cadence®

Encounter® Test: Guide 4: Faults

Product Version 12.1.101
February 2013

Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Contents

List of Figures

Preface

About Encounter Test and Diagnostics

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

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

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

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

Typographic and Syntax Conventions

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

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

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

- Variables appear in Courier italic type.

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

- Optional arguments are enclosed in brackets.

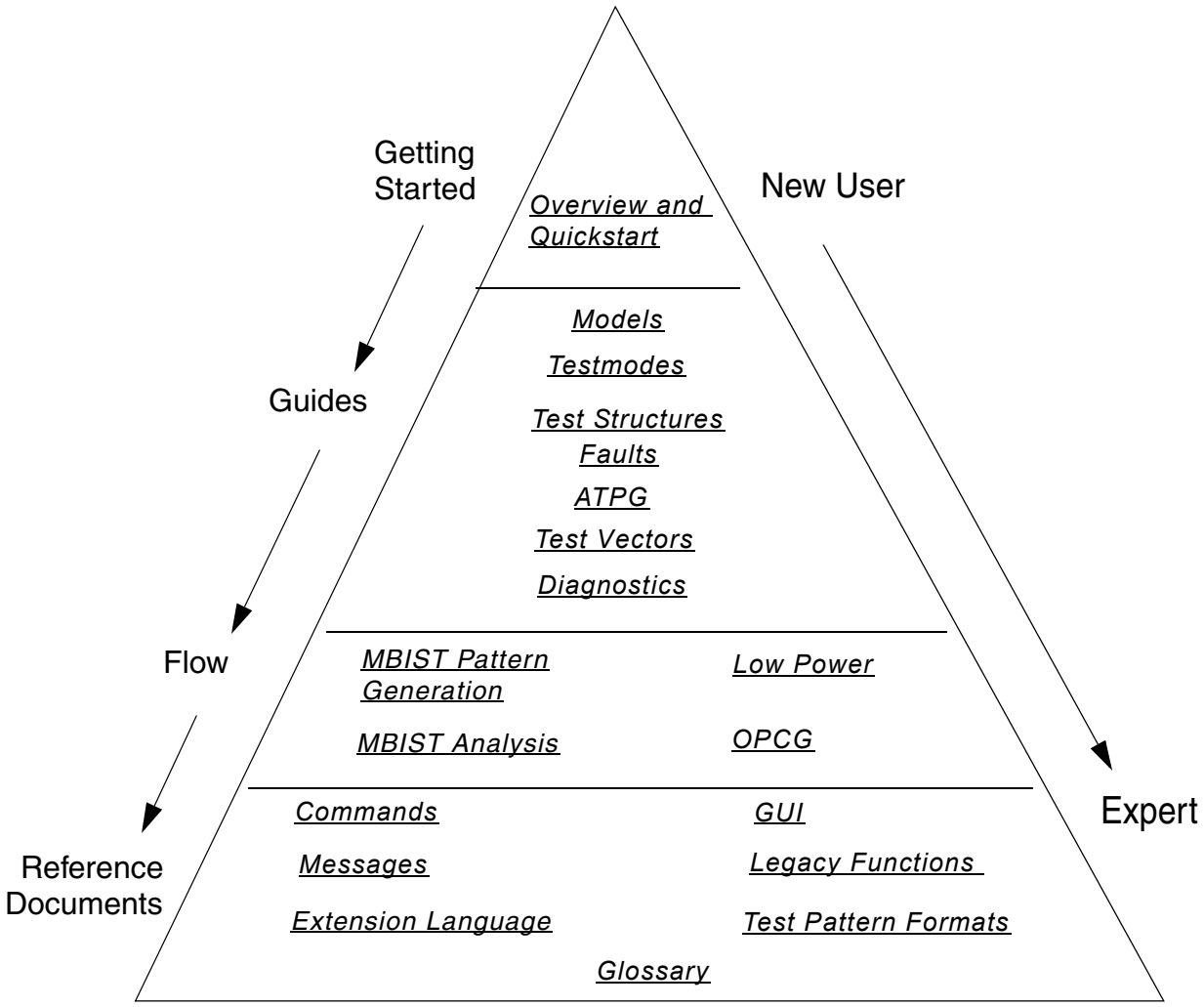
Example: `[simulation=gp|hsscan]`

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

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

Encounter Test Documentation Roadmap

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



Getting Help for Encounter Test and Diagnostics

Use the following methods to obtain help information:

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

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

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

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

Contacting Customer Service

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

- Cadence Online Customer Support

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

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

- Cadence Customer Response Center (CRC)

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

- IBM Field Design Center Customers

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

Encounter Test And Diagnostics Licenses

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

Using Encounter Test Contrib Scripts

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

What We Changed for This Edition

This is a new manual created using the information that was previously included in the *Modeling* and *Verification* Guides.

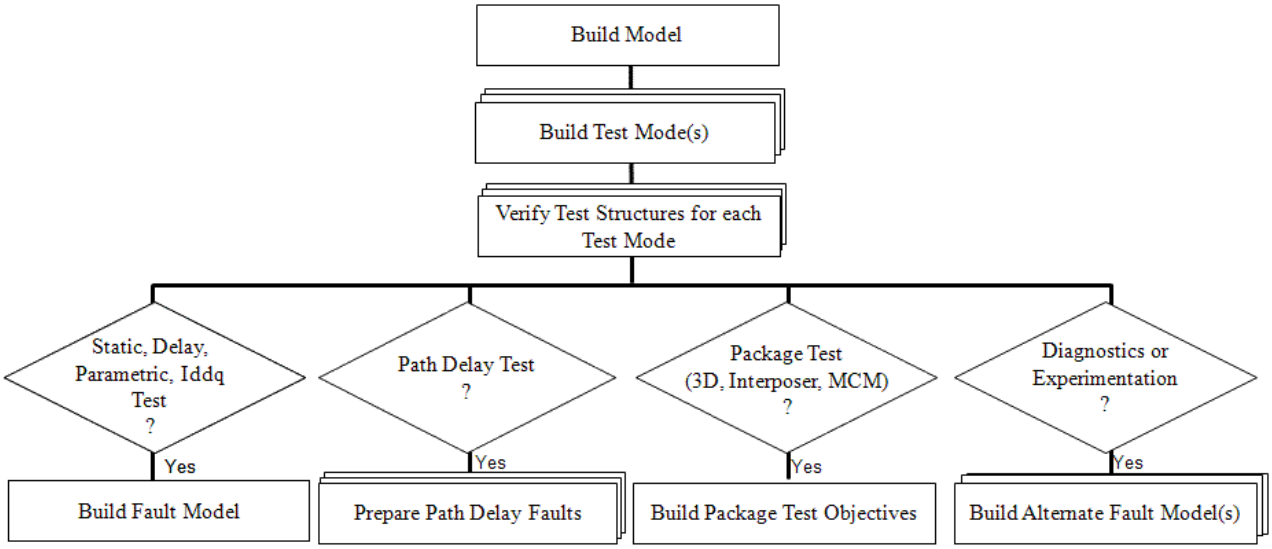
1

Building Fault Models

Building a fault model identifies the faults that ATPG, simulation, and diagnostics processes use. Refer to Concepts for descriptions of the Encounter Test fault types, attributes, status, and modeling options. The fault models you need depend on your test generation methodology.

Figure 1-1 on page 15 shows when to select various types of fault models and where building fault models fits in the processing flow. Each fault model and associated status is unique; faults detected in one fault model are not marked in any other fault model. Note that fault models may be built either before or after test modes are built. However, it is recommended that you build your test modes and verify them before building the fault model in order to avoid locking issues.

Figure 1-1 Fault Selection



The following sections describe how to build each of these types of fault or objective models:

Build Fault Model

The fault model can include Static (Stuck-at), Dynamic (Transition), Iddq, and Parametric (Driver/Receiver) faults and other pattern faults defined with fault rules. By default, the fault model includes static, dynamic, and Iddq faults. Parametric (driver/receiver) faults are included only if they are explicitly selected. Dynamic faults can be excluded if you are not going to do any delay testing on this design.

To build a fault model using the graphical interface, select *Models-Fault Model* from the Verification pulldown on the main GUI window. Refer to "Build Fault Model" in the *Encounter Test: Reference: GUI*.

To build a fault model using command line, use "build_faultmodel", which is documented in *Encounter Test: Reference: Commands*.

Example syntax for various uses of the `build_faultmodel` command are provided in the following sections:

- Build Fault Model Examples for Including/Excluding Faults
- Build Fault Model Examples for Cell Boundary Fault Model
- Build Fault Model Examples with Fault Rule Files
- Build Fault Model Examples with Special Handling of Ignored Faults

Build Fault Model Input

An Encounter Test Model is the only required input for creating the default fault model.

If test modes exist, they are also used as input to build fault model. The process of building the fault model is independent of any testmode(s) that might be created and therefore can be run either before or after defining the testmode(s). The process of identifying active faults per testmode and initializing the testmode fault status is done during `build_faultmodel` for all existing testmodes and is done during `build_testmode` for any testmodes built after the fault model exists.

Tip

It is recommended that you build the fault model after creating and verifying the testmodes in order to enable building multiple testmodes simultaneously and to avoid locking issues. However, if you need to build additional testmodes after the fault model exists, you can do that without having to re-build the fault model.

Fault Rule Files are used as input to building the fault model if you have pattern faults to add or want to modify the status or inclusion of specific faults (that is, ignore faults, mark faults redundant, or mark faults detected).

Build Fault Model Output

The binary data files are created and stored in the workdir/tbdata directory for use by subsequent fault oriented tasks such as ATPG, fault simulation, and diagnostics.

Build Fault Model Examples for Including/Excluding Faults

Example 1: Excluding Dynamic Faults

The following syntax turns off dynamic faults in the fault model. The default is to include dynamic faults.

```
build_faultmodel workdir=<directory> includedynamic=no
```

Example 2: Including Faults for Parametric Tests

The following syntax is used to include the driver and receiver faults which are the target of parametric tests:

```
build_faultmodel workdir=<directory> includedrvrcvr=yes
```

Example 3: Including Precollapsed Faults

Identifying Pre-Collapsed faults is common in the industry. However, some test tools include the faults in their fault models and simply mark their test status based on the faults they target for test generation. To provide a similar capability, Encounter Test allows specification of the following syntax to include the pre-collapsed faults for each AND/NAND, OR/NOR, and BUF/INV in the fault model and simply marks the extra faults as collapsed unless collapsefaults=no is also specified as in Example 4: Including Collapsed Faults.

```
build_faultmodel workdir=<directory> precollapsed=yes
```

Example 4: Including Collapsed Faults

In addition to the pre-collapsed faults discussed in the previous example, build fault model also analyzes the design and collapses faults that are logically equivalent between gates (such as faults on a string of buffers where the test for the s-a-0 fault on the output of the first BUF in the string will also test the s-a-0 faults on the outputs of all the other BUFs in that string). Fault collapsing, which is industry standard, allows the test generator to work on a single fault (called the Independent fault or the Equivalence Class Representative, ECR) and then fault simulation marks all the equivalent faults with the same status as the Independent fault. The following syntax turns off fault collapsing so every fault is independent.

```
build_faultmodel workdir=<directory> collapsefaults=no
```

Note that this does not include the pre-collapsed faults unless you include them with `precollapsed=yes` on the same command line.

Example 5: Excluding Scan Chain Faults

Excluding scan chain faults reduces the size of the fault model and reduces simulation time during create_scanchain_tests. Test generation will not have any scan faults to work on in this case so scanchain tests need to be run with good machine simulation (create_scanchain_tests `gmonly=yes` or `create_scanchain_delay_tests gmonly=yes`).

To exclude scan faults from the fault model, prior to build_faultmodel, create a full scan test mode and then specify its name in the excludescanmode keyword as shown in the example below:

```
build_faultmodel excludescanmode=FULLSCAN
```

Build fault model processes the specified testmode and removes faults along the scan chain that would be easily tested by scanchain or logic tests.

Notes

Although this technique is supported, it is rarely used.

An alternative that reduces test generation time while keeping the faults in the coverage is to use a priori fault markoff. See Removing Scan Faults from Consideration.

Build Fault Model Examples for Cell Boundary Fault Model

Encounter Test provides comprehensive fault modeling capabilities that includes modeling at technology cell boundaries and at the primitive level. The default is to fault at the primitive level.

Example 6: Building Industry Compatible Fault Model

The following syntax on `build_model` causes Encounter Test to model faults at the cell boundary level with a single pair of faults at the cell boundary even when a net fans out internally from a cell input pin. This fault modeling is the default behavior in when the model is built with `industrycompatible=yes`. The `build_model` process inserts a buffer into the flattened model so a single pair of faults can be built at the cell boundary, as shown in Figure 1-2 (`industrycompatible=yes`).

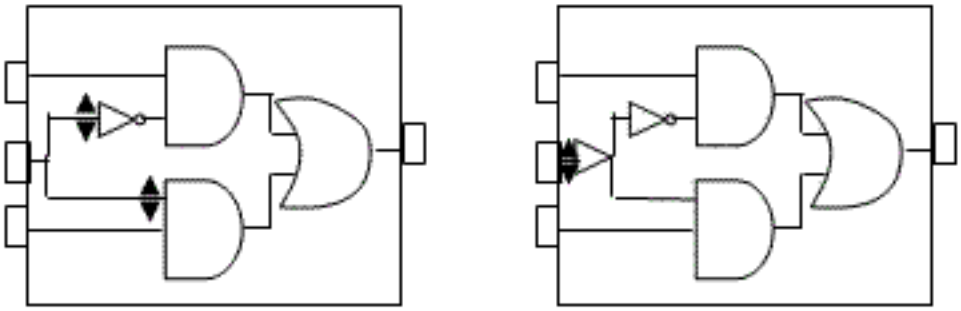
```
build_model workdir=<directory> industrycompatible=yes
build_faultmodel workdir=<directory>
```

Example 7: Building Cell Fault Model without Industry Compatible

The following syntax causes Encounter Test to model faults at the cell boundary level with multiple pairs of faults included when a net fans out internally from a cell input pin. This is depicted in Figure 1-2 on page 19 (`industrycompatible=no`).

```
build_faultmodel workdir=<directory> cellfaults=yes
```

Figure 1-2 Fault Model Compatibility with Other Tools



Build Fault Model Examples with Fault Rule Files

Example 8: Using Fault Rule Files named by Verilog Module Name

The following syntax selects fault rule files to be applied based on cell block name (`cellnamefaultrules=yes` causes `build_faultmodel` to search the directories specified in faultpath looking for each cell in the design) :

```
build_faultmodel cellnamefaultrules=yes faultpath=/dir1/faultrules1:/dir1/
faultrules2:/dir2/faultrules3
```

This assumes the fault rule files are named based on the cell name (Verilog module name). That is, the file name and the name on the Entity statement in the fault rule match.

```
File name is:  tech_AND
```

Fault rule file contains: Entity = tech_AND

When any instance of tech_AND is found in the model, this fault rule will be applied to modify the default faulting assumptions on that cell.

Example 9: Using Fault Rule Files with User-selected Names

A user-specified fault rule name may be identified with the FAULTS attribute in the Verilog (see Using Design Source Attributes to Add/Remove Faults) or may be identified directly on the command line with the faultrulefile keyword on the `build_faultmodel` command line.

The following syntax selects the fault rule files to be applied based on the file names in the FAULTS attributes and the directories listed in the faultpath keyword.

```
build_faultmodel  faultpath=/dir/faultrule1:/dir/faultrule2:/dir2/faultrule3
```

The following syntax selects the fault rule file to be applied based on the names of the files in the faultrulefile keyword:

```
build_faultmodel faultrulefile=/dir1/myfaultrule1,/dir1/myfaultrule2,/dir2/
myfaultrule3
```

Notes

If faultpath and faultrulefile are both specified on the command line files specified with the faultrulefile keyword are processed before the rules from the faultpath keyword. This is especially important for fault rules containing statements where the order of the statements matters, such as NOFAULT or FAULTONLY statements.

If is recommended that you not include more than one fault rule that contains rule data for the same entity (Verilog module, hierBlock in the model).

If the same filename happens to exist in more than one of the directories specified in FAULTPATH, the one found in the first directory is used (searching left to right in the list provided to the keyword).

If you use FAULTS attributes and the files are not found during `build_faultmodel`, a message is printed to let you know it is missing. If you use cellnamefaultrules=yes and name your files according to the cell name, `build_faultmodel` checks for fault rules for every cell and since it doesn't expect to find them for every cell, no message is printed for missing names.

Dynamic faults defined in a fault rule file are always included in a fault model, regardless of whether dynamic faults are requested to be created when the fault model is built.

All faults defined in a fault rule are included regardless of any NOFAULTing or other fault removal. If the fault is defined in a fault rule, it is assumed that you want it to be included.

Build Fault Model accepts compressed or gzipped versions of fault rule files to conserve memory when processing large files.

Build Fault Model Examples with Special Handling of Ignored Faults

Example 10: Including Ignored Faults in the Fault Model

By default, ignored faults are not included in the Encounter Test fault model; unless the model is built with `industrycompatible=yes`. A count of the number of uncollapsed ignored faults that would have been included is calculated and printed, but the actual faults cannot be reported. This behavior reduces the size of the fault model for customers who don't need details about these faults. An example of the report in the Build Fault Model log is below:

```
Global Ignored Static Fault Count:    100
Global Ignored Dynamic Fault Count:   200
```

The following syntax includes the ignored faults in the fault model and provides the ability to list them. However, the ignored faults are not collapsed, are not categorized, and are not included in any fault calculations. The output in the Build Fault Model log looks the same as without includeignore=yes; the same as the 2 lines above that provide the counts.

```
build_faultmodel includeignore=yes
```

Example 11: Accounting for Ignored Faults in Reports/Statistics

The following syntax analyzes the globally ignored static and dynamic faults and categorizes the ignored attribute.

```
build_faultmodel includeignore=yes ignorefaultstatistics=yes
```

On specifying the above keywords, build fault model accumulates global data (uncollapsed) fault counts for each of the categories for static and dynamic faults separately. These are then printed in the Build Fault Model log in place of the 2 line fault counts. An example is below.

```
Global Static Total Ignored (undetectable) Fault Count:   100
Ignored-Tied Fault Count:                   40
Ignored-Unconnected Fault Count:            20
Ignored-Blocked Fault Count:                40

Global Dynamic Total Ignored (undetectable) Fault Count:  200
Ignored-Tied Fault Count:                  80
Ignored-Unconnected Fault Count:           40
Ignored-Blocked Fault Count:               80
```

Notes

Using `includeignore=yes` without ignorefaultstatistics (or with `ignoredfaultstatistics=no`) causes the output of `report_faults ignored=yes` to report the ignored faults as `I` without any additional categorization.

In addition to categorizing the total fault counts in the `build_faultmodel` log, specifying `ignoredfaultstatistics=yes` also:

- causes the output of `report_faults ignored=yes` to categorize the ignored faults as It (tied), Iu (unconnected), or Ib (blocked). See Report Faults
- causes the ignored faults to be included in the global test coverage calculation.

Build Alternate Fault Models

Encounter Test supports the creation of alternate fault models as a means to temporarily replace or augment the standard Encounter Test fault model without having to rebuild the files associated with the standard fault model. One example of how the alternate fault model could

be advantageous is during diagnostic simulation where shorted net fault descriptions not typically found in the standard fault model, could be found in an alternate fault model, and thus able to be considered in diagnostic simulation. The alternate fault model file names are differentiated from the standard fault model files by adding your own identifier as a file name qualifier using either the Build Alternate Fault Model window or command line.

The following syntax builds an alternate fault model named SHORTEDNET with the shorted-nets faults from `prepare_shorted_nets` included in with the standard faults.

```
build_alternate_faultmodel workdir=<mydir> ALTFAULT=SHORTEDNET faultpath=<path to
faultrules created by prepare_shorted_nets> cellrulefaultnames=yes
```

Notes

For alternate fault models to be accessed by Encounter Test applications, the `ALTFAULT=`*`user_specified_name`* keyword must be specified or set in the environment. If this is not set, the standard Encounter Test fault model (assuming it has been created) is made available to the application.

Uncommitted fault status results for an alternate fault model cannot be committed against a master as there is no master alternate fault model. If `commit_tests` is run, patterns are commited but not the fault coverage. To gain credit in the master falut model built by `build_faultmodel`, the patterns must be resimulated without the `ALTFAULT` keyword prior to `commit_tests`.

To perform Build Alternate Fault Model using the graphical interface, refer to Build Alternate Fault Model in the *Encounter Test: Reference: GUI*.

For a complete description of the build_alternate_faultmodel syntax, refer to build_alternate_faultmodel in the *Encounter Test: Reference: Commands*.

`build_alternate_faultmodel` has the same fault selection options and input file requirements as build_faultmodel. The only difference in the output is the identification of the data by the ALTFAULT name.

Prepare Path Delay Faults

The faults for path delay test are identified by specifying a list of paths to be included and are saved by a user specified path name. You may build the path delay faults using `prepare_path_delay`. Or, you may have the path delay faults built as part of the `create_path_delay_tests` command by specifying a pathfile. In either case you can run one or more `create_path_delay_tests` experiments against those faults by specifying the pathname.

Prepare Path Delay is run using a command line such as the one below. Refer to prepare_path_delay in the *Encounter Test: Reference: Commands* for complete syntax.

```
prepare_path_delay workdir=myworkdir testmode=FULLSCAN pathfile=/dir/mypathfile
maxnumberpaths=10 pathname=mypaths
```

The `maxnumberpaths` is an optional keyword that indicates how many paths are to be created along the paths specified in the pathfile. The pathname is an optional keyword that allows you to name the path delay fault model; by default it is named by the name of the pathfile.

Prepare Path Delay Inputs

- Encounter Test Model from Build Model
- Test Mode from Build Test Mode
- Path file with the identification of paths for which path faults are to be created

Refer to Path File in *Encounter Test: Guide 5: ATPG* for syntax of this file.

Prepare Path Delay Outputs

A path delay fault model identified by the pathname or pathfile name.

To reference this path delay fault model in `create_path_delay_tests` use `pathname=<pathname specified on prepare_path_delay>` or, if no pathname was specified, `pathname=<name of the pathfile specified on prepare_path_delay>`.

To reference this path delay fault model on the GUI schematic, set the analysis context ALTFAULT to `Paths.<pathname>` or `Paths.<pathfile name>`

Example 1: prepare_path_delay pathfile=/dir/mypathfile

```
create_path_delay_tests pathname=mypathfile
GUI  ALTFAULT:  Paths.mypathfile
```

Example 2: prepare_path_delay pathfile=/dir/mypathfile pathname=mypaths

```
create_path_delay_tests pathname=mypaths
GUI ALTFAULT:  Paths.mypaths
```

The output log reports the number of path groups created as shown below. Use `report_path_faults` for more information about the paths.

```
INFO (TPT-315):  18 path groups are defined.  [end TPT_315]
```

Build Package Test Objectives

The package test objectives are stuck-driver, shorted-nets and optionally slow-to-disable objectives described in Concepts. The command is `build_sdtsnt_objectives` (`sdtsnt` means stuck-driver test/shorted-nets test).

The following syntax builds the stuck-driver and shorted-nets test objectives as well as the slow-to-disable objectives:

```
build_sdtsnt_objectives workdir=myworkdir objectivefile=myslowdisables
```

Note: The package test objectives can also be built with the fault model by specifying `sdtsnt=yes` on the `build_faultmodel` command line.

Build Package Test Objectives Input

- Encounter Test Model from Build Model
- Test Modes built with a mode definition file that indicates the test type is INTERCONNECT (for MCM, 3D or Interposer test) or IOWRAP for chip test. See Mode Definition file Statement TEST_TYPES in *Encounter Test: Guide 2: Testmode*s. Note that the testmode keyword is not specified on the `build_sdtsnt_objectives` command line; the objectives are built and made active for every applicable testmode.
- Objective file (optional). Identfies slow-to-disable faults for interconnect tests.

  The objectivefile is used to notify Encounter Test where to define slow-to-disable objectives. The syntax for the objective file:

  PIN<*short_chip_pin_name1*> PIN<*short_chip_pin_name2*>

  where the first *short_chip_pin_name1* identifies the backtrace point for the driver getting the disable transition (i.e. to be slow-to-disable tested) and the second *short_chip_pin_name2* identifies the backtrace point for the driver getting the enable transition.

  Each pair of pin names identifies a single net driven by both chips. This net is the target slow-to-disable objective net. Two objectives are defined for each pin pair where 1->Z and 0->Z are the required transitions on the driver being tested for slow disable.

Build Package Test Objectives Output

There are multiple representations of these objectives:

- The package test objectives model

- The stuck-driver test faults included in alternate fault model ##TB_SDT

When you run `create_interconnect_tests` or `create_iowrap_tests`, there is no need to identify the objectives, they are known from the existence of the package test objectives model.

If you want to analyze the stuck-driver objectives on the GUI or report the stuck-driver test faults, specify ALTFAULT in the analysis context or on the command line as ##TB_SDT.

2

Modifying Fault Model/Fault Status

Encounter Test allows specification of attributes in the model to add faults that would not be included by default or to remove faults that would be included by default.

Using Model Attributes to Add/Remove Faults

This section discusses the following topics:

- Fault Attribute Specification in Verilog Design Source
- Fault Attribute Specification in Edit Model File

Fault Attribute Specification in Verilog Design Source

Encounter Test allows specification of attributes in technology library or design source to add faults that would not be included by default or to remove faults that would be included.The following is a set of fault specifications that can be placed on instance pins of logic primitives and the values that are allowed for the attribute:

- PFLT (Pin FauLT) -- these are stuck-at (also known as static) faults
  - PFLT=”NO” – remove all stuck-at faults from this pin
  - PFLT=”0” – include only a stuck-at-0 on this pin
  - PFLT=”1” – include only a stuck-at-1 on this pin
  - PFLT=”01” – include both stuck-at-0 and stuck-at-1 on this pin
- TFLT (Transition FauLT) – these are transition (also known as delay or dynamic) faults
  - TFLT=”NO” – remove all transition faults from this pin
  - TFLT=”0” – include a slow-to-fall transition fault on this pin
  - TFLT=”1” – include a slow-to-rise transition fault on this pin

- TFLT="01" – include both slow-to-fall and slow-to-rise transition faults on this pin

- DFLT (Driver FauLT) -- these are driver objectives for parametric test
  - DFLT="NO" – remove the driver objective from this pin
- RFLT (Receiver FauLT) – these are receiver objectives for parametric test
  - RFLT="NO" – remove the receiver objective from this pin

The following attributes can be placed on the instance of a primitive or a cell (Verilog module) depending on the value of the attribute:

- `FAULTS=NO` - specified on an instance of a primitive to remove all faults from that primitive
- `FAULTS=<filename>` - specified on a cell (Verilog module) to include faults in the fault rule with the specified filename.

The following is an example of Verilog using PFLT and FAULT syntax – this example is unrealistic and is included just to show placement of the attributes:

Figure 2-1 Fault Attributes in Verilog

```
(* FAULTS="mypflt" *) module pflt_test (y, a1, a2, a3, b1, b2, b3);
input a1, a2, a3, b1, b2, b3;
output y;
wire nb1_S001, na1_S003;
(* FAULTS="NO" *)  nor y_G001(y, na1_S003, nb1_S001);
and nb1_G (nb1_S001, (* PFLT="01" *) b2, b1,(* PFLT="0" *) b3);
and na1_G (na1_S003, a2, a1,(* PFLT="NO" *) a3);
endmodule
```

In the above example,

- Pattern faults defined in file mypflt in one of the directories specified with the `build_faultmodel FAULTPATH` keyword will be included for each instance of this module
- All faults will be removed from the nor gate y_G001 for all instances of this module
- Faults SA0 and SA1 will be placed on pin b2 of instance nb1_G for each instance of this module (instead of only SA1 that would be included by default)
- Fault SA0 will be placed on pin b3 of instance nb1_G for each instance of this module (instead of both SA1 and SA0 that would be included by default)
- All static faults will be removed from pin a3 of instance na1_G for each instance of this module (instead of both SA1 and SA0 that would be included by default)

Notes

- The right parenthesis and the semicolon on the and instances MUST NOT be on the same line as the PFLT attribute as this is illegal Verilog. Encounter Test will issue a syntax error in this case.
- The PFLT, TFLT, and FAULTS attributes are valid only on primitives.

Fault Attribute Specification in Edit Model File

The PFLT, TFLT, DFLT, RFLT, and FAULTS attributes discussed in the previous section can be included in the model during `build_model` by specifying them in an Edit Model file rather than directly in the Verilog Design Source. To do that:

- create a file with Edit Model ADD ATTRIBUTE statements.
- run: `build_model editfile=<filename>`
- run: `build_faultmodel <your normal options>` and the faults defined with the attributes in the edit file will be included/removed as appropriate.

To include the same attributes as shown in the Verilog in the previous section, the file would look like this:

```
ADD ATTRIBUTE FAULTS=mypflt on CELL pflt_test ;
ADD ATTRIBUTE FAULTS=NO on INSTANCE y_G001 of CELL pflt_test ;
ADD ATTRIBUTE PFLT=01 on PIN b2 on INSTANCE nb1_G of CELL pflt_test ;
ADD ATTRIBUTE PFLT=0 on PIN b3 on INSTANCE nb1_G of CELL pflt_test ;
ADD ATTRIBUTE PFLT=NO on PIN a3 on INSTANCE na1_G of CELL pflt_test ;
```

Note: Attributes in the Verilog source can be overridden with an edit file with CHANGE ATTRIBUTE or DELETE ATTRIBUTE edit model statements.

Using Fault Rule Specification to Add/Remove Faults

A fault rule may be used to:

- add user-defined pattern faults to the fault model. These faults are defined with a pattern of required module input values and expected module output values with and without the presence of the fault. For dynamic pattern faults, there is also an initial value on the module inputs.

- identify faults as IGNORE, PTAB, DETECTED (i.e., Tested) , REDUNDANT.
- remove all faults from an instance within module (NOFAULT INSTANCE name IN MODULE name)
- remove all faults from an instance of a module (NOFAULT BLOCK name)
- remove all faults from the entire module (NOFAULT MODULE name)
- include faults only on an instance within a module (FAULTONLY INSTANCE name IN MODULE name)
- include faults only in an instance of a module (FAULTONLY BLOCK name)
- include faults only in a module (FAULTONLY MODULE name)

There can be multiple NOFAULT and FAULTONLY statements to identify the complete set of blocks to be faulted or not faulted. See Fault Rule File Syntax in Appendix A for complete details on the syntax and usage of fault rule file statements including examples.

Preparing a Fault Subset

A fault subset is a set of static, dynamic, and/or parametric faults that are used as input to a Test Generation or Fault Simulation task. Preparation of a fault subset includes reading an input fault list which specifies which faults to include and optionally assigns status to the specified faults. The output is an experiment with the fault subset for use as input to Test Generation/Fault Simulation runs.

The command line to use is shown in the following syntax:

```
prepare_fault_subset workdir=myworkdir testmode=FULLSCAN experiment=fltsubset1 /
faultlist=/dir/faults4fs.subset1
```

The fault list input may be specified as a file using the faultlist keyword as shown above or may be created with report_faults and piped into the `prepare_fault_subset` command. These options are discussed in more detail in the Prepare Fault Subset Input section below.

Notes

If the status of the faults are updated during `prepare_fault_subset`, the vectors created using this subset will not be able to be committed.

A GUI form is not available for invocation of prepare_fault_subset; it must be run with a command line.

For a complete description of the `prepare_fault_subset` syntax, refer to "prepare_fault_subset" in the *Encounter Test: Reference: Commands*

Prepare Fault Subset Input

- Encounter Test model from Build Model
- Encounter Test fault model from Build Faultmodel
- Encounter Test test mode from Build Testmode
- A fault list created manually or from report faults. See the following sections for more information.

Using the Output from Report Faults to Create an Input Fault List

Typical output from `report_faults` uses pin names, fault type, and indexes to identify the faults.

The input to `prepare_fault_subset` must be referenced by pin name and fault type or by fault index. The default is to use the pin name and fault type. If the input is to be processed using fault index, specify `prepare_fault_subset -I`.

Use one of the following methods to process the output of `report_faults` with `prepare_fault_subset`.

- Direct the output of the report_faults command to `prepare_fault_subset` using the following syntax:

  ```
  report_faults faults=1:10 | prepare_fault_subset -I
  ```

- The output of the `report_faults` command can be written to a file, edited (if desired), then read by `prepare_fault_subset` using the `faultlist=` *<keyword>* as shown in the following steps:

  ```
  report_faults faults=1:10 > report_faults.out
  ```

```
prepare_fault_subset -I faultlist=report_faults.out
```

or

```
report_faults faults=1:10 > report_faults.out
prepare_fault_subset faultlist=report_faults.out
```

- The output of `report_faults` can be directed to a file using the following syntax:

  ```
  report_faults faults=1:10 > myfaultlist
  ```

  `myfaultlist` can then be edited, then either piped or passed into `prepare_fault_subset` using the following syntax:

  ```
  cat myfaultlist | prepare_fault_subset <parameter>
  ```

  or

  ```
  prepare_fault_subset faultlist=myfaultlist <other parameters>
  ```

  `report_faults` output can also be piped directly into `prepare_fault_subset`.

Notes:

1. The selection of faults in report_faults may be done with ranges (as shown in the previous examples, `report_faults faults=1:10` includes 10 faults with indexes 1 through 10), or lists (`faults=1,3,5,7` includes only those 4 faults) or a combination of these (`faults=1:10,13,27:29,50` includes 15 faults 1 through 10, 13, 27 through 29, and 50).
2. The previous examples all show how to obtain the global status for the faults. To obtain the status from a test mode, specify `testmode=`*`testmode_name`*. The list shows faults for the specified testmode and also shows the gloabl status of all faults. If you use the list with two sets of status as input to `prepare_fault_subset`, the last one should win; but it is not recommended since this will cause confusion.
3. Refer to the *Encounter Test: Reference: Commands* for additional report_faults and prepare_fault_subset options.
4. The fault status characters processed by `prepare_fault_subset` are the characters viewed in the output of Report Faults and are limited to `U` (untested), `T` (tested), `P` (possibly tested), `3,X, or C` (possibly tested at best), and `R` (redundant).

Manually Creating an Input Faultlist File

An input file for Report Faults and Prepare Fault Subset can be manually created. The following two file formats are acceptable:

The file must include at least two columns: fault index and fault status for fault index references, as shown in the following example:

```
21 U
22 T
23 U
```

or

The file must include pin name, fault type (SA0, SA1, SR, SF), and a fault status character for fault pin name references, as shown in the following example:

```
A                   OSA0 T
A                   OSA1 U
hadd12.carry2.A0    ISA1 U
```

Prepare Fault Subset Output

An experiment in tbdata consisting of the fault subset data and an empty vectors file identified by the specified experiment name.

Using Fault Subset for ATPG / Fault Simulation

When you run ATPG or fault simulation specify the same experiment name as you specified for `prepare_fault_subset` and specify append=yes.

For example, if you run: `prepare_fault_subset workdir=dir testmode=FULLSCAN_DELAY experiment=subset1 faultlist=/dir/mysubsetfile`

To use this fault subset in one of these commands you would run:

- `create_logic_tests testmode=FULLSCAN_DELAY experiment=subset1 append=yes`
- `create_logic_delay_tests testmode=FULLSCAN_DELAY experiment=subset1 append=yes`
- `create_parametric_tests testmode=FULLSCAN_DELAY experiment=subset1 append=yes`
- `analyze_vectors testmode=FULLSCAN_DELAY inexperiment=2analyze experiment=subset1 append=yes`

Note: Once you have run a command using the fault subset, the status of the faults in that subset has changed to match the results from the command you just ran. And the vectors generated by that run are identified with the experiment from the subset. You may append

again using the same or another command, but the vectors and status from the preceding run will be used as input (like a normal append run).

Removing Scan Faults from Consideration

A high number of scan faults can be detrimental to the speed of throughput of downstream test generation applications. The command, `prepare_apriori_faults`, can be used to mark scan faults as tested prior to beginning the test generation process. `prepare_apriori_faults` marks scan faults as tested but does not remove them from the fault model.

An alternative method to removing scan faults from consideration is to exclude them from the fault model. Refer to "Example 5: Excluding Scan Chain Faults" on page 9 for details.

Refer to "prepare_apriori_faults" in the *Encounter Test: Reference: Commands* for syntax information. The syntax for the command is shown below:

```
prepare_apriori_faults WORKDIR=<directory> TESTMODE=<testmode name>
```

Prepare Apriori Faults Inputs

- Encounter Test model from build_model
- Full Scan Test mode from build_testmode
- Fault model from build_faultmodel

Prepare Apriori Faults Outputs

The fault status for the testmode updated with scan faults marked as tested.

Preparing an Ignore Faults File

The `prepare_ignore_faults` command creates a fault rule file for a cell with a list of faults to be marked as ignored when processing the cell at a higher packaging level where the faults were already proven untestable (redundant) at the cell level. By default, the command processes Redundant faults but options exist to select Possibly Testable at Best (PTAB) faults or faults untestable due to X-source. There is also an option to mark the redundant faults as Redundant instead of Ignored.

To prepare ignore faults, refer to "prepare_ignore_faults" in the *Encounter Test: Reference: Commands*. A sample command line is shown below:

```
prepare_ignore_faults WORKDIR=mywd TESTMODE=FULLSCAN OUTFAULTDIR=./myfaultdir
```

Prepare Ignore Faults Inputs

- Encounter test model from `build_model`
- Test mode from `build_testmode`
- Fault model and status from `build_faultmodel`
- For redundant or untestable due to X-source faults to be processed an experiment from ATPG or committed ATPG data must be provided.

Prepare Ignore Faults Outputs

The output from `prepare_ignore_faults` is the file specified by the `outfaultfile` and/or `OUTFAULTDIR` keywords.

The following is an example of the fault rule data in the output file:

```
Entity = BTTP
/* Start prepare_ignore_faults fault rule creation. Version 7.3 Date 19970306215136
*/
IGNORE { PATTERN 1 BLOCK
BLTI_p_model_p_blth_p_B_LATCH_p_BLT_DATAI_OUT$OMX104.0000100.master }
IGNORE { PATTERN 2 BLOCK
BLTI_p_model_p_blth_p_B_LATCH_p_BLT_DATAI_OUT$OMX104.0000100.master }
IGNORE { SA0 PIN BLTI_p_DRV103.0000101.01 }
IGNORE { SA0 PIN BLTI_p_DRV106.0000101.01 }
IGNORE { SA0 PIN BLTI_p_DRV110.0000101.01 }
                .
                .
                .
IGNORE { SA1 PIN btext_p_mb_data_cio000.EnbAnd.A2 }
IGNORE { SA1 PIN btext_p_mb_data_cio001.EnbAnd.A2 }
IGNORE { SA1 PIN btext_p_mb_data_cio002.EnbAnd.A2 }
IGNORE { SA1 PIN btext_p_mb_data_cio003.EnbAnd.A2 }
IGNORE { SA1 PIN btext_p_mb_data_cio004.EnbAnd.A2 }
```

Note: If you edit this fault rule file or generate your own fault rule file, keep in mind that the ENTITY statement is required for all fault rule files.

Preparing Detected Faults File

The `prepare_detected_faults` command creates a fault rule file for a cell (ordinarily a full chip) with a list of faults to be marked as detected when processing the cell at a higher packaging level (ordinarily the SCM containing just this chip). This capability is useful if it is known that the same tests applied to the chip are to be applied to the SCM. The fault rule file created by this step can be read in when the fault model is later created for the SCM. This will cause the faults that were detected at the chip level to be initialized to detected status in the fault model for the SCM, thus circumventing the need to explicitly perform test generation for these faults at the SCM level.

To prepare detected faults, refer to "prepare_detected_faults" in the *Encounter Test: Reference: Commands*. A sample of the command is shown below:

```
prepare_detected_faults WORKDIR=mywd TESTMODE=FULLSCAN EXPERIMENT=tg1
OUTFAULTDIR=./myfaultdir
```

Prepare Detected Faults Inputs

- Encounter test model from build_model
- Test mode from build_testmode
- Fault model from build_faultmodel
- Fault status from test generation on the cell; experiment or committed. Refer to "Test Generation and Fault Simulation" in the *Encounter Test: Guide 5 : ATPG*.

Prepare Detected Faults Outputs

The output from `prepare_detected_faults` is the file specified by the `outfaultfile` and/or `OUTFAULTDIR` keywords.

The following is an example of the fault rule data in the output file:

```
Entity = newpart
DET { PATTERN 1 BLOCK c1.b1.f1.dff_primitive.slave }
DET { PATTERN 4 BLOCK c1.b1.f1.dff_primitive.slave }
DET { PATTERN 5 BLOCK c1.b1.f1.dff_primitive.slave }
```

3

Reporting Faults, Test Objectives, and Statistics

This chapter covers the Encounter Test tasks for reporting individual test objectives and reporting test coverage statistics.

Report Faults

To produce a fault report using the graphical interface, refer to "Report Faults" in the *Encounter Test: Reference: GUI*.

For a complete description of the `report_faults` syntax, refer to report_faults in the *Encounter Test: Reference: Commands*.

The syntax for the report_faults command is given below:

```
report_faults workdir=<directory> testmode=<name> experiment=<name> /
faultstatus=untested faulttype=static inputcommitted=no ignored=yes
```

To view a list of faults in the fault model or objective model, select Report - Faults - Logic Faults from the main menu. Report Faults displays the contents of a fault model and/or an objective model in the Encounter Test Log window.

Report Faults Output

Report Faults produces the following information:

- Fault - Each fault is assigned a unique identifier called the fault index.
- Status - The status of the fault. Refer to for more information.
- Type - The type of the fault in the fault model. Refer to for more information.
- Func - The simulation function of the logic driving the net where the fault effect origin.
- Fault pin name/Propagation net name.

Additional ouptut may be displayed through the use of Advanced options.

Report_faults Options

The fault status (status) and fault type (type) keywords are used to specify the data to report. Refer to "report_faults" in the *Encounter Test: Reference: Commands* for more information.

- `Fault Status`

  Fault status refers to the set of attributes of a fault that change during processing by Encounter Test applications. Of primary interest among these attributes are those that denote whether a test has been generated for the fault, but there are several other attributes that affect Encounter Test processing or which may be useful in the analysis of a design's testability.

  The report faults function optionally prints a status key as shown in Figure 3-1, which explains the notation that is used in the fault listing to denote fault status. The key includes some Fault Attributes and Fault Test Status.

Figure 3-1 Fault Status Key - available through msgHelp TFM-705 or optionally included in log (reportkey=yes)

```
Status key:
i=   inactive in testmode (Inactive faults ( i))
c=   collapsed - listed after equiv. class representative fault(Collapsed (C))
u=   untested  - not processed (Untested – Not Processed (u))
I=   ignored   (unclassified) (Ignored Faults (I, Iu, Ib, It))
It=  ignored   (tied) (Ignored Faults (I, Iu, Ib, It)
Iu=  ignored   (unconnected) (Ignored Faults (I, Iu, Ib, It)
Ib=  ignored(blocked) (Ignored Faults (I, Iu, Ib, It)
T=   tested by simulation (Tested by Simulation (T))
Ti=  tested by implication (Tested by Implication (Ti))
P=   possibly tested (limited not reached ) (Possibly Tested (P))
Tpdl=tested (possibly detected limit reached) (Tested by Possibly Detected Limit (Tpdl))
Tm=  tested in another mode (Tested in Another Mode (Tm))
Tus= tested user specified ( DETECTED Statement ) (Tested User Specified (Tus))
A=   Aborted (Aborted (A))
R=   redundant (Redundant (R))
Rus= redundant:user specified (Redundant: User Specified (Rus))
Ud=  untestable:undetermined reason (Untestable)
Uus= untestable:user specified (Untestable: User Specified (Uus))
Ulh= untestable:linehold inhibits fault control/observe (Untestable: User Specified
(Uus))
Utm= untestable:testmode inhibits fault control/observe (Untestable: Testmode inhibits
fault control/observe (Utm))
Ucn= untestable:constraint
Uxs= untestable:X sourced or sinked (Untestable: X-sourced or sinked (Uxs))
Usd=  untestable:sequential depth (may be testable with more sequential depth)(U
ntestable: Sequential depth (Usd))
Ugt= untestable:global termination (Untestable: Global term (Ugt))
```

```
Unio=untestable:Seq or control not specified to target fault (PI to PO Path)
(Untestable: Seq or control not specified to target fault (Unio , Unil, Unlo, Unra, Uner))
Unil=untestable:Seq or control not specified to target fault (PI to LATCH
Path)(Untestable: Seq or control not specified to target fault (Unio , Unil, Unlo, Unra, Uner))
Unlo=untestable:Seq or control not specified to target fault (LATCH to PO
Path)(Untestable: Seq or control not specified to target fault (Unio , Unil, Unlo, Unra, Uner))
Uner=untestable:Seq or control not specified to target fault (interdomain LATCH
to LATCH) (Untestable: Seq or control not specified to target fault (Unio , Unil, Unlo, Unra, Uner))
Unra=untestable:Seq or control not specified to target fault (intradomain LATCH
to LATCH)(Untestable: Seq or control not specified to target fault (Unio , Unil, Unlo, Unra, Uner))


PTAB (Possibly Tested at Best) status:
        3   =    PTAB 3-State
        P3  =    PTAB 3-State        :Possibly Tested (Possibly Tested (P))
        T3  =    PTAB 3-State        :Tested(limited reached)(Tested (T))
        X   =    PTAB X-state        :
        PX  =    PTAB X-state        :Possibly Tested (Possibly Tested (P))
        TX  =    PTAB X-state        :Tested(limited reached)(Tested (T))
        C   =    PTAB Clock Stuck Off
        PC  =    PTAB Clock Stuck Off   :Possibly Tested (Possibly Tested (P))
        TC  =    PTAB Clock Stuck Off   :Tested(limited reached)(Tested (T))
        TiC =    PTAB Clock Stuck Off   :Tested by implication (Tested by Implication
(Ti))
```

Refer to Fault Attributes on page 68 and Fault Test Status on page 73 for more information.

- `Fault Type`

  Fault type refers to the types of faults to be reported. The `report_faults` command optionally prints a type key, as shown in Figure 3-2, which explains the notation in the listing.

Figure 3-2 Fault Type key - available through msgHelp TFM-305 or optionally included in log (reportkey=yes)

```
Type key:
&  = AND Grouped          |  = OR Grouped
ISA0 = stuck-at-zero fault on an input pin (Static Pin Faults)
ISA1 = stuck-at-one fault on an input pin(Static Pin Faults)
ISR  = slow-to-rise fault on an input pin (Dynamic Pin Faults)
ISF  = slow-to-fall fault on an input pin (Dynamic Pin Faults)
OSA0 = stuck-at-zero fault on an output pin (Static Pin Faults)
OSA1 = stuck-at-one fault on an output pin (Static Pin Faults)
OSR  = slow-to-rise fault on an output pin (Dynamic Pin Faults)
OSF  = slow-to-fall fault on an output pin (Dynamic Pin Faults)
DPAT = dynamic pattern fault (Dynamic Pattern Faults)
SPAT = static pattern fault (Static Pattern Faults)
DRV0 = driver fault driving 0 (Parametric (Driver/Receiver))
DRV1 = driver fault driving 1 (Parametric (Driver/Receiver))
RCV0 = receiver fault receiving 0 (Parametric (Driver/Receiver))
RCV1 = receiver fault receiving 1 (Parametric (Driver/Receiver))
PSH0 = net shorted to 0 fault (Shorted Nets Pattern Faults)
PSH1 = net shorted to 1 fault (Shorted Nets Pattern Faults)
IDDQ0= Iddq logic 0 fault (IDDq)
IDDQ1= Iddq logic 1 fault (IDDq)
QPAT = Iddq pattern fault (IDDq)
```

- `Faults`

  This keyword may be used to report specific fault indexes. Use `status=all` with faults.

- `Faultlocation=cellpin`

  If the `faultlocation` keyword is set as `cellpin`, the output log reports the highest level cell boundary pin associated with the fault for a cell boundary fault model, as shown in the following sample output:

```
INFO (TFM-705):  Global   Fault List:
Fault Status   Type   Func    Fault pin name
1 u            OSA0    PI       a[0]
2 u            OSA1    PI       a[0]
5 u            OSA0    PI       b[0]
6 u            OSA1    PI       b[0]
9 u            OSA0    PO       out
10 u           OSA1    PO       out
13 u           ISA1    test1    t1.in1
16 u           ISA1    test1    t1.in2
19 Pu          OSA0    test1    t1.out1
20 Pu          OSA1    test1    t1.out1
23 u           ISA1    test1    t1.in1
26 u           ISA1    test1    t1.in2
29 Pu          OSA0    test1    t1.out1
30 Pu          OSA1    test1    t1.out1
```

- `Faultlocation=pin`

  If the `faultlocation` keyword is set as `pin`, the output log reports the primitive pin name associated with the fault, as shown in the following sample output:

```
INFO (TFM-705):  Global   Fault List:

Fault Status    Type    Func      Fault pin name
   1 u          OSA0     PI       Pin.f.l.topcell.nl.a[0]
   2 u          OSA1     PI       Pin.f.l.topcell.nl.a[0]
   5 u          OSA0     PI       Pin.f.l.topcell.nl.b[0]
   6 u          OSA1     PI       Pin.f.l.topcell.nl.b[0]
   9 u          OSA0     PO       Pin.f.l.topcell.nl.out
  10 u          OSA1     PO       Pin.f.l.topcell.nl.out
  13 u          ISA1     AND      Pin.f.l.topcell.nl.t1.__i0.A1
  16 u          ISA1     AND      Pin.f.l.topcell.nl.t1.__i0.A2
  19 Pu         OSA0     AND      Pin.f.l.topcell.nl.t1.__i0.01
  20 Pu         OSA1     AND      Pin.f.l.topcell.nl.t1.__i0.01
  23 u          ISA1     AND      Pin.f.l.topcell.nl.t1.t2.__i0.A1
```

- `Faultlocation=net`

  If the `faultlocation` keyword is set as `net`, the output log reports the name of the net where the fault effect originates, as shown in the following sample output:

```
INFO (TFM-705):  Global   Fault List:
Fault Status    Type   Func      Propagation net name
    1 u         OSA0    PI       Net.f.l.topcell.nl.a[0]
    2 u         OSA1    PI       Net.f.l.topcell.nl.a[0]
    5 u         OSA0    PI       Net.f.l.topcell.nl.b[0]
    6 u         OSA1    PI       Net.f.l.topcell.nl.b[0]
    9 u         OSA0    PO       Net.f.l.topcell.nl.out
```

```
   10 u          OSA1    PO       Net.f.l.topcell.nl.out
   13 u          ISA1    AND      Net.f.l.topcell.nl.t1.out1
   16 u          ISA1    AND      Net.f.l.topcell.nl.t1.out1
   19 Pu         OSA0    AND      Net.f.l.topcell.nl.t1.out1
   20 Pu         OSA1    AND      Net.f.l.topcell.nl.t1.out1
   23 u          ISA1    AND      Net.f.l.topcell.nl.t1.t2.out21
```

- `hierstart`

  Specifies the index value or the name of the hierblock. This keyword is used to request the fault list for the specific heirarchical block. The default value is a non-hierarchical format.

- `hierend`

  Specifies the number of hierarchical levels up to which faults are to be listed. This keyword can be set to values <`depth`>, `techcell`, or `primitive`. The default value is `primitive`. Following is a sample of the report format for a hierarchical fault listing. The name of the hierblock is displayed before each hierblock fault heading and fault list.

```
INFO (TFM-705):  Global   Fault List:
Hier Block/Cell: f.l.topcell.nl/topcell, Hier Block ID: 0
Fault Status    Type   Func      Fault pin name
    1 u         OSA0    PI       Pin.f.l.topcell.nl.a[0]
    2 u         OSA1    PI       Pin.f.l.topcell.nl.a[0]
    5 u         OSA0    PI       Pin.f.l.topcell.nl.b[0]
    6 u         OSA1    PI       Pin.f.l.topcell.nl.b[0]
    9 u         OSA0    PO       Pin.f.l.topcell.nl.out
   10 u         OSA1    PO       Pin.f.l.topcell.nl.out
   13 u         ISA1    AND      Pin.f.l.topcell.nl.t1.__i0.A1
   16 u         ISA1    AND      Pin.f.l.topcell.nl.t1.__i0.A2
   19 Pu        OSA0    AND      Pin.f.l.topcell.nl.t1.__i0.01
   20 Pu        OSA1    AND      Pin.f.l.topcell.nl.t1.__i0.01
   23 u         ISA1    AND      Pin.f.l.topcell.nl.t1.t2.__i0.A1
Hier Block/Cell: t22/test2, Hier Block ID: 11
Fault Status    Type   Func      Fault pin name
   63 u         ISA1    AND      Pin.f.l.topcell.nl.t22.__i0.A1
   66 u         ISA1    AND      Pin.f.l.topcell.nl.t22.__i0.A2
   69 Pu        OSA0    AND      Pin.f.l.topcell.nl.t22.__i0.01
   70 Pu        OSA1    AND      Pin.f.l.topcell.nl.t22.__i0.01
```

Report Fault Coverage Statistics

The effectiveness of the test vectors produced by logic test generation is expressed as the percentage of modeled faults that were detected by simulation of the test patterns. This percentage is generally called test coverage or fault coverage.

To produce a fault statistics report using the graphical interface, refer to "Report Fault Statistics" in the *Encounter Test: Reference: GUI*.

To produce a fault statistics report using command line, refer to ""report_fault_statistics" in the *Encounter Test: Reference: Commands*.

An example syntax for the `report_fault_statistics` command is given below:

```
report_fault_statistics  workdir=mywd testmode=FULLSCAN experiment=tg1
```

Encounter Test calculates and prints the following calculations:

- %TCov

  Percentage Test Coverage calculated as the number of active faults with a status of *tested* or *tested in another mode* divided by the total number of active faults.

- %ATCov

  Percentage Adjusted Test Coverage calculated as the number of active faults with a status of *tested* or *tested in another mode* divided by the number of non-redundant, active faults.

- %PCov

  Percentage Possibly Detected Coverage calculated as the number of active faults with a status of *tested* or *tested in another mode* or *possibly tested* divided by the total number of active faults.

- %APCov

  Percentage Adjusted Possibly Detected Coverage calculated as the number of active faults with a status of *tested* or *tested in another mode* or *possibly tested* divided by the number of non-redundant, active faults.

Note: The test coverage fault statistics represent the count and percentage of the faults that are defined in the fault model and are active in at least one test mode.

Encounter Test displays the faults that are defined in the fault model but are not active in any test mode and the maximum attainable global test coverage due to the inactive faults in a separate section in the log.

These statistics may be reported for:

- The global design

  Global fault statistics reflect the committed results for all test modes combined.

- An experiment on a test mode

  Uncommitted fault statistics reflect the cumulative results for that test mode up to and including the experiment. This includes all results that were committed at the time the experiment was created, and includes the *tested in another mode* status.

- The committed results for a test mode

  Committed test mode fault statistics reflect the cumulative results for that test mode, including only the committed results for all test modes. Note that the statistics for a test mode are affected by the results of some other test mode if some faults have *tested in another mode* status.

- A comet

  Comet fault statistics reflect the cumulative results for all of its member test modes combined, including only results that have been committed. The calculation and interpretation of comet fault statistics is the same for both cross-mode markoff comets and statistics-only comets.

The statistics are reported for each of several different types of faults: static, dynamic, pattern, PI, PO, etc. Each variable includes only faults that are active; that is:

- For global statistics, all non-ignored faults are counted
- For test mode statistics, only faults active in the test mode are counted
- For comet statistics, a fault is counted if it is active in any test mode in that comet

Following are some sample outputs of fault statistics:

- Default output of fault statistics is shown below:

```
Fault Statistics for Global   :
INFO (TFM-701):  Fault Statistics for Global:
                 -- ATCov --   ------------ Global Faults ------------------

                  Global   Total   Tested   Possibly   Redundant   Untested
Total Static       0.00     30       0         0           0          30
Collapsed Static   0.00     30       0         0           0          30
PI Static          0.00      4       0         0           0           4
PO Static          0.00      2       0         0           0           2

Total Dynamic      0.0      42       0         0           0          42
Collapsed Dynamic  0.00     42       0         0           0          42
PI Dynamic         0.00      4       0         0           0           4
PO Dynamic         0.00      2       0         0           0           2

Parametric

IDDq               0.00     30       0                               30
```

- Fault statistics output with testmode specified:

```
INFO (TFM-701):  Fault Statistics for Testmode:COMPRESSION
            --- ATCov -----   -------- Testmode Faults-------------- --------
Global Faults -----------
```

```
                Testmode  Global  Total  Tested  Possibly  Redundant  Untested
                Total     Tested Possibly Redundant Untested
TotalStatic      0.00      0.00    30      0        0          0          30
                   30        0      0      0       30
CollapsedStatic 0.00      0.00    30      0        0          0          30
                   30        0      0      0       30
PIStatic         0.00      0.00     4      0        0          0           4
                    4        0      0      0        4
POStatic         0.00      0.00     2      0        0          0           2
                    2        0      0      0        2
Dynamic          0.00      0.00    42      0        0          0          42
                   42        0      0      0       42
CollapsedDynamic0.00       0.00    42      0        0          0          42
                   42        0      0      0       42
PI Dynamic       0.00      0.00     4      0        0          0           4
                    4        0      0      0        4
PO Dynamic       0.00      0.00     2      0        0          0           2
                    2        0      0      0        2

Parametric

IDDq             0.00      0.00    30      0                              30
                  30         0                     30

Path              0.00      0.00     0       0                             0
                     0       0                       0

[end TFM_701]
```

- Fault statistics output with options `reportpiporows=yes`, `reporttype=static,dynamic` `reportptab=yes` and `coveragecredit=tested,possibly`:

```
INFO (TFM-701):  Fault Statistics for Global:
                  -- PCov --  ------------- Global Faults ----------------
                   Global    Total   Tested   Possibly   Redundant   Untested


Total Static         0.00      74       0         0          0           74
Collapsed Static     0.00      61       0         0          0           61
PI Static            0.00      16       0         0          0           16
PO Static            0.00       2       0         0          0            2
Total Dynamic        0.00      74       0         0          0           74
Collapsed Dynamic    0.00      64       0         0          0           64
PI Dynamic           0.00      16       0         0          0           16
PO Dynamic           0.00       2       0         0          0            2

[end TFM_701]

INFO (TFM-702): Possibly Testable at Best Fault Statistics and Reasons Report
for  Global:

          -PTBCov-  ----Global Faults-----
           Global    3-state   TIE X   CSO

Static      0.00       10        0       0

Dynamic     0.00       10        0       0

[end TFM_702]
```

Following is an example of the `report_fault_statistics` output for Global TCov statistics with ignored fault statistics:

```
Global Static Total Ignored (undetectable) Fault Count:      100
Ignored-Tied Fault Count:          40
Ignored-Unconnected Fault Count:   20
Ignored-Blocked Fault Count:       40

Global Dynamic Total Ignored (undetectable) Fault Count:     200
Ignored-Tied Fault Count:          80
Ignored-Unconnected Fault Count:   40
Ignored-Blocked Logic Fault Count:80

INFO (TFM-701):   Fault Statistics for Global   [end TFM_701]
              --- TCov ---  ------------- Global Faults ----------------------

                 Global    Total   Tested   Possibly   Redundant   Untested

Total Static      80.00     1000      800         25          25         50

Collapsed Static  77.50      800      620         20          20         40

Total Dynamic     80.00     2000     1600         50          50        100

Collapsed Dynamic 77.50     1600     1240         40          40         80
```

Following is an example of the `report_fault_statistics` output for Global ATCov statistics with ignored fault statistics:

```
Global Static Total Ignored (undetectable) Fault Count:       100
Ignored-Tied Fault Count:           40
Ignored-Unconnected Fault Count:    20
Ignored-Blocked Fault Count:        40

Global Dynamic Total Ignored (undetectable) Fault Count:      200
Ignored-Tied Fault Count:           80
Ignored-Unconnected Fault Count:    40
Ignored-Blocked Logic Fault Count:  80

INFO (TFM-701):   Fault Statistics for Global   [end TFM_701]
              --- ATCov ---  ------------- Global Faults --------------------
                 Global    Total   Tested   Possibly   Redundant   Untested

Total Static      91.43     1000      800         25          25         50

Collapsed Static  91.18      800       620         20          20         40

Total Dynamic     91.43     2000       1600        50          50        100

Collapsed Dynamic 91.18     1600       1240        40          40         80
```

Options for Hierarchical Fault Statistics

- `hierstart`

  Specifies the index value or the name of the hierblock. This keyword is used to request the fault statistics for the specific heirarchical block. The default value is a non-hierarchical format.

- `hierend`

Specifies the number of hierarchical level upto which fault statistics are to be displayed. This keyword can be set to values <depth>, techcell, or primitive. The default value is 1.

Following is a sample output of the heirachical fault statistics for a given testmode.

```
Hierarchical Fault Stats selected:Total Static
       -APCov-  ------------------------- Faults -------------------------
---------------
Depth  Global  Total  Untested  Tested  Possibly  Redundant  HierName
1       0.00     30       30        0        0         0          topcell /
topcell
2       0.00      8        8        0        0         0          t2/test2
2       0.00      4        4        0        0         0          t11 / test2
2       0.00      8        8        0        0         0          t2 / test1
2       0.00      4        4        0        0         0          t22 / test2
[end TFM_701]
```

The maximum Global Test Coverage statistics is a part of the Global Fault Statistics. In this the test modes are listed after the statistics and the details of the statistics is available in the help for the message TFM-074 and not printed in the output log. A sample output of the Maximum Global Test Coverage Statistics is shown below:

```
INFO (TFM-704): Maximum Global Test Coverage Statistics:
                   %Active     #Faults     #Active     #Inactive
Total Static       100.00        30          30          0
Collapsed Static   100.00        30          30          0
PI Static          100.00         4           4          0
PO Static          100.00         2           2          0


Total Dynamic      100.00        42          42          0
Collapsed Dynamic  100.00        42          42          0
PI Dynamic         100.00         4           4          0
PO Dynamic         100.00         2           2          0

Parametric

There are 2 test mode(s) defined:
FULLSCAN
COMPRESSION

[end TFM_704]

There are no PPIs for Test Mode:  COMPRESSION

Information for Test Mode:  COMPRESSION

-------------------------

        Scan Type = GSD
```

Report Domain Faults

To report a list of faults within clock domain, use the report_domain_faults function. This function will generate a report that contains the faults that meet the specified criteria. The

faults included are in the domain(s) identified by the clock constraints or test sequences. To list faults in an individual domain, select one sequence or use a clockconstraints file with the clock(s) for the single domain. To see the faults in each domain, run the command multiple times selecting an individual domain each time. The syntax for this function is given below:

```
report_domain_faults experiment=tg1 testmode=FULLSCAN_TIMED workdir=mywd
faultlocation=net clockconstraints=clk_const
```

Following is a sample output of this function for a domain in a clock constraint file, `clk_const`:

```
INFO (TFM-705): Testmode: FULLSCAN_TIMED   Fault List:
Fault List for domains from Clock Constraints File clk_const:
  Fault Status    Type    Sim Func/Cell Name   Fault pin name
    380 T         OSF            PI           Pin.f.l.feq.nl.rx_clk
    379 T         OSR            PI           Pin.f.l.feq.nl.rx_clk
    428 T         OSF            PI           Pin.f.l.feq.nl.wb_clk
    427 T         OSR            PI           Pin.f.l.feq.nl.wb_clk
    738 T         ISF           MUX           Pin.f.l.feq.nl.apb_1.comp_done_reg
.I1.I0.DATA1
    737 T         ISR           MUX           Pin.f.l.feq.nl.apb_1.comp_done_reg
.I1.I0.DATA1
    736 Unio      ISF           MUX           Pin.f.l.feq.nl.apb_1.comp_done_reg
.I1.I0.DATA0
    1124629c      OSF           MUX           Pin.f.l.feq.nl.apb_1.i_277092.I0.I
0.DOUT
    735 Unio      ISR           MUX           Pin.f.l.feq.nl.apb_1.comp_done_reg
.I1.I0.DATA0
```

For a complete description of the `report_domain_faults` syntax, refer to report_domain_faults in the *Encounter Test: Reference: Commands*.

Report Domain Fault Coverage Statistics

To report fault coverage statistics for a clock domain, use the `report_domain_fault_statistics` function. This command reports the domain coverage for the superset of faults in all clocking sequences in the clock constraint file or the specified test sequence(s). To get separate statistics for multiple domains, run the command multiple times. The syntax for this function is given below:

```
report_domain_fault_statistics experiment=printCLKdom testmode=FULLSCAN /
workdir=mywd coveragecredit=tested testsequence= myclkseq
```

Note: To get a list of domains with the total number of faults in each domain, run `create_logic_tests` or `create_logic_delay_tests reportsequencefaults=yes`.

Inputs

- Encounter Test model from `build_model`
- Test mode from `build_testmode`
- Fault_model from `build_faultmodel`
- Experiment or committed results from ATPG

The following is a sample report for experimental test coverage based on clocking sequences.

```
              Experiment Statistics: FULLSCAN.printCLKdom
              #Faults   #Tested  #Possibly   #Redund  #Untested   #PTB  #TestPTB
  %TCov  %ATCov  %PCov  %APCov  %PTBCov  %APTBCov
Total         3152       1107     68          0     1977        0        0  35.12
35.12  37.28  37.28  35.12  35.12
Total Static              1584      690       47        0      847        0
0  43.56  43.56  46.53  46.53  43.56  43.56
Total Dynamic             1568      417       21        0     1130        0
0  26.59  26.59  27.93  27.93  26.59  26.59
Collapsed                 1730      612       45        0     1073        0
0  35.38  35.38  37.98  37.98  35.38  35.38
Collapsed Static           928      411       33        0      484        0
0  44.29  44.29  47.84  47.84  44.29  44.29
Collapsed Dynamic          802      201       12        0      589        0
0  25.06  25.06  26.56  26.56  25.06  25.06
```

Report Path Faults

To report path faults using the graphical user interface, refer to Report Path Faults in the *Encounter Test: Reference: GUI*.

For a complete description of the `report_pathfaults` syntax, refer to report_pathfaults in the *Encounter Test: Reference: Commands*.

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

```
report_pathfaults experiment=<experiment_name> testmode=<testmode_name> /
workdir=<directory> globalscope=no
```

Sample output for the short and long format (specified using the `reportdetails =yes|no` keyword) of the `report_pathfaults` command output is shown below:

Short form

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Listing of individual path faults with status and groups
```

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

There are 16 path faults and 8 path groups in this fault model
faultID: 1  transition: 0->1  size: 4  status: TG Fail Near Robust  group: Path_1_01
faultID: 2  transition: 0->1  size: 4  status: TG Fail Near Robust  group: Path_1_01
faultID: 3  transition: 1->0  size: 4  status: TG Fail Near Robust  group: Path_1_10
faultID: 4  transition: 1->0  size: 4  status: TG Fail Near Robust  group: Path_1_10
faultID: 5  transition: 0->1  size: 4  status: TG Fail Near Robust  group: Path_2_01
...
...
Path group: Path_4_10 has 1 Path Faults
Path group: Path_4_01 has 1 Path Faults
Path group: Path_3_10 has 2 Path Faults
Path group: Path_3_01 has 2 Path Faults
...
...
```

Long Form

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Listing of individual path faults with status and groups
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -


There are 16 path faults and 8 path groups in this fault model

faultID: 1  transition: 0->1  size: 4  status: TG Fail Near Robust  group: Path_1_01
Pin Pin.f.l.TESTCASE1.nl.FF_2.__i0.dff_primitive.Q,
Pin Pin.f.l.TESTCASE1.nl.FF_2.__i4.01,
Pin Pin.f.l.TESTCASE1.nl.FF_3.mux1.DOUT,
Pin Pin.f.l.TESTCASE1.nl.FF_3.__i0.dff_primitive.Q

faultID: 2  transition: 0->1  size: 4  status: TG Fail Near Robust  group: Path_1_01
Pin Pin.f.l.TESTCASE1.nl.FF_2.__i0.dff_primitive.Q,
Pin Pin.f.l.TESTCASE1.nl.FF_2.__i4.01,
Pin Pin.f.l.TESTCASE1.nl.FF_3.mux1.DOUT,
Pin Pin.f.l.TESTCASE1.nl.FF_3.__i0.dff_primitive.Q
...
...
...
Path group: Path_4_10 has 1 Path Faults
Path group: Path_4_01 has 1 Path Faults
Path group: Path_3_10 has 2 Path Faults
Path group: Path_3_01 has 2 Path Faults
...
...
...
```

The fault status can be any of the following:

- Tested Hazard Free
- TG Fail Near Robust
- Not Processed

- Test Gen NearRobust

Report Path Fault Coverage Statistics

To report path statistics using the graphical interface, refer to Report Path Fault Statistics in the *Encounter Test: Reference: GUI*.

For a complete description of the `report_pathfault_statistics` syntax, refer to report_pathfault_statistics in the *Encounter Test: Reference: Commands*.

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

```
report_pathfault_statistics experiment=<experiment name> /
testmode=<testmode name> workdir=<directory> pathname=<value>
```

An example of path faults statistics follows:

```
        Path Fault Coverage Definitions

    #Faults     : Number of Path Faults
    #Tested     : Number of Path Faults marked tested.
    #HzFree     : Number of Path Faults marked tested and hazard free
    #Robust     : Number of Path Faults marked tested and  robust
    #NrRob      : Number of Path Faults marked tested and  nearly robust (almost
    robust)
    #NoRob      : Number of Path Faults marked tested and  Non Robust
    #NoTest     : Number of Path Faults for which Path Test Generation could not
    generate a test

    %TCov     (%Test Coverage)      : #Tested/#Faults
    %HFree    (%Hazard Free TCov)   : #HzFree/#Faults
    %Rob      (%Robust TCov)        : #Robust/#Faults
    %NrRob    (%Nearly Robust TCov) : #AlRob/#Faults
    %NoRob    (%Non Robust TCov)    : #NoRob/#Faults
    %NoTest   (%Untesable)          : #NoTest/#Faults

Test Mode Statistics for Path Faults


        #Faults   #Tes  #Hz   #Rob  #NrR  #NoR  #NoTes  %TCov  %HFre  %Rob    %NrRob  %NoRob  %NoTe
                  ted   Free  ust   ob    ob    t              e                              st

 Paths  1000      800   200   150   200   250   100     80.0   20.0   15.0    20.0    25.0    10.0
```

Report Package Test Objectives (Stuck Driver and Shorted Nets)

Report Package Test Objectives produces information for objectives defined in the objectiveModel and objectiveStatus files. Stuck driver and shorted net objectives are defined

for I/O wrap and interconnect test purposes. Both of these objectives are based upon target nets. Stuck driver objectives are defined to check the integrity of the path from a driver to each of its receivers. Shorted nets objectives are defined to check the integrity of the paths for a set of target nets.

To report package test objectives, use `report_sdtsnt_objectives`. Refer to report_sdtsnt_objectives in *Encounter Test: Reference: Commands* for more information.

The following information is produced for static and dynamic stuck driver objectives:

For stuck driver objectives:

- Objective - Each objective is assigned a unique identifier called the objective index.
- Type - Static or Dynamic
- Value - The objective value
- Status - Test status of objective
- Intrachip - A yes or no indicator. Intrachip indicates whether the driver and receiver are on the same chip.
- PkgPinNet - A yes or no indicator. PkgPinNet indicates whether the driver or receiver are associated with a net which has a net which has a primary input, output or bidi pin.
- Net - The net which connects the driver and receiver.
- Driver - The logic model block identified as the driver.
- Receiver - The logic model block identified as the receiver.

The shorted nets objective list includes the number of target nets and the corresponding number of objectives. Each net is listed in a table showing the logic value required on that net for a given objective.

Refer to Stuck Driver and Shorted Net Objectives in the *Encounter Test: Reference: Legacy Functions* for more information.

Report Package Test Objectives Coverage Statistics

To report package test objectives coverage, use `report_sdtsnt_objective_statistics`. Refer to report_sdtsnt_objective_statistics in *Encounter Test: Reference: Commands* for more information.

4

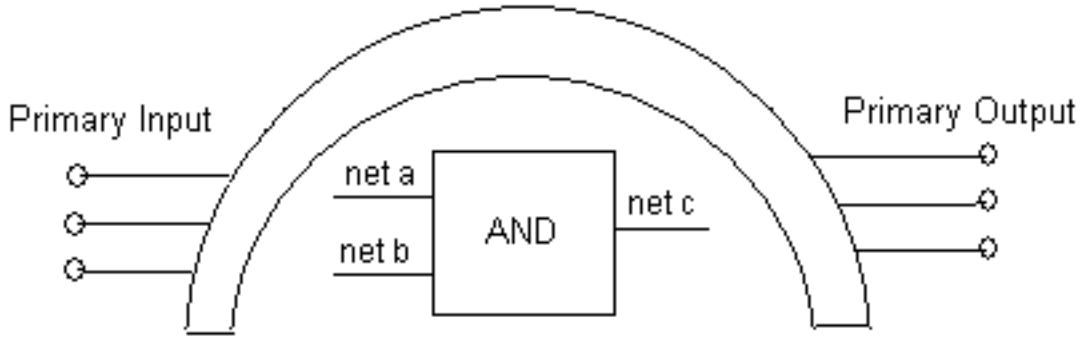
Analyzing Faults

Fault analysis is an early test generation process that determines if there are untestable static logic faults in the design. When the early test generation process ends, data about these untestable faults is provided to you for analysis.

Fault Analysis Process

A simplified view of the early test generation/fault analysis process is shown in the following figure:

Figure 4-1 Simple AND Block for Fault Analyzer Example



The early test generation process includes the following steps:

- Set up the test for a single fault. For the example, assume the test generator is trying to find a test for the first input of the AND (neta) stuck at one. In order to detect the difference between the good device and the device with the stuck-at-1 fault, the first step is to set `neta=0`.
- Next put a non-controlling value on netb to allow the effect of neta to be seen at the output. `Set netb=1`.

- Expect a value of zero on netc when the device is good and a value of one if the defect represented by the fault on net a occurs. This is represented as `0/1` in net c.
- The input values (on neta and netb) must be justified back to the PIs or to flops or latches whose value can be scanned in.
- The output values (on netc) must be propagated forward to the POs or to flops or latches whose value can be scanned out.

If the values cannot be set up, or cannot be observed, or if conflicting values are required to do the justification and propagation, then the fault is untestable.

Analyze Deterministic Faults

Use the `analyze_deterministic_faults` command to analyze untested faults and identify observe and control test points to test such faults. The command does not identify test points for any untested faults for the following since test points in this logic could invalidate the design or testmode and cause failures in verify test structures:

- Inactive logic
- Outputs of RAMs, ROMs, or non-scan flops
- Clock paths, scan paths, tied logic paths, Test Inhibit/Test Constraint paths, or the outputs of RAMs, ROMs and non-scan latches

The total number of test points that the command identifies is determined by the number of untested faults in the design and the ability to combine as many test points as possible and still retain the ability to test the faults. In many cases, a test point might provide testability for only a single fault. In these cases, a high test coverage requires a large number of test points.

For example, if there are over 100K untested faults in a design, more than 20K test points might be required to achieve a test coverage of 99.9% or higher. The `analyze_deterministic_faults` command identifies how many faults should be tested with each identified test point, thus allowing you to use only those test points that test large number of faults.

The following is a sample usage of the `analyze_deterministic_faults` command:

```
analyze_deterministic_faults workdir=workdir inexperiment=experiment_name
testmode=testmode
```

The previous command creates all possible test points for the experiment. Refer to analyze_deterministic_faults in *Encounter Test: Reference: Commands* for more information.

Input Files

- Encounter Test Model from `build_model`
- Testmode from `build_testmode`
- Fault model from `build_faultmodel`
- Experiment from previous test generation run. This provides the starting point for coverage for untested faults.
- `tgfaultlist` file (optional) containing a list of faults on which `analyze_deterministic_faults` is to work. This allows you to target specific pieces of the design.

Outputs

`TestPointInsertion.<testmode name>.dfa` in the `testresults` directory. If a `testresults` directory does not exist then the command places the output file in the `workdir`. The output file lists the test points in weighted format, starting with the test points that test the highest number of faults. If an observe and control test point each test the same number of faults, then the observe test point will be listed first.

A sample of the file is given below:

```
entity XYZ is
attribute INTERNAL_TEST_POINT_LIST of XYZ: constant is
"\NB/NBBST/UU_BistPauseCnt_bp0_1_X9 [pin:Z] [type:O] [faults:9]," &
"\NB/NBBST/UU_BistPauseCnt_bp0_1_X11 [pin:Z] [type:O] [faults:8]," &
"\NB/XBPML/UU_NBXBR_PerfMonXbar_2_1 [pin:Y] [type:C0] [faults:7]," &
"\NB/XBPML/UU_NBXBR_PerfMonXbar_3_1 [pin:Z] [type:O] [faults:6]," &
"\NB/NBBST/UU_BistPauseCnt_bp0_1_X8 [pin:Z] [type:O] [faults:6]," &
"\NB/NBBST/UU_BistPauseCnt_bp0_1_X10 [pin:Y] [type:C1] [faults:6]," &
"\NB/XBL2S/Ldt2ToXbarBuf/UU_invCrcError_bit1 [pin:Z] [type:O] [faults:6]," &
"\NB/XBL1D/XbarToLdt1Buf/UU_tPktCnt_n2_bit2 [pin:Z] [type:O] [faults:1]," &
"\NB/NBBST/UU_BistCamState_bp0_int3_1 [pin:Z] [type:O] [faults:1]," &
"\NB/XBL1S/Ldt1ToXbarBuf/UU_LdtToXbarr_xp0/UU_Z0 [pin:Z] [type:O] [faults:1]";
end XYZ;
```

In the output:

- The first entry, for example `\NB/NBBST/UU_BistPauseCnt_bp0_1_X9` in the given output, is the block name
- The second entry, for example `[pin:Z]` in the given output, is the pin name
- The third entry, for example `[type:O]` in the given output, is the type of test point. The following types of test points are available:
  - `[type:O]` - observe test point

- [type:C0] - control to 0 test point
- [type:C1] - control to 1 test point
- [type:CB] - control both test point

- The last entry, for example [faults:9] in the given output, depicts the number of faults tested

You can edit this file to remove any test points. However, you must ensure that the last line ends with "; .

If you specify traceinactive=yes, the command creates an additional file named InactiveFaults.*<testmode name>* in the testresults or workdir directory. A sample is given below:

```
Inactive fault analysis summary
-------------------------------

pin BS1 creates 34 inactive fault(s)
2140 2133 2123 2072 2065 2055 1870 1863 1853 1802 1795 1785 1600 1593 1583 1532
1525 1515 1330 1323
1313 1262 1255 1245 1127 1114 1104 1041 1028 1018  128  118  108    9

pin DI1 creates 42 inactive fault(s)
2211 2210 2209 2208 2193 2192 2189 2186 1949 1948 1947 1946 1931 1930 1927 1924
1917 1916 1679 1678
1677 1676 1661 1660 1657 1654 1647 1646 1409 1408 1407 1406 1391 1390 1387 1384
1377 1376  967  966
30   29

pin RI creates 29 inactive fault(s)
2207 2206 2199 2099 1945 1944 1937 1920 1829 1675 1674 1667 1650 1559 1405 1404
1397 1380 1289  970
191  190  181  180  163  162  148  147   49

pin BS2 creates 12 inactive fault(s)
2109 2096 2086 1839 1826 1816 1569 1556 1546 1299 1286 1276
```

In the output:

- Each line starting with pin *<pinname>* is a cause of the specified number of inactive faults.
- The line(s) under each pin entry lists the fault indexes of the faults that are inactive because of that pin.

Sample Methodology for Large Parts

For large designs that typically contain thousands of untested faults, it is recommended that you use the following methodology to analyze untested faults:

1. Run `report_fault_statistics`, as shown below, to identify areas in the design that have low coverage:

   ```
   report_fault_statistics workdir=workdir testmode=testmode
   experiment=exp_name hierlevel=macro colsuntested=yes globalscope=no
   ```

2. Create a fault list from the identified areas of the design by running `report_faults`, as shown below:

   ```
   report_faults workdir=workdir testmode=testmode experiment=exp_name
   reporthierarchical=yes globalscope=no statuspossibly=no statusincomplete=yes
   statusaborted=yes statusuntestable=yes typedrvrcvr=no inputcommitted=no
   hierrange=range
   ```

   The `hierrange` keyword is the list of hierblock indexes for which you want the untested faults.

3. Feed this list of faults to the `analyze_deterministic_faults` command by specifying the `tgfaultlist` keyword, as shown below.

   ```
   analyze_deterministic_faults workdir=workdir testmode=testmode
   inexperiment=exp_name tgfaultlist=fault_list
   ```

A disadvantage of this methodology is that it does not target the complete design but only specific areas in the design, therefore, resulting in limited test coverage.

Analyze Faults

The `analyze_faults` command is basically the beginning of the `create_logic_tests` process; `analyze_faults` does not produce vectors, it only produces TFA messages to report testability problems. You also can create TFA messages by running `create_logic_tests`. The messages do not print to the `create_logic_tests` log but you can view them with GUI message analysis.

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

To perform Analyze Faults using command lines, refer to "analyze_faults" in the *Encounter Test: Reference: Commands*.

Restrictions

Encounter Test applies the following restrictions on deterministic fault analysis:

- Deterministic Fault Analysis processes only static (stuck-at or pattern) faults.
- Specific analysis capability is not provided for:

- Driver and receiver faults
- Testability of faults by IDDq tests.

- Fault Analysis is generally incapable of resolving all untestable faults in the presence of non-scannable memory elements. Faults that require multiple-time frames to be proven untestable are classified as incomplete.
- Limitations of the interactive analysis of Deterministic Fault Analysis messages:
  - Analysis of aborted faults is limited. During analysis of an aborted fault, only the associated fault block is displayed.
  - Only a single set of fault analysis data can exist at any point in time.

Input Files

- Encounter Test Model from `build_model`
- Testmode from `build_testmode`
- Fault model from `build_faultmodel`

Output

- Fault status information
- TFA messages in the log that indicate testability problems. These messages can be analyzed interactively with GUI message analysis.

Analyzing TFA Messages from Create Logic Tests or Analyze Faults

1. Select the *Messages* Tab and click the *View Messages* icon; or *Window - Messages - Analyze Faults.* The *Analyze Faults Message Summary* list is displayed.
2. Select a message number, then select *View*. The *Analyze FaultsSpecific Message List* is displayed. There are three messages generated. The reason is included as the last sentence in the text of the message.
   - TFA-001 - Untestable faults
   - TFA-020 - Redundant faults
   - TFA-030 - Aborted faults

See "Specific Message List Window" in the *Encounter Test: Reference: GUI* for more information.

Select a specific fault to analyze by selecting a specific message and then *Analyze*.

3. Analyze the message using Actions on the View Schematic Window. Refer to *Encounter Test: Reference: GUI* for details.

   When you analyze a TFA message, an informational message dialog pops up along with the display of the logic containing the fault and any additional logic that was found to contribute to the controlability or observability problem.

   The message dialog provides additional information about the cause of the testability issue and/or recommends a test point that would improve the testability of that fault.

Refer to "Message Analysis Windows" in the *Encounter Test: Reference: GUI* for additional information.

GUI Schematic Fault Analysis

You can use Analyze Faults on a fault in the Graphical User Interface Schematic to use fault analysis on specific faults. This allows you to graphically view testability problems in the design.

Note: Analyze Fault on the GUI provides an optimistic analysis of the fault; ATPG includes additional processes that may prove a fault untestable that the GUI reports as testable.

Analyze Fault uses a dual-machine nomenclature to represent the good machine and fault machine values associated with each entity. These values are displayed on the design and in the Information Window.

The net values displayed by Deterministic Fault Analysis are expressed in terms of good and fault machine logic value pairs. The first value is for the good machine and the second value is for the fault machine (design behavior in the presence of a given fault). For example, a 1/0 represents a design value of one for the good machine, and a design value of zero in the presence of the fault. The logic values of entities (blocks, pins, or nets) unaffected by a particular fault are normally expressed as V/V, where V can be any of the valid logic values.

Deterministic Testability Measurements for Sequential Test

Deterministic Testability Analysis (*Testability Measurements*) provides four design testability analysis functions.

- The first function provides potentially-attainable logic values (LV) at output pins of each block, by means of Possible Value Set (PVS) simulation. See "Possible Value Set (PVS) Concepts" on page 61 for details.
- The second function provides deterministic controllability/observability measures at output pins of each primitive block.
- The third function provides sequentiality analysis at input and output pins of each primitive block
- The fourth function provides latch tracing information.

These deterministic testability analysis (TTA) functions are performed for the good design and at the test mode level.

Deterministic Testability Analysis prints out results on output pins based on pin types being selected. If selected, one results table and one stats table are printed for each of the two analysis functions. The stats tables provide summary of the analysis. The results tables provide the analysis results for each output pins of selected types.

The analysis data can be included in the GUI Schematic Window by selecting corresponding Information Window Display options. Note that these options are not retained because of the long run time associated with these tasks. It is therefore recommended that the options be used only when analyzing small sections of sequential logic.

Notes:

- To perform Deterministic Testability Analysis, refer to "report_testability_measurements" in the *Encounter Test: Reference: Commands*
- To perform Latch Tracing Analysis using command lines, refer to "report_sequential_trace" in the *Encounter Test: Reference: Commands*

Performing Deterministic Testability Analysis

To perform Deterministic Testability Analysis, refer to "report_testability_measurements" in the *Encounter Test: Reference: Commands.*

Input Files

- Encounter Test Model from `build_model`
- Testmode from `build_testmode`

Fault model from `build_faultmodel`

Output

Testability information in the log

Note: A blank entry in the controlability column indicates that the net is two-state instead of three-state.

Possible Value Set (PVS) Concepts

While the Encounter Test Logic Values (LV) technique describes signal simulation responses in a logic network and for a given instance, PVS is a logic algebra identifying potentially-attainable logic values at a pin at and beyond a given time instance. These two techniques complement each other, and together, they form the complete (static) logic value system of Encounter Test.

Information about attainable logic signal values is the basis of Deterministic Testability Analysis. To be consistent with the logic scope defined by Encounter Test Logic Values, PVS uses the five fully-specified logic values (0, 1, Z, L, H) in its base set. These five fully-specified logic values are 0, 1, Z, L, H. PVS consists of the 32 unique subsets of these five fully-specified logic values.

For two-state, three-state, and weak signals, the initial PVS values are {0, 1 }, {0, 1, Z }, and {Z, L, H }, respectively, while for PFET and NFET, their initial PVS values include all five fully-specified logic values, {0, 1, Z, L, H }. After TTA calculations, the signal's PVS may result in smaller sets than their initial PVS sets.

For example, for an AND with an X-source at one of its inputs, its output PVS would be {0 }. Similarly, the PVS at the output of a three-state driver is {Z } if its enable signal's PVS is {0 }.

Deterministic Controllability/Observability (CO) Measure Concepts

CO analysis is a measure combining the number of primary inputs (PI's) and the (compounded) logic depth needed in setting a signal value. For controlling logic values of primitives, the controllability of the controlling values at the outputs are the minimum controllability of the controlling values at the inputs plus 1 (for the added logic depth going from an input to the output of a primitive logic function). For non-controlling logic values, the controllability of the non-controlling values at the outputs are the sum of the controllabilities of the non-controlling values at the inputs plus 1.

For example, with a 2-input AND gate, its 0-controllability measure is 1 plus the smaller 0-controllability of its two inputs; its 1-controllability measure is 1 plus the sum of the 1-controllabilities of its two inputs. Initially, 0-controllability and 1-controllability for all PI's are set to 1 and for all internal signals set to the largest integer supported by the software. During TTA calculations, all internal signals converge to smaller numbers. If a signal's controllability measure remains at the maximum integer, it indicates that it is impossible to control this signal to the respective logic value, which may cause difficulties in test generation. A surprisingly large controllability measure, though less than the maximum integer may also indicate potential testability problems.

Similar to controllability measures, the observability measure combines the number of PI's and the (compounded) logic depth needed to observe a signal through primary outputs (PO's). Initially, observability measures would assume an integer 1 for all PO's and the maximum integer for all internal signals. During TTA's observability calculations, all observability internal signals would converge to integer numbers between 1 and the maximum integer. A maximum integer or a surprisingly large observability measure would indicate it impossible or very difficult to observe the signal at PO's.

CO analysis imposes a SCAN penalty on Scan controllable/measurable signals, to distinguish these Scan controllable/measurable signals from PI's and PO's for cost considerations. The default Scan penalty is set to 2. A feedback penalty of 10000 is also imposed on signals in feedback loops. Non-scan latches and memories would also be penalized by 10000 to distinguish them from combinational and Scan logics. CO calculation penalties allow user overrides.

Sequential Depth Measure Concepts

Similar to controllability/observability measures, sequential depth analysis measures the sequential elements to go through in order to control/observe a node. Currently, this analysis computes six sequential depth measures for each node. These six measures are: minimum and maximum sequential depth from Topological Control Points (TCP), minimum and maximum sequential depth from Topological Measure Points (TMP), minimum sequential depth for setting a logic ZERO and ONE to a node.

TCP analysis starts by setting sequential depth of 0 to primary inputs, tied nodes, lineholds, TI's, fixed value latches and test constraint latches. For the rest of the nodes, the minimum and maximum TCP measures are calculated as the minimum and maximum TCP measures among the inputs of a node, except for non-controllable latches and memory for which a 1 is added to the output nodes. Controllable latches are considered as directly controllable so their minimum and maximum TCP measures are the same as primary inputs.

Minimum and maximum TMP measures are calculated in a similar way as TCP measures, except it starts by setting sequential depth of 0 to primary outputs and that measurable latches are considered as if they were primary outputs.

The minimum sequential depth of setting a logic ZERO and ONE combines the above concepts of deterministic controllability and minimum TCP measures.

Latch Tracing Analysis Concepts

Contrary to the deterministic controllability/observability measures and sequential depth measures which would start processing from inputs to outputs or from outputs to inputs, latch tracing starts with a node and traces to both inputs and outputs. Latches are separated into the input and output cones of the focal node and are sorted according to the relative levels to the node.

Random Resistant Fault Analysis (RRFA)

Analyze Random Resistance identifies sections of a design that are resistant to testing by flat, uniformly distributed random patterns and provides analysis information to help improve the random pattern testability of the design. This analysis information is based on signal probabilities computed by simulating random patterns and counting the number of times each net takes on the values 0, 1, X and Z. Analyze Random Resistance then uses the faults that were not tested by the random pattern simulation to determine points in the design that tend to block fault activation and propagation to observable points.

The analysis information provided by Analyze Random Resistance falls into four key areas:

- Testpoint Insertion
- Cluster and region identification
- Correlated net identification
- Hard to Control/Observe Nets

Note: The primary use of the information provided by Analyze Random Resistance is for testpoint insertion, which is dicusssed in this section. Other areas are discussed in Encounter Test: Reference: Legacy Function.

Analyze Random Resistance

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

To perform *Analyze Random Resistance* using command lines, refer to "analyze_random_resistance" in the *Encounter Test: Reference: Commands.*

Input Files

- Encounter Test Model from `build_model`
- Testmode from `build_testmode`
- Fault model from `build_faultmodel`

User Input (Optional)

- Linehold

  This user-created file is an optional input to test generation. It specifies design PIs, latches, and nets to be held to specific values for each test that is generated. See "Linehold File" in *Encounter Test: Guide 5: ATPG* for more information.

- Test Sequence

  This points to user specified clock sequences that have been read in using `read_sequence_definitions`. The test sequence is required if you are processing an LBIST testmode.

  See "Coding Test Sequences" in *Encounter Test: Guide 5: ATPG* for an explanation of how to manually create test (clock) sequences.

- Probability input file

  A file where controllabilities of inputs and observabilities of outputs are specified for Analyze Random Resistance to use as starting values. The file is specified using the `analyze_random_resistance` keyword `probabilityfile`.

Output

- Fault status information

  Analyze Random Resistance experiments cannot be committed because no vectors are produced.

- TRA (Analyze Random Resistance) messages which can be analyzed with GUI Message Analysis.

- Signal Probability information

- TestPointInsertion.testmode.experiment – a file in testresults (or in the workdir if testresults does not exist). It contains the list of recommended test points in a format that can be used for inserting into the Encounter Test Model or in DFT Synthesis. For additional information, refer to Testpoint Insertion.

Test Points

This section discusses the following topics:

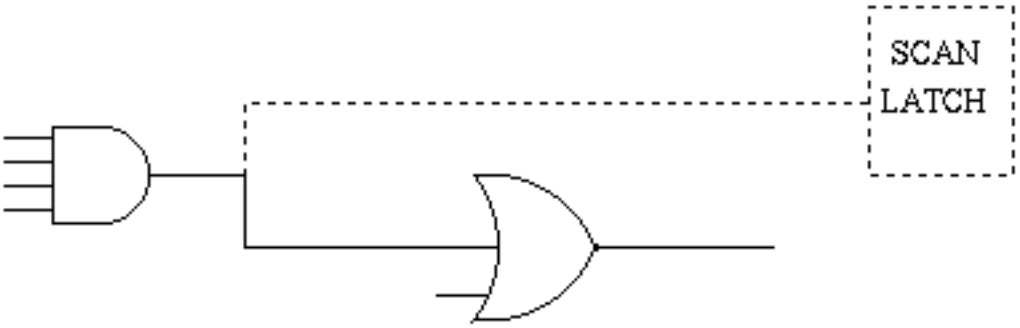
- Recommendations
- Insertion

Recommendations

In some cases, the random testability of a design may be increased by connecting a net to a scannable latch or primary output, thus improving the observability of the net. Analyze Random Resistance recommends places in the design where these Observe Points should be added.

Figure 4-2 shows how an observe point is added in the design.

Figure 4-2 An Observe Test Point

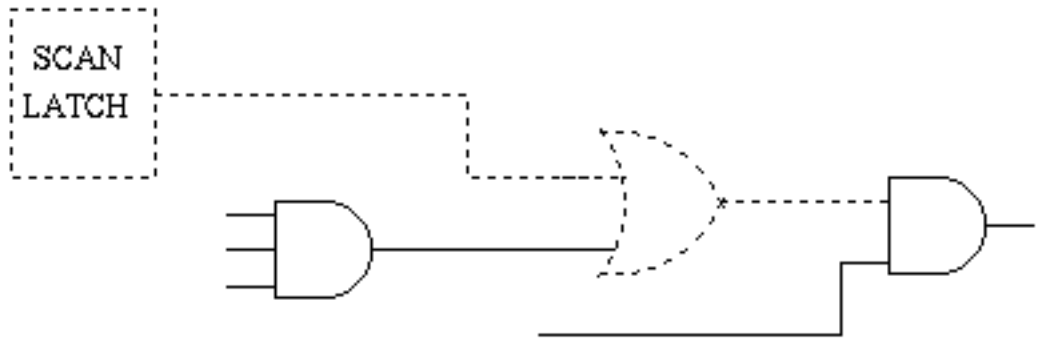


In other cases, the random testability may be increased by adding an OR gate or an AND gate to a particular net with the other input connected to a scannable latch or primary input. Analyze Random Resistance recommends places in the design where these Control-1 and Control-0 test points should be added.

Figure 4-3 shows how a control-1 test point is added in the design.

For control-0, the dashed OR in the figure would be AND.

Figure 4-3 Control-1 Test Point



In addition to graphical message analysis showing the recommended location for test points, Analyze Random Resistance creates a TestPointInsertion file containing test point recommendations written in TSDL format. This file may be modified if desired and used as input to RC-DFT Synthesis to insert test points automatically into the design.

This file may also be specified on the *GUI Schematic Edit* pull-down or with the `edit_model` command to add the test points to the Encounter Test model strictly for experimentation purposes. In this case, the test points are modeled using primary inputs and primary outputs rather than scannable latches.

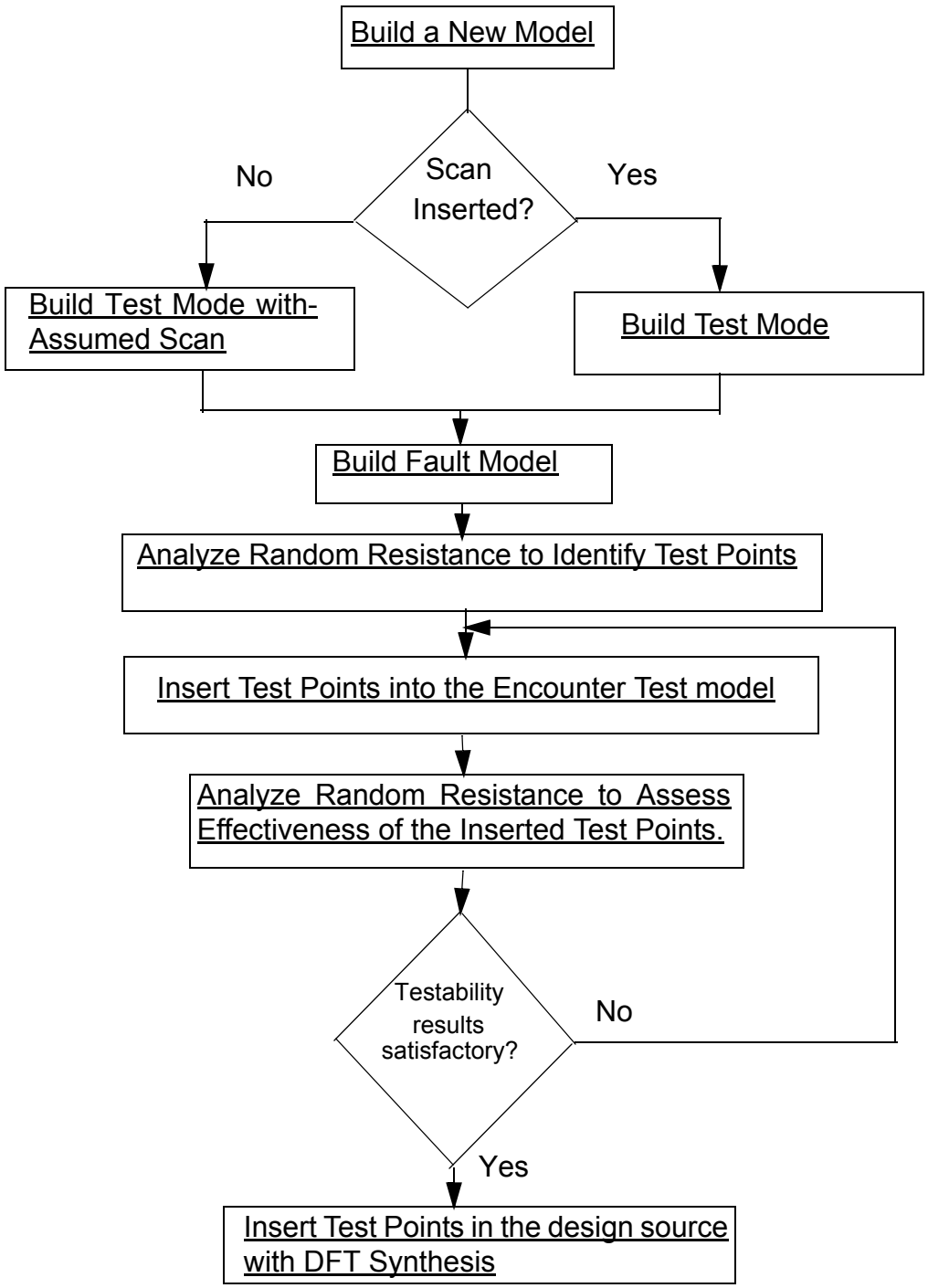
Refer to the following for additional information:

- "Testpoint Insertion" on page 66
- "Edit Test Points" in the *Encounter Test: Reference: GUI*

Testpoint Insertion

The following figure shows a typical processing flow for Test Point Insertion.

Figure 4-4 Test Point Insertion Flow



1. Build a New Model

   Run your normal `build_model` command to build an Encounter Test model. There are no unique requirements for building the model for this flow.

   For complete information on *Build Model,* refer to "Performing Build Model" in the *Encounter Test: Guide 1: Models*.

2. Build Test Mode

   Run `build_testmode` to build the testmode. The results of Random Resistant Fault Analysis will be more accurate if you use a testmode that configures the scanchains as they will be when the design is tested.

   However, if the test structures have not been inserted in the design (i.e. the latches are not scannable), you can use the Assumed Scan feature to simulate the latches as scannable. This feature can be used to identify specific latches to assume as scannable (for a partial scan implementation) or will assume all latches to be scannable.

   For complete information, refer to:

   - "Performing Build Test Mode" in the *Encounter Test: Guide 2: Testmodes.*
   - "Assumed Scan" in the *Encounter Test: Guide 2: Testmodes*.
   - "Build Test Mode" in the *Encounter Test: Reference: GUI.*

3. Build Fault Model

   Run `build_faultmodel` to build a fault model. As Random Resistant Fault Analysis analyzes only static faults, the fault model does not need to include any other faults. However, you can have other types; RRFA will ignore all except the static faults.

   Refer to Build Fault Model for more information.

4. Analyze Random Resistance to Identify Test Points

   Run `analyze_random_resistance tpi=yes` (default) to identify the test points to improve testability.

   Ensure you enter the experiment name for the output.

   Refer to "Analyze Random Resistance" on page 63 for additional details.

5. Insert Test Points into the Encounter Test model

   Command line: use the following command to insert the testpoints into the Encounter Test model

   ```
   edit_model tsdlfile=testresults/TestPointInsertion.testmode.experiment
   ```

You can edit the test point insertion file prior to using it as input to edit model.

GUI: Use the following scenario to insert test points into the Encounter Test model:

a. Bring up the Encounter Test GUI and set the Analysis Context to select the testmode and experiment used for `analyze_random_resistance`. The easiest way to do this is to select the `analyze_random_resistance` task from the Task View.

b. Select *View Schematic* to bring up the schematic display. There is no need to display any logic on the schematic, but you may display anything you like.

c. Select *Edit* then select *Test Points.*

d. In the *Edit Test Points* window, select the *File* button next to *File Name* and navigate to the TestPointInsertion file containing the testpoints from RRFA (that is, TPI file). Select that file and click *OK*.

e. Specify the number of desired test points from the TPI file (you may include All or a specified number of top test points). Note there is also an additional selection on this window to add a specific type of testmode on a specific pin; you can ignore that for this scenario. When you have the desired list, click *OK* at the bottom of the window.

f. Click the *Apply* button next to the *File* button and you will see the top of the window is populated with the details of the test point information. You can edit the list by selecting and using the right mouse button.

g. Click *OK* at the bottom of the window. The edits will be included in the *Edits Pending* window and the hierarchical model in memory will be updated to reflect the changes.

Note: The actual design does not change at this point. Step **i** actually modifies the design.

h. You may trace through the design to view what has been done. If you want to add a test point at a specific pin you may select that pin on the design view and use the right mouse button to edit test points. A small test point edit window will allow you to select the type(s) of test points you want to add at that point. These will also be added to the *Edits Pending* window.

i. When you have all the desired edits , select *Edits Complete, Reinitialize* on the *Edits Pending* window. The hierarchical model will be written to disk and the flat model, test mode, and fault model will be recreated.

6. Analyze Random Resistance to Assess Effectiveness of the Inserted Test Points.

Run `analyze_random_resistance` again and see the test coverage information. If you inserted test points with the GUI, the easiest way to do this is to go back to the

analyze_random_resistance task on the *Task View* to bring up the Form that is set up the way you ran it the last time. Use a different experiment name if you do not want to change the TestPointInsertion file.

If the results are not satisfactory, you can insert additional test points.

7. Insert Test Points in the design source with DFT Synthesis

Bring up RC-DFT Synthesis and use the command:

```
insert_dft rrfa_test_points -input_tp_file ./testresults/
TRAtestPointInsertionData.testmode.experiment
```

To select less than all test points in the file, use `-max_number_of_testpoints integer`.

Refer to Using Encounter Test to Automatically Select and Insert Test Points in *Design For Test in Encounter RTL Compiler Guide* for more information.

5

Deleting Fault Model Data

This chapter covers the Encounter Test tasks for deleting fault model, fault status and test objectives data.

Delete Fault Model

This action removes all the faultModel and faultStatus files and all dependencies on these files for the specified fault model. This enables you to build multiple test modes in parallel without contention.

Note: Deleting the fault model does not affect test mode data and does not require any rebuilding of test modes.

To delete a fault model using the graphical interface, refer to "Delete Fault Model" in the *Encounter Test: Reference: GUI*.

To delete a fault model using command line, refer to "delete_faultmodel" in the *Encounter Test: Reference: Commands*.

An example of the `delete_faultmodel` command is given below:

```
delete_faultmodel workdir=/local/dlx sdtsnt=<yes|no>
```

If the command does not find the faultModel file to delete, it continues to remove files that are dependent on these files and registration records under the assumption that the file has been manually deleted. This provides a way of removing registration records and dependency records from globalData even if the files do not exist.

Delete Fault Status for All Existing Test Modes

Use command `build_faultmodel overwrite=no` to rebuild the fault status information for the design without rebuilding the fault model. This effectively resets the status for all faults in all testmodes and invalidates any existing fault oriented test data.

Delete Committed Fault Status for a Test Mode

Use command `delete_committed_tests` to delete all the committed data for a testmode and reset the fault status back to its initial state from `build_faultmodel`.

Delete Alternate Fault Model

Use the `delete_alternate_faultmodel` command to delete a faultmodel built using `build_alternate_faultmodel`. The alternate fault model to be deleted is identified with the ALTFAULT keyword.

Delete Package Test Objectives

Use the `delete_sdtsnt_objectives` command to delete the objective model and status built using `build_sdtsnt_objectives`.

The objective model and status can also be deleted using delete_faultmodel sdtsnt=yes. This will delete both the fault model and the objective model.

Note: The alternate fault model named ##TB_SDT that is built along with the objective model is not deleted through either of the above mentioned commands. Use `delete_alternate_faultmodel altfault=##TB_SDT` to delete this model.

Delete Fault Model Analysis Data

Use the `delete_fautmodel_analysis_data` command to delete the register array and random resistant analysis data built using `prepare_faultmodel_analysis` or `build_faultmodel registerarray=yes` and `randomresistant=yes`. Refer to Building Register Array and Random Resistant Fault List Files for Pattern Compaction for more information.

6

Concepts

To generate tests for a design the first thing that must be done is identify the behaviors that require testing and the targets of those tests. In the digital logic test industry, the most common targets for testing are called stuck-at and transition faults. In Encounter Test we refer to static and dynamic faults. Static faults, such as stuck-at faults, affect the behavior of the design without regard to timing. Dynamic faults, also called transition or delay faults, affect the time-dependent behavior of the design. Since these static and dynamic faults are the logic model equivalent of physical defects, you will sometimes see these referenced as logic faults. Other faults or objectives are identified for special testing purposes such as ensuring that a driver in the physical design is working prior to applying tests to the entire design.

Encounter Test supports several different fault types and other test objectives. These faults are analyzed and some of them are classified with fault attributes that allow test generation and fault simulation to know that they cannot be tested so they will not be selected for processing. Once the faults are included in a fault model, the ones that are selected for processing may be used in test generation or fault simulation processes which mark them with a fault status; or they may be used in a diagnostics process to identify a failing location on the physical device.

Fault Types

The following sections describe the types of faults created by Encounter Test.

Static (Stuck-at)

Static Pin Faults

A static pin fault is stuck-at logic zero or logic one. This can be on any pin in the model. By default Encounter Test generates the following static pin faults:

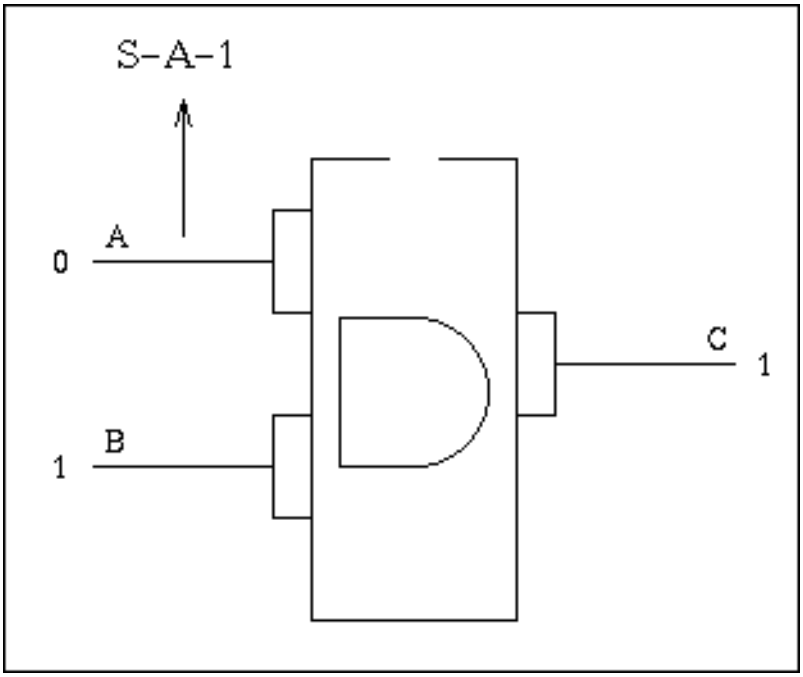
- Inputs of standard primitives stuck-at non-controlling
- Inputs of other primitives stuck-at both zero and one

- Outputs of primitives stuck-at both zero and one

A stuck-at fault assumes a logic gate input or output is fixed to either a logic 0 or 1.

Figure 6-1 shows an AND gate with input A Stuck-At-1 (S-A-1).

Figure 6-1 S-A-1 AND gate



The faulty AND gate perceives the “A” input as a logic 1 value regardless of the logic value placed on the input. The pattern applied to the fault-free AND gate has an output value of 0. The pattern applied to the faulty AND gate has an output value of 1.

The pattern shown in Figure 6-1 is a valid test of the input “A” S-A-1 because there is a difference between the faulty gate (fault machine) and the good gate (good machine). When this condition occurs, this set of input patterns detects the fault. If they had responded with the same output value, the pattern applied would not have constituted a test for that fault (S-A-1).

A good machine has no faults and a fault machine has one, and only one, of the stuck faults (S-A-1 or S-A-0). In other words, multiple faults are not considered in Encounter Test.

Static Pattern Faults

A pattern fault is defined by a pattern specified on a set of inputs and the resulting good machine and fault machine values to expect on the outputs in order to detect the fault.

Encounter Test automatically generates pattern faults for the following primitive logic functions:

- XOR (exclusive OR - ODD)
- XORN (exclusive OR - EVEN)
- TSD (Three-State Driver)
- LATCH (Latch Primitive)
- MUX (MUXes)
- __DFF (Flip-Flops)

Pattern faults ensure that virtually all possible defects associated with these logic functions can be effectively modeled. For example, a two-input XOR gate generates four static pattern faults, which represent all four possible input value combinations (00, 01, 10, 11), guaranteeing that all potential static defects will be covered.

Externally defined pattern faults may be specified in addition to those automatically generated by Encounter Test. This is done by creating fault rules and placing them in a file which is referenced by the Cell Library or netlist. If the filename is the same as the name of the cell to which it applies, it does not have to be referenced in the design source. An option during *Build Fault Model* will automatically look for fault rules that are named the same as the cells. See Build Fault Model Examples with Fault Rule Files for more information.

The purpose for pattern faults on MUXes is to guarantee that adequate tests have been applied to test the defects that can occur on these designs. S-A-1 or S-A-0 faults on the I/O pins of a mux would not force these robust test patterns. MUXes are generally constructed from transistor-level designs that can exhibit Z or Contention states when faulty. Static pattern faults for a MUX require each input to be selected and observed at both a 1 and 0 while all other data inputs are at the opposite state.

__MUX2 primitive static pattern faults are also implemented in Encounter Test.

Additional static pattern faults may be provided by the technology supplier or designer in a fault rule.

Shorted Nets Pattern Faults

A shorthand method to represent two nets that are shorted together. The shorted nets pattern faults are a form of static pattern faults.

Additional shorted nets pattern faults may be provided by the technology supplier or designer in a fault rule.

Dynamic (Transition)

Dynamic Pin Faults

Pin transitions from zero to one (slow-to-rise) or from one to zero (slow-to-fall). This can be on any pin in the model. By default Encounter Test generates the following dynamic pin faults:

- Inputs of primitives slow-to-rise and slow-to-fall
- Outputs of primitives slow-to-rise and slow-to-fall

More information about dynamic faults is included in "Delay Defects" in *Encounter Test: Guide 5: ATPG*.

Dynamic Pattern Faults

A sequence of patterns to set up the inputs and the resulting good machine/fault machine values to expect on the outputs in order to detect the fault. By default Encounter Test generates dynamic pattern faults for the following primitives: XOR, LATCH, RAM, ROM, TSD. The technology supplier or designer of a fault rule may provide additional dynamic pattern faults.

More information about dynamic faults is included in "Delay Defects" in *Encounter Test: Guide 5: ATPG.*

Parametric (Driver/Receiver)

A set of pattern faults to force all drivers to drive zero and one (and Z for three-state drivers) and to force all receivers to receive zero and one. These are not really faults in the sense of modeling a physical defect. Instead they are used to facilitate the testing of the drivers and receivers under worst-case conditions. The technology supplier or designer cannot provide pattern faults for driver/receiver objectives.

IDDq

IDDq faults represent defects that cause a CMOS device to use excessive current (IDD) compared with the normal current into the Vdd bus. In Encounter Test, the static faults are used for IDDq Tests. When an IDDq fault report is requested, the faults are identified as IDDq with a status of T (tested) or U (untested).

Fault Attributes

These classes of attributes are determined during the process of building the fault model or initializing the fault status for a testmode. After the fault is placed in one of these categories, it is not changed.

Ignored Faults (I, Iu, Ib, It)

When a fault is marked as Ignored, it means that there is no way to test the fault; it has no effect on the design operation. These faults are omitted from the fault model entirely unless `includeignore=yes` is specified during "build_faultmodel".

The following categories of faults are omitted from the fault model by default:

- Dangling/unconnected/unused logic (Iu). Faults on logic that does not feed any primary outputs nor observable latches and cannot be observed.

  Note: If all paths from the logic to observation points pass through blocks whose simulation function is unknown to Encounter Test (Blackboxes), the logic is treated as if it is dangling.

- Blocked logic (Ib). These are fault that cannot be observed because all paths from the logic to observation points are blocked by values propagated from TIE blocks.

- Tied Logic (It). These are faults that cannot be activated (excited) because the logic values required from control points are blocked by TIE values.

  Note: Outputs of blackboxes are treated as TIE blocks. They are tied to X unless blackboxoutputs was specified with a different tie value on `build_model`.

The following categories of faults are included in the fault model but are marked with an Ignored attribute:

- Faults from previous categories when `includeignore=yes` is specified.

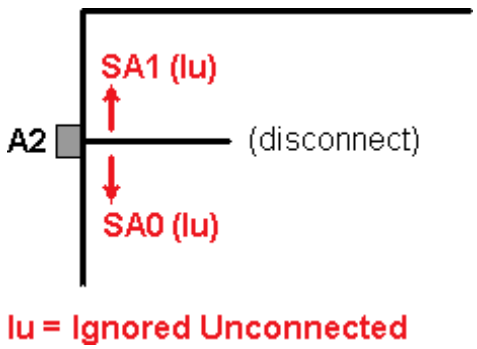
- Faults that are identified as Ignored in a fault rule.

See "Preparing an Ignore Faults File" on page 25 for information on automatically creating a fault rule file with Ignores for a level of hierarchy. See "Pattern Faults and Fault Rules" on page 86 for information on manually creating a fault rule.

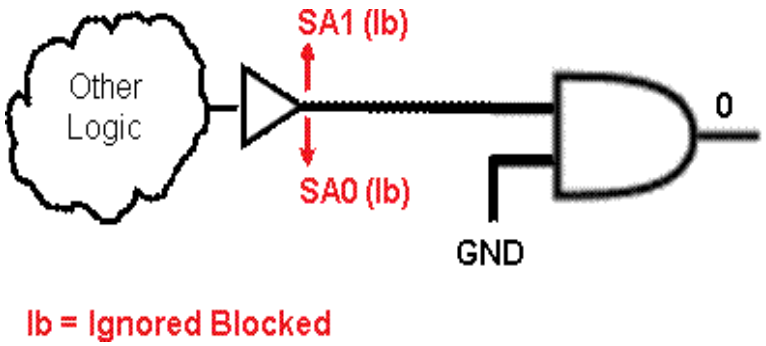
Ignored faults are not processed by test generation applications and are not included in the divisor when computing fault coverage unless `ignorefaultstatistics=yes` is specified on `build_faultmodel`.

See Build Fault Model Examples with Special Handling of Ignored Faults and Figure 6-3 on page 89 for more information.

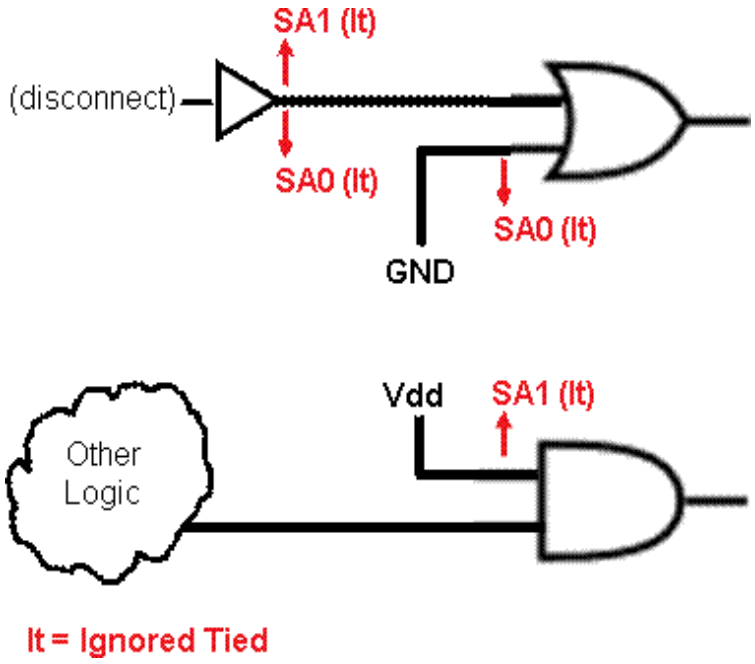
Example 1: Ignore Faults for Unconnected Logic



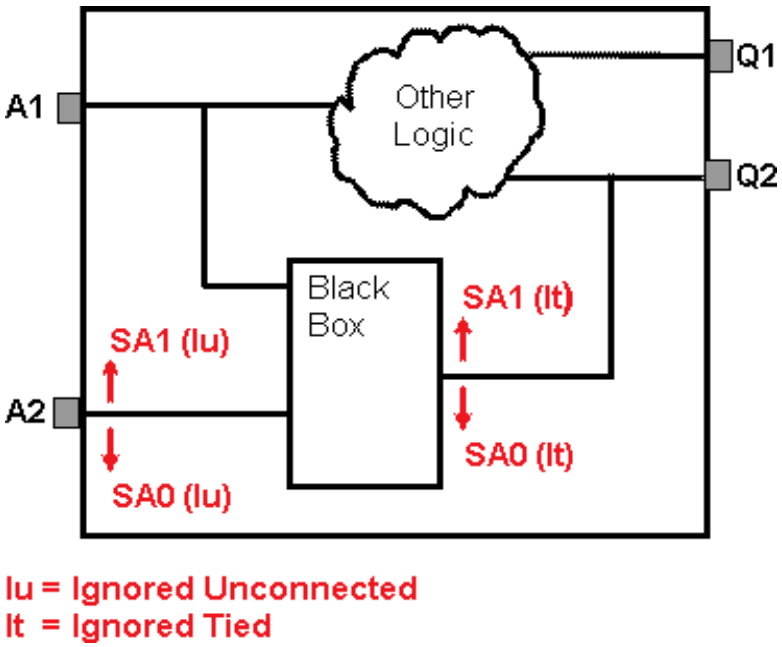
Example 2: Ignore faults that are blocked:



Example 3: Ignored faults for tied logic



Example 4: Ignore Faults feeding into/out of Blackboxes



Collapsed (C)

Faults that are logically equivalent are collapsed to a single representative fault. The representative fault is called an independent fault or an Equivalence Class Representative (ECR). The faults that are logically equivalent to an independent fault are called collapsed (or reduced) faults. Generating tests only for independent faults is a commonly used time-saving technique. A collapsed fault always has the same status as its representative fault. Faults are collapsed from left to right except faults on primary inputs (which are retained).

Pre-Collapsed

By default, the fault model includes only the following faults on single AND/NAND, OR/NOR, BUF/INV gates: stuck-at non-controlling on the inputs and both faults on the output. This optimizes test generation/fault simulation by focusing their efforts on a single fault even though it actually represents multiple potential defects in the hardware. Encounter Test terms these faults as pre-collapsed since they are collapsed by definition rather than through an analysis of the design. The faults on these blocks are:

- AND/NAND gates have s-a-1 faults on the inputs and both s-a-1 and s-a-0 on the output; s-a-0 faults on the inputs are pre-collapsed
- OR/NOR gates have s-a-0 faults on the inputs and both sa-1 and s-a-0 on the output; s-a-1 faults on the inputs are pre-collapsed
- BUF/INV gates have no faults on the input and both s-a-0 and s-a-0 on the output- s-a-0 and s-a-1 faults on the input are pre-collapsed

Grouping (&, I)

Fault grouping is used for modeling defects whose behavior is complex.

- OR Group (I) - If any one of the faults is tested, they are all tested.
- AND Group (&) - All faults in the group must be tested to expose the defect.

While Encounter Test supports fault grouping for better modeling of defects, it does not provide any test coverage statistics for the groups (defects). All coverage reporting is in terms of the individual faults. One effect of this is to give partial credit for an AND group where some, but not all of the faults in the group are tested.

When any fault in an OR group is detected, all faults within that group are marked as detected. This eliminates any further test pattern generation or fault simulation effort on that group, to help reduce the run time.

A fault group is identified by the & (AND) or I (OR) symbol next to each fault in the group. All faults within a group have consecutive indexes, and so will appear consecutively in a fault list.

Possibly testable at best faults (PTAB)

This is a characteristic of the fault that means there is no way Encounter Test can generate a test that is guaranteed to test the fault; but it may be able to be Possibly Tested (P). Possibly testable at best faults are not processed by test generation applications (by default). There are separate statistics maintained for these faults.

Possibly testable at best faults may be identified in a fault rule. Refer to Preparing an Ignore Faults File on page 25.

The possibly testable at best faults can be identified as:

- (3) -- PTAB faults that cause three-state contention on one or more nets or that cause High-Z on one or more three-state nets without Termination or Keepers.

- (X) -- PTAB faults that cause TIE-X signals to propagate to observation points (POs or scannable latches).
- (C) -- PTAB faults that cause all clocks (or the only clock) on one or more memory elements to be Stuck Off.

If a possible test is generated for a PTAB fault, the status is marked as P3, PX, or PC. If, during ATPG, you request the possibly tested faults marked as Tested after some number of possible tests have been generated, the status is marked as T3, TX, or TC.

Active and Inactive (i)

When the testmode fault status is initialized, every fault that is not ignored is either “active” in the testmode or “inactive”.

Active faults

An active fault is any fault that propagates to a PO or scannable flop while the circuit is in the scan state or test constraint state for the testmode. Active faults can be testable, untestable, redundant, or aborted. It is only the active faults that are given a Fault Test Status.

Inactive faults (i)

An inactive fault is one that cannot be detected in this test mode due to being blocked by a TI (Test Inhibit) primary input or fixed-value latch. See Identifying Inactive Logic in the *Encounter Test: Guide 2: Testmodes* for more explanation of inactive logic. Inactive faults are not processed by test generation or simulation run in this testmode. Unlike Ignored faults, Inactive faults may be active, and therefore, tested in a different testmode.

Fault Test Status

Logic Faults, IDDq Faults, Driver/Receiver Objectives, Stuck Driver and Shorted Nets Objectives may all be given a *Test Status* based on the success or failure of the test generation/fault simulation process to generate test patterns for them.

An active fault is classified in exactly one of these categories:

Tested (T)

Some test has been generated in this test mode that will fail in the presence of this fault. Thus, application of the test stimulus will cause a difference (0/1 or 1/0) at a tester-observable point (primary output or scannable reg) in the expected good response vs. the response in the presence of the fault. A fault can be marked tested in one of several ways:

Tested by Simulation (T)

The fault simulation determined that the fault is detected by the patterns (may be patterns from ATPG or manual patterns)

Tested by Implication (Ti)

The fault is expected to be detected when patterns are applied at the tester, but no fault simulation was performed to ensure it. One example is scan chain faults that are marked tested by apriori fault markoff. Another example are faults that cause single port, non-scannable memory elements to have their clocks "stuck off". If such memory elements are seen to write both zero and one during ATPG fault imply process, then the "clock stuck off" faults are detected by "implication".

Tested by Possibly Detected Limit (Tpdl)

The fault is really possibly tested but you have requested that it be marked tested after a specific number of patterns possibly tested it. If a fault is possibly tested many times, it is possible that there is a difference that could be observed between the good value and the value in the presence of the fault. Encounter Test fault simulation provides a keyword, pdl=n, to specify how many times a possibly detected fault is going to be simulated against additional patterns and another keyword, `markpdlfaultstested=yes`, to specify that if the pdl limit is reached the fault should be marked tested rather than possibly tested.

Tested in Another Mode (Tm)

The fault is active in more than one testmode that are participating in Cross-Mode Markoff (MARKOFF) and it was marked Tested in another mode. It is not known whether patterns created in this testmode also would have detected the fault since it is removed from processing as soon as the other testmode's results were committed.

Tested User Specified (Tus)

The fault has not been processed with ATPG or fault simulation in this design. A fault rule with the DETECTED statement was used as input to mark the fault tested during the building of the fault model.

Possibly Tested (P)

The fault may or may not be tested by the patterns. A test has been generated in this test mode where the good value is known (0 or 1) but the value in the presence of the fault is unknown by the simulator (X). When the patterns are applied at the tester the value will be seen as a 0 or 1; if it differs from the good value then the fault is tested.

Aborted (A)

ATPG processed a fault but was unable to generate a test or prove it untestable. This happens when some part of the ATPG process exceeds an internal limit or there is a problem that prevents the test from being completed. In some cases increasing the setting of the effort keyword in ATPG will allow the fault status to be resolved.

Redundant (R)

Test generation or Fault Analysis determined that the fault is untestable due to logical redundancy or tied logic that have nothing to do with the design state caused by any test mode. Faults marked redundant in the master fault status are not processed in any test mode.

Redundant: User Specified (Rus)

The fault was not processed by test generation or fault analysis in this design. A fault rule with the REDUNDANT statement was used as input to mark the fault redundant during the building of the fault model.

Untested – Not Processed (u)

This is the status of all active faults before they are processed by test generation and/or fault simulation. Faults can remain untested – not processed after ATPG if some run stopping criterion (set by keywords starting with “max”) was reached before these faults were considered for test generation or fault analysis and no patterns generated for other faults happened to detect these too.

Untestable

The test generator was unable to create a test for the fault in this test mode. The following untestable reasons are identified and can be reported. Note that if you run ATPG multiple times it is possible for the untestable reason to be different due to a variety of reasons including, but not limited to, different lineholds, ignoremeasures, test sequence, constraints, timings, or keyword settings.

Untestable: Undetermined (Ud)

No test can be generated for this fault, but no single reason to explain why could be determined.

Untestable: User Specified (Uus)

The fault was not processed in this experiment. A fault subset was created with the fault marked as untestable and that fault subset was used for the experiment that resulted in this status.

Untestable: Linehold inhibits fault control/observe (Ulh)

The test generator encountered logic value(s) originating from linehold(s) (LH flagged test function pin, or specified via a user file), which are inconsistent with logic value(s) required to test the fault.

The linehold file used during ATPG prevents testing of this fault.

Untestable: X-sourced or sinked (Uxs)

A test for this fault requires values on one or more nets that cannot be set to known values (the nets are at X). Non-terminated three-states and clock-stuck-off are specific types of X-sources that are categorized separately. This general category of other X-sources includes the following:

- TIEX
- sourceless logic
- feedback loop (not broken by a latch)
- ROM with unknown contents
- RAM contents if the Read_enable is off

- Faults that cause more than one port of a latch to turn on

  As an example, if a `TIEX` block feeds an AND gate, this makes it impossible to drive a `1` on the output of the AND gate. This, in turn, will prevent testing the `OSA0` fault on the AND gate.

Untestable: Seq or control not specified to target fault (Unio , Unil, Unlo, Unra, Uner)

The fault is in a portion of the logic that cannot be tested due to user control or test sequence. For example, if `create_logic_delay_tests` is set to process only intra-domain faults that require repeating clocks, then faults that are not in intra-domain are untestable. The user can control the selected sequence filter, a test sequence, test mode constraints (TCs) or lineholds. Refer to Delay and Timed Test in *Encounter Test: Guide 5: ATPG* for more information.

This status is further categoried by the path along which the fault needs to be detected:

- (Unio) -- untestable fault between Primary Inputs (PIs) and Primary Outputs (POs).
- (Unil) -- untestable fault between Primary Inputs (PIs) and Latches or Flip-Flops.
- (Unlo) -- untestable fault between Latches or Flip-Flops and Primary Outputs (POs).
- (Unra) -- untestable fault within intra-domain logic.
- (Uner) -- untestable fault within inter-domain logic.

Untestable: Testmode inhibits fault control/observe (Utm)

A test for this fault cannot be produced due to conflicts with pins that are already at value from setting the design into the state set up by the testmode.

A test for this fault conflicts with the TC (and perhaps the TI) pins or fixed value regs. Faults conflicting with a TI pin are normally identified as inactive in the testmode.

Untestable: Sequential depth (Usd)

This fault was discovered to be untestable during multiple time image processing. If the sequential test generator was not used, it may be able to test the fault.

Untestable: Global term (Ugt)

A test for this fault requires termination on some three-state primary output, but the tester-supplied termination does not permit termination to the state required by the test. If your IC manufacturer can support a different termination value, you can change the global tester termination value during test generation with the `globalterm` keyword. Note that changing the tester termination may cause the fault to become testable or to become untestable for a different reason.

Untestable: Constraint (Ucn)

A test for this fault cannot be produced without violating the clocking constraints specified by the automatic or user specified clock sequence(s).

Fault/Test Coverage Calculations

The normal static test coverage is calculated as follows:

$$\%\%\text{TCov (Test Coverage)} = \frac{\#\text{ Tested static faults}}{\#\text{ globally active static faults}} \times 100$$

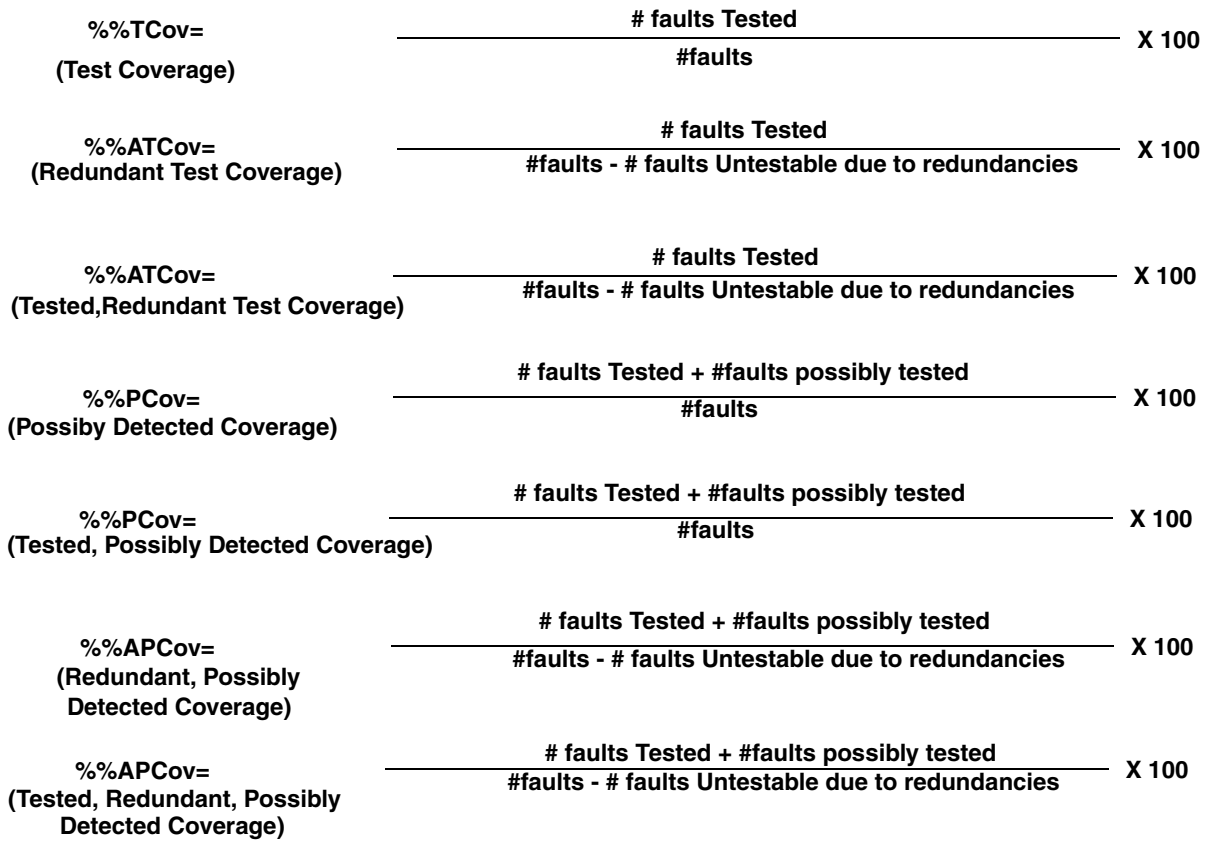
The ignore fault coverage is calculated as follows:

$$\%\%\text{TCov (Fault Coverage)} = \frac{\#\text{ Tested static faults}}{\#\text{ globally active static faults} + \text{globally ignored static faults}} \times 100$$

The following figures show the complete set of coverage calculations.

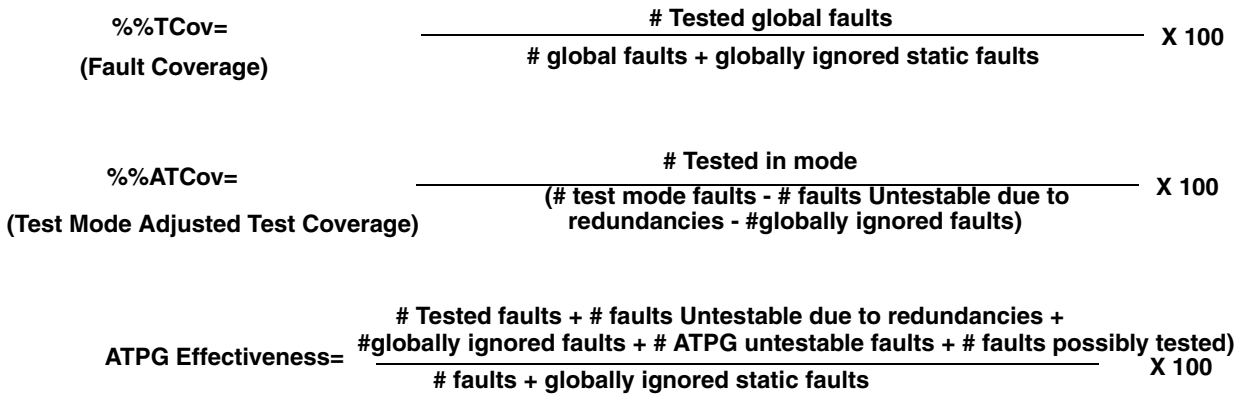
The calculations for dynamic faults are done similarly and the ignored counts are added to the denominator in the test coverage calculation. For ATCov (adjusted test coverage), the globally ignored faults are subtracted from the denominator. For more details, refer to "Report Fault Coverage Statistics" on page 32.

Figure 6-2 Test Coverage Formulas Used By Encounter Test



The following testmode coverage calculations are used when `build_faultmodel` option `ignorefaultstatistics=yes` is specified:

Figure 6-3 Test Coverage Formulas with Ignore Faults



Fault Modeling

The faults defined in the previous section are related to the design model in what is termed a *Fault Model*.

- For most test generation tasks, the standard Encounter Test fault model is created and used as input. This fault model includes all the static and dynamic pin and pattern faults as well as the driver/receiver objectives and Iddq faults.
- Stuck-Driver and Shorted Nets testing (Interconnect/IO Wrap Testing), requires a different fault model that includes only the objectives for these tests. This *Objective Model* can be created at the same time as the standard fault model, or may be created separately.
- The Path Delay fault model can be used as an input to Path Delay Test Generation. See "Path Delay" on page 92 for additional information.
- Any number of alternate fault models may also be created. These fault models can be an exact copy of the regular fault model if desired. These unique fault models were initially supported for use with diagnostics -- to allow for better modeling of some types of manufacturing defects such as bridging faults. However, their support has been generalized so they may be used for test generation if the need arises.

Note: Patterns generated using one fault model do not cause faults to be marked off in any of the other fault models. If that is desired, the patterns would need to be resimulated against the other fault model. Refer to Figure 1-1 on page 6 for a depiction of the flow.

Pin Faults and Pattern Faults

These two basic fault types are modeled in Encounter Test as:

- Pin Faults
  - Static (Stuck-At) Faults
  - Dynamic (Transition) Faults
- Pattern Faults
  - Static Pattern Faults
  - Dynamic Pattern Faults
  - Shorted net faults

Refer to "Fault Types" on page 73 for additional information.

Pin faults do not always accurately model certain static defects. For better defect modeling, Encounter Test provides support for pattern faults. A static pattern fault represents a static defect by identifying a set of values required on specific logic nets to excite the particular defect being modeled. Once excited, the defect's effect appears at the output of a logic gate or on some specific net and must be propagated to an observable point just like any other fault.

Pattern faults can be grouped such that two or more pattern faults represent a single defect. The group can be identified as either an ORing group or an ANDing group. An ORing group implies the defect is considered detected if any faults in the group are detected. An ANDing group implies the defect is considered detected only if all faults in the group are detected. Partial credit is given for those faults in an ANDing group which have been detected.

As an example, a short between two nets can be represented by two pattern faults in an ORing group. The first fault requires net A to be zero and net B to be a one, with the fault effect propagating at net B as a good design 1/faulty design zero (this assumes logic zero dominates a logic one). The second fault in the group requires net A to be a one and net B to be a zero, with the fault effect propagating from net A as a 1/0. Detecting either of these faults automatically gives credit to the other fault. Encounter Test provides a shorthand means of defining a two-net short defect within a fault rule file for an entity in the hierarchy of the design.

Cross-Mode Markoff (MARKOFF)

When you create a testmode, you can specify that the testmode belongs to one or more groups called comets. Comets are used to group testmodes for the purpose of reporting fault coverage. There are two types of comets:

■ Cross-Mode Markoff (MARKOFF) Comet

The cross-mode markoff comet allows you to obtain cumulative fault coverage across multiple testmodes and reduces test generation time by not reprocessing the fault in several different testmodes.

It also reduces tester time by only generating a single test for each fault. By default, a cross-mode markoff comet is created for each tester description rule name and all testmodes that use that tester description rule are automatically included in the same comet. This ensures that if a fault is tested in one testmode, it will not have another test generated in another testmode targeted for the same tester.

If a single cross-mode markoff comet is defined for the design, or if there are multiple markoff comets but they do not have any testmodes in common, then the rules are simple. When an experiment is committed, the tested status of the faults for that testmode is saved. If a tested fault is active in other testmodes, then commit_tests also processes the other testmodes that have the same markoff comet defined and marks the fault as tested in another mode (Tm).

The rules get a little more complicated if a testmode belongs to more than one markoff comet, so this is not recommended. Here is an example of how it works when testmodes are included in more than one markoff comet. Consider a design with three testmodes, A, B, and C, with three faults f1, f2, and f3 active in all three testmodes. Assume two comets have been defined: Comet1 consisting of modes A and C, and Comet2 consisting of modes B and C. Now, suppose a test generation run is committed for testmode A that tested both faults f1 and f2. These faults are marked tested in mode A, but are not marked in modes B and C, because no faults have yet been tested by modes belonging to Comet2. Next, suppose a test generation is run in mode B, which tests fault f1. When this experiment for mode B is committed, fault f1 is marked tested (T) in mode B and tested in another mode (Tm) in mode C. Note that fault f1 has now been tested for each comet, and mode C does not belong to any other markoff comets. Next, a test generation is run in mode C that tests fault f3. When the experiment for mode C is committed, fault f3 is marked tested (T) in mode C and tested in another mode (Tm) in both modes A and B.

When you report the tested faults in a testmode belonging to a markoff comet, the status indicates which faults are tested in that testmode and which faults are tested in another mode (Tm).

■ Statistics-Only (STATS_ONLY) Comet

The statistics-only comet allows you to obtain fault statistics for a group of testmodes without having cross-mode markoff.

Note: As stated previously, Encounter Test automatically creates a mark-off comet for the Tester Description Rule (TDR), therefore, if you have multiple testmodes that use the

same TDR, they will automatically get the benefit of cross-mode markoff. If you choose to define your own comets, it is recommended that you define the TDR comet as `stats_only` to avoid the complexity of having a testmode in more than one comet. This must be done in the mode definition file for each testmode that uses that TDR.

As an example, assume you are using a tester named HALLEY and you have four testmodes named MODE1, MODE2, MODE3, MODE4.

You want to define two comets; BORELLY to include MODE1 and MODE2 and HARTLEY to include MODE3 and MODE4.

By default MODE1 and MODE2 would be in BORELLY and HALLEY; and MODE3 and MODE4 would be in HARTLEY and HALLEY which would make the cross-mode markoff more complex. To avoid this, include the following comets statement in the mode definition files for MODE1 and MODE2:

```
COMETS =  HALLEY STATS_ONLY,  BORELLY MARKOFF ;
```

- This makes HALLEY a `stats_only` comet so it will not be considered during cross-mode markoff for this testmode.
- BORELLY will be used for cross-mode markoff so once a fault is tested in MODE1 it will be marked off in MODE2 and vice versa.

In the mode definition files for MODE3 and MODE4, include the following comets statement:

```
COMETS =  HALLEY STATS_ONLY,  HARTLEY MARKOFF ;
```

- This makes HALLEY a `stats_only` comet so it will not be considered during cross-mode markoff for this testmodes.
- HARTLEY will be used for cross-mode markoff so once a fault is tested in MODE1 it will be marked off in MODE2 and vice versa.

If you want to disable cross-mode markoff altogether, simply define the TDR as a `stats_only` comet in every testmode. However, this is not recommended as there is significant benefit to tester time, test generation time and the volume of vector data when using cross-mode markoff.

Other Test Objectives

Path Delay

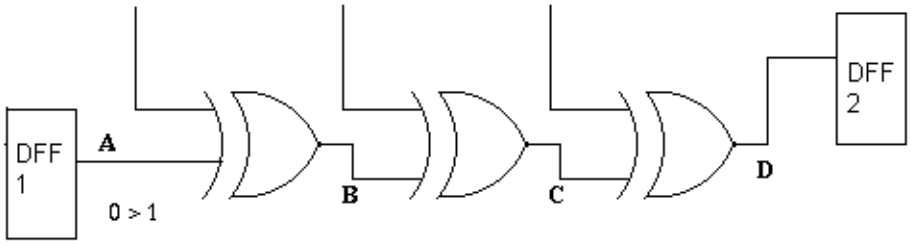
A signal path with a total delay exceeding the clock interval is a path delay fault, represented by functional pins or signals. A path delay fault consists of the combinational network between

a latch/primary input and a latch/primary output for which a transition in the specified direction does not arrive at the path output in time for proper capture.

A fault model may be built for use in *Path Delay Test Generation* using prepare_path_delay with a pathfile input.

Figure 6-4 shows an example of a path delay test objective. The transition from 0 to 1 is released from DFF1 along path A, B, C, D and captured at DFF2.

Figure 6-4 Path Delay Fault Example



Package Test Objectives

There are three types of test objectives for Package Test:

- Stuck Driver Test (SDT)
- Shorted Nets Test (SNT)
- Slow-to-Disable

Stuck Driver and Shorted Net Objectives

Stuck driver and shorted net objectives are defined for I/O wrap and interconnect test purposes. Both of these objectives are based upon target nets. For a chip, the target nets are those connected to bidirectional package pins. For higher level packages, the target nets are the package pin nets and the nets that interconnect the components.

Stuck Driver Objectives

Stuck driver objectives are defined to check the integrity of the path from a driver to each of its receivers. A driver is any combinational primitive logic function which is not a buffer, resistor or mux. When the net is a multi-source net, the drivers are the primitives that source the net. A receiver is any combinational primitive logic function which is not a buffer or resister. Typical examples of receivers are ANDs/ORs, and POs.

Shorted Nets Objectives

Shorted nets objectives are defined to check the integrity of the paths for a set of target nets. Each net participating in a shorted nets test will have one of its drivers supplying a value to the net at same time as other nets participating in the test. The values driven on to the nets is determined by a logorithmic counting pattern known as O=2*logN (or counting and complement) where N is the number of target nets and O is the number of shorted nets objectives. For example, six shorted nets objectives are defined for eight target nets (N=*) as follows:

```
Target        Objective Number
Net             1 2 3 4 5 6
------------------------------
n1              0 0 0 1 1 1
n2              0 0 1 1 1 0
n3              0 1 0 1 0 1
n4              0 1 1 1 0 0
n5              1 0 0 0 1 1
n6              1 0 1 0 1 0
n7              1 1 0 0 0 1
n8              1 1 1 0 0 0
```

Slow-To-Disable Objectives

A slow-to-disable objective is a stuck driver objective defined to detect a driver which is slow to transition to its inactive (disable) state. Definition of, and test generation for, slow-to-disable objectives is optional and only applies to interconnect testing. You must provide an objectivefile to the Build Package Test Objectives process for slow-to-disable objectives to be defined.

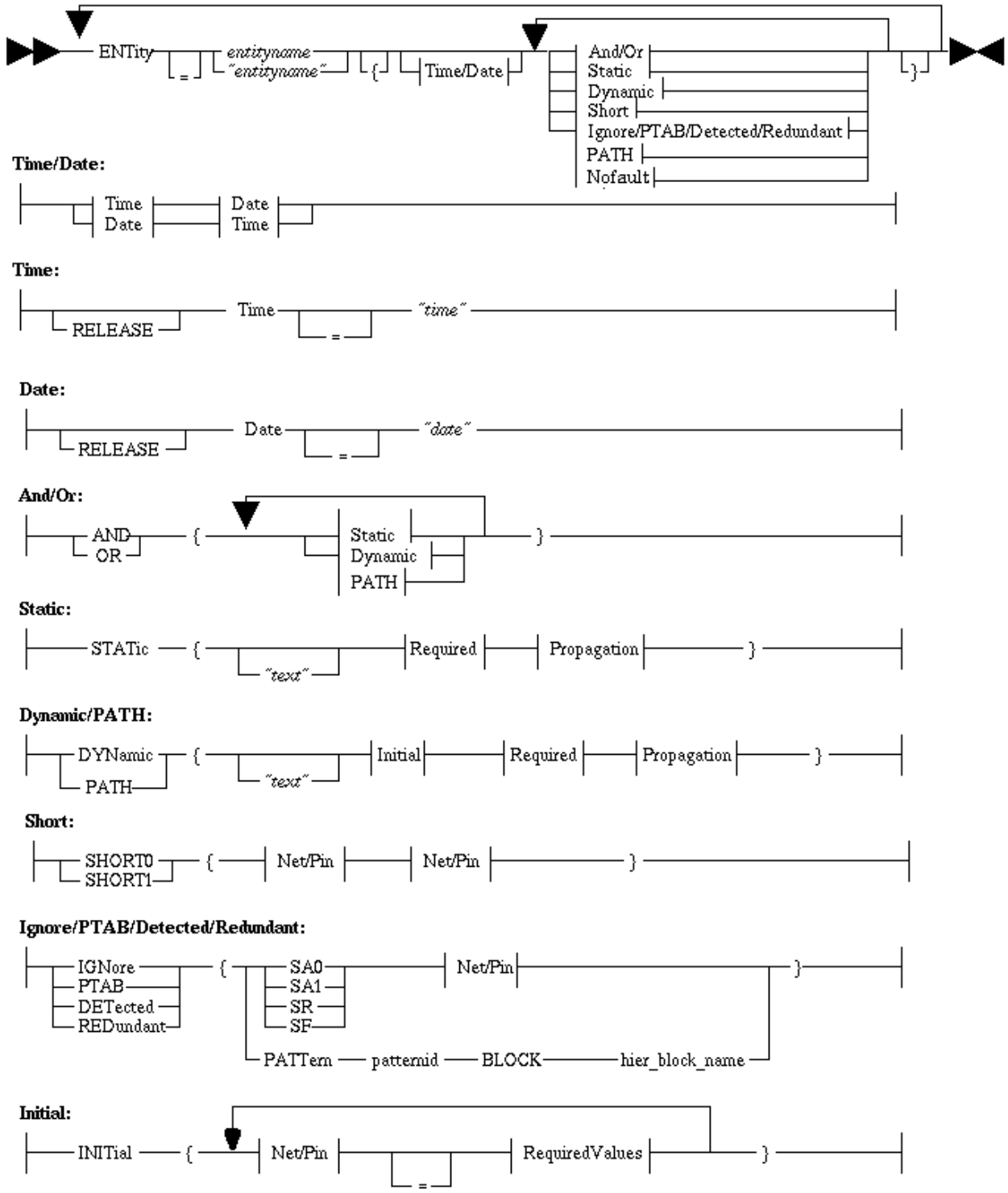
A

Pattern Faults and Fault Rules

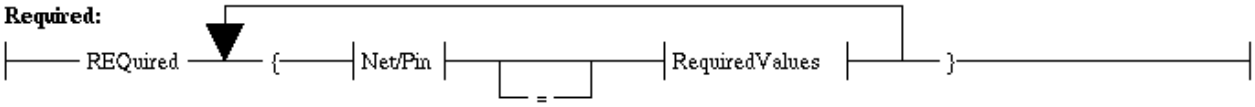
Fault Rule File Syntax

Following is the syntax specification for the fault rule file, which always starts by identifying the entity to which it pertains.

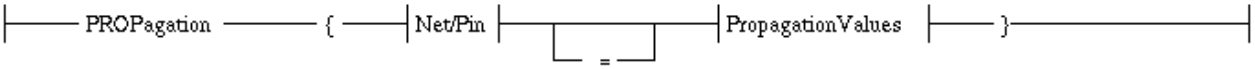
Figure A-1 Fault Rule File Syntax



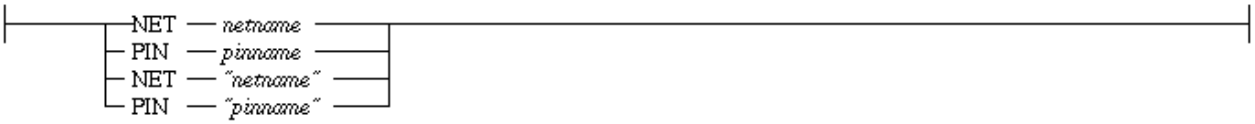
Required:



Propagation:



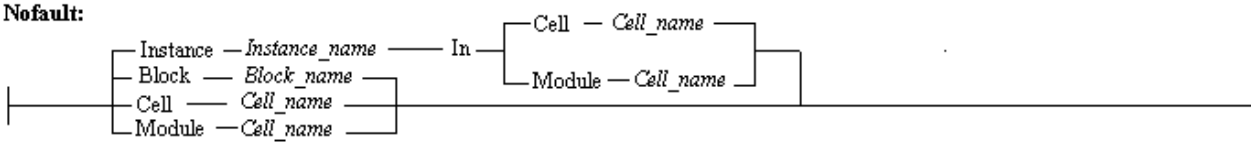
Net/Pin:



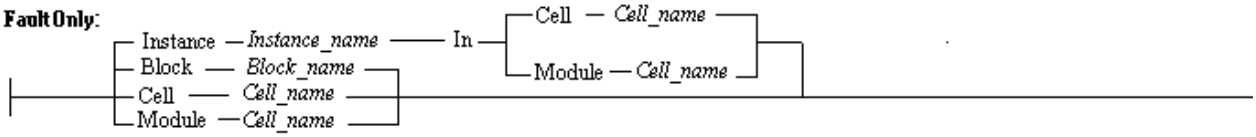
RequiredValues:

PropagationValues:

Nofault:



Fault Only:



Syntax Notes:

1. Comments delineated with a /* */ can be included wherever a space can exist. A line containing only a comment is valid. However, a comment may not span lines. That is, a comment cannot contain any newline characters. A single line comment delineated with `//` is also supported.
2. Keywords may be specified in upper, lower or mixed case within the fault rule file. The characters that appear in uppercase indicate the abbreviated form of the keyword. For example, the keyword "entity" is recognized when specified as either ENTITY or ENT.
3. Lower case characters indicate variables.
4. *`entityname`*, *`netname`,* and *`pinname`* can be any character string that does not contain blanks, tabs, newlines, or the { }, " ", /* */ delimiters.

   If the name matches any of the allowed fault rule keywords, it must be enclosed in quotes (for example, "z" or "short0").
5. "text" (a quoted string delineated with " ") may not span lines. It can contain any characters except newline and the double quote delimiter (").
6. Newline characters may be inserted anywhere in the fault rule file, except within a comment, keyword, variable or quoted string.
7. NOFAULT and FAULTONLY cannot be specified within the context of an ENTITY statement. NOFAULT and FAULTONLY are the only two fault rule statements that are allowed outside an ENTITY.

Fault Rule File Element Descriptions

The following describes the elements of the fault rule file syntax:

- ENTITY

  This defines the name of the containing block (a Verilog module) to which this fault rule file applies. It is acceptable to have more than one model entity included in the same fault rule file.

  Note: A curly brace is required in the syntax if specifying multiple entities as shown in Example A-1.

Example A-1 Multiple ENTITY Specification

```
############################################################
ENTITY = CLKINVX1
{
 IGNORE { SA0 PIN "Pin.f.l.CLKINVX1.nl.I0.A0" }
```

```
 IGNORE { SA1 PIN "Pin.f.l.CLKINVX1.nl.I0.01" }
}

ENTITY = PDUDGZ
{
 IGNORE { SA0 PIN "Pin.f.l.PDUDGZ.nl.__i0.tsd.ENABLE" }
 IGNORE { SA1 PIN "Pin.f.l.PDUDGZ.nl.__i0.tsd.DOUT" }
}
##########################################################
```

- TIME

  This provides a mechanism for auditing the fault rule.

- DATE

  This provides a mechanism for auditing the fault rule.

- AND

  Provides for the grouping of two or more pattern faults that model a single defect. An AND group implies the defect is considered detected only if all faults in the group are detected. Refer to Grouping (&, I) for more information.

- OR

  Provides for the grouping of two or more pattern faults that model a single defect. An OR group implies the defect is considered detected if any fault in the group is detected. Refer to Grouping (&, I) for more information.

- STATIC

  Begins the definition of a static pattern fault. A static pattern fault is used to model a defect which can be detected regardless of the speed at which the test patterns are applied. A static pattern fault specifies a set of REQUIRED net or pin values to excite the defect and a single PROPAGATION net or pin value to identify where the effect of the defect first appears.

- DYNAMIC

  Begins the definition of a dynamic pattern fault. A dynamic pattern fault is used to model a defect which requires a sequence of patterns to be applied to the design within a specific period of time. A dynamic pattern fault specifies both a set of INITIAL net or pin values and a set of REQUIRED net or pin values which together identify the nets which must go through a specific value change (transition), and a PROPAGATION net value to identify where the effect of the defect first appears.

- PATH

Identifies a fault that is to be marked as a "pathfault" in the fault model that is to be built with this fault rule. Refer to “Path Pattern Faults” on page 86 for additional information.

- REQUIRED

  A list of nets or pins and their values which are required to excite the defect.

- PROPAGATION

  Specifies the net or pin where the defect's effect first appears. Also specifies the good machine/faulty machine values for that net or pin in the presence of the fault excitation conditions.

- INITIAL

  A list of nets or pins and their values that represent the initial conditions required to excite a dynamic pattern fault. These nets and values are used in conjunction with the REQUIRED net value pairs to fully specify the excitation requirements for a dynamic pattern fault or a path pattern fault. In order for the defect to be considered "excited", the INITIAL net values must exist immediately before a clocking event that produces the REQUIRED net values.

- SHORT0

  A shorthand means of specifying two pattern faults which represent two nets shorted together with the effect that whichever net has a logical value of zero pulls the other net to zero as well.

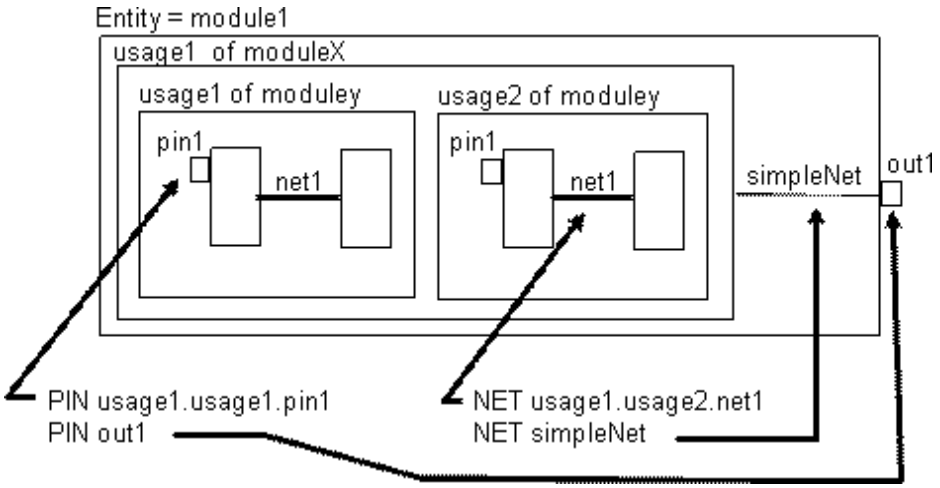
- SHORT1

  A shorthand means of specifying two pattern faults which represent two nets shorted together with the effect that whichever net has a logical value of one pulls the other net to one as well.

- NET

  The net must be within the context of module on the ENTITY statement. To specify a net defined in the module, use the simple name of the net, for example “net01”. To specify a net within a lower level entity, use the hierarchical name of the net, for example “usage1.net01”. See example in Figure A-2.

Figure A-2 Example of NET Statement



- PIN

  A pin name can be used instead of a net name when it is more convenient to do so; however, the pins being specified should not be pins on primitive gates since the Encounter Test Build Model process may rename the pins on primitive gates. The only pin on a logic gate which could be reliably specified in a fault rule file is the single output pin, whose pin name should always be "01". Pin names may be either simple; for example, "pin01", or they may be hierarchical; for example, "usage1.pin01".

  A hierarchical name is comprised of instance names for each level in the hierarchy down to the block containing the net or pin, and the simple name of the net or pin within the block which defines it. The instance names and simple name are joined with periods (.) between them. For example, usage1.usage2.net1 would refer to usage-block "usage1" within the definition block and within that block, the "usage2" usage-block and the net "net1" within it.

- IGNORE

  Identifies a fault that is to be ignored. These faults will be marked ignore in the fault model that is built with this fault rule. Unlike faults that are removed with a PFLT pin attribute, all faults that are found to be equivalent to an ignored fault (during fault collapsing) are also marked ignore.

  This element supports the use of wildcards as shown below. The wildcard is `*` character.

  ```
  IGNORE { SA1/SA0/SR/SF PIN rlm.usage*.pinZ}
  ```

  Note: `IGNORE NET` and `IGNORE PATTERN` do not allow wildcards

- PTAB

Identifies a fault that is to be marked "Possibly Testable at Best" in the fault model that is built with this fault rule.

The Write Ignore Faults application uses this syntax for faults that are identified as:

- untestable due to unknown (X) source.
- untestable due to three-state contention.
- untestable due to non-terminated three-state.
- possibly testable at best.

- DETECTED

  Identifies a fault that is to be marked "detected" in the fault model that is to be built with this fault rule.

- REDUNDANT

  Identifies a fault that is to be marked "redundant" in the fault model that is to be built with this fault rule.

- SA0

  A pin stuck-at-zero fault.

- SA1

  A pin stuck-at-one fault.

- SR

  A pin slow-to-rise fault.

- SF

  A pin slow-to-fall fault.

- PATTERN

  A pattern fault. The pattern fault identification is the unique identifier that is printed in the fault listing for this fault. Each pattern fault owned by a block is assigned a unique identifier.

  - PIN

    The pin containing the fault. For Ignore and PTAB faults, the hierarchical pin name refers to the specific pin (input or output) that contains the specified fault.

  - BLOCK

The block containing the pattern fault. This is the hierarchical name of the block that contains the pattern fault. Either short or proper form of the name is allowed.

- Text

  Text is a comment that is not discarded by the Fault Model Builder, but is carried forward into the Fault Model File. No syntax checking is done on the text inside the quotation marks, which may include any characters except carriage return or newline.The maximum length is 1024 characters. Text is used only for informational purposes; it is printed in Fault Model listings, may be viewed on the GUI, and is printed in the diagnostics callout report, but is ignored by all other Encounter Test applications and tools.

- RequiredValues

  A logic value or value pair used to represent the logic value(s) required for a net or pin in order to excite the defect. The basic syntax allows for specifying both the good design and the faulty design logic values required to excite the defect. When both are specified, the good design value is first, with a slash (/) used to separate the good and faulty design values. If only one value is specified, it is assumed that the same value is required in both good and faulty designs to excite the defect.

  In some cases, the logic value on the net or pin may not matter (a "don't care" condition). In these cases a logic value of X should be specified. For example, to model an input pin stuck-at-1 defect, the pattern fault required value should be specified as 0/X for the pin with the fault since the defect itself will cause the faulty design to assume a logic 1 value if all other requirements are met. A value of X is achieved in the good machine by the absence of a pin specification.

  Encounter Test allows a required value to be "don't care" in the faulty design, such as 1/X or 0/X. It is also possible to specify that the good design is a "don't care", such as X/0 or X/1. The former can be used to specify additional requirements to enable test generation to get the propagation node to its proper state so that there will be a difference between the good design and the faulty design. The latter can be used when the propagation node will have the correct value regardless of what the node has on it at the time, but the fault requires the value in order to excite the defect (cause the fault simulator to inject it).

  A value of P is used only in the specification of a path pattern fault. It is placed at other points along the path, including the master latch or primary output where the result of the transition is to be observed. Path pattern faults are normally defined only by the path test applications based on user specification of paths. Refer to Refer to Path File in *Encounter Test: Guide 5: ATPG* for additional information.

- PropagationValues

A logic value pair used to represent the logic values that will propagate from the specified net or pin whenever the defect has been excited. The values are specified for both the good design and the faulty design respectively, with a slash (/) used to separate them. It is important that the good design propagation value that is specified should be the value obtained whenever the good design required values are obtained. If the listed required values are not sufficient to produce the specified good design propagation value, it is unpredictable whether the fault will be detected by any tests that are automatically generated. V means a value of 0 or 1, that is, a "don't care", and ~V is used in conjunction with V to specify the opposite value of V. For example, 1 if V ends up being a 0, or 0 if V ends up being a 1.

- NOFAULT

  Identifies faults to be excluded by identifying logic that is not to have faults. The logic that is not be faulted is specified with Block, Instance, Cell, or Module.

  The type of faults to be excluded are static, dynamic, or both.

  A NOFAULT statement with no STATIC/DYNAMIC designation excludes both STATIC and DYNAMIC faults.

  - Instance - The name of the instance whose faults are to be excluded. Wildcards can be specified for the Instance name, for example, NOFAULT STATIC Instance a*c in CELL xyz (wildcard is allowed only for the instance name, not the cell name in this statement).

    Note: MODULE is a synonym for cell in this statement.

  - Block - The short or proper form of the hierarchical name of the block whose faults are to be excluded. Wildcards can be specified for the block name. For example, NOFAULT DYNAMIC Block *Z.

  - Cell or Module - The name of the module whose faults are to be excluded. Cell is a synonym of Module. Wildcards can be specified for the Cell or Module name. For example, CELL x*z or Module ab*.

  Names containing special characters must be enclosed in dowble quotes. Wildcards represent one or more alphanumeric characters and are not case sensitive. It is recommended not use wildcards for NOFAULT within FAULTONLY blocks.

  Note: When using a "*" wildcard character, if square brackets are part of the search string, a syntax error is generated on Linux platform. For example, if a string with the wildcard pattern "*rry[0]" is specified while searching for a block name "carry[0]" in the fault model, the search is successful on AIX and Solaris but not on Linux. Therefore while specifying patterns with "[" and "]" brackets, it is necessary to enclose each bracket character in a set of brackets "[]" in order for each bracket character to be treated as part of the name. The pattern "*rry[0]" needs to be specified as "*rry[[]0[]]". This is a known

problem.

- FAULTONLY

  Identifies specific logic to fault while excluding faults on all other blocks. The identifier can be specified in mixed case. Faults are placed on all appropriate blocks within this block, all the way down the hierarchy. You can specify more than one FaultOnly statement in a fault rule file. All blocks specified will be faulted.

  Following is the syntax of the `FAULTONLY` statement:

```
FAULTONLY <STATIC/DYNAMIC> Block <hier block name>
FAULTONLY <STATIC/DYNAMIC> Instance <instance_name> In Module <module_name>
FAULTONLY <STATIC/DYNAMIC> Cell <cell_name>
FAULTONLY <STATIC/DYNAMIC> Module <module_name>
```

  The type of faults included only on this logic can be static, dynamic, or both.

  A `FAULTONLY` statement with no STATIC/DYNAMIC designation affects both STATIC and DYNAMIC faults.

  Following is an explanation of the keywords used in the `FAULTONLY` statement:

  - Instance - The name of the instance for the logic to fault. You may also use wildcards such as asterisk (*) for the Instance name, for example `INSTANCE < name / wildcard name>` IN CELL name.

    For example, specifying FAULTONLY STATIC Instance a*c in CELL xyz will match FAULTONLY STATIC Instance abc in CELL xyz (wildcard is allowed only for the instance name, not the cell name).

  - Block - The short or proper name of the hierarchical name of the block for the logic of faults. Wildcards may be specified for the block name such as BLOCK `<name/ wildcard name>`. For example, BLOCK abc* will match BLOCK abcde, abcdef etc.

  - Cell or Module - The name of the module with the logic for faults. You may specify wildcards for the Cell name such as CELL `<name/wildcard name>`. For example, CELL *z will match CELL xyz and CELL 123xyz.

    Note: Wildcards represent one or more alphanumeric characters and are not case sensitive. It is recommended not use wildcards for NOFAULT within FAULTONLY blocks.

  - Module - Statement keyword that precedes a cell name. This can be specified in mixed case.

    Note: Cell and module are synonyms.

- Block name - Identifies a specific hierarchical block in the Encounter Test hierarchical model. This can be specified in both short and proper form of the block name. If a name contains special characters, it must be specified within double quotes.
- Cell name - Identifies a specific library cell name. If a name contains special characters, it must be specified within double quotes.
- Instance name - Identifies a specific instance within a cell or module. If a name contains special characters, it must be specified within double quotes.

NOFAULT can be used on modules and instances inside a FAULTONLY block to remove faults inside the FAULTONLY block. If NOFAULT and FAULTONLY are specified on the same block:

- For CELL and INSTANCE in CELL, the last statement specified in the cell will win.
- For an instance specific BLOCK, NOFAULT will override FAULTONLY, irrespective of the order in which the statements are specified.

How to Code a Fault Rule File

Example 1: Using Normal Pattern Files

```
/* This is the fault rule file for lsrfhxx  */

ENTITY = lsrfhxx       /* this is specified up front */

/* Here are all the static pattern fault definitions...  */

  Static {
     "internal resistive short to ground"  /* optional text */
     Required    /* required values */
     {Net netone 0
      Net nettwo 1
      net three 1
      }
      Propagation {     /* propagation value */
      net three 1/0                /* fault effect          */
     }
  }

  STATIC { "internal pin zz01 not connected (open)"
     REQ {
      Pin a20 0
      Net net2 1
     }
     PROP {
      net three 0/1
     }                         /* comment */
                               /* comment */
  }                            /* comment */
```

```
/*  Here are all the dynamic faults...  */

Or {
       /* the following two pattern faults represent one defect */
  Dynamic {
     "bridging fault between internal nodes X and Y"
     Initial {
       PIN a01    1
       PIN a02    0
     }
     REQ {
      PIN a01     1
      PIN a02     1
      PIN 02      1
     }
     PROP{ net abc 0/1 }
  }
  DYNAMIC {
     INIT {
       PIN a01    0
       PIN a02    1
     }
     REQ {
      PIN a01     1
      PIN a02     1
      PIN 02      1
     }
      PROP{ net abc 0/1 }
  }
} /* end OR */

/* shorted nets...  */
Short1 {
    NET netone
    NET nettwo
} /* end SHORT 1 */

Short0 {
    NET net3
    NET net4
} /* end SHORT 0 */

IGNORE { SA0  PIN "block1.pin10" }
IGNORE { PATT fault10 BLOCK "block2.latch" }
PTAB { SA1  PIN "block2.pin3" }
```

Example 2: Removing Faults

```
NoFault  Instance  reset_reg_0  In Module sys_reset_block
NoFault  Block  dig_top.u_sys_reset_block.reset_reg_0
NoFault  Cell SDFFRX1
NoFault  Module SDFFRX1
```

Note: An entity name is not required at the top of the file

Example 3: Using a Fault Rule File

```
create_diag_tests_transition faultrulefile=<file with listed Instances and
blocks>
```

Example 4: Combining Nofault and Individual Pattern Fault Specifications

The following example shows how to combine user-defined fault specification and `nofault` statement for a specific block. Note that the section for `Entity=top` should have both braces `{` and `}` before the `nofault` statement.

```
Entity=top {
IGNORE { SR PIN LOG.Dmux.SEL }
}
Nofault BLOCK Block.f.l.top.nl.COMPRESSION
```

Creating Shorted Net Fault Definitions

Shorted Net Faults can be created using the `create_shorted_net_faults` command.

Specifying a Shorted Net Fault in a Fault Rule File

A shorted net fault can be specified in a Encounter Test fault rule file using one of the following methods:

1. The first is "`SHORT0 NET=xyz NET=asdf`". This specification in the fault rule file creates two pattern faults in the Encounter Test fault model, with the specification of an OR to group the patterns together. In effect, the "SHORT0" specification is a short way of writing a full two-pattern ORed pattern fault.
2. A "SHORT1" specification is also a short way of writing two ORed pattern faults in the Encounter Test fault model. The SHORT0 specification describes a bridged net where the bridge acts like an AND gate. A SHORT1 specification describes a bridged net where the bridge acts like an OR gate.
3. Other kinds of behavior caused by shorted nets, such as a dominant net short, which cannot be described with the SHORT specification, require that you write the full specification of the two pattern faults in the Encounter Test fault rule file. The following table demonstrates this point for shorting two nets, A and B.

4. Use the command `create_shorted_net_faults`. Refer to "Creating Shorted Net Fault Definitions" on page 54.

Fault Rule File	Equivalent Pattern Fault	Fault Model

`SHORT0{NET A NET B}`	`OR{`
	`STATIC { "Net A=0 forces Net B=0"`
	`REQUIRED {NET A=0 NET B=1}`
	`PROPAGATION {NET B=1/0}`
	`}`
	`STATIC {"Net B=0 forces Net A=0"`
	`REQUIRED {NET A=1 NET B=0}`
	`PROPAGATION {NET A=1/0}`
	`}`
	`}`
		`Fault Index 1 SPAT at net B`
		`Fault Index 2 SPAT at net A`

`SHORT1{NET A NET B}`	`OR {`
	`STATIC {"Net A=1 forces Net B=1"`
	`REQUIRED {NET A=1 NET B=0}`
	`PROPAGATION {NET B=0/1}`
	`}`
	`STATIC {"Net B=1 forces Net A=1"`
	`REQUIRED {NET A=0 NET B=1}`
	`PROPAGATION {NET A=0/1}`
	`}`
	`}`
		`Fault Index 3 SPAT at net B`
		`Fault Index 4 SPAT at net A`

`OR{`
`STATIC {"Net A=1 Dominates Net B=0"`
`REQUIRED {NET A=1 NET B=0}`
`PROPAGATION { NET B=0/1}`
`}`
`STATIC {"Net A=0 Dominates Net B=1"`
`REQUIRED {NET A=0 NET B=1}`
`PROPAGATION { NET B=1/0}`
`}`
`}`
	`Same as fault rule file specification`	`Fault Index 5 SPAT at net B`
		`Fault Index 6 SPAT at net B`

To simulate the effects of the "SHORT1" from a fault rule file, you must specify "gmach=3,4" when resimulating the patterns in order to get the full effect of the shorted net. You must specify both components of the ORed pattern fault.

Using Create Shorted Net Command

Faults may be automatically defined to model shorted net defects using the command `create_shorted_net_faults`. This command uses an input list of net pairs and generates a set of shorted net faults. The resulting output file can then be used as fault rule input to Build Fault Model.

- To view the command parameters, refer to "create_shorted_net_faults" in the *Encounter Test: Reference: Commands*.

An example of the `create_shorted_net_faults` command is shown below:

```
create_shorted_net_faults cellname=moduleA outputfile=moduleA_flts \
filename=netpairfile
```

The following is a sample of a simple net pair file:

```
THE_REG_FILE.n_44 THE_REG_FILE.n_2403
THE_REG_FILE.n_4156 THE_REG_FILE.n_2403
THE_REG_FILE.n_88 THE_REG_FILE.n_2403
```

Net-Pair File Syntax

The following is an example of the syntax for each line of the net-pair file:

```
static fault spec dynamic fault spec netname1 netname2 ...
```

The static and dynamic fault specs are optional. The specified keyword values for `create_shorted_net_faults` indicate a static or dynamic fault spec for a net pair not associated with any specific fault spec.

Encounter Test supports the following static faults:

`staticOr`	Short1 of net1 with net2
`staticAnd`	Short0 of net1 with net2
`staticBoth`	Short1 and Short0 of net1 with net2 - 2 fault definitions. This fault type only cannot specify more than two nets.
`staticDom`	net1 dominates net2 - ORed pattern faults. This fault type can specify more than two nets.

`staticDomBoth`	net1 dominates net2 and net2 dominates net1 - 2 sets of ORed faults. This fault type is the default for static faults.
`staticAll`	`staticBoth` and `staticDomBoth` between net1 and net2
`split`	This optional modifier indicates individual components are not defined as ORed pattern faults.

Encounter Test supports the following dynamic faults:

`delayOr`	net1 is slow to fall if net2 is 1 and vice versa - ORed
`delayAnd`	net1 is slow to rise if net2 is 0 and vice versa - ORed
`delayBoth`	Both `delayOr` and `delayAnd`. This fault type cannot specify more than two nets.
`delayDom`	net1 makes net2 slow to get to the opposite value - ORed. This fault type can specify more than two nets.
`delayDomBoth`	Both net1 dominates net2 and net2 dominates net1
`delay`	Creates the delay version of the defined static fault spec. This fault type is the default for dynamic faults.
`delayAll`	`delayBoth` and `delayDomBoth` between net1 and net2
`delayTrans`	Transition on net1 slows net2 for opposite trans - ORed. This fault type can specify more than two nets.
`delayTransBoth`	Both net1 trans slow net2 and net2 trans slow net1
`split`	This optional modifier indicates individual components are not defined as ORed pattern faults.

Note: Net names may be coded with single or double quotes to avoid confusion with a fault specification.

Fault Rule File Output

The following table shows examples of lines from an input net-pair file and the associated output fault rule definitions.

Input File Line	Resulting Output
staticOr netA netB	`Short1 { Net "netA" Net "netB" }`
staticAnd netA netB	`Short0 { Net "netA" Net "netB" }`
staticBoth netA netB	`Short0 { Net "netA" Net "netB" }` `Short1 { Net "netA" Net "netB" }`

staticDom netA netB	`OR{`
	` Static{`
	`  "dominant short"`
	`  REQ { Net "netA"=0 Net "netB"=1/X }`
	`  PROP { Net "netB"=1/0 }`
	` }`
	` Static{`
	`  "dominant short"`
	`  REQ { Net "netA"=1 Net "netB"=0/X }`
	`  PROP { Net "netB"=0/1 }`
	` }`
	`}`

staticDomBoth netA netB	`AND{`
	` Static{`
	`  "dominant short"`
	`  REQ { Net "netA"=0 Net "netB"=1/X }`
	`  PROP { Net "netB"=1/0 }`
	` }`
	` Static{`
	`  "dominant short"`
	`  REQ { Net "netA"=1 Net "netB"=0/X }`
	`  PROP { Net "netB"=0/1 }`
	` }`
	` Static{`
	`  "dominant short"`
	`  REQ { Net "netB"=0 Net "netA"=1/X }`
	`  PROP { Net "netA"=1/0 }`
	` }`
	` Static{`
	`  "dominant short"`
	`  REQ { Net "netB"=1 Net "netA"=0/X }`
	`  PROP { Net "netA"=0/1 }`
	` }`
	`}`

Input File Line	Resulting Output

staticAll netA netB	`Short0 { Net "netA" Net "netB" }`
	`Short1 { Net "netA" Net "netB" }`
	`AND{`
	`Static{`
	`"dominant short"`
	`REQ { Net "netA"=0 Net "netB"=1/X }`
	`PROP { Net "netB"=1/0 }`
	`}`
	`Static{`
	`"dominant short"`
	`REQ { Net "netA"=1 Net "netB"=0/X }`
	`PROP { Net "netB"=0/1 }`
	`}`
	`Static{`
	`"dominant short"`
	`REQ { Net "netB"=0 Net "netA"=1/X }`
	`PROP { Net "netA"=1/0 }`
	`}`
	`Static{`
	`"dominant short"`
	`REQ { Net "netB"=1 Net "netA"=0/X }`
	`PROP { Net "netA"=0/1 }`
	`}`
	`}`

staticDom Split netA netB	`Static{`
	`"dominant short"`
	`REQ { Net "netA"=0 Net "netB"=1/X }`
	`PROP { Net "netB"=1/0 }`
	`}`
	`Static{`
	`"dominant short"`
	`REQ { Net "netA"=1 Net "netB"=0/X }`
	`PROP { Net "netB"=0/1 }`
	`}`

delayAnd netA netB	`OR {`
	`Dynamic {`
	`"Delay short 0"`
	`INIT { Net "netB"=0 }`
	`REQ { Net "netA"=0 Net "netB"=1/X }`
	`PROP { Net "netB"=1/0 }`
	`}`
	`Dynamic {`
	`"Delay short 0"`
	`INIT { Net "netA"=0 }`
	`REQ { Net "netA"=1/X Net "netB"=0 }`
	`PROP { Net "netA"=1/0 }`
	`}`
	`}`

Input File Line	Resulting Output

delayOr netA netB	`OR {`
	` Dynamic {`
	`  "Delay short 1"`
	`  INIT { Net "netB"=1 }`
	`  REQ { Net "netA"=1 Net "netB"=0/X }`
	`  PROP { Net "netB"=0/1 }`
	` }`
	` Dynamic {`
	`  "Delay short 1"`
	`  INIT { Net "netA"=1 }`
	`  REQ { Net "netA"=0/X Net "netB"=1 }`
	`  PROP { Net "netA"=0/1 }`
	` }`
	`}`

delayBoth netA netB	`OR {`
	` Dynamic {`
	`  "Delay short 0"`
	`  INIT { Net "netB"=0 }`
	`  REQ { Net "netA"=0 Net "netB"=1/X }`
	`  PROP { Net "netB"=1/0 }`
	` }`
	` Dynamic {`
	`  "Delay short 0"`
	`  INIT { Net "netA"=0 }`
	`  REQ { Net "netA"=1/X Net "netB"=0 }`
	`  PROP { Net "netA"=1/0 }`
	` }`
	`}`
	`OR {`
	` Dynamic {`
	`  "Delay short 1"`
	`  INIT { Net "netB"=1 }`
	`  REQ { Net "netA"=1 Net "netB"=0/X }`
	`  PROP { Net "netB"=0/1 }`
	` }`
	` Dynamic {`
	`  "Delay short 1"`
	`  INIT { Net "netA"=1 }`
	`  REQ { Net "netA"=0/X Net "netB"=1 }`
	`  PROP { Net "netA"=0/1 }`
	` }`
	`}`

Input File Line	Resulting Output
delayDom netA netB	`OR { Dynamic { "Delay dominant short" INIT { Net "netB"=0 } REQ { Net "netA"=0 Net "netB"=1/X } PROP { Net "netB"=1/0 } } Dynamic { "Delay dominant short" INIT { Net "netB"=1 } REQ { Net "netA"=1 Net "netB"=0/X } PROP { Net "netB"=0/1 } } }`
delayDomBoth netA netB	`OR { Dynamic { "Delay dominant short" INIT { Net "netA"=0 Net "netB"=0 } REQ { Net "netA"=0 Net "netB"=1/X } PROP { Net "netB"=1/0 } } Dynamic { "Delay dominant short" INIT { Net "netA"=1 Net "netB"=1 } REQ { Net "netA"=1 Net "netB"=0/X } PROP { Net "netB"=0/1 } } } OR { Dynamic { "Delay dominant short" INIT { Net "netB"=0 Net "netA"=0 } REQ { Net "netB"=0 Net "netA"=1/X } PROP { Net "netA"=1/0 } } Dynamic { "Delay dominant short" INIT { Net "netB"=0 Net "netA"=1 } REQ { Net "netB"=1 Net "netA"=0/X } PROP { Net "netA"=0/1 } } }`
delayDom Split netA netB	`Dynamic { "Delay dominant short" INIT { Net "netB"=0 } REQ { Net "netA"=0 Net "netB"=1/X } PROP { Net "netB"=1/0 } } Dynamic { "Delay dominant short" INIT { Net "netB"=1 } REQ { Net "netA"=1 Net "netB"=0/X } PROP { Net "netB"=0/1 } }`

Input File Line	Resulting Output

delayAll netA netB	OR {
	Dynamic {
	"Delay short 0"
	INIT { Net "netB"=0 }
	REQ { Net "netA"=0 Net "netB"=1/X }
	PROP { Net "netB"=1/0 }
	}
	Dynamic {
	"Delay short 0"
	INIT { Net "netA"=0 }
	REQ { Net "netA"=1/X Net "netB"=0 }
	PROP { Net "netA"=1/0 }
	}
	}
	OR {
	Dynamic {
	"Delay short 1"
	INIT { Net "netB"=1 }
	REQ { Net "netA"=1 Net "netB"=0/X }
	PROP { Net "netB"=0/1 }
	}
	Dynamic {
	"Delay short 1"
	INIT { Net "netA"=1 }
	REQ { Net "netA"=0/X Net "netB"=1 }
	PROP { Net "netA"=0/1 }
	}
	}
	AND {
	Dynamic {
	"Delay dominant short"
	INIT { Net "netA"=0 Net "netB"=0 }
	REQ { Net "netA"=0 Net "netB"=1/X }
	PROP { Net "netB"=1/0 }
	}
	Dynamic {
	"Delay dominant short"
	INIT { Net "netA"=1 Net "netB"=1 }
	REQ { Net "netA"=1 Net "netB"=0/X }
	PROP { Net "netB"=0/1 }
	}
	}
	OR {
	Dynamic {
	"Delay dominant short"
	INIT { Net "netB"=0 Net "netA"=0 }
	REQ { Net "netB"=0 Net "netA"=1/X }
	PROP { Net "netA"=1/0 }
	}
	Dynamic {
	"Delay dominant short"
	INIT { Net "netB"=1 Net "netA"=1 }
	REQ { Net "netB"=1 Net "netA"=0/X }
	PROP { Net "netA"=0/1 }
	}
	}

Input File Line	Resulting Output

delayTrans netA netB	`OR {`
	` Dynamic {`
	`  "Delay transition interference"`
	`  INIT { Net "netA"=1 Net "netB"=0 }`
	`  REQ { Net "netA"=0 Net "netB"=1/X }`
	`  PROP { Net "netB"=1/0 }`
	` }`
	` Dynamic {`
	`  "Delay transition interference"`
	`  INIT { Net "netA"=0 Net "netB"=1 }`
	`  REQ { Net "netA"=1 Net "netB"=0/X }`
	`  PROP { Net "netB"=0/1 }`
	` }`
	`}`

delayTransBoth netA netB	`OR {`
	` Dynamic {`
	`  "Delay transition interference"`
	`  INIT { Net "netA"=1 Net "netB"=0 }`
	`  REQ { Net "netA"=0 Net "netB"=1/X }`
	`  PROP { Net "netB"=1/0 }`
	` }`
	` Dynamic {`
	`  "Delay transition interference"`
	`  INIT { Net "netA"=0 Net "netB"=1 }`
	`  REQ { Net "netA"=1 Net "netB"=0/X }`
	`  PROP { Net "netB"=0/1 }`
	` }`
	`}`
	`OR {`
	` Dynamic {`
	`  "Delay transition interference"`
	`  INIT { Net "netB"=1 Net "netA"=0 }`
	`  REQ { Net "netB"=0 Net "netA"=1/X }`
	`  PROP { Net "netA"=1/0 }`
	` }`
	` Dynamic {`
	`  "Delay transition interference"`
	`  INIT { Net "netB"=0 Net "netA"=1 }`
	`  REQ { Net "netB"=1 Net "netA"=0/X }`
	`  PROP { Net "netA"=0/1 }`
	` }`
	`}`

Input File Line	Resulting Output

delayTrans netA netB netC	`OR {`
	` Dynamic {`
	`  "Delay transition interference"`
	`  INIT { Net "netA"=1 Net "netB"=1 Net "netC"=0 }`
	`  REQ { Net "netA"=0 Net "netB"=0 Net "netC"=1/X }`
	`  PROP { Net "netC"=1/0 }`
	` }`
	` Dynamic {`
	`  "Delay transition interference"`
	`  INIT { Net "netA"=0 Net "netB"=0 Net "netC"=1 }`
	`  REQ { Net "netA"=1 Net "netB"=1 Net "netC"=0/X }`
	`  PROP { Net "netC"=0/1 }`
	` }`
	`}`

staticDom delay netA netB	`OR{`
	` Static{`
	`  "dominant short"`
	`  REQ { Net "netA"=0 Net "netB"=1/X }`
	`  PROP { Net "netB"=1/0 }`
	` }`
	` Static{`
	`  "dominant short"`
	`  REQ { Net "netA"=1 Net "netB"=0/X }`
	`  PROP { Net "netB"=0/1 }`
	` }`
	`}`
	`OR {`
	` Dynamic {`
	`  "Delay dominant short"`
	`  INIT { Net "netB"=0 }`
	`  REQ { Net "netA"=0 Net "netB"=1/X }`
	`  PROP { Net "netB"=1/0 }`
	` }`
	` Dynamic {`
	`  "Delay dominant short"`
	`  INIT { Net "netB"=1 }`
	`  REQ { Net "netA"=1 Net "netB"=0/X }`
	`  PROP { Net "netB"=0/1 }`
	` }`
	`}`

Input File Line	Resulting Output

netA netB
 	
Command line options:
defaultStatic=staticDomBoth
defaultDelay=delay split=no
	AND{
	Static{
	"dominant short"
	REQ { Net "netA"=0 Net "netB"=1/X }
	PROP { Net "netB"=1/0 }
	}
	Static{
	"dominant short"
	REQ { Net "netA"=1 Net "netB"=0/X }
	PROP { Net "netB"=0/1 }
	}
	Static{
	"dominant short"
	REQ { Net "netB"=0 Net "netA"=1/X }
	PROP { Net "netA"=1/0 }
	}
	Static{
	"dominant short"
	REQ { Net "netB"=1 Net "netA"=0/X }
	PROP { Net "netA"=0/1 }
	}
	}
	AND {
	Dynamic {
	"Delay dominant short"
	INIT { Net "netA"=0 Net "netB"=0 }
	REQ { Net "netA"=0 Net "netB"=1/X }
	PROP { Net "netB"=1/0 }
	}
	Dynamic {
	"Delay dominant short"
	INIT { Net "netA"=1 Net "netB"=1 }
	REQ { Net "netA"=1 Net "netB"=0/X }
	PROP { Net "netB"=0/1 }
	}
	}
	OR {
	Dynamic {
	"Delay dominant short"
	INIT { Net "netB"=0 Net "netA"=0 }
	REQ { Net "netB"=0 Net "netA"=1/X }
	PROP { Net "netA"=1/0 }
	}
	Dynamic {
	"Delay dominant short"
	INIT { Net "netB"=1 Net "netA"=1 }
	REQ { Net "netB"=1 Net "netA"=0/X }
	PROP { Net "netA"=0/1 }
	}
	}

B

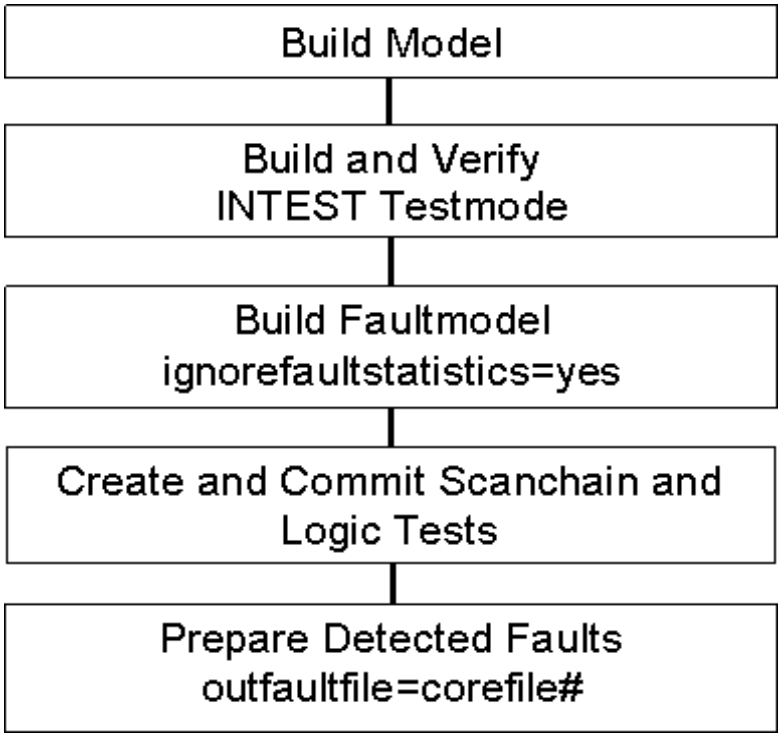
Hierarchical Fault Processing Flow

This is a flow for taking faults tested at the core level out of consideration when the 1500 wrapped cores are used on an SOC

Core Level Flow

At the core level the following steps are run for each 1500 wrapped core to be placed on the SOC.

Figure B-1 Core Level Flow



1. Build Model – run `build_model` with the options you normally use

2. Build and Verify INTEST Testmode – run `build_testmode` with a mode definition file that includes `Scan_type gsd boundary=internal`
3. Build the faultmodel including ignorefaults in the fault statistics – run `build_faultmodel ignorefaultstatistics=yes includeignore=yes`
4. Create and Commit scanchain and logic tests using the following command lines:

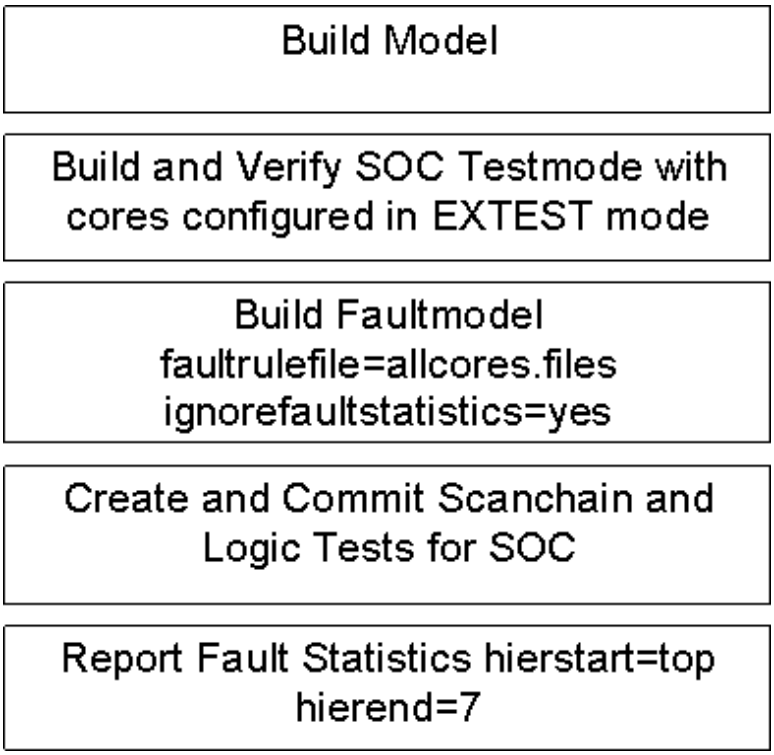
```
create_scanchain_tests testmode=INTEST experiment=srt
commit_tests testmode=INTEST inexperiment=srt
create_logic_tests testmode=INTEST experiment=tg1
commit_tests testmode=INTEST inexperiment=tg1
```

5. Prepare Detected Faults – run `prepare_detected_faults testmode=INTEST outfaultfile=corefile#` (where # is an arbitrary number to make the corefiles unique)

SOC Level Flow

At the SOC level the following steps are run to include each 1500 wrapped core.

Figure B-2 SOC Level Flow



1. Build Model – run `build_model` with your normal options; instead of pointing to Verilog for the core, you may point to the tbdata/hierModel2 for the core. This is somewhat faster than parsing the Verilog and ensures that the source is exactly what was used for ATPG.
2. Build and Verify SOC testmode with cores configured in EXTEST mode – run `build_testmode` and then `verify_test_structures` using your normal options and an assignfile that configures the testmode into EXTEST mode.
3. Build Faultmodel—create a file named allcores.files that lists the location of the corefile# from `prepare_detected_faults` on each testmode one per line with no punctuation. Then `build_faultmodel faultrulefile=allcores.files ignorefaultstatistics=yes includeignore=yes`
4. Create and Commit Scanchain and Logic Tests for SOC.

```
create_scanchain_tests testmode=SOC experiment=srt
commit_tests testmode=SOC inexperiment=srt
create_logic_tests testmode=SOC experiment=tg1
commit_tests testmode=SOC inexperiment=tg1
```

5. Report Fault Statistics – run `report_fault_statistics testmode=SOC hierstart=top hierend=`*`n`* (where *n* is the number of levels down from the top required to print the core level statistics)

C

Building Register Array and Random Resistant Fault List Files for Pattern Compaction

As designs grow larger, various techniques are employed by Encounter Test to reduce pattern counts. The command `prepare_faultmodel_analysis` can be used to identify large register arrays, also known as low-power registers, and identify special random pattern resistant structures. The benefit of the analysis is that faults that can be tested using the same write or read port address can be merged into a single test. When combined with test generation compaction options, a reduction in test pattern counts for large designs with arrays is possible.

The register array and random fault list files can also be produced as part of the Build Fault Model process by selecting the screen options to *Identify random resistant faults* and *Identify register array faults*, or via the command line by specifying `build_faultmodel` keywords `randomresistant=yes` and `registerarray=yes`.

To prepare fault model analysis, refer to "prepare_faultmodel_analysis" in the *Encounter Test: Reference: Commands*. The syntax for the command is shown below:

```
prepare_faultmodel_analysis WORKDIR=<directory>
```

where `workdir` is the name of the working directory.

Input Files

- Encounter Test model from `build_model`.
- If using `prepare_faultmodel_analysis` to generate these lists of faults, a fault model must exist from `build_faultmodel`.

Output Files

Two sets of data are produced in the tbdata directory:

- registerArrayList is produced if analysis is performed for register arrays.
- resistantFaultList is produced if analysis is performed for random resistant faults.

These are optional inputs to Logic Test Generation and IDDq Test Generation. Use the following command line syntax to use this data:

```
create_logic_tests workdir=mywd testmode=FULLSCAN experiment=stg1
randomresistant=yes registerarray=yes

create_logic_delay_tests workdir=mywd testmode=FULLSCAN experiment=dtg1
randomresistant=yes registerarray=yes

create_iddq_tests workdir=mywd testmode=FULLSCAN experiment=itg1
randomresistant=yes registerarray=yes
```

Index

A

C

D

F

G

H

I

L

M

N

O

P

R

S

T

U

W

X