active pulse during the waveform. This definition is supported only for path-delay MUX operation, and the timing of this pulse must match the timing of the first pulse of the D waveform.

**Table 1 Supported WaveFormCharacter Interpretations
(Continued)**

WFC	Interpretation
H	Measure-high (STIL “CompareHigh”, or “CompareHighWindow” followed by “CompareUnknown”) during the waveform. In bidirectional contexts, this waveform must also define a drive-inactive (STIL Z) state before performing the measure.
L	Measure-low (STIL “CompareLow”, or “CompareLowWindow” followed by “CompareUnknown”) during the waveform. In bidirectional contexts, this waveform must also define a drive-inactive (STIL Z) state before performing the measure.
T	Measure-inactive (STIL “CompareOff”, or “CompareOffWindow” followed by “CompareUnknown”) during the waveform. In bidirectional contexts, this waveform must also define a drive-inactive (STIL Z) state before performing the measure.
X	This waveform is used by TetraMAX to indicate a no-measure operation. From a STIL perspective, the contents of this waveform may be empty; the absence of activity might imply ATE operations to inhibit previous output measures. This waveform assumes the previous drive state continues to be asserted when used in bidirectional contexts; TetraMAX will define a drive state before specifying an X that continues to be applied here. Note this waveform should not contain a P state to provide the drive state because the P state does not maintain a drive-inactive value.

IEEE Std. 1450.1 Extensions Used in TetraMAX

The IEEE Std. 1450.1 extensions used by TetraMAX ATPG are identified and described in the following sections:

- [Vector Data Mapping Using \m](#)
- [Vector Data Mapping Using \j](#)

- [Signal Constraints Using Fixed and Equivalent](#)
 - [ScanStructures Block](#)
-

Vector Data Mapping Using \m

The vector data mapping function allows for a new waveform definition to be selected for a given waveform character in a vector. This is most useful in the case of parameter passing to a macro or procedure; however, it can be used anywhere a waveform character string is formed.

In certain scan test styles (such as the LSI “LNI” protocol), it is necessary to measure the output of the design under test’s (DUT) bidirectional signals in one cycle and then drive the same logical values on the same bidirectional signals from the tester in the next cycle, while also turning the internal bidirectional signal drivers off. A test pattern thus has the following format:

1. Load scan chains.
2. Force values on primary inputs (all clocks are off, bidirectional signals are driven by design under test (DUT)).
3. Measure primary outputs and bidirectional signals (all clocks are off).
4. Force values on primary inputs (values are the same as in cycle 2, except the internal bidi drivers are turned off by asserting a special `bidi_ctrl` input), force values on bidirectional signals (same logical values as measured in previous cycle).
5. Pulse capture clock.
6. Unload scan chains.

Turning off the internal bidirectional drivers in cycle 4 avoids possible contentions that can result in cycle 5 due to capturing new data into the state elements. The additional data to be applied on bidirectional signals in cycle 4 is redundant (can be computed from the data of cycle 3.) This test style needs to be supported without adding extra data to the STIL patterns and without changing the waveformcharacters in the patterns. Also, ATPG rules checking can verify the correctness of the patterns (for example, the internal bidirectional signals are turned off in cycle 4) before actually generating test data.

Note that ATPG-generated patterns are typically guaranteed to be contention-free on all bidirectional signals, both pre- and post-capture. Thus, the example protocol might not be required to avoid bidi contentions. However, ATPG tools might support this protocol for customers that have already designed their test flow with this protocol.

It is important that the `bidi_ctrl` input turns off ALL internal bidi drivers in cycle 4 above. Otherwise, a contention-free pattern could be transformed into a pattern with contentions by the very protocol that attempts to avoid bidi contentions! For example, consider the following example, where `BIDI1` and `BIDI2` are bidirectional signals, and `BIDI_CTRL` is an input that, when 0, turns off the internal driver of `BIDI1`, but not of `BIDI2`:

ATPG-generated contention-free pattern:

1. Load scan chains.
2. Force values on primary inputs and bidirectional signals (`force BIDI_CTRL=1; BIDI1 = Z; BIDI2 = Z;`).
3. Measure primary outputs and bidirectional signals (`measure BIDI1=L; BIDI2=H;`).

4. Pulse capture clock (this has the effect of switching the internal drivers such as now both the BIDI1 and BIDI2 internal drivers are driving 0. There is no contention, because the tester continues to drive Z on both bidirectional signals, as in cycle 2.).
5. Unload scan chains.

The pattern above is changed, after it has been generated, to match the LNI protocol:

1. Load scan chains.
2. Force values on primary inputs and bidirectional signals (`force BIDI_CTRL=1; BIDI1 = Z; BIDI2 = Z;`).
3. Measure primary outputs and bidirectional signals (`measure BIDI1=L; BIDI2=H;`).
4. Force values on primary inputs (`force BIDI_CTRL=0; BIDI1=0; BIDI2=1;`).
5. Pulse capture clock (this has the effect of switching the internal drivers such as now both the BIDI1 and BIDI2 internal drivers are driving 0. This causes a contention on BIDI2: its internal driver, not turned off, drives 0 while the tester drives 1, as in cycle 4).
6. Unload scan chains.

Syntax

The mapping operation is specified in either the Signals or the SignalGroups block as follows:

```
Signals {
    sig_name < In | Out | InOut | Pseudo > {
        ( WFCMap (from_wfc)* -> to_wfc; )*
    }
}
SignalGroups (domain_name) {
    groupname = sigref_expr {
        ( WFCMap (from_wfc)* -> to_wfc; )*
    }
}
```

WFCMap is an optional statement that, when used, indicates that any pattern data assigning from_wfc to the signal or signalgroup, should be interpreted as having been assigned from to_wfc instead.

To use the mapping of a signal or signalgroup, a new flag is added to the cyclized pattern data:
\m Indicates that the defined mapping should be used.

If the vector mapping \m is used, but no WFCMap has been defined for the waveformcharacter to be mapped, the same waveformcharacter is used unchanged.

Example

In the following example, the vectors are labeled to correspond to the LNI-protocol cycles above. Cycle 3 uses the arguments passed in for _io first (HL), then cycle 4 uses them again, this time mapped to (10), which remain in effect for cycle 5 as well.

```
Signals {
    a In; ck In; bidi_enable In; b Out; q1 InOut; q2 InOut;
}
Procedures procdomain {
    "capture_sysclk" {
```

```

W specWFT; // where all waveformchars are defined
"cycle 2": V { a=#; ck=0; bidi_enable=1; b=X; _io=ZZ ; }
"cycle 3": V { b=#; _io=%%; }
"cycle 4": V { bidi_enable=0; b=X; _io=\m ##; }
"cycle 5": V { ck=P; }
}
}

Pattern "__pattern__" {
W specWFT;
"cycle 1": Call "load_unload" { ... }
Call "capture_sysclk" { a=0; b=H; _io=HL; }
"cycle 6": Call "load_unload" { ... }
}

```

Vector Data Mapping Using \j

The “join” function allows you to have multiple waveform characters against the same signal in one vector. This enables structuring of STIL pattern output using procedures.

Syntax

Refer to [“Vector Data Mapping Using \m”](#) for the syntax definition of the WFCMap statement.

General Example

The following is an example usage of the join function.

```

Signals {
    b InOut { WFCMap 0x -> k; }
}
Pattern p {
    V { b = 0; b = x; }
}

```

[Table 2](#) shows examples of “two data” conditions on an InOut.

Table 2 “Two Data” Conditions

Force	Measure
0, 1, Z, N	X
Z	L, H, T
0	L
1	H
0	H

Table 2 “Two Data” Conditions (Continued)

Force	Measure
1	H, T

The rules for handling multiple definitions of a signal are

- The normal behavior of a WFC-assignment to a signal is to replace the last-assigned WFC value.
- This behavior might be OVERRIDDEN on a per-vector bases, through the use of a “join” escape sequence. The “join” allows both WFCs to be evaluated, using the WFCMap, to resolve or specify a single new WFC that is the required effect of these two WFCs.

For instance, take the case where two SignalGroups have a common element in them (signal 'b'):

```
_pi = '...+b';
(po = '...+b';
```

A procedure might “join” these two groups in a vector:

```
proc { cs { V { _pi=#; _po= \j #; } }}
```

Signal 'b' needs to be resolved based on the combinations of WFCs that might be seen by these two groups. It might have a WFCMap declaration like

```
WFCMap 0x -> 0;
WFCMap 1x -> 1;
... etc. ...
```

This mechanism provides for the explicit resolution of “joined” data without creating new combinations of waveforms on-the-fly.

“Joining” requires the WFCMap to define two WFCs to equate to a single new resolved WFC. The WFCMap never requires more than two WFCs as [Figure 1](#) presents:

Figure 1 WFC Example

```
WFCMap 0A -> n;
WFCMap cn -> m;
V { b = 0; b = \jA; }
    |
    |
    | join 0 and A; get n
```

Usage Example

Consider a design with one input, one output and two bidirectional signals. STIL would declare them like this:

```
Signals { i:In; o:Out; b1:InOut; b2:InOut; }
```

STIL also defines signal groups:

```
Signalgroups {
    "_pi" = 'i + b1 + b2';
    "_po" = 'o + b1 + b2';
}
```

STIL patterns are written out using capture procedures. Unlike a “flat” vector-only STIL output, using capture procedures has many advantages:

- Pattern cyclization is encapsulated in the capture procedures; timing in the Timing block and data in the Pattern block. This allows easy understanding and maintenance of the three separately.
- Rules checking (DRC) is done only on the small Procedures block, independent of the huge Pattern block.
- Patterns are CTL (P1500) ready: only the procedures need to be modified, not the Patterns.

Capture procedures are defined like this:

```
Procedures {
    "capture" {
        V { "_pi"="### ; "_po"="###; }
    }
}
```

All of the previous examples are fully STIL 1450-1999 compliant.

Now let's consider a STIL pattern that includes the following:

```
force_all_pis { i=0; b1=Z; b2=1; }
measure_all_pos ( o=H; b1=H; b2=X; )
```

The STIL output would translate the previous example into the following:

```
Call capture { "_pi"=0Z1; "_po"=HHX; }
```

Because of how STIL is interpreted by the consumer of the patterns, the actual arguments are substituted for the formal arguments # in the body of the procedure, and the signalgroups _pi and _po are expanded to their signals, resulting in:

```
V { i=0; b1=Z; b2=1; o=H; b1=H; b2=X; }
```

However, STIL also specifies that if multiple waveform characters are assigned to the same signal in a given vector, all but the last one are ignored. Thus, the previous example vector is equivalent to:

```
V { i=0; o=H; b1=H; b2=X; }
```

Now, how should the bidirectional assignments be interpreted? The first one, b1=H, means “measure High on b1”. This is consistent with the intention of the ATPG pattern, although it leaves the ambiguity of also implying that the tester driver should be tri-stated (b1 driven to Z) to take a measure. The STIL consumer application is supposed to take this into account.

The second bidirectional, b2=X, means “measure X (no measure) on b2”. Unfortunately, the drive part (b2 driven to 1) is lost. This is a real problem.

The 1450.1 solution is very simple and general: Provide a mapping to explicitly explain what to do with two waveform character assignment. Thus, the 1450.1 procedure would be written as:

```
Procedures {
    "capture" {
        V { "_pi"= \j ### ; "_po"= \j ###; }
    }
}
```

Notice the addition of the “join” modifier `\j`. The `\j` refers to the WFCMap mapping table that would be defined as:

```
Signalgroups {
    "_pi" = 'i + b1 + b2';
    "_po" = 'o + b1 + b2' {
WFCMap 0X -> 0; WFCMap 1X -> 1; WFCMap ZX -> Z; WFCMap NX -> N;
    }
}
```

This provides an unambiguous interpretation of the previous example:

```
V { i=0; b1=Z; b2=1; o=H; b1=H; b2=X; }
```

to the required:

```
force_all_pis { i=0; b1=Z; b2=1; }
measure_all_pos ( o=H; b1=H; b2=X; )
```

Signal Constraints Using Fixed and Equivalent

Structured test patterns often have signals constrained to have a certain value or waveform during a pattern sequence. This might be required, for example, for ATPG scan rules checking (such as a test mode signal always active) or for differential scan or clock inputs. The Fixed STIL construct defines signals that must have a constant waveform character and the Equivalent construct defines signals that are linked to other signals. These constructs help reduce pattern volume, because the value of a constraint signal does not need to be specified explicitly in the pattern data. Also, ATPG rules checking requires signal constraint information as input.

ScanStructures Block

Simulation of scan patterns outside the test-pattern generator is often performed to verify timing, design functionality, or the library used to generate the patterns. The speed of the simulator is limited by simulating the load/unload (Shift) operation of scan chains. Optimal simulation performance is achieved with no shifts, bypassing scan chain logic and asserting/verifying the scan data directly on the scan cells.

The ScanStructures block is extended to include additional information required for efficient simulation of scan patterns, that is, eliminating the need to simulate load/unload (shift) cycles. Additional constructs are defined on the ScanCell statement inside the ScanChain block. In addition, the capability is added to the ScanStructures block to support referencing previous ScanChain definitions from other ScanStructure blocks.

Elements of STIL Not Used by TetraMAX

The following sections list the STIL output and input constructs that are not used in this version of TetraMAX ATPG.:

- [TetraMAX STIL Output](#)
- [TetraMAX STIL Input](#)

Note that this list is provided to you as a guide for tools that are designed to read the STIL output file generated from TetraMAX ATPG.

The only elements of 1450.1 used by TetraMAX ATPG are identified previously in 1450.1 Extensions Used in TetraMAX; all other elements of 1450.1 are not used by TetraMAX ATPG.

This information is provided specifically from the context of TetraMAX ATPG as a standalone tool. TetraMAX ATPG-generated STIL output is applied in several contexts and tool flows, for example as part of the CoreTest environment (CoreTest is in development). These contexts will use additional STIL constructs and behaviors not used by TetraMAX ATPG alone.

TetraMAX STIL Output

Here is a list of output constructs that TetraMAX ATPG does not currently support. To understand more about these constructs, refer to the numbered paragraph in IEEE Std. 1450-1999 as indicated in the list.

Note:

The TetraMAX ATPG internal pattern source will not write or produce STIL with the constructs described in this section. Future versions might make use of these constructs as necessary.

11. UserKeywords

TetraMAX ATPG does not generate any UserKeywords. All keywords used are those defined in the standard.

12. UserFunctions

TetraMAX ATPG does not generate any UserFunctions. All timing expressions use expressions that are defined in the standard.

17. PatternBurst > SignalGroups, MacroDefs, Procedures, ScanStructures (named domains)

TetraMAX ATPG does not generate any references to named domains from within a PatternBurst. All references are to the globally defined blocks only. Other contexts of STIL generation may provide named domain blocks.

17. PatternBurst > Start, Stop

TetraMAX ATPG does not generate any start/stop information within a PatternBurst. All

patterns are expected to execute from the beginning to the end of the pattern.

17. PatternBurst > PatList > pat_name {...} (optional statements per pattern)

TetraMAX ATPG does not generate any pattern names in a PatList that contain block information. The default generation of STIL data will rely on definitions in a global SignalGroups, MacroDefs, Procedures, and ScanStructure blocks only. Named block reference statements may be specified from other STIL contexts

18. Timing > WaveformTable > Inherit...

TetraMAX ATPG does not use the Inherit statement within WaveformTables. All waveform tables are completely self-contained.

18. Timing > WaveformTable > SubWaveforms

TetraMAX ATPG does not use the SubWaveform block within the Timing definition. All waveforms are composed of single drive and compare events.

18. Timing > WaveformTable > event_label

TetraMAX ATPG does not generate any event labels. All timing information is composed of timing values that are relative to the beginning of the period.

18. Timing -> WaveformTable > [event_num]

TetraMAX ATPG does not use multiple events in a waveform. All data from a pattern is made up of single waveform character references.

18. Timing -> WaveformTable > @ label references in timing expressions

TetraMAX ATPG does not generate any relative timing. All timing values are specified as absolute times from the start of the period.

18.2 Waveform event definitions > expect events

TetraMAX ATPG does not generate any expect events. Only drive and compare events are used.

19. Spec, Selector

TetraMAX ATPG does not generate either Spec or Selector blocks. All timing values are specified as absolute numbers.

21.1 Cyclized data > \d

TetraMAX ATPG does not generate data using the decimal notation, which is selected by use of the \d escape sequence.

21.2 Multiple-bit cyclized data

TetraMAX ATPG does not generate any multiple bit vector information. All vectors contain only one wfc per signal.

21.3 Non-cyclized data

TetraMAX ATPG does not generate any non-cyclized data. All vectors are made up of WFC data that refers to cyclized waveform data in a Timing block.

22.6 Loop statement

TetraMAX ATPG support of Loop operations is very restrictive to certain contexts.

Generally, all pattern vectors are executed in a straight-line sequence.

22.7 MatchLoop statement

TetraMAX ATPG does not generate any MatchLoop conditions. All patterns vectors are

executed in a straight-line sequence.

22.8 Goto statement

TetraMAX ATPG does not generate any Goto statements. All patterns vectors are executed in a straight-line sequence.

22.9 Breakpoint

TetraMAX ATPG does not generate any Breakpoint statements. It is assumed that all vectors will fit into available ATE memory.

22.11 Stop

TetraMAX ATPG does not generate any Stop statements within a pattern. All patterns are expected to execute from the beginning through to the last vector.

23.1 Pattern > TimeUnit

TetraMAX ATPG does not generate the TimeUnit statement. This statement is only used with uncyclized data, which is not generated by TetraMAX ATPG.

TetraMAX STIL Input

Here is a list of constructs that TetraMAX ATPG can read, but ignores. These will not be written out. To understand more about these constructs, refer to the numbered paragraph in IEEE Std. 1450-1999 as indicated in the list.

10. Include Statement

Supported (by the reader, not produced by the writer).

11. UserKeywords Statement

Ignored (by the reader, not produced by the writer).

12. UserFunctions Statement

Ignored (by the reader, not produced by the writer).

17. PatternBurst block syntax

References to named SignalGroups, MacroDefs, Procedures, and ScanStructures statements are supported (by the reader, not produced by the writer). Start, Stop and Termination statements are not supported by the reader.

F

STIL99 Versus STIL

This appendix provides an overview of the differences between the STIL99 and STIL pattern formats.

Table 1 STIL 99 Versus STIL

STIL 99	STIL
statement STIL 1.0; STIL 1.0 { TRC 2006; } (only if <>>resource_tags present)	statement STIL 1.0 { Design 2005; } STIL 1.0 { TRC 2006; } (only if <>>resource_tags present)
Header { Title " TetraMAX ..."; Date "Tue Feb ..."; Source "comment"; History { Ann {* ...previous Annotations in the History section *} Ann {*} ...fault, pattern, drc reports, clocks and constrained pins *} } }	Header { Title " TetraMAX ..."; Date "Tue Feb ..."; Source "comment"; History { Ann {*}...previous Annotations in the History section *} Ann {*} ...fault, pattern, drc reports, clocks and constrained pins *} } }

Table 1 STIL 99 Versus STIL (Continued)

STIL 99	STIL
UserKeywords ScanChainGroups (conditionally) ActiveScanChains (conditionally) ; Conditionally means with respect to the type of the design being parsed through TetraMAX ATPG. ObserveValue (seq-comp only) ScanChainPartition (seq-comp only) SeqCompressorStructures (seq-comp only) SerializerStructures (dftmax with serializer only)	UserKeywords BistStructures (conditionally) ClockStructures (conditionally) FreeRunning (conditionally) DontSimulate ScanChainGroups and ActiveScanChains are keywords in 1450.1. They are used as UserKeywords only in -stil99 format. Conditionally means with respect to the type of the design being parsed through TetraMAX ATPG. ObserveValue (seq-comp only) ScanChainPartition (seq-comp only) SeqCompressorStructures (seq-comp only) SerializerStructures (dftmax with serializer only)
Ann { * ANNOTATION * } Used only in the Header History section STL procedure file flow has options to preserve Ann in output STIL for top-section of the STIL data (not patterns). Special blocks for Ann { * load_unload <var> <cnt> * } and Ann{ * end load_unload * }, reseed_partial_overlap found in stil99 patterns for sequential compressor.	Ann { * ANNOTATION * } Used only in the Header History section STL procedure file flow has options to preserve Ann in output STIL for top-section of the STIL data (not patterns).

Table 1 STIL 99 Versus STIL (Continued)

STIL 99	STIL
	Variables { (seq-comp only, - stil only)
	Integer "var_name"; (seq-comp only)
	}
	(seq-compr only)
Signals {	Signals {
"sig":	"sig":
1. Always quoted	1. Always quoted
2. Does not use [] array notation; used for Pseudo only in seq-comp	2. Does not use [] array notation; used for Pseudo only in seq-comp
In	In
Out	Out
InOut	InOut
Pseudo	Pseudo
(Used for internal chain scan references on some BIST environments)	(Used for internal chain scan references on some BIST environments)
;	;
Instead of using semicolon, {} bracket format used if the following attributes are present:	Instead of using semicolon, {} bracket format used if the following attributes are present:
ScanIn; (no integer number)	ScanIn; (no integer number)
ScanOut; (no integer number)	ScanOut; (no integer number)
	WFCMap {
	0X->0; 1X->1; ZX->Z; NX->N; PX->P; P[0 1]->P; } * // end
	WFCMap
]
	}
	See Appendix E for more details on WFCMap. The mapping operation is specified in either the Signals or the SignalGroups.

Table 1 STIL 99 Versus STIL (Continued)

STIL 99	STIL
SignalGroups { (No signalgroups domain name)	SignalGroups "user_name" { (No signalgroups domain name)
Supports user names and generates specific groups: <code>_pi</code> lists all inputs+bidirections, <code>_po</code> lists all outputs+bidirectionals.	Supports user names and generates specific groups: <code>_pi</code> lists all inputs+bidirections, <code>_po</code> lists all outputs+bidirectionals.
{ } format used if the following attributes are present: <code>ScanIn</code> ; (no integer number) <code>ScanOut</code> ; (no integer number)	{ } format used if the following attributes are present: <code>ScanIn</code> ; (no integer number) <code>ScanOut</code> ; (no integer number)
	WFCMap { <code>0X->0; 1X->1; ZX->Z; NX->N;</code> (<code>_pi _po</code> , -stil only) <code>PX->P;</code> } FreeRunning { <code>Period time;</code> <code>LeadingEdge time;</code> <code>TrailingEdge time;</code> <code>OffState <D U>;</code> }

TetraMAX ATPG will accept the following Predefined SignalGroups:

- `_in` = input pins
- `_out` = output pins
- `_io` = bidirectional pins
- `_pi` = inputs + bidirectional pins
- `_po` = outputs + bidirectional pins
- `_si` = scan chain inputs
- `_so` = scan chain output

Table 1 STIL 99 Versus STIL (Continued)

STIL 99	STIL
<pre>PatternExec { (optionally named) Patternburst "_burst_"; (Fixed burst name) }</pre>	<pre>PatternExec "user_name" { (optionally named) Patternburst "_burst_"; (Fixed burst name) }</pre>
	<p style="text-align: center;">Default generation if no input patternexecs is a single unnamed patternexec. If named patternexecs are input, you must identify one patternexec to be used by TetraMAX ATPG, and only this patternexec is written out.</p> <pre>Patternburst "_burst_" (SignalGroups "user_name" ;)* (user spec'ed) (MacroDefs "user_name" ;)* (user spec'ed) (Procedures "user_name" ;)* (user spec'ed) (ScanStructures "user_name" ;)* (user spec'ed) PatList { user_specified_pattern_name -or- "_pattern_"; (Fixed pattern name) (ParallelPatList (SyncStart Independent LockStep) { (PAT_NAME_OR_BURST_NAME { (Extend;) })* // end ParallelPatList} // end of PatternBurst</pre>

Table 1 STIL 99 Versus STIL (Continued)

STIL 99	STIL
	Default generation if no patternbursts are specified on input will use the name "_burst_".
<pre>Timing { (No name generated) WaveformTable user_name { (Default name: _default_WFT_) fixed names for features: "_launch_WFT_", etc. << resource_tag >> preserved and passed through. No generation. Supported in 2008.09-sp2. Period integer_time_units;</pre>	<pre>Timing { (No name generated) WaveformTable user_name { (Default name: _default_WFT_) fixed names for features: "_launch_WFT_", etc. << resource_tag >> preserved and passed through. No generation. Supported in 2008.09-sp2. Period integer_time_units;</pre>
	Note current environment supports integer units of time only.
<pre>Waveforms { groups_or_signal_names { << resource_tag >> preserved and passed through. No generation. Supported in 2008.09-sp2. WFC usage in tmax is fixed to the following: inputs: 01 Z N outputs: H L T X clocks: PD E (Always single-WFC references, separated)</pre>	<pre>Waveforms { groups_or_signal_names { << resource_tag >> preserved and passed through. No generation. Supported in 2008.09-sp2. WFC usage in tmax is fixed to the following: inputs: 01 Z N outputs: H L T X clocks: PD E (Always single-WFC references, separated)</pre>

Table 1 STIL 99 Versus STIL (Continued)

STIL 99	STIL
<pre>ScanStructures { (Unnamed) ScanChain name { ScanLength integer ; ScanCells name_list ; ScanIn signal_name ; ScanOut signal_name ; ScanMasterClock signals ; ScanSlaveClock signals ; ScanInversion 0,1 ; } ObserveValue { vectored_pseudo_sig_assignment } (userkeyword statement, seq-comp only)</pre>	<pre>ScanStructures "user_name" { (user spec'ed) -or- ScanStructures { (Unnamed) ScanChain name { ScanLength integer ; ScanCells name_list ; ScanIn signal_name ; ScanOut signal_name ; ScanMasterClock signals ; ScanSlaveClock signals ; ScanInversion 0,1 ; } ObserveValue { vectored_pseudo_sig_assignment } (userkeyword statement, seq-comp only)</pre>
<pre>ScanChainGroups { (Used for some BIST designs) Generates groups of chains AND groups of groups. Groups are used in UserKeyword blocks and ActiveScanChains statements.</pre>	<pre>ScanChainGroups { (Used for some BIST designs) Generates groups of chains AND groups of groups. Groups are used in UserKeyword blocks and ActiveScanChains statements.</pre>
<pre>}</pre>	<pre>}</pre>
<pre>ScanChainPartition "name" { ... }</pre>	<pre>ScanChainPartition "name" { ... }</pre>
<pre>(userkeyword statement, seq-comp only)</pre>	<pre>(userkeyword statement, seq-comp only)</pre>
<pre>}</pre>	<pre>}</pre>

Table 1 STIL 99 Versus STIL (Continued)

STIL 99	STIL
<pre>sigref_expr = vec_data; //STIL Cyclized Pattern data LIST OF WFCs — for example "_po"= HHHL</pre> <p>In STIL, only assignment of WFC characters is allowed, except \r to repeat one WFC character. No \h for hex or other options used in the data.</p> <p>No Base statement in declarations; all assignments are by WFC.</p> <p>\r(integer) WFC — only one from the list of choices...</p>	<pre>sigref_expr = vec_data; //STIL Cyclized Pattern data LIST OF WFCs — for example "_po"= HHHL</pre> <p>In STIL, only assignment of WFC characters is allowed, except \r to repeat one WFC character. No \h for hex or other options used in the data.</p> <p>No Base statement in declarations; all assignments are by WFC.</p> <p>\r(integer) WFC — only one from the list of choices...</p>
<p>TetraMAX ATPG supports a fixed context for WaveFormCharacter (WFC) interpretation in STIL data. A minimum set of requirements are validated against the waveforms associated with these WFCs to avoid undue constraints and support test behaviors as necessary. For a listing of WFC Support, see Table 1 in Appendix E, "STIL Language Support."</p>	<p>\m</p> <p>\j Usage change for dot-1 compliance See Appendix E for details on: Vector Data Mapping Using \m Vector Data Mapping Using \j</p>
<pre>sigref_expr = serial_data;</pre>	<pre>sigref_expr = serial_data; variable_expr := variable_data; (-stil only, seq-comp only)</pre>

Table 1 STIL 99 Versus STIL (Continued)

STIL 99	STIL
(LABEL :)	(LABEL :)
"precondition all signals": on initial C in Pattern "pattern N": used in patterns.	"precondition all signals": on initial C in Pattern "pattern N": used in patterns.
User labels allowed in procedures and macros	User labels allowed in procedures and macros
V(ector) { (cyclized data) V { cyclized_data_assignments_only }	V(ector) { (cyclized data) V { cyclized_data_assignments_only }
}W(aveformTable) NAME ; W name ;	}W(aveformTable) NAME ; W name ;
C { cyclized_data_assignments_only }	C { cyclized_data_assignments_only }
Call name ; Call name { scan_and_cyclized_arguments }	Call name ; Call name { scan_and_cyclized_arguments }
Macro name ; Macro name { scan_and_cyclized_arguments }	Macro name ; Macro name { scan_and_cyclized_arguments }
Loop integer { ... } Allowed in setup procedures & some BIST procs; also used in seq-comp -stil99 Pattern blocks	Loop integer { ... } Allowed in setup procedures & some BIST procs; also used in seq-comp -stil99 Pattern blocks
	LoopData { ... } (-stil only, seq-comp only) Loop "var_name" { ... } (-stil only, seq-comp only)

Table 1 STIL 99 Versus STIL (Continued)

STIL 99	STIL
Vector statements only with constant WFC assignments	Vector statements only with constant WFC assignments
}	}
IDDQ TestPoint;	IDDQ TestPoint;
ScanChain CHAINNAME ;	ScanChain CHAINNAME ;
ActiveScanChains group_or_chain_names ;	ActiveScanChains group_or_chain_names ;
Used around Shift blocks; also in seq-comp load_unload procedures (without Shift)	Used around Shift blocks; also in seq-comp load_unload procedures (without Shift)
	F (ixed) { (cyclized-data)* (non-cyclized-data)* }
	F { cyclized_data_assignments }
	Used in procedures
	E (quivalent) ((\m) sigref_expr)* ;
	E sig \m sig ;
	Used in procedures
	See Appendix E under "Signal Constraints Using Fixed and Equivalent"

Table 1 STIL 99 Versus STIL (Continued)

STIL 99	STIL
<pre> Pattern "_pattern_" { Standard pattern structure: "precondition all signals": C { _po = ... _pi = ... } Structure change to this for proper bidi representation: W default_WaveformTable_name ; Macro "test_setup"; "pattern 0": ... pattern sequences follow } </pre>	<pre> Pattern user_specified_pattern_name { -or- Pattern "_pattern_" (fixed name by default) Standard pattern structure: "precondition all signals": C { _po = ... _pi = ... } Assignment change to this for proper bidi representation: W default_WaveformTable_name ; Macro "test_setup"; "pattern 0": ... pattern sequences follow } </pre>
<pre> Procedures { (Unnamed Procedures block) Procedures "diagnosis" { (In some BIST contexts) (PROCEDURE_NAME { TetraMAX ATPG flow uses fixed name to identify application. (pattern-statements)* support# and % assignment to specific types of groups: _po, _pi, and in some circumstances groups of bidi-only and clock- only signals. } } </pre>	<pre> Procedures "user_name" ; (user spec'ed) -or- Procedures { (Unnamed Procedures block) Procedures "diagnosis" { (In some BIST contexts) (PROCEDURE_NAME { TetraMAX ATPG flow uses fixed name to identify application. (pattern-statements)* support# and % assignment to specific types of groups: _po, _pi, and in some circumstances groups of bidi-only and clock- only signals. } } </pre>

Table 1 STIL 99 Versus STIL (Continued)

STIL 99	STIL
<pre>MacroDefs { (Unnamed MacroDefs block) (MACRO_NAME { TetraMAX ATPG flow uses fixed name to identify application test_setup, bist_setup macros do not use parameters W { }, C { }, V { } statements } } PROCEDURE_OR_MACRO_NAME { "load_unload" { (Scan procedure has fixed name) W { }, C { }, V { } statements Scan parameters may be specified before the Shift. Shift { W { }, C { }, V { } statements Scan parameters applied. } W { }, C { }, V { } statements Scan parameters may be specified after the Shift. }</pre>	<pre>MacroDefs "user_name" ; (user spec'ed) -or MacroDefs { (Unnamed MacroDefs block) (MACRO_NAME { TetraMAX ATPG flow uses fixed name to identify application test_setup, bist_setup macros do not use parameters W { }, C { }, V { } statements } } PROCEDURE_OR_MACRO_NAME { "load_unload" { (Scan procedure has fixed name) W { }, C { }, V { } statements Scan parameters may be specified before the Shift. Shift { W { }, C { }, V { } statements Scan parameters applied. } W { }, C { }, V { } statements Scan parameters may be specified after the Shift. }</pre>

Parameters Supported in Specific Contexts Only

In TetraMAX ATPG the # sign is primarily used — not the % sign. You should only use the # sign in very specific situations within certain procedure types. With a fixed name, like load_unload, the # sign is associated with groups associated as scanin and scanoutputs. The # references the scanins and scanouts. You can parameterized the _pi group on last_shift_launch. The parameters are constrained to _so_si_po_pi.

Predefined Signal Groups in STIL

To minimize typing that you can have to perform to define a DRC file by hand, TetraMAX ATPG has a number of predefined signal groups it recognizes. A SignalGroup is a method in STIL for describing a list of pins using a symbolic label. Symbolic labels allow a large number of pins to be referenced without a large amount of typing.

TetraMAX ATPG will accept the following predefined SignalGroups that:

- `_in` = input pins
- `_out` = output pins
- `_io` = bidirectional pins
- `_pi` = inputs + bidirectional pins
- `_po` = outputs + bidirectional pins
- `_si` = scan chain inputs
- `_so` = scan chain outputs

If your STIL DRC description defines a symbolic group with the same name as the predefined TetraMAX groups, then your definition supersedes the predefined definition.

Note: There is not a predefined signal group called `_clks`. TetraMAX ATPG does not create an `_clks` group the user needs to define the signals they want to be clocks in the flow, and put those signals into the `_clks` group. If the user is using the extended capture procedures with multiple cycles, then the user needs to create and define this group and reference that signal group in these procedures.

G

Defective Chain Masking for DFTMAX

The following sections of this appendix describe the flow for masking defective scan chains in DFTMAX compression:

- [Introduction](#)
- [Running the Flow](#)
- [Examples](#)
- [Limitation](#)

Introduction

Prior to the introduction of this feature, the flow for masking defective scan chains in DFTMAX compression was extremely inefficient. For example, if you found a scan hold violation on a chain from a chip returned from fabrication, you would want to generate patterns as if the entire compression chain was masked. The old flow for masking the defective chain used the `add_cell_constraints` command to place a constraint of “XX” on all the cells in the chain. However, this flow was problematic when the chain contained padding bits (the additional shift cycles required for every pattern; this situation occurs when the chain is either shorter than the longest compression mode chain or if the chain contains pipeline stages). The existing cells of the chain can be easily masked using the `add_cell_constraints` command. However, there’s no simple way to mask the padding bits. These additional bits, which also require masking, had to be manually identified. In order to resolve the patterns, the constraints were externally read in via a separate file.

In the solution described in this appendix, the external file is not required. Instead, TetraMAX ATPG identifies defective scan chains based on cell constraints during DRC. If every scan cell of a particular scan chain has a cell constraint of X or XX or OX, then the scan chain is treated as a defective scan chain when you specify both the `run_simulation` and `run_atpg` commands. As a result, padding measures originating from the defective chain is masked.

Running the Flow

The following sections describe the various processes for masking defective scan chains in DFTMAX compression:

- [Placing Constraints on the Defective Chain](#)
- [Generating Patterns](#)
- [Regenerating Patterns](#)

Note: Both the `run_simulation` and `run_atpg` commands support this flow.

Placing Constraints on the Defective Chain

Before running DRC, you will need to use the `add_cell_constraints` command to place the constraints on the defective chain. This solution uses a more relaxed condition for identifying defective chains during DRC. Before, every cell of the chain had to have cell constraint “XX” in order for it to completely mask it out. Now, to identify a chain as defective, every cell in a chain needs to have a cell constraint of “X,” “OX,” or “XX.”

Some examples, in increasing complexity, are as follows.

Example 1

```
add_cell_constraints XX c2 -all
```

Example 2:

```
add_cell_constraints X c2 0 2
```

```
add_cell_constraints XX c2 3
add_cell_constraints OX c2 4 5
```

Note that in Example 2, there are overlapping constraints in one cell: cell “3.”

Example 3:

```
add_cell_constraints X c2 0
add_cell_constraints XX c2 1 3
add_cell_constraints OX c2 4 5
```

Note in Example 3, that there are overlapping constraints in more than one cell: cells “1,” “2,” and “3.”

It is important to note that any new `add_cell_constraints` commands you specify override previous `add_cell_constraints` commands you specified if cells overlap between the two commands.

During the next step, DRC, TetraMAX ATPG will identify one or more defective scan chains, and the following message will appear:

```
Scan chain c2 has been identified as a defective chain
```

There are several different flow options available for generating or regenerating patterns. These flows are described in the following sections.

Generating Patterns

For generating patterns, you can use any of the following four flows:

- **Option 1:**

```
set_patterns external <pat_file>
run_simulation -store
write_patterns <new_pat_file>
```

- **Option 2 (Note that the simulation model needs to be complete):**

```
set_patterns external <pat_file>
run_simulation -resolve | -override
write_patterns <new_pat_file> -external
```

- **Option 3:**

```
set_patterns external <pat_file>
run_simulation -failure_file <fail_file>
set_patterns external <pat_file> -resolve <fail_file>
write_patterns <new_pat_file> -external
```

- **Option 4:**

```
set_patterns external <pat_file>
run_atpg -resolve
write_patterns <new_pat_file> -external
```

Regenerating Patterns

For regenerating patterns, use the following set of commands:

```
set_patterns -external <old_patterns>
add_faults -all
run_atpg -auto
write_patterns <new_pat_file> -internal
```

After generating or regenerating the patterns, the following report message will appear:
<number> scan chains have been completely masked.

Examples

This section shows several flow examples.

Figure 1 Generating Patterns

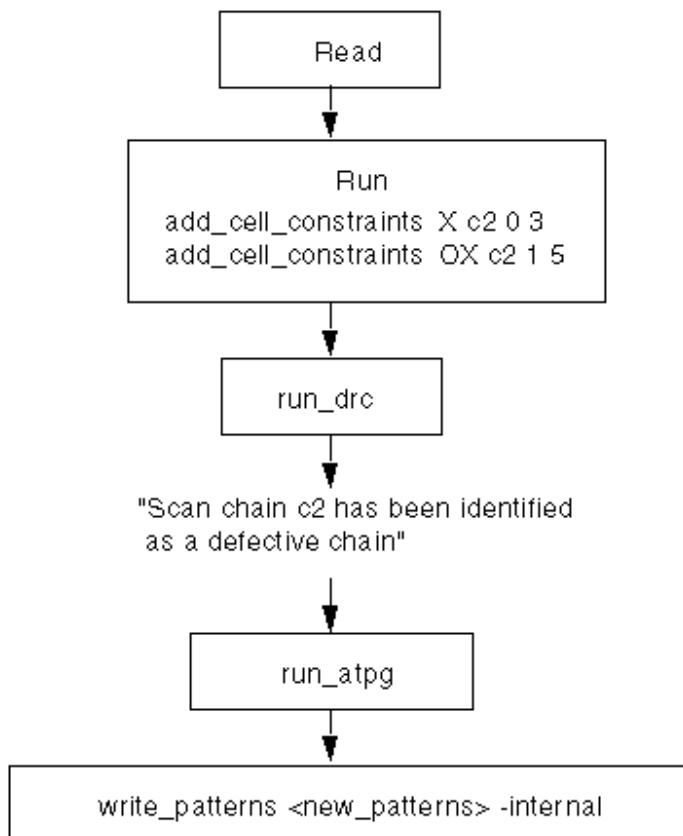


Figure 2 Regenerating Patterns

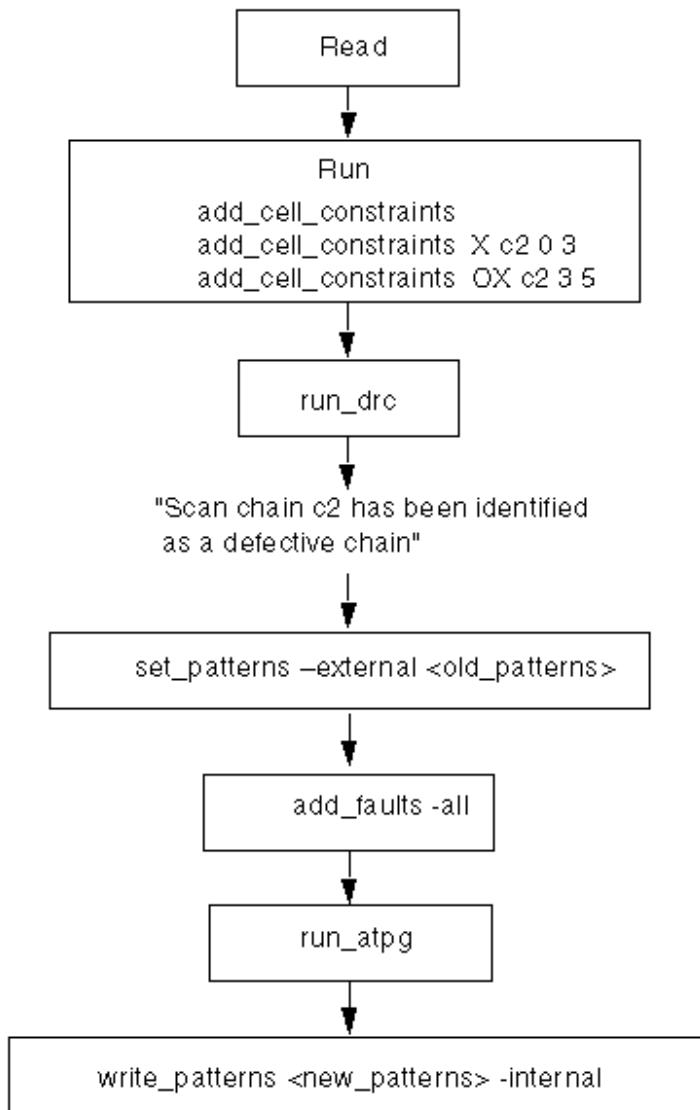
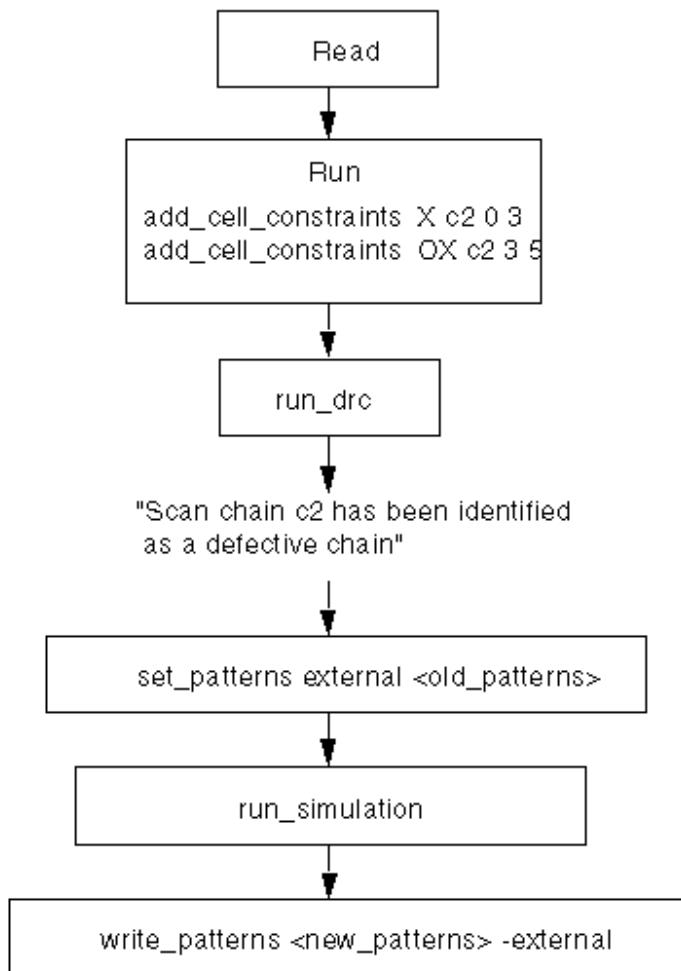


Figure 3 Re-Simulation and Updating Pattern Values

Note: In this case, scan cell 3 of chain c2 has a hold time violation and needs to be



Limitation

This flow does not include support for Full-Sequential patterns.